



Rapport de Stage :

*Développement d'un add-on pour
l'application de messagerie instantanée
Libon sous iOS*

Jérémie Foucault

Maître de Stage : Fabien Crapiz
Tuteur : Mathieu Faverge
Troisième année, filière informatique
Bordeaux, le 31 août 2015

Table des matières

1	Introduction	4
2	Présentation du stage	5
2.1	Le groupe <i>Orange</i>	5
2.2	Nature du poste et environnement de travail	6
2.3	Sujet de stage	7
3	Réalisation du projet	8
3.1	Conception de l'application de démonstration	8
3.1.1	Storyboard	8
3.1.2	Modèle de données	9
3.1.3	Architecture de l'application	11
3.1.3.1	Input View :	12
3.1.3.2	Navigation Stack :	13
3.1.3.3	Collection View & Flow Layout :	15
3.2	De l'application de démonstration à <i>Libon</i>	17
3.2.1	Contexte et spécificités de <i>Libon</i>	17
3.2.2	Réutilisation du code	18
3.3	Communication et interactions avec l'application hôte : <i>Libon</i>	19
3.3.1	Un processus de développement loin d'être industrialisé.	19
3.3.2	Chargement de la librairie <i>WittiZ</i>	19
3.3.3	Interactions avec le champ de saisie du texte	22
3.3.3.1	Un mécanisme puissant : les <i>Delegates</i>	22
3.3.3.2	Affichage d'images d'aperçu avec le texte	24
3.3.3.2.1	Modification du <i>parsing</i> :	26
3.3.3.2.2	Requête asynchrone et <i>parsing</i> :	27
3.3.4	Algorithme pour l'envoi des vidéos	29
3.3.4.1	Découpage du champ texte en morceaux de même type	29
3.3.4.2	Chaîne d'envoi et de réception des <i>Events</i>	30
3.3.4.3	Un <i>Event</i> adapté à la transmission des vidéos <i>WittiZ</i> .	33
4	Conclusion	36
5	Bilan	36

Remerciements

Le projet de fin d'étude marque la fin de mes études et constitue un véritable tremplin pour mon début de carrière professionnelle. Ainsi je remercie le groupe Orange et sa direction qui joue le jeu des stages en nous recevant dans de très bonnes conditions. Par ailleurs, je tiens tout d'abord à remercier mon maître de stage Fabien Crapiz qui a contribué au bon déroulement de mon stage dans un contexte parfois difficile. Merci également à Mathieu Favergé pour son suivi et son implication en tant que tuteur pédagogique tout au long de mon stage. Je souhaite également remercier toute l'équipe pour leur accueil, leur bonne humeur et leur disponibilité pour m'aider dès que nécessaire. Enfin, merci à Alexis Carrel-Billiard qui a su me faire confiance en me recrutant pour ce stage.

1 Introduction

La dernière année du cycle ingénieur à l'*ENSEIRB-MATMECA* est une année de spécialisation qui oriente fortement le secteur d'activité dans lequel nous pourrons travailler après l'école. Mon choix de filière a été motivé par plusieurs facteurs. Tout d'abord l'attrait que j'avais pour le domaine et le secteur visé, mais également les compétences enseignées et enfin le potentiel du secteur d'activité. En la matière, il semble raisonnable de dire que l'internet des objets est un des secteurs en pleine explosion et promis à un avenir certain. Pour ces raisons j'ai choisi la filière Génie Logiciel des Réseaux de Télécommunications qui est transverse avec le département télécom. Cette spécialité m'a permis d'acquérir des compétences sur des sujets orientés télécom, mais également d'être initié à certaines technologies pour l'internet des objets. Mon seul regret, le manque de veille technologique dans un secteur pourtant en pleine ébullition.

Mon stage de *2^{ième}* année, s'est déroulé dans une PME, où le département Soft était constitué de 3 personnes. J'avais beaucoup apprécié le travail dans cette petite équipe avec un fonctionnement start-up. Fort de cette expérience, mon souhait était alors de débuter ma carrière dans une start-up ou PME. Cependant n'ayant pas d'expérience dans un grand groupe, je souhaitais pour ce dernier stage, trouver une mission dans une société de taille importante. Ceci était ma première exigence durant ma recherche de stage. Ma seconde exigence concernait les technologies, n'ayant pas vraiment d'expérience dans le développement mobile, j'ai souhaité trouver une mission avec du développement mobile, de préférence, pour la plateforme *iOS*. Enfin, le dernier critère était le domaine ; j'aurais souhaité en priorité travailler dans le domaine de l'internet des objets. Malheureusement les grands groupes ne s'intéressent, du moins publiquement, que peu à ce domaine. Finalement j'ai accepté l'offre de stage d'*Orange* qui consistait à réaliser un add-on pour l'application *Libon* sous *iOS* et qui réunissait tout de même deux de mes trois critères de sélection pour mon stage.

2 Présentation du stage

2.1 Le groupe Orange

Le marché des télécoms a connu de grands changements ces dernières années mais ces changements n'ont pas forcément été synonymes de croissance et profits. Ainsi, *Orange* a connu plusieurs années difficiles dernièrement et commence seulement à voir les bénéfices des changements de stratégie imposés. Le groupe *Orange* est avant tout une société internationale qui compte aujourd'hui plus de 244 millions de clients dans le monde. Elle est aussi bien connue pour ses produits et services grand public que ses solutions destinées aux professionnels. *Orange*, c'est également une infrastructure réseau qu'elle doit entretenir et développer à la fois sur le territoire national avec le déploiement de la 4G dernièrement, mais aussi international avec par exemple le déploiement de câbles sous marin entre les continents. En résumé, le groupe *Orange* aujourd'hui c'est :



Fig. 1 : Le groupe *Orange* en quelques chiffres

Orange Labs Products and Services (OLPS) est une entité d'*Orange* née en 2013 de la fusion entre *Orange Labs* et *DPS* qui réalisait, déployait et maintenait les plates-formes de services. *OLPS* a désormais la charge de s'occuper du cycle de vie complet (stratégie, maintenance, ...) des produits et services mis en œuvre partout dans le monde par *Orange*. Les missions et objectifs de *OLPS* sont variés, ils sont aussi bien techniques : se centrer sur la recherche et être moteur de l'innovation pour le groupe, que managériales avec de nouvelles méthodes de conduite de projet et le renforcement de la notion de *bout en bout* lors de la création de nouveaux produits et services. *OLPS* est découpé en 9 sous entités qui ont à leur charge un domaine particulier (Service d'accès, TV, Open Services, Usage et Expérience client, ...). Pour chacune de ces entités, des équipes sont réparties sur les différents sites *Orange* en France et dans le monde.

Ainsi sur le site de Pessac la division *OLPS* compte 36 personnes, reparties parmi 3 services : *COMSERV* qui s'occupe des services réseaux et média, *IVA* qui fait principalement de la qualification et de l'intégration et enfin *SOFT* qui s'occupe des développements logiciels.

2.2 Nature du poste et environnement de travail

Au sein de l'entité *OLPS* du site *Orange* de Pessac, j'ai intégré l'équipe *SOFT* composée de 13 personnes qui travaillent sur différents projets rattachés à *Orange* ou à certaines de ses filiales, notamment *Orange Vallée*. Une partie de l'équipe travaille sur le projet *Libon*, c'est sur ce projet que j'ai travaillé. *Libon* est une application mobile de voix sur IP et de messagerie instantanée principalement à destination des clients mobile *Orange* qui peuvent alors appeler depuis l'international un numéro national gratuitement sous réserve d'avoir une connexion internet. Le domaine de la messagerie instantanée est un secteur extrêmement concurrentiel et est largement dominé par quelques acteurs Américains, *Whatsapp*, *Facebook Messenger* et *Snapchat* ont su faire leur preuve et s'imposer grâce à des fonctionnalités spécifiques. Ainsi sur un marché aussi saturé *Libon* mise sur une différentiation par les fonctionnalités. C'est dans ce cadre que plusieurs projets ont été lancés afin de délivrer aux utilisateurs une expérience de *chat* enrichie. Cela passe notamment par la signature de nombreux partenariats permettant d'utiliser des services externes au sein du *chat*. Cela permet, par exemple, de proposer une grande variété d'émoticones ou de repérer des informations dans les messages du *chat* et de donner des informations contextualisées avec l'aide de *Yelp*.

Dans cette optique un partenariat avec la société *WittiZ* est envisagé, afin de permettre à l'utilisateur d'envoyer de courts extraits vidéo en lieu et place des traditionnels émoticones. Il s'agit dans un premier temps de réaliser un *POC*¹, afin d'avoir une idée concrète du rendu final tout en mettant au jour les défis techniques et d'éventuelles solutions, ce qui mènera par la suite, sous condition de validation, à l'intégration de cette extension dans l'application *Libon* en production.

Le travail a été réalisé pour l'application *iOS* de *Libon*. Un environnement de développement *Mac Os X* a donc été utilisé, avec le langage *Objective C* et son IDE *Xcode*. Le projet *Libon* utilise la méthode *Kanban* pour la gestion de projet. Ainsi des tâches sont ajoutées à une liste de tâches à réaliser dans laquelle les différentes personnes travaillant sur le projet peuvent piocher. Toutefois ces personnes ne travaillent pas nécessairement sur le même site *Orange*, d'ailleurs la version *iOS* de *Libon* a principalement été sous-traitée par la société *BeTomorrow* basée à Bordeaux, qui assure toujours la maintenance et l'ajout de fonctionnalités dans l'application. Concrètement, les nouvelles fonctionnalités et améliorations sont demandées par le Technocentre à Paris qui est le *PO*². Les tâches, de priorités plus ou moins importantes, sont ensuite réparties entre les différentes équipes ; *BeTomorrow* se chargeant de la majorité d'entre elles pour la partie client *iOS*. Ainsi le projet d'intégration de *WittiZ* à *Libon* n'étant pas de priorité élevée, cette mission m'a été confiée et j'ai travaillé seul sur le code de ce projet. La gestion du code pour l'application de démonstration est assurée grâce à un dépôt *Git* sur la forge *Orange*. Pour l'application *Libon*, le *versioning*³ du code est assuré par un dépôt *Mercurial* hébergé chez *BeTomorrow* et auquel nous avons accès via un *VPN*⁴.

¹Preuve de concept (de l'anglais : Proof of Concept), est une réalisation courte ou incomplète d'une certaine méthode ou idée pour démontrer sa faisabilité.

²Product Owner, il s'agit de l'équivalent SCRUM du maître d'ouvrage (MOA), c'est donc la personne ou le groupe qui exprime le besoin.

³Mécanisme qui consiste à conserver la version d'une entité logicielle quelconque, de façon à pouvoir la retrouver facilement, même après l'apparition et la mise en place de versions plus récentes.

⁴Virtual Private Network, est un système permettant de créer un lien direct et sécurisé entre des ordina-

2.3 Sujet de stage

Les communications par messageries instantanées ont beaucoup évolué ces dernières années et de nouvelles fonctionnalités viennent en permanence enrichir cette expérience. Ainsi l'idée première du sujet de stage était de mettre en place de nouvelles formes de communication. Cela aurait pu passer par l'envoi de sons, de répliques cultes ou d'onomatopées sous forme d'extraits audio afin d'exprimer simplement et rapidement ce que les mots ne permettent de faire. Cela pouvait également passer par des images, avec notamment les émoticônes tels que nous les connaissons aujourd'hui, mais également via des stickers de plus en plus utilisés ou enfin via des extraits de films ou vidéos amusantes et expressives. Ce dernier exemple est déjà relativement exploité notamment via des claviers proposant l'envoi de *GIF*⁵ créées à partir de toute sorte de vidéo. Après étude, c'est cet usage qui a été retenu comme meilleur candidat pour une intégration dans *Libon*.

Lors de la phase de conception, plusieurs questions se sont posées.

Une d'entre elles, essentielle mais qui demeure assez floue dans les solutions existantes est la question des droits d'exploitation. En effet tout extrait sonore ou vidéo peut être soumis à des droits d'auteur et c'est bien souvent le cas pour les extraits issus de film. Ainsi avant d'intégrer une telle solution à *Libon* il est essentiel que ces questions soient réglées tout comme l'aspect financier et le modèle économique qui en incombe.

D'autre part, collecter, sélectionner et mettre en forme les extraits vidéos est un travail long et fastidieux pouvant difficilement être automatisé et donc par conséquent coûteux.

Enfin au-delà des contenus il est nécessaire de pouvoir les servir et donc d'avoir l'infrastructure réseau capable de tenir la charge. Ce qui représente un coût supplémentaire non négligeable.

Ces interrogations ont conduit à repenser la stratégie à adopter afin de proposer un tel service. En effet certaines sociétés proposent déjà ce genre de service. C'est notamment le cas de *WittiZ*, qui propose des extraits vidéo, via une application mobile, pour lesquels ils possèdent des droits de diffusion. Ainsi un partenariat avec cette société semble bien plus envisageable et économiquement viable pour le projet. Plusieurs semaines de discussion entre la société *WittiZ* et le *PO* auront été nécessaires pour trouver un accord, impliquant un partage des ressources et connaissances. L'objectif du stage est donc de travailler sur l'intégration des vidéos de la plate-forme *WittiZ* sous forme d'un clavier dans l'application *Libon* en utilisant l'API REST de *WittiZ*. Des spécifications ciblées sur le design, visibles figures 2 et 3, ont été réalisées en début de stage afin d'avoir une idée concrète du rendu final.

teurs distants.

⁵Format d'image permettant de stocker plusieurs images dans un fichier. Ceci permet de créer des animations si les images sont affichées à un rythme suffisamment soutenu.

3 Réalisation du projet

3.1 Conception de l'application de démonstration

La majorité des projets sont initiés par un *POC* qui se traduit en général par le développement d'une application de démonstration implémentant une grande partie des fonctionnalités demandées par le *PO*. Ainsi une application iOS, permettant de sélectionner une vidéo *WittiZ* puis de l'envoyer, a été développée en partant de zéro. Dans un premier temps, les données et leur format disponible en entrée vont être détaillés ce qui permettra par la suite d'exposer l'architecture de cette application.

3.1.1 Storyboard

Le but de cette application de démonstration est de prouver qu'il est possible d'exploiter l'API REST de *WittiZ* afin de proposer de manière visuellement attractive le partage de vidéos directement au sein d'un *chat*. L'application réalisée peut donc être vue en deux parties. La première brique va constituer le support de développement et il s'agira de reconstituer un *chat* au fonctionnement basique et sans fonctionnalités réseau. La seconde brique constituera la librairie *WittiZ* qui sera construite à partir de l'API REST fournie. Cette seconde brique aura un cycle de vie indépendant de la première.

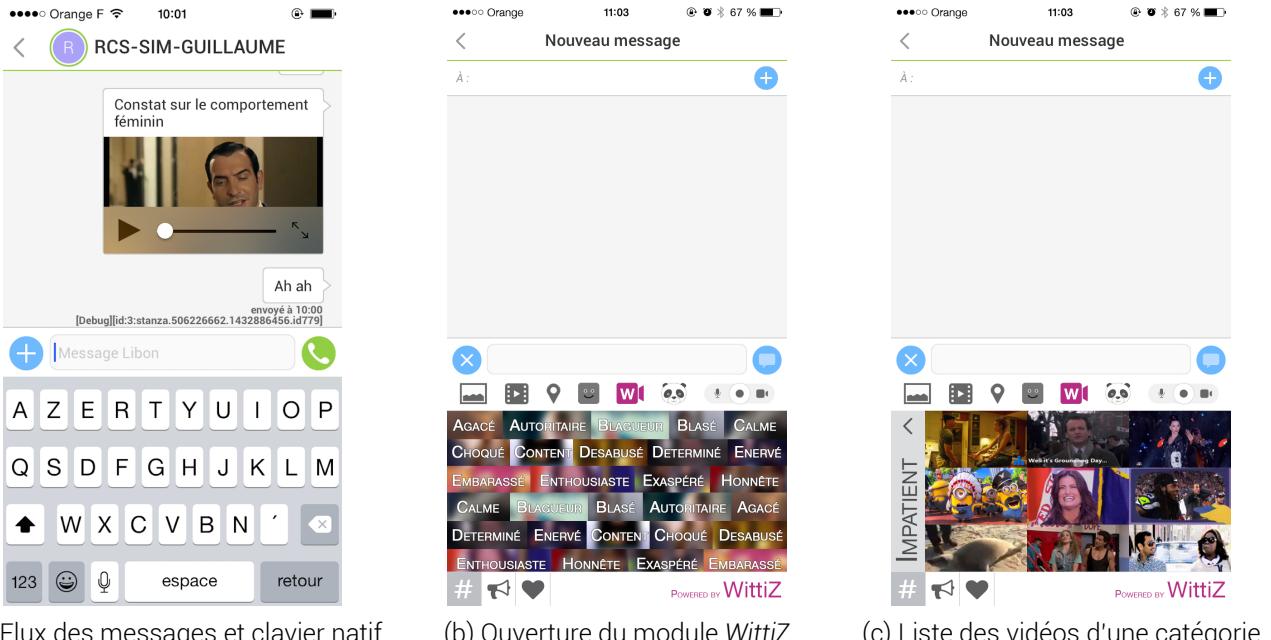


Fig. 2 : Aperçu des spécifications réalisées en accord avec le *PO*

Ainsi, l'utilisateur va ouvrir l'application et retrouver le flux de ses derniers messages (figure 2a), il va pouvoir composer un message et l'envoyer de manière classique. Cependant ne sachant que répondre au dernier message reçu, il préfère s'exprimer au travers d'un extrait vidéo. Il appuie alors sur le bouton *WittiZ* situé au dessus

de son clavier, ce qui ouvre un nouveau clavier remplaçant les traditionnelles lettres. Ce clavier, visible figure 2b, propose d'abord une liste scrollable⁶ de mots représentant des émotions. Lorsque l'utilisateur trouve une catégorie qui lui convient, il peut accéder à la liste des vidéos correspondantes en effectuant un tap sur le mot. L'utilisateur peut alors parcourir les différentes vidéos (figure 2c), en sélectionner une pour lire la vidéo, puis l'envoyer (3a). Lorsque l'utilisateur choisit d'envoyer une vidéo, une image d'aperçu se place dans le champ de composition du message comme le montre la figure 3b ce qui permet à l'utilisateur de joindre du texte à la vidéo. L'utilisateur peut aussi revenir en arrière (figure 3c) grâce à la navigation interne au module. Il peut également accéder rapidement aux vidéos les plus populaires ou à celles qu'il a marqué favorites, grâce aux deux onglets dédiés situés dans la partie inférieure du module donnant de la même manière accès aux vidéos.

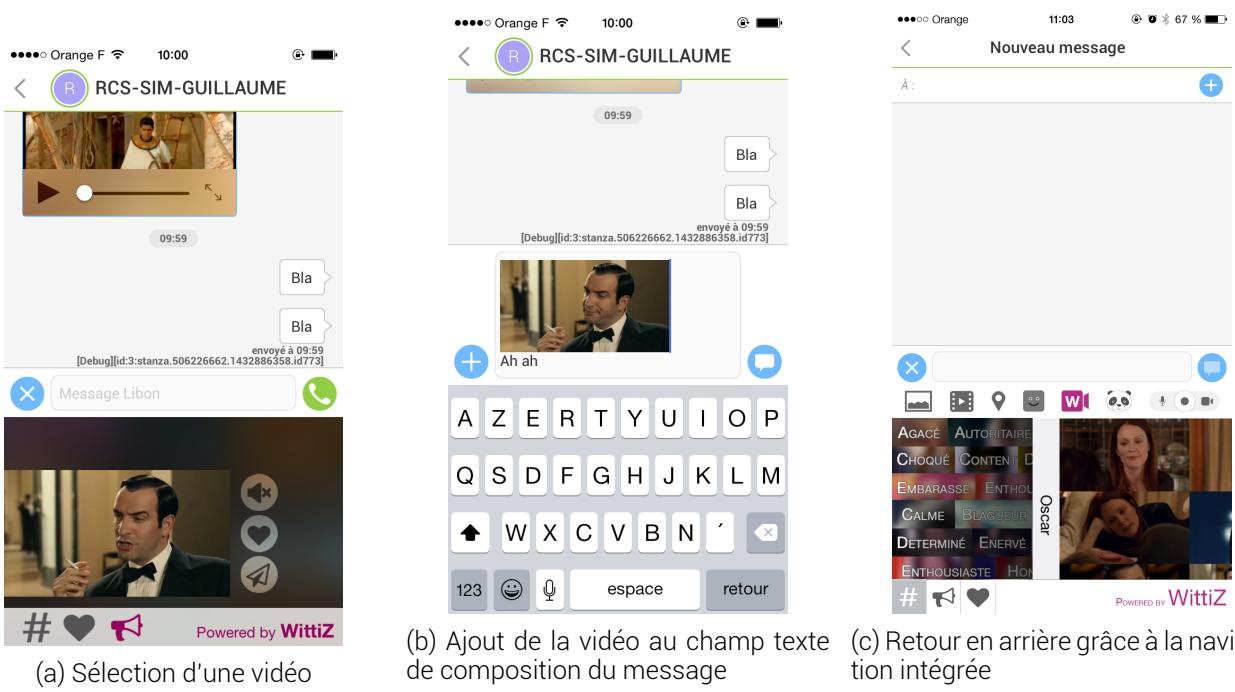


Fig. 3 : Suite des spécifications réalisées en accord avec le PO

3.1.2 Modèle de données

La société Wittiz fourni une API à laquelle il est possible de se connecter pour récupérer les données nécessaires au fonctionnement du module en développement. L'API fournie possède une architecture de type REST, c'est à dire que l'on utilise une url et une méthode (GET, POST, PUT, DELETE) pour accéder à une ressource particulière. Le serveur retourne alors les données demandées au format JSON.

⁶Liste que l'on peut faire défiler horizontalement en déplaçant son doigt de bas en haut.

POST	/artists/:id/subscribe	Subscribe current user to artist
GET	/artists	Get all artists minimal informations
GET	/artists/:id/scenes	Get scenes where artist is present
POST	/current_user/seen_scenes	Mark scenes as seen or partially seen
POST	/current_user/scenes/like	Add a scene to current user favorite list
POST	/current_user/notifications/:group_id/seen	Mark notification as seen
GET	/current_user/notifications	Get current user's notifications
GET	/current_user/status	Get current user's status
POST	/login	Signin user. You must at least provide email
POST	/playlists/:id/scenes	Add a scene to a playlist
DELETE	/playlists/:id	Delete a playlist
GET	/playlist_categories	Get playlist categories
PUT	/playlists/:id	Update a playlist for current user

Fig. 4 : Extrait des ressources accessibles via l'API de WittiZ

Cette API est utilisée par l'application propre à la société WittiZ ainsi elle répond à un grand nombre de requêtes et expose de nombreuses ressources. Un échantillon de ces ressources est présenté figure 4. Cependant toutes ces ressources sont inutiles pour l'utilisation que nous allons en faire, c'est pourquoi seulement quatre ressources ont été retenues :

- Les *scènes*, il s'agit d'un item représentant un court extrait vidéo provenant d'un film, d'une pub, etc. Une scène est composée d'un titre, un lien vers une image d'aperçu, un lien vers la vidéo, ainsi que le titre et l'identifiant de la ressource d'où provient la scène.
- Les *tags*, représentent des catégories dans lesquelles sont classées les vidéos. Un tag est composé d'un identifiant unique, d'un nom et du nombre de scènes qu'il contient.
- Les (*scènes*) *populaires*, ne sont pas une ressource à proprement parlé mais plutôt un ensemble de ressources. En clair, il s'agit simplement d'un tableau contenant les scènes les plus populaires auprès des utilisateurs.
- Les (*scènes*) *favorites*, tout comme les populaires constituent un ensemble de scènes que l'utilisateur a marqué afin de pouvoir y accéder facilement par la suite. Cependant l'accès à cette ressource est liée à l'utilisateur, ainsi un compte est nécessaire afin que les informations de l'utilisateur soient enregistrées. C'est

pourquoi une phase d'authentification est nécessaire pour accéder à ces informations.

Lors de la navigation dans l'application ces ressources vont être téléchargées puis parsées afin d'être transformées en objets facilement accessibles dans l'application, la structure de ces objets est détaillée en figure 5.

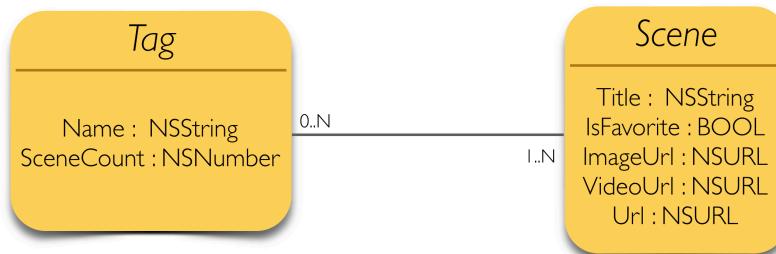


Fig. 5 : Format des données

3.1.3 Architecture de l'application

Le but de cette application de démonstration n'est pas de recréer une application de *chat* totalement fonctionnelle mais seulement de se rapprocher le plus simplement possible du fonctionnement attendu. Dans cette optique seul l'aspect interface a été implémenté et afin de ne pas perdre de temps, une librairie open source⁷ permettant de reproduire rapidement l'interface de l'application Message de l'iPhone a été utilisée.

Les applications *iOS* sont en général conçues en respectant le *Design Pattern*⁸ *MVC*⁹. Ainsi ce patron de conception induit qu'une vue est rattachée à un contrôleur qui va se charger d'y injecter les données contenues dans le modèle. De fait, le module *WittiZ* a été conçu en respectant ce *design pattern*, il est donc composé de plusieurs contrôleurs, vues et modèles qui sont détaillés en figure 6. Le contrôleur *WittizLib* est le point d'entrée de la librairie. Le *WittizKeyboardController* va s'occuper de la gestion des onglets en affichant la vue correspondant à l'onglet sélectionné par l'utilisateur. Les contrôleurs *TagsContainerViewController* et *TagsCollectionViewController* ont à leur charge l'affichage de la liste des tags. *FavoriteScenesViewController* ainsi que *TrendingScenesViewController* héritent de *ScenesViewController* dans la mesure où ils manipulent le même format de données, seule la source de ces données va différer d'un contrôleur à l'autre. Ils vont ainsi afficher la liste des scènes Favorites et Populaires. Enfin le *SceneViewController* permettra de visualiser la vidéo et d'effectuer quelques opérations telles que l'envoi de la vidéo, l'enregistrement dans les favoris ou la gestion du son.

Nous avons vu en section 3.1.1 que les spécifications établies par le *PO* recommandait un affichage sous forme de clavier secondaire et que la navigation au sein

⁷<https://github.com/jessesquires/JSQMessagesViewController>

⁸En français patron de conception, est un ensemble de bonnes pratiques pour la conception d'une architecture logicielle.

⁹Modèle - Vue - Contrôleur

de ce module doit être indépendante du reste de l'application tout en offrant la possibilité d'afficher des listes *scrollable* avec un agencement personnalisé pour les tags. Dans cette optique, les paragraphes suivants nous permettront d'exposer comment obtenir ces trois fonctionnalités.

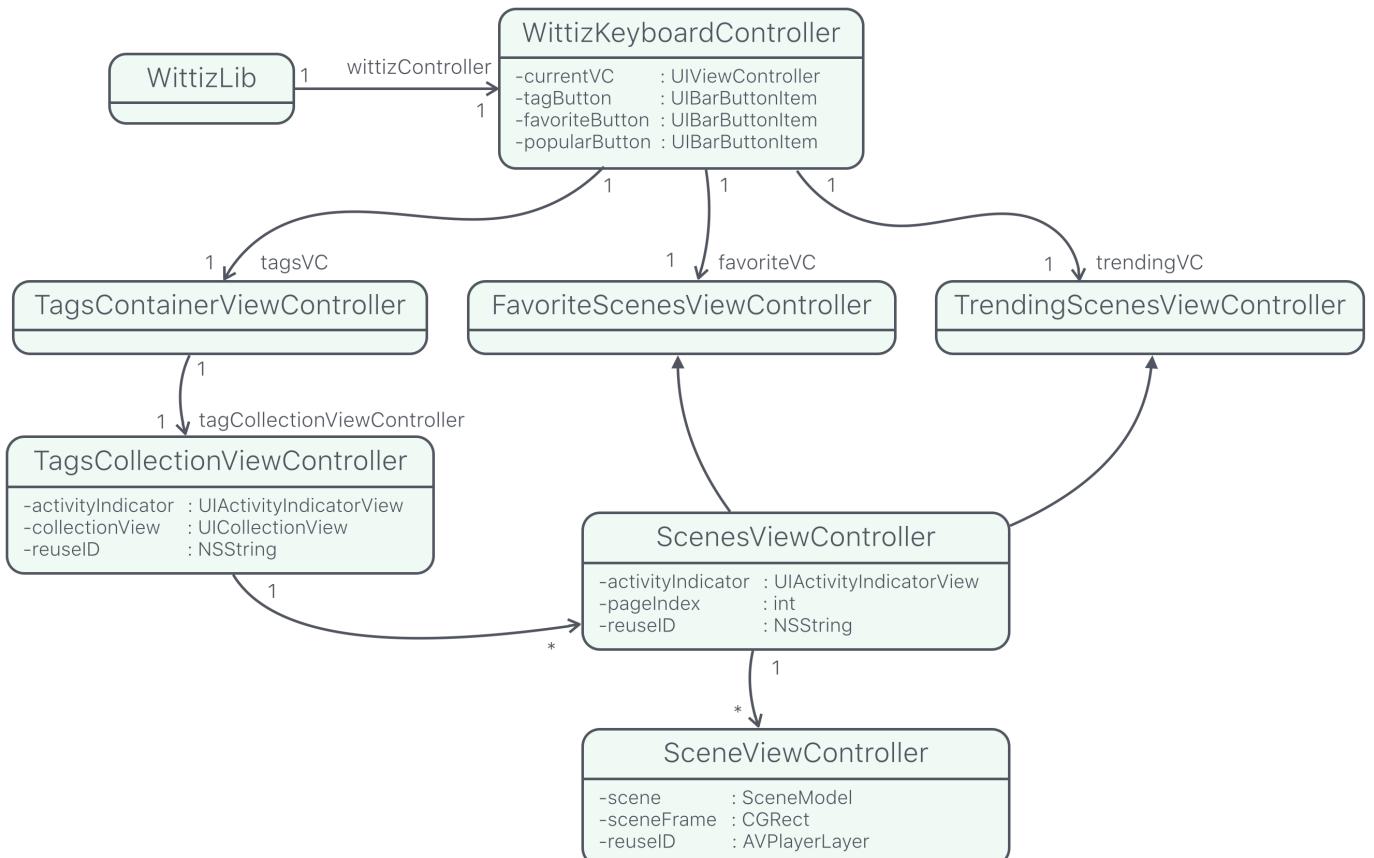


Fig. 6 : Diagramme de classe du module Wittiz

3.1.3.1 Input View : Un clavier alternatif doit respecter plusieurs aspects du clavier natif ; la taille par exemple doit coïncider afin que le passage de l'un à l'autre soit "transparent". De plus il doit pouvoir répondre aux mêmes gestes de l'utilisateur et adopter un comportement similaire notamment lorsque l'on souhaite fermer ce dernier. Par exemple il doit pouvoir être fermé en le faisant glisser vers le bas, ou en tapant sur la vue principale. Il est également important de mentionner que ce clavier, bien qu'au sein de l'application, va agir de manière isolée par rapport à l'application hôte. En effet il peut être considéré comme une petite application au sein de l'application globale dans la mesure où les vues viendront se placer en surcouche et la navigation en son sein sera indépendante du reste.

Ces considérations faites, peu de solutions existent pour la conception de cet add-on, la première solution consisterait à tout implémenter à la main et s'affranchir des recommandations d'Apple sur l'architecture de l'application notamment pour la hié-

rarchie des vues. Bien qu'offrant le plus de libertés cette solution s'avère être complexe à mettre en œuvre et requiert un bon niveau de connaissances du framework *Cocoa Touch*¹⁰, connaissances que je n'avais guère à ce moment-ci. La seconde solution, d'ailleurs retenue, consistait à utiliser un des mécanismes du framework permettant de réaliser facilement ce que nous souhaitons : l'attribut `inputView`. Cet attribut est disponible pour tous les éléments de saisie de texte `inputText`. Le type attendu est une `UIView`, qui correspond à la vue à afficher à la place du clavier traditionnel lorsque le champ texte est actif. Cependant il n'est pas non plus souhaitable que ce clavier alternatif apparaisse systématiquement lorsque l'utilisateur sélectionne le champ texte, auquel cas il ne pourrait plus saisir de texte. Il doit seulement s'ouvrir lorsque l'utilisateur presse le bouton associé (figure 7). L'astuce consiste donc à ajouter dynamiquement la vue contenant le clavier à l'attribut `inputView` et à rendre le champ texte actif afin que le clavier s'ouvre. À contrario, lors de la fermeture du clavier, l'attribut sera repositionné à `nil` afin qu'à la prochaine ouverture ce soit bien le clavier natif avec lettres qui soit affiché.

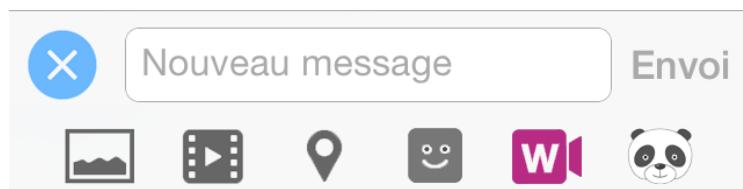


Fig. 7 : Bouton d'ouverture du module *WittiZ*

Nous pouvons désormais afficher une vue à la manière d'un clavier, nous allons désormais voir comment hiérarchiser ces vues afin d'avoir une notion de navigation.

3.1.3.2 Navigation Stack : L'un des inconvénients de l'attribut `inputView` est qu'il ne permet d'attacher qu'une seule vue et que cet attribut est seulement accessible depuis le contrôleur lié à un champ texte (`inputText`). Ceci s'avère être problématique dans la mesure où ce nouveau clavier ne sera pas composé d'un unique contrôleur. De plus si l'on adopte une solution simpliste pour la navigation qui consisterait à modifier la vue assignée à l'attribut `inputView` à chaque changement de vue, toute la gestion de la navigation se ferait dans le contrôleur ayant la paternité du champ texte, contrôleur appartenant à la *brique chat* de l'application. Or cela brise la séparation du code entre les deux briques *chat* et module *WittiZ*, nécessaire si l'on garde en tête que le module *WittiZ* pourrait être par la suite intégré dans l'application *Libon*.

Cependant la documentation *Apple* nous apprend qu'une autre solution plus adaptée existe, les *Container View Controller* qui permettent d'imbriquer dans une vue principale d'autres vues. L'avantage de ce mécanisme est qu'il nous permet de mettre en place un point d'entrée pour le module *WittiZ* ne requérant que peu de modifications dans le code de la brique *chat*. Le *WittizKeyboardController* sera donc le contrôleur à instancier pour initialiser le module ; c'est également ce dernier qui se chargera de la navigation. Les vues imbriquées fonctionnent comme un arbre de profondeur maximale 1. Ainsi le contrôleur père peut avoir n fils et chacun de ses fils

¹⁰API native d'*Apple* pour le développement orienté objet sur son système d'exploitation *iOS*.

a une référence vers son unique père. Il est donc possible de réaliser une navigation entre deux contrôleurs, avec la possibilité de passer de l'un à l'autre puis de revenir en arrière. Cependant au vu des spécifications, une navigation à trois niveaux (Liste des Tags / Liste des Scènes / Aperçu Scene) doit être utilisée. Il est donc nécessaire d'avoir une vraie pile de navigation afin de conserver une référence vers les contrôleurs qui étaient ouverts précédemment. Apple met à disposition des développeurs la classe `UINavigationController` qui permet de mettre en place un historique de navigation entre les différents contrôleurs. Une limite de cette classe est qu'elle ne peut être utilisée qu'une seule fois dans l'application et ne peut donc pas être utilisée avec notre module puisque l'application hôte pourrait utiliser ce contrôleur pour la navigation. La solution choisie a été de réimplémenter les fonctionnalités de base de ce contrôleur. Le fonctionnement est relativement simple, il s'agit tout simplement d'une pile, qui est initialisée et gérée par un contrôleur principal, en l'occurrence `UINavigationController`, ce contrôleur implémente entre autres deux méthodes (`void`)`pushViewController` qui permet d'ouvrir un nouveau contrôleur-vue et de l'ajouter à la pile de navigation. La seconde, (`UIViewController *`)`popViewController`, permet de revenir à la vue précédente en dépliant le contrôleur-vue, actuellement affiché, de la pile de navigation. L'algorithme 8 détaille le fonctionnement de cette dernière fonction qui utilise pour fonctionner la pile de navigation et le principe des vues imbriquées.

```

function popViewController() {
    if (self.stack.size > 1) {
        ViewController fromViewController = self.stack.pop();
        ViewController containerViewController = self;

        fromViewController.willMoveToParentViewController();
        fromViewController.view.removeFromSuperview();
        fromViewController.removeFromParentViewController();

        if (self.stack.size > 0) {
            ViewController toViewController = self.stack.head();

            containerViewController.addChildViewController(toViewController);
            containerViewController.view.addSubview(toViewController);
            toViewController.didMoveToParentViewController(containerViewController);
        }
    }
}

```

Fig. 8 : Algorithme en pseudo-code de la méthode `popViewController`

Une solution a donc été trouvée afin de mettre en place une navigation au sein du module *WittiZ*, nous allons désormais nous intéresser à un autre challenge des spécifications, l'affichage en grille des mots correspondant aux différentes catégories de vidéos.

3.1.3.3 Collection View & Flow Layout : Pour rappel, la figure 2b présentait de quelle manière l'affichage de ces tags devait être fait ; une grille avec des mots de taille variable encadrés d'un fond de couleur délimitant sans marge les différents éléments. Afin de réaliser ce genre d'agencement, Apple met à disposition des développeurs la classe `UICollectionView`.

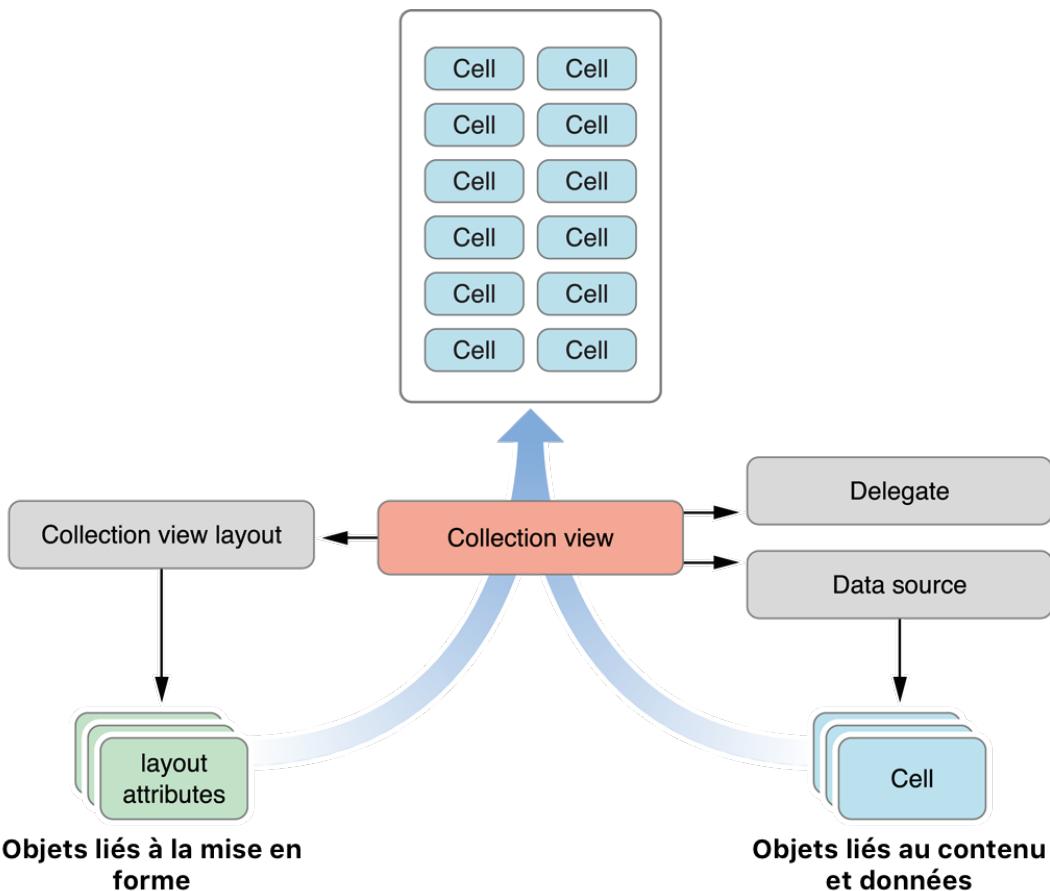


Fig. 9 : Fonctionnement du `UICollectionViewController`

Comme le montre la figure 9, cette classe va combiner plusieurs sources d'information afin de créer la grille souhaitée. D'une part elle va récolter les données à afficher, dans notre cas un simple mot et d'autre part la manière d'afficher ces données grâce au *Collection View Layout* fourni. C'est ce *layout*¹¹ qui va principalement nous intéresser puisque c'est lui qui régit l'affichage des différents éléments. Par défaut c'est le `UICollectionViewFlowLayout` qui va être utilisé pour l'affichage. Il permet d'afficher rapidement et simplement des données en plaçant les cellules les unes à la suite des autres sur la même ligne jusqu'à ce que la place manque, une nouvelle ligne est alors ajoutée et ainsi de suite. La figure 10 résume parfaitement ce fonctionnement.

¹¹Ensemble des règles permettant la mise en forme de l'interface.

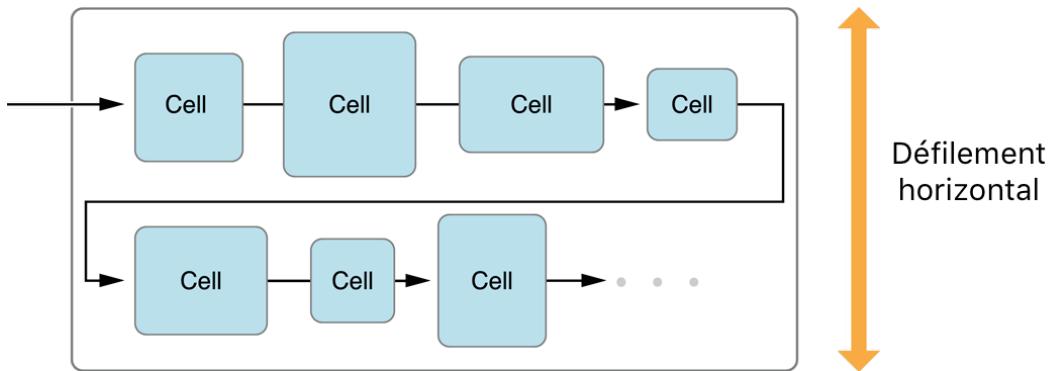


Fig. 10 : Fonctionnement du *flowLayout*

Il est possible de configurer plusieurs éléments de ce *layout* notamment la taille des cellules et l'espacement minimum et maximum entre chaque cellule. Cependant dans notre cas, même en le configurant, ce *layout* ne pourra pas répondre à notre besoin puisque l'espacement entre les cellules doit être nul et l'espace restant doit être réparti entre les différentes cellules en tant que *padding*¹². Afin d'obtenir la disposition attendue, un nouveau *layout* héritant du `UICollectionViewFlowLayout` a été implémenté.



Fig. 11 : Affichage des tags sans *padding*

Afin d'obtenir le résultat escompté, deux étapes sont nécessaires. La première consiste à générer un premier alignement des cellules avec un espacement inter-cellule nul en utilisant la méthode `layoutAttributesForItemAtIndexPath` de la classe mère. Une fois ce premier calcul fait, le résultat visible figure 11 est obtenu. La seconde étape consiste à calculer en début de ligne le *padding* à appliquer à chaque cellule. Pour se faire deux choses sont nécessaires, savoir quand une nouvelle ligne

¹²Espacement entre le bord de la cellule et le texte qu'elle contient. Se distingue de la marge qui est l'espacement entre le bord de deux cellules différentes.

commence et la valeur du *padding* à appliquer. Pour calculer le *padding*, la taille de chaque élément va être additionnée jusqu'à ce qu'ils dépassent la largeur de l'écran. La différence entre la largeur de l'écran et cette somme est alors divisée par le nombre d'éléments additionnés. Finalement, on obtient le *padding* à repartir dans chaque cellule afin de centrer le texte. Afin de déterminer le commencement d'une nouvelle ligne, un petit test reposant sur ses coordonnées va être réalisé pour chaque cellule. Ainsi si la cellule courante se trouve à une ordonnée qui est supérieure ou égale à la somme de la hauteur et de l'ordonnée de la cellule précédente alors il s'agit d'une nouvelle ligne ; sinon la ligne n'a pas changé.

3.2 De l'application de démonstration à Libon

Une fois l'application de démonstration terminée, une présentation a été organisée afin de présenter cette première version au *PO*. Satisfait du travail réalisé il a donné son feu vert pour commencer l'intégration de *WittiZ* à *Libon*.

3.2.1 Contexte et spécificités de Libon

Le développement de *Libon* a commencé en 2010, on était alors à la version 4 d'*iOS* le système d'exploitation mobile d'*Apple*. Les API disponibles pour les développeurs sont encore limités et l'*IDE* de développement *Xcode* est loin d'avoir toutes les fonctionnalités connues et appréciées des développeurs aujourd'hui. Ainsi la création d'interfaces n'était pas très simple surtout lorsqu'il s'agissait de travailler collaborativement sur les mêmes fichiers. De plus la gestion de l'internationalisation était incroyablement lourde. Pour ces raisons, entre autres, la société *BeTomorrow*, prestataire en charge du développement de l'application, a choisi d'utiliser pour le développement de l'application certains de leurs outils développés en interne.

On retrouve par exemple un éditeur permettant de réaliser des interfaces utilisateurs en *WYSIWYG* qui génère des fichiers *xml*, un peu à la manière de ce qui se fait pour *Android*. Par ailleurs, pour une raison que je n'ai pu obtenir, *BeTomorrow* a vendu à *Orange* une application avec un socle potentiellement multiplate-forme en *C++* et pouvant donc être réutilisé avec la plate-forme *Android* ou *Windows Phone*. Ce qui s'avère étonnant c'est que l'application *Libon* pour *Android* a été confiée à une autre équipe qui n'utilise absolument pas ce socle commun. Ainsi l'application embarque une très grosse partie de code *C++*, bien loin du langage *Objective-C*, pour la logique de l'application. Les vues, quant à elles, sont écrites en *Objective-C++* afin de pouvoir communiquer avec le reste du code source.

Libon se différencie également par son framework dédié aux interfaces utilisateurs. *Apple* met à disposition des développeurs le framework **UIKit** afin de concevoir les interfaces, il se compose de nombreuses classes¹³ et méthodes, notamment la **UIView**, une vue, de laquelle de nombreux éléments héritent. Pour *Libon*, la plupart des classes d'interface ont été réécrites ; il s'agit d'une pratique assez courante qui permet d'avoir un style cohérent au sein de toute l'application facilement et rapidement. En général c'est le mécanisme de l'héritage qui est utilisé afin de seulement modifier les propriétés intéressantes. Cependant *BeTomorrow* a préféré utiliser un lien *a-un* pour ses classes dédiées aux interfaces en plaçant dans un attribut l'objet de base d'**UIKit**.

¹³Elles sont prefixées par les lettres UI.

Le problème est qu'il est alors impossible d'utiliser de nombreuses méthodes des librairies standards et externes qui prennent en paramètre une classe du framework `UIKit`. Malgré mes demandes, je n'ai pu avoir de réponse claire sur ce choix, je suppose donc qu'il s'agit de limiter au maximum le champ d'action du développeur afin qu'il n'utilise pas de composants standards *Apple* qui pourraient ne pas correspondre à la charte graphique de l'application.

Enfin *Libon*, c'est surtout et avant tout, 187346 lignes de codes constituant un noyau monolithique où les évolutions technologiques sont très difficiles à mettre en place. C'est notamment le cas d'`ARC`¹⁴ introduit par *Apple* en 2011 qui permet de gérer facilement la mémoire mais qui n'a pas été intégré au projet rendant donc obligatoire la gestion de la mémoire à la main.

Dans ce contexte de développement peu conventionnel pour du développement *iOS*, l'intégration du code de l'application de démonstration dans l'application *Libon* a débuté.

3.2.2 Réutilisation du code

Dès la conception de l'application de démonstration, un des objectifs était de pouvoir par la suite réutiliser le plus de code pour l'intégration. De fait, pour intégrer le code du module *WittiZ*, trois solutions sont possibles.

- La première consiste à ajouter les fichiers directement au projet de l'application en production. Cette solution bien que simpliste en apparence, puisqu'il s'agit d'ajouter des fichiers sources à compiler comporte plusieurs aspects négatifs. Tout d'abord cela signifie ajouter du code ne dépendant pas complètement de l'application au sein de cette dernière ce qui complique fortement la maintenance d'autant plus si le module est utilisé dans plusieurs applications. De plus le projet n'étant pas compatible `ARC`, il ne faut pas que les fichiers sources utilisent ce mécanisme, or ce n'est pas le cas des fichiers du module *WittiZ* développés avec l'application de démonstration.
- La seconde consiste à réaliser une librairie statique à partir du code du module, cela permet de compiler le code source de notre module et d'en générer un fichier .a. Cette librairie est ensuite chargée dans le projet en instanciant les méthodes publiques disponibles dans cette librairie. L'inconvénient majeur de la librairie statique seule est qu'il faut fournir à part les fichiers en-tête ainsi que les ressources telles que les images et les vues.
- La dernière solution consiste à regrouper dans un seul package tous les éléments nécessaires au fonctionnement du module, il s'agit alors d'un framework. Un framework statique va contenir une ou plusieurs librairies statiques, les fichiers d'en-tête et les ressources que sont les images et les fichiers de vue.

C'est cette dernière solution, le framework, qui a été utilisée dans le cadre de l'intégration de *WittiZ* à *Libon*. Par ailleurs afin de faciliter le développement, un nouveau projet Xcode permettant de compiler le framework a été créé puis inclus en tant que sous projet dans celui de *Libon*. Ceci permet avec un peu de configuration d'ajouter une étape lors de la compilation de *Libon* afin que le framework soit (re)compilé s'il ne l'avait pas déjà été ou que certains de ses fichiers sources ont été modifiés.

¹⁴Automatic Reference Counting, mécanisme qui s'occupe d'identifier automatiquement les objets utilisés et ceux pouvant être marqué comme à libérer.

3.3 Communication et interactions avec l'application hôte : Libon

Packager le code sous forme de framework pour l'inclure au projet est finalement assez simple et ne constitue qu'une infime partie du travail d'intégration. En effet que le projet compile après l'intégration du framework ne veut pas dire que le module soit en mesure de communiquer avec le reste de l'application, ainsi le plus dur est d'activer les interactions entre le module et l'application hôte. Les sections suivantes détailleront les différents aspects à considérer afin de pouvoir pleinement tirer partie des fonctionnalités offertes par l'add-on *WittiZ*. Dans un premier temps nous allons rapidement revenir sur la montée en compétence nécessaire afin de réaliser cette intégration.

3.3.1 Un processus de développement loin d'être industrialisé.

En section 3.2.1, l'architecture lourde et complexe de l'application *Libon* a déjà été évoquée. Cependant à cette première difficulté il faut ajouter l'inexistence de documentation sur l'architecture de l'application et l'absence de commentaires dans le code puisque étant systématiquement supprimés par *BeTomorrow* si « le code est suffisamment explicite ». Par ailleurs aucun test unitaire ou de bout en bout n'a été implémenté afin de vérifier, de manière automatique, qu'aucune régression n'est introduite par les nouveaux développements.

Cet environnement de développement n'est donc pas très accueillant pour les nouveaux arrivants ainsi un gros travail de compréhension et d'appropriation du code a été nécessaire avant de pouvoir commencer réellement le travail d'intégration. Cette phase d'apprentissage s'est principalement faite par lecture du code, modifications mineures et compilation pour constater les effets. Ces opérations sont extrêmement chronophages notamment car chaque compilation, aussi rapide soit elle, entraîne également l'exécution de divers scripts permettant le déploiement de l'application sur le terminal de test. Ainsi trois à quatre minutes se seront écoulées entre le moment où le bouton d'exécution du débogage sera pressé et le moment où l'application sera ouverte et opérationnelle sur l'*iPhone*.

3.3.2 Chargement de la librairie *WittiZ*

Libon permet le partage de différents éléments comme des photos, vidéos, sa position GPS, ou des émoticônes grâce au menu de la figure 12a. Ce menu était composé, avant intégration, de deux pages que l'utilisateur pouvait faire glisser de gauche à droite et inversement pour accéder aux émoticônes (figure 12b) ou aux boutons de partage (12a). L'intégration de *WittiZ* devait se faire de manière naturelle, ainsi l'option consistant à ajouter une page au *slider* après les émoticônes a rapidement été éliminée pour son manque d'ergonomie. La solution retenue est présentée figure 12c, il a donc été préféré l'ajout d'une ligne de boutons, avec un bouton pour *WittiZ* et un pour les émoticônes en remplacement du *slider*.



Fig. 12 : Clavier de partage de *Libon*

Des modifications au niveau des interfaces graphiques ont donc été nécessaires afin de prendre en compte ces choix ergonomiques. Les éléments d'interface sont regroupés par modules, celui dédié au clavier secondaire se nomme *compad-media.pad*. Un module peut être composé de modules imbriqués, de vues ou d'éléments standards. Il est également composé d'états dans lesquels les éléments du module peuvent être masqués ou positionnés différemment. Le module intéressant est visible figure 13, il est composé de trois vues :

- **file-transfer-page** : La vue affichée par défaut qui contient les différents boutons de partage.
- **emoticons-page** : Cette vue est composée d'un container dans lequel seront injectés les différents émoticônes pouvant être envoyés ainsi que quelques boutons de contrôle (ajout, suppression).
- **wittiz-page** : Il s'agit essentiellement d'un container dans lequel les vues du module *WittiZ* seront injectées.

Ce module comporte également quatre états qui vont permettre d'en faire varier l'aspect.

- **default** : L'état utilisé par défaut et dans lequel tout les boutons sont affichés.
- **file-transfer-disabled** : État dans lequel certains boutons sont désactivés. Sur les terminaux plus âgés le partage de vidéos et photos n'est pas possible pour des raisons techniques.
- **emoticons** : Utilisé afin d'afficher le sélecteur d'émoticônes.
- **wittiz** : Utilisé pour l'affichage du clavier Wittiz.

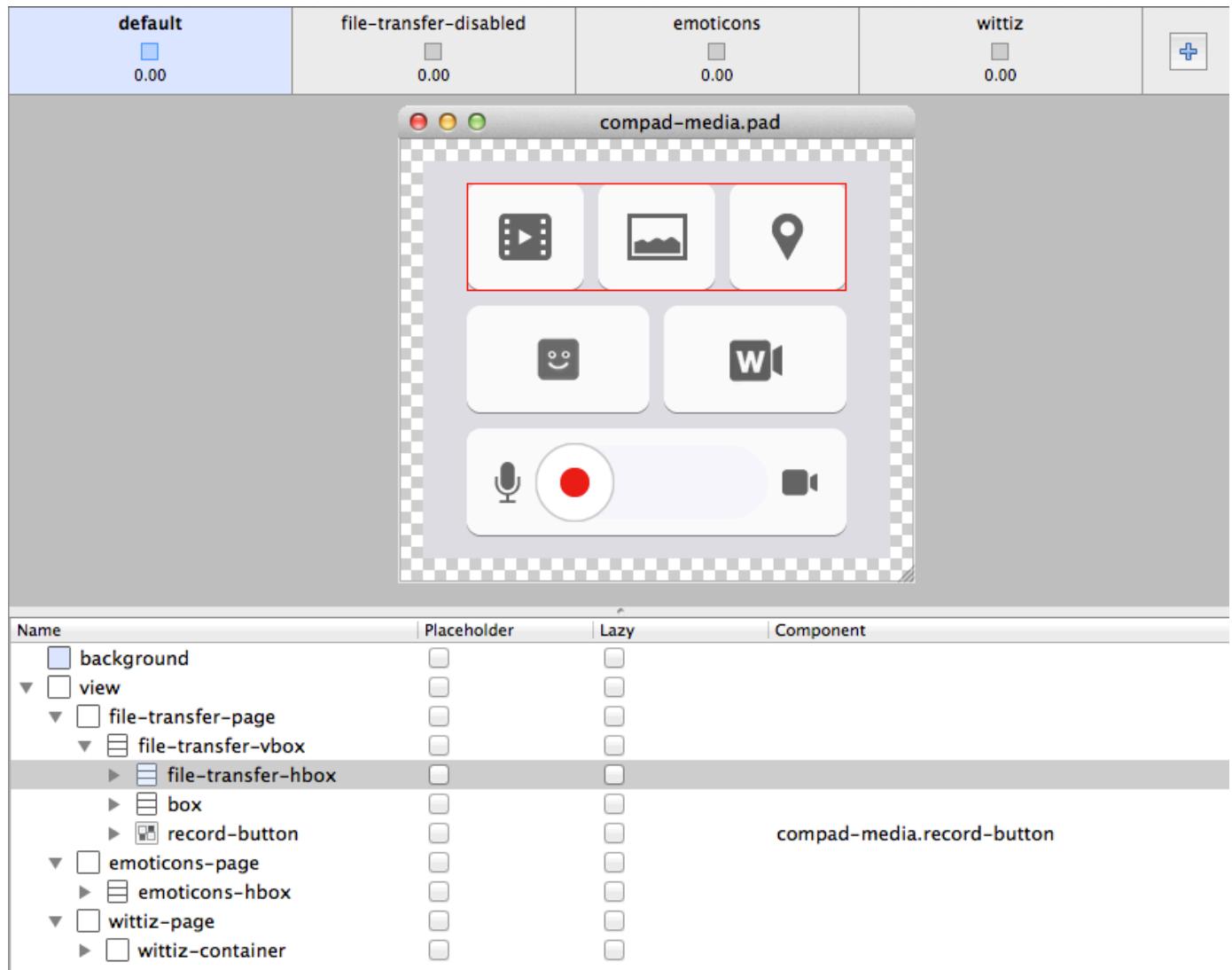


Fig. 13 : Interface de l'outil de création d'interfaces *LibonDesignTool*

Les modifications de code sont quant à elles limitées grâce au travail de packaging sous forme de framework réalisé préalablement. Il suffit simplement d'instancier le

module, lui donner la taille qu'il peut occuper et l'injecter en tant que sous-vue dans le container prévu à cet effet dans le module `compad-media.pad`. Enfin il faut indiquer au bouton `WittiZ` fraîchement ajouté qu'il doit, lorsqu'il est pressé, passer l'état du module à `wittiz` et initialiser le module si cela n'a pas déjà été fait.

Grâce à cette intégration, le module est désormais visible et fonctionnel au sein de l'application, mais envoyer une vidéo n'aura aucun effet car il ne communique pour le moment pas avec le champ de composition des messages. La section suivante détaille comment les données vont transiter de notre librairie `WittiZ` jusqu'au champ texte de `Libon`.

3.3.3 Interactions avec le champ de saisie du texte

Deux aspects sont à considérer afin de pouvoir interagir avec le champ texte. Tout d'abord il faut déterminer de quelle manière ajouter du contenu à ce champ texte depuis la librairie. Ensuite il faut considérer quel contenu ajouter à ce champ sachant que plusieurs informations doivent être transmises : l'url de la vidéo, son titre et un lien vers une image d'aperçu qui d'après les spécifications est censé être le seul élément affiché dans le champ de texte. Nous allons d'abord nous intéresser à comment ajouter du contenu au champ texte.

3.3.3.1 Un mécanisme puissant : les Delegates

La délégation est un patron de conception qui est très utilisé dans le framework *Cocoa Touch* et qui va nous permettre d'ajouter du contenu au champ texte directement depuis la librairie `WittiZ`. Apple définit ce principe comme suit : « Un delegate est un objet qui agit au nom de, ou en coordination avec un autre objet lorsque cet objet rencontre un événement dans un programme. »

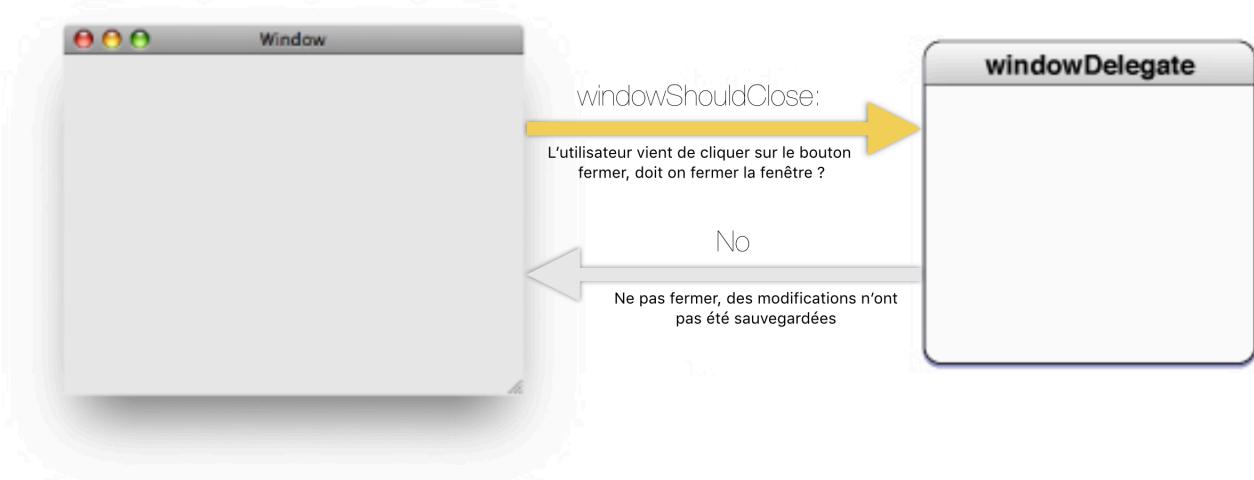


Fig. 14 : Principe de la délégation

De manière plus explicite, ce *design pattern* implique deux objets : le délégué et l'objet délégué. L'objet délégué met en place un protocole auquel le délégué doit se conformer. Un protocole étant une liste de méthodes que le délégué doit obligatoirement implémenter. Enfin l'objet délégué définit une propriété `delegate` au travers de laquelle il enverra des signaux à son délégué. Considérons l'exemple de la figure 14, à gauche, une instance de `NSWindow` est l'objet délégué que l'on nommera `window`, à droite, l'objet `windowsDelegate` est son délégué. L'objet `Window` a déclaré un protocole avec plusieurs méthodes dont `windowShouldClose` : . Lorsqu'un utilisateur va cliquer sur le bouton de fermeture de la fenêtre, l'objet `Window` va envoyer à son délégué un signal via la méthode `windowShouldClose` : afin d'avoir confirmation de la fermeture de la fenêtre. Ce dernier répond en retournant un booléen, ce qui lui permet d'avoir un contrôle sur la fermeture de l'application.

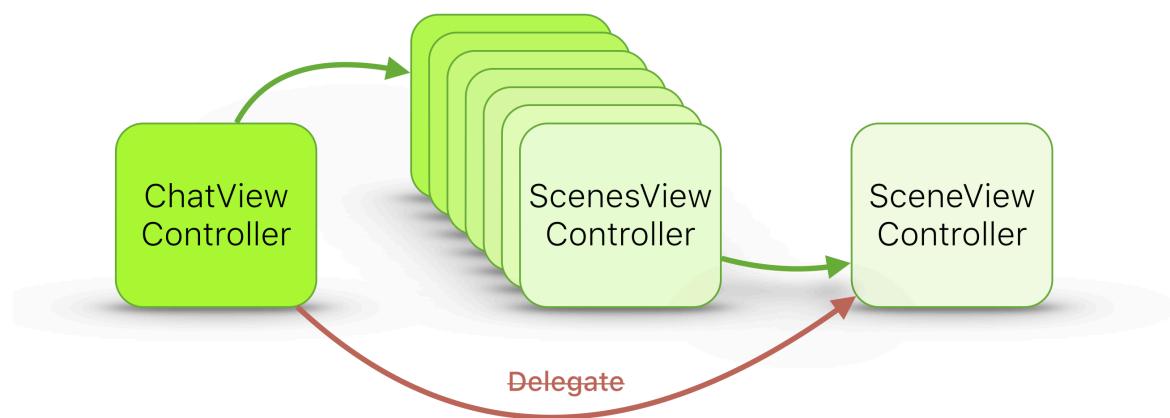


Fig. 15 : Un objet délégué, ne peut déléguer qu'à un objet dont il connaît l'existence

Nous allons donc mettre en application ce mécanisme afin d'accéder au champ texte. Comme nous venons de le voir, un objet peut déléguer des méthodes à un délégué, ainsi il est nécessaire que celui qui délégue ait la connaissance de son délégué, or dans notre cas (figure 15) l'instance de `SceneViewController` qui va déléguer, n'a pas de connaissance directe de l'objet contenant le champ texte, en l'occurrence une instance de `chatViewController`. Afin de pallier à ce problème, la solution retenue, illustrée figure 16, consiste à relayer via des "délégués passerelles" le signal jusqu'au délégué réellement concerné.

Ces deux objets peuvent désormais communiquer et l'ajout de contenu se fera grâce à la méthode `addText : (NSString*)text`, qui prend en paramètre le contenu à ajouter dans le champ texte, utilisée par l'objet délégué. Nous allons nous intéresser dans la section suivante au format des données à ajouter, le prototype de la fonction d'ajout du texte pouvant donner un premier indice sur le type utilisé.

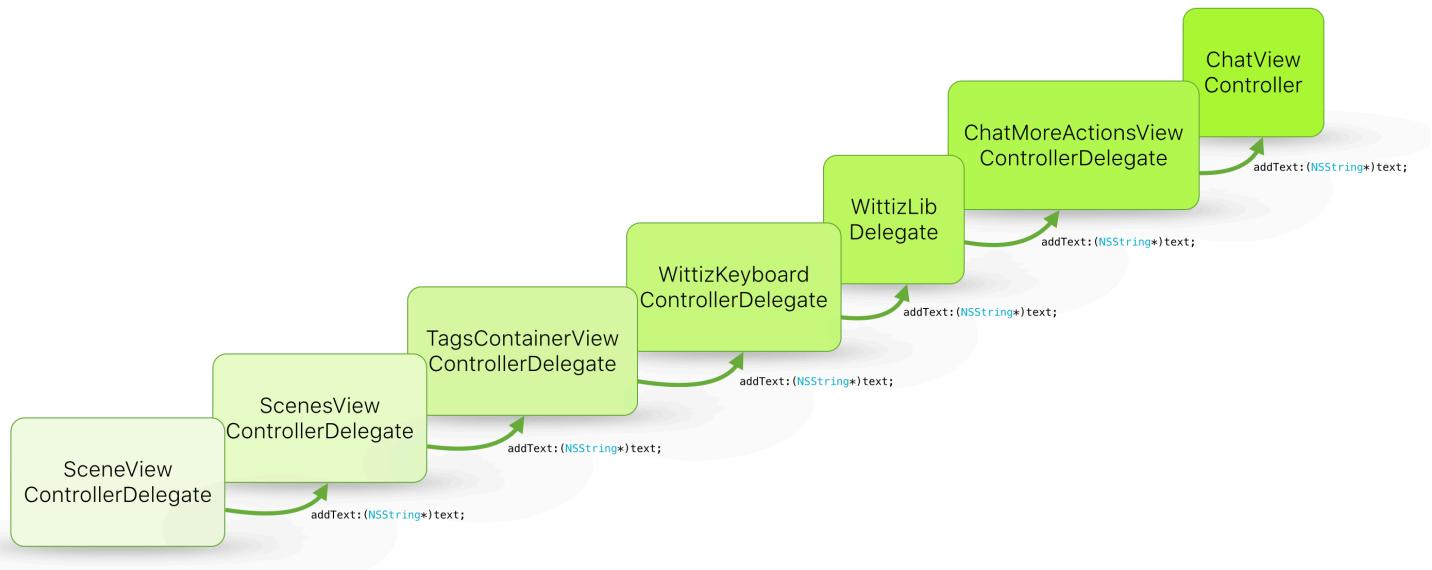


Fig. 16 : Plusieurs délégués n'assurant qu'un rôle d'intermédiaire sont nécessaires

3.3.3.2 Affichage d'images d'aperçu avec le texte

Le champ texte utilisé pour composer le message est une instance de la classe `UITextField`. Cette classe comporte, entre autres, deux attributs permettant d'y insérer du texte. Le premier, nommé `text` est de type `NSString*` et contient le texte présent dans le champ texte. Cette attribut peut être modifié afin de mettre à jour le contenu du champ texte. Le second attribut qui nous intéresse est l'`attributedText` de type `NSAttributedString*`. Ce dernier permet de personnaliser la mise en forme du texte affiché, ainsi il est possible d'insérer des mots en gras avec d'autres mots en italiques ou surlignés (figure 17).

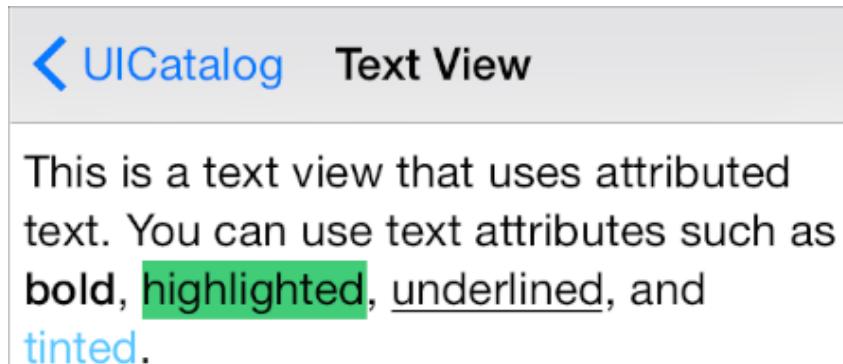


Fig. 17 : Texte mis en forme grâce à l'attribut `attributedText`

Il est également possible d'inclure dans le texte une "pièce jointe" comme le montre la figure 18. Cette pièce jointe pouvant être par exemple un fichier sonore ou bien une photo. D'ailleurs ce dernier cas sera largement exploité par la suite.

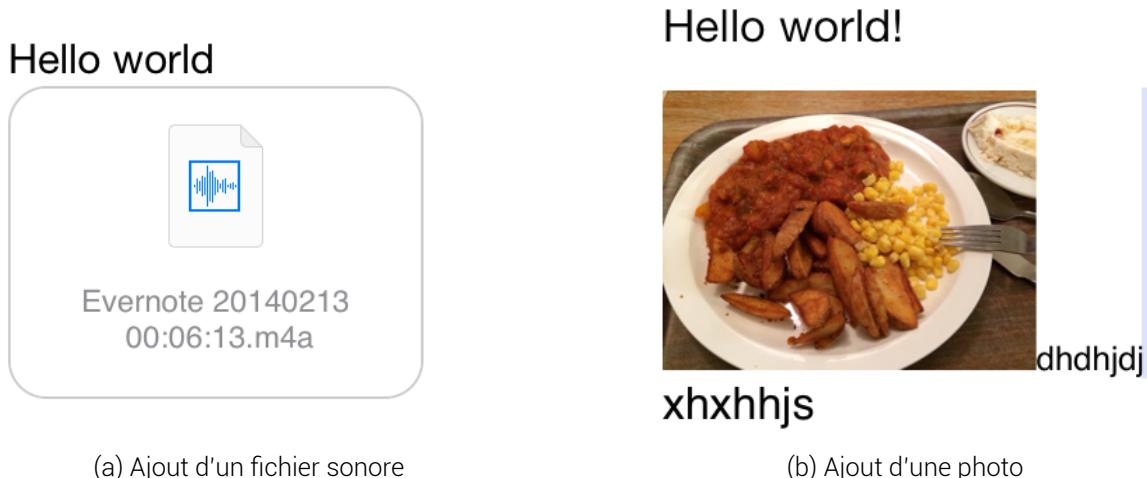


Fig. 18 : Possibilité d'intégrer directement dans le champ texte des fichiers

Dans *Libon*, les deux attributs présentés précédemment sont utilisés. L'attribut `attributedText` est utilisé pour l'affichage des émoticônes au sein du texte. L'attribut `text` est utilisé quant à lui pour la persistance des données. En effet, un des inconvénients de l'`attributedText` est qu'il n'est exploitable que sous forme binaire ce qui rend difficile la persistance des données notamment lorsque celle-ci se fait localement et sur un serveur distant comme c'est le cas pour *Libon*. Ainsi le texte au format brut est conservé dans l'attribut `text` et est utilisé lorsque le texte doit être sauvegardé. Par exemple, lorsque l'utilisateur quitte l'application ou change de téléphone et restaure l'application. Ainsi concrètement que l'utilisateur ajoute du texte ou un émoticône au champ texte, c'est du texte qui est ajouté. En l'occurrence pour les émoticônes c'est leur équivalent texte qui sera ajouté. Ce texte brut est ensuite parsé afin de générer dynamiquement une `NSAttributedString*` qui contient les images correspondant aux émoticônes et pouvant être utilisée avec l'attribut `attributedText`.

Ce mécanisme fonctionne parfaitement avec les émoticônes qui ont un équivalent texte, qui sont en nombre fini et dont les images sont disponibles localement. Cependant l'intégration des aperçus *WittiZ* aux cotés des émoticônes dans ce champ texte représente un défi pour plusieurs raisons. D'une part car plusieurs informations doivent être transmises : le titre de la vidéo, l'url vers la vidéo, l'url vers l'image d'aperçu et enfin l'url vers la page publique. Ainsi de la même manière que les émoticônes, une balise va être insérée dans le texte afin de pouvoir par la suite identifier les éléments *WittiZ*. Le format utilisé est présenté figure 19. D'autre part il va falloir modifier intelligemment le *parsing*¹⁵ du texte en trouvant un système possiblement extensible à d'autres modules afin d'éviter de le parser deux fois (une fois pour les émoticônes et une autre pour *WittiZ*). Enfin contrairement aux émoticônes, les vidéos *WittiZ* sont ex-

¹⁵En français décomposition analytique. Découpe un flux continu de caractères en informations, puis en construit un arbre d'analyse.

trêmement nombreuses et la seule ressource disponible localement est une url vers l'image d'aperçu, il va donc falloir concilier le *parsing* en temps réel du texte et la requête asynchrone permettant de récupérer l'image à afficher.

```
Hello, :), toujours malade ?
<wittiz-addition>
{thumbnailUrl ::http ://p.com/img.jpg
 videoUrl ::http ://p.com/vid.mp4
 publicUrl ::http ://p.com/public
 label ::'My Label'}
<wittiz-addition>
```

Fig. 19 : Chaîne de caractère utilisée pour caractériser une vidéo Wittiz

3.3.3.2.1 Modification du parsing : Avant intégration, le *parsing* des émoticônes fonctionnait grâce à deux méthodes, la méthode `autodetectEmoticons` de la classe `TextFieldAdditions` se chargeait de parcourir le contenu du champ texte et pour chaque nouvelle position appelait la méthode `searchFirstEmoticon` de la classe `Emoticon` qui pour un texte et une position donnés tentait d'identifier par comparaison de chaînes de caractères, l'équivalent texte d'un des émoticônes. Si un émoticône était identifié, une structure contenant sa position, sa taille ainsi que son image était renvoyée à la première méthode qui se chargeait alors de générer une `attributedString` avec cet émoticône pouvant être assigné à l'attribut `attributedText`.

Afin de pouvoir réutiliser un maximum de ce code une classe abstraite `Addition` a été créée, elle contient la structure de données qui est retournée lorsqu'un élément (émoticône ou vidéo Wittiz) est identifié dans le texte. Elle contient également le prototype de la méthode `searchFirstAddition` qui devra être implémenté par les classes qui hériteront de cette dernière. On a donc également deux classes `EmoticonAddition` et `WittizAddition` qui héritent de `Addition`. Le code de la première `EmoticonAddition` n'a pas été modifié et reste donc identique à ce qui se faisait avant l'intégration. Pour la classe `WittizAddition`, la méthode `searchFirstAddition` va utiliser des *regex*¹⁶ afin d'identifier rapidement le *pattern*¹⁷ Wittiz. Enfin dans la classe `TextFieldAdditions` la méthode `autodetectAdditions` utilise désormais l'introspection afin de dynamiquement appeler la méthode de détection des *additions* dans toutes les classes qui implémentent cette méthode. Dans le cas où pour une même itération plusieurs éléments sont détectés, ce qui est théoriquement impossible, seul l'élément de plus grande taille et qui inclut donc l'autre trouvé est retenu.

¹⁶Expression rationnelle.

¹⁷Chaîne de caractère avec des parties fixes et des parties variables constituant schéma facilement identifiable.

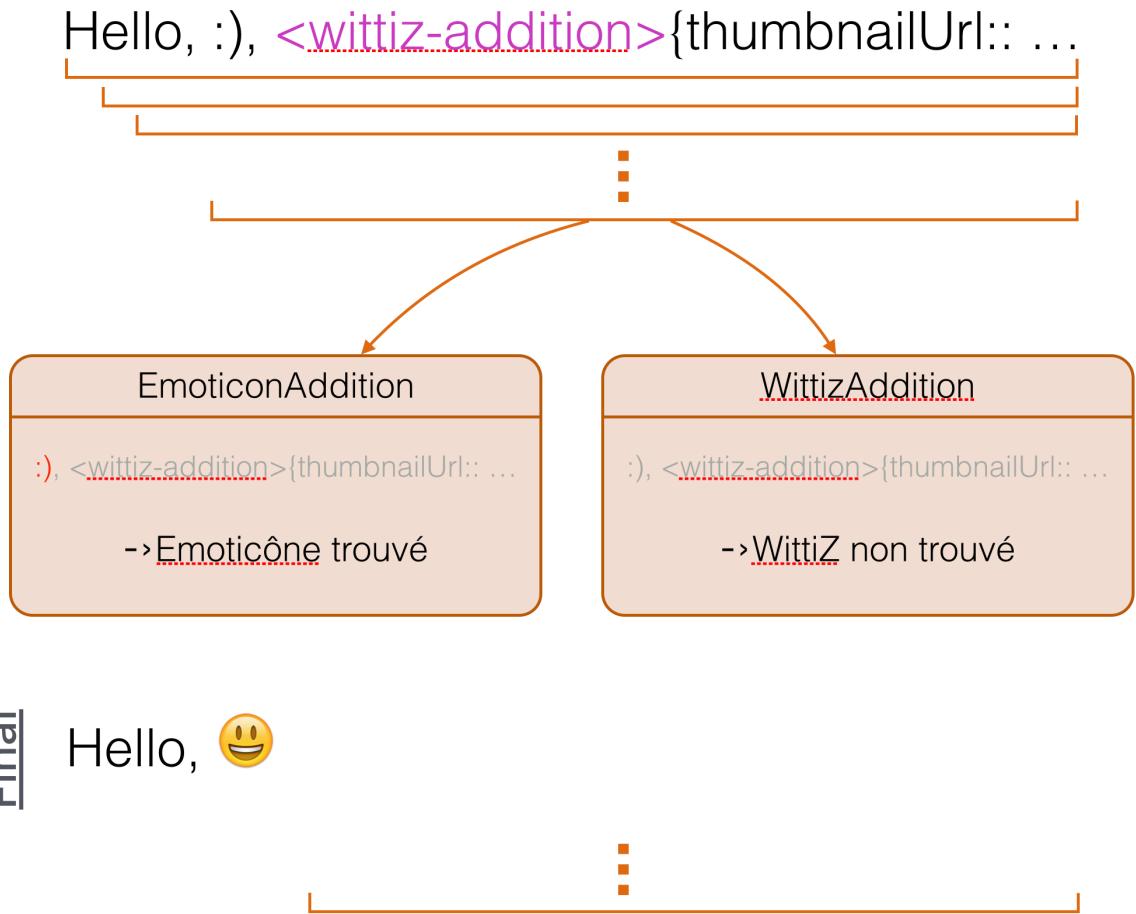


Fig. 20 : Schéma de fonctionnement du *parsing*

3.3.3.2.2 Requête asynchrone et parsing : Par définition le *parsing* se fait en temps réel, or les requêtes web se font de manière asynchrone, c'est à dire que la requête est lancé mais le fonctionnement de l'application continue sans attendre la réponse, c'est seulement quand la réponse arrive qu'elle est traitée. Deux solutions étaient possibles pour traiter ce problème. Effectuer un traitement bloquant, c'est à dire bloquer le traitement de l'application, du *thread*¹⁸ en l'occurrence, jusqu'à ce que l'on ait reçu une réponse du serveur web. Cette solution aurait pu être envisagée si le *thread* alors bloqué n'effectuait que ce traitement, cependant ce traitement a lieu sur le *thread* principal qui a à sa charge l'interface utilisateur ; ainsi bloquer le *thread* entraînerait un blocage de l'interface, ce qui n'est pas acceptable. L'alternative est donc de rendre synchrone la requête web du moins en apparence puisqu'il est techniquement impossible de la rendre synchrone. L'idée est donc lorsqu'un élément *WittiZ* est détecté de retourner une image vide puis de continuer le *parsing* tout en ayant conservé une référence vers cette image afin de pouvoir la remplacer par l'image finale lorsque cette

¹⁸Fil d'exécution représentant l'exécution d'un ensemble d'instructions en langage machine du processeur.

dernière a été téléchargée. Le challenge étant de conserver les différentes références vers les différentes images vides puis de les remplacer dynamiquement. Les **Futur** un type d'objet introduit par les équipes ayant développé *Libon* permet de solutionner ce problème. Un **Futur** est un type générique qui permet de retourner de façon synchrone un objet factice du type spécifié puis de le remplacer dynamiquement par l'objet final une fois que ce dernier est disponible. La figure 21 détaille le fonctionnement de cet objet dans notre situation.

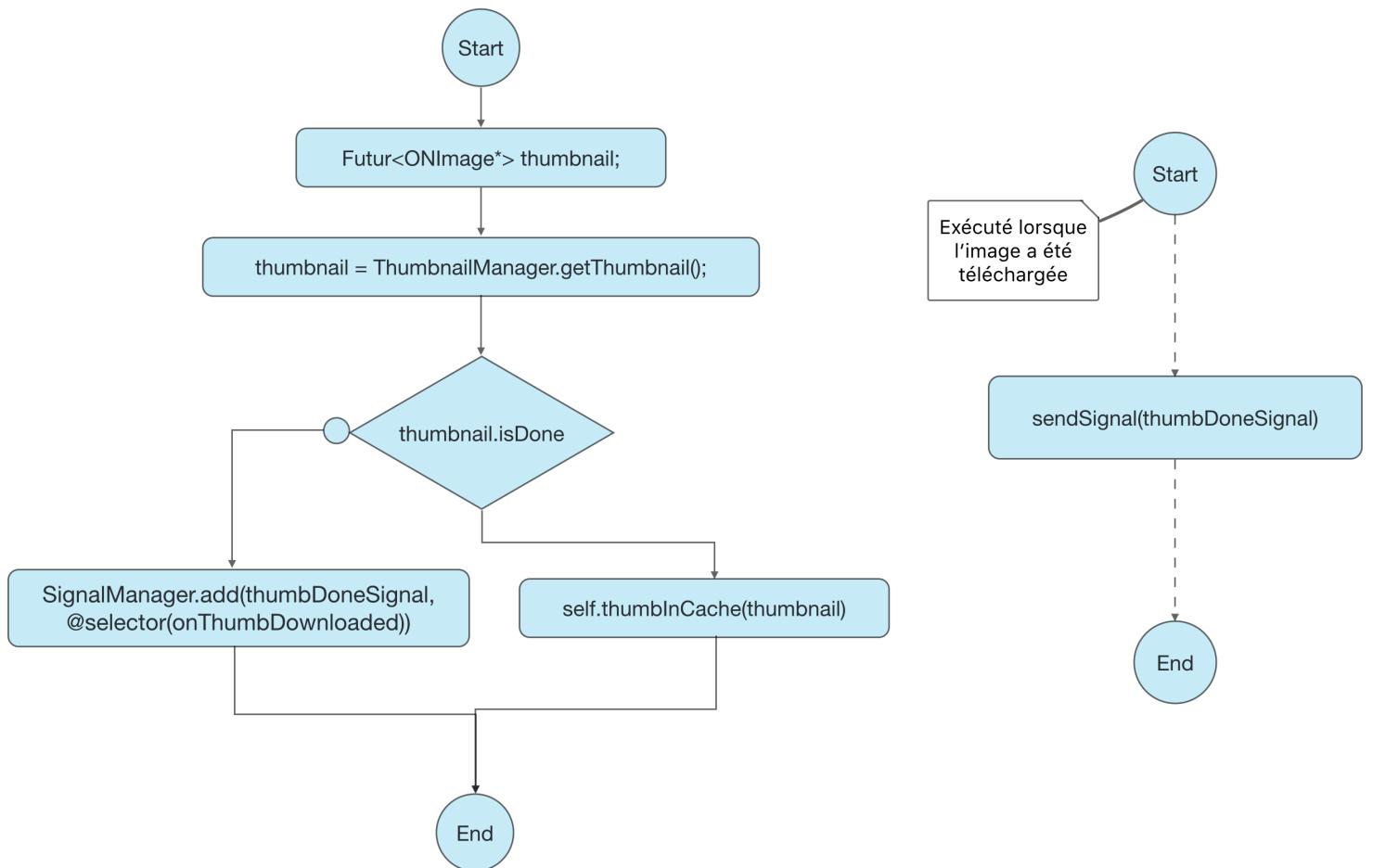


Fig. 21 : Schéma de fonctionnement des *Futur*

Le seul inconvénient de cette solution est qu'un carré blanc de la taille de l'image est affiché dans l'interface utilisateur le temps que l'image se télécharge. Cependant, suite à des tests et dans la mesure où le téléchargement se faisait en moyenne en 1,3s, il a été jugé en accord avec le *PO* que ce détail n'était pas très gênant.

L'intégration du module *WittiZ* se fait donc désormais jusqu'au champ texte avec la possibilité d'afficher une image d'aperçu de la vidéo directement au sein de ce dernier. Cependant à ce stade si l'utilisateur venait à presser le bouton *Envoyer*, rien ne se passerait. Nous allons donc nous intéresser dans la section suivante à comment la vidéo à partager va arriver jusqu'au téléphone du destinataire.

3.3.4 Algorithme pour l'envoi des vidéos

Lors de l'établissement des spécifications il avait été convenu que les vidéos partagées apparaîtraient dans le flux des messages sous la forme d'un lecteur intégré permettant de lire directement la vidéo. La partie gauche de la figure 22 montre le contenu du champ texte tel qu'il est affiché à l'utilisateur lorsqu'il a composé son message. La partie droite quant à elle représente comment le contenu doit être présenté dans le flux des messages une fois qu'il a été envoyé. Ce que l'on peut noter est que le message composé en une seule fois a été découpé en deux messages : un premier de type texte suivi d'un lecteur vidéo avec la vidéo partagée.

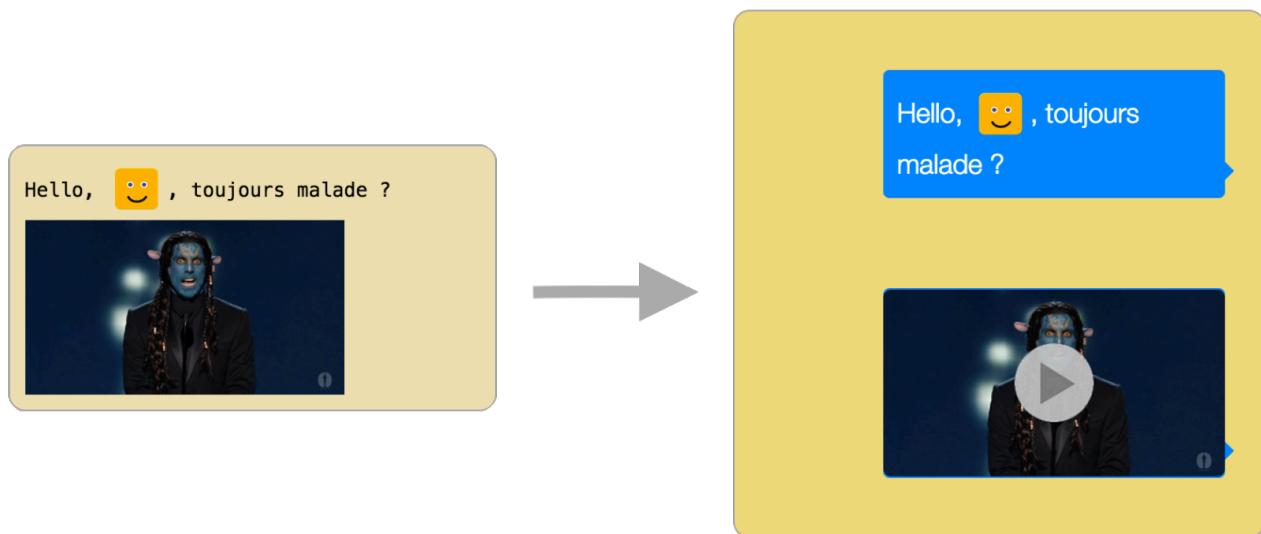


Fig. 22 : Passage du champ texte au flux des messages

Concrètement le processus d'envoi va se faire en deux grandes étapes que sont la transformation du contenu du champ texte en texte exploitable (que l'on va de nouveau parser et découper) et l'envoi concret des messages sous un format à déterminer.

3.3.4.1 Découpage du champ texte en morceaux de même type

La transformation du texte brut en objet dynamique correspondant à l'image d'un émoticône ou de Wittiz a été détaillée en section 3.3.3.2. Pour l'envoi du message ce processus doit être inversé, heureusement l'opération est bien plus aisée qu'en sens

inverse puisqu'il s'agit dans ce cas de parcourir l'`attributedString` et pour chaque élément de type `NSTextAttachment` le remplacer par son équivalent texte sauvegardé préalablement dans un de ses attributs. On obtient alors une chaîne de caractères, du type de celle présentée en figure 19. Une dernière opération de découpage est nécessaire avant de pouvoir être exploitée. L'objectif est d'isoler les différents types d'éléments constituant cette *String*¹⁹ tout en gardant la chronologie utilisée lors de la rédaction du message. Ainsi la chaîne de caractère `<wittiz-addition>` va être utilisée comme délimiteur pour ce découpage. On enverra alors autant de messages que de morceaux créés. Si l'on conserve l'exemple de la figure 19, l'envoi se fera en deux messages avec en premier un message de type texte puis un de type vidéo.

Il ne reste désormais plus qu'à faire l'envoi des messages sur le réseau. Pour se faire l'application possède des *Events* qui correspondent à divers événements se produisant dans l'application et que nous allons détailler dans la section suivante

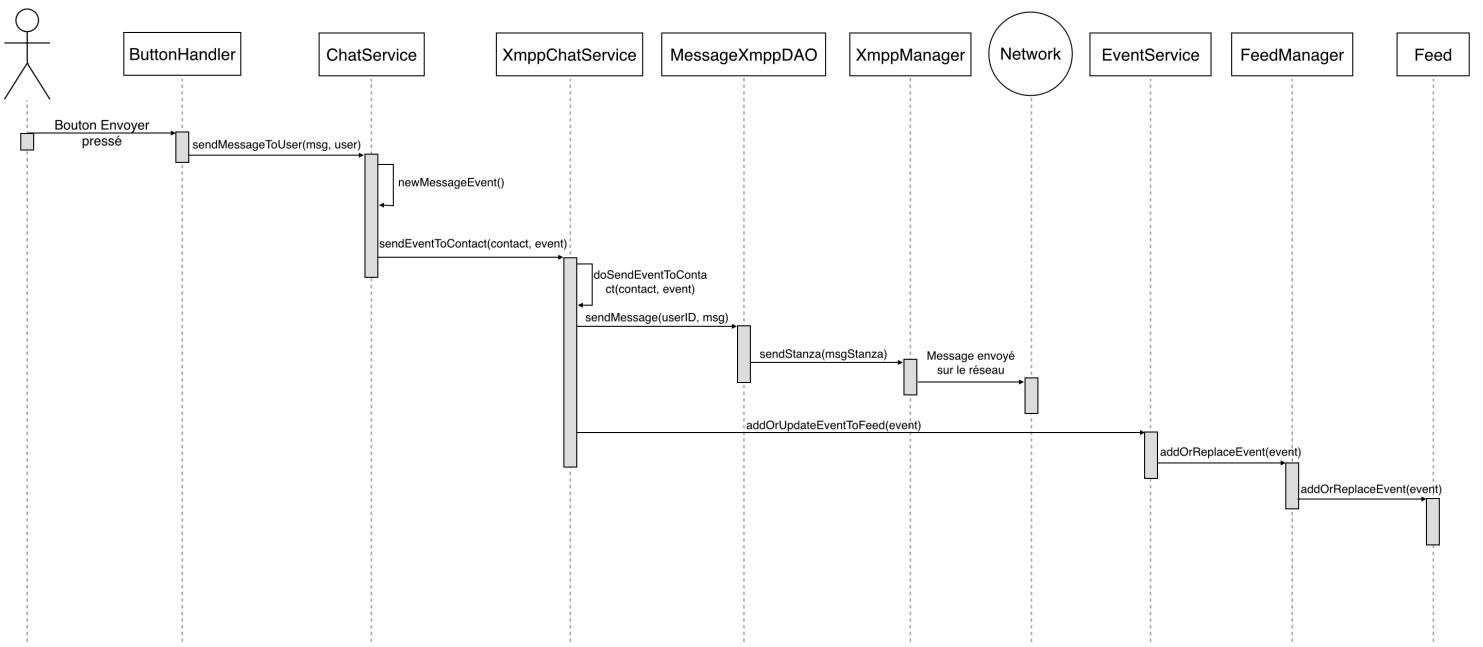
3.3.4.2 Chaîne d'envoi et de réception des Events

Les *Events* dans *Libon* sont nombreux et n'ont pas tous le même fonctionnement. De manière non exhaustive je peux mentionner le *CallEvent*, le *ConferenceEvent*, le *FileTransferEvent*, le *MessageEvent*, le *NotificationEvent* et le *LocationEvent*. Nous allons seulement nous intéresser à trois de ces événements qui ont un fonctionnement similaire et qui seront utiles à la résolution de la problématique.

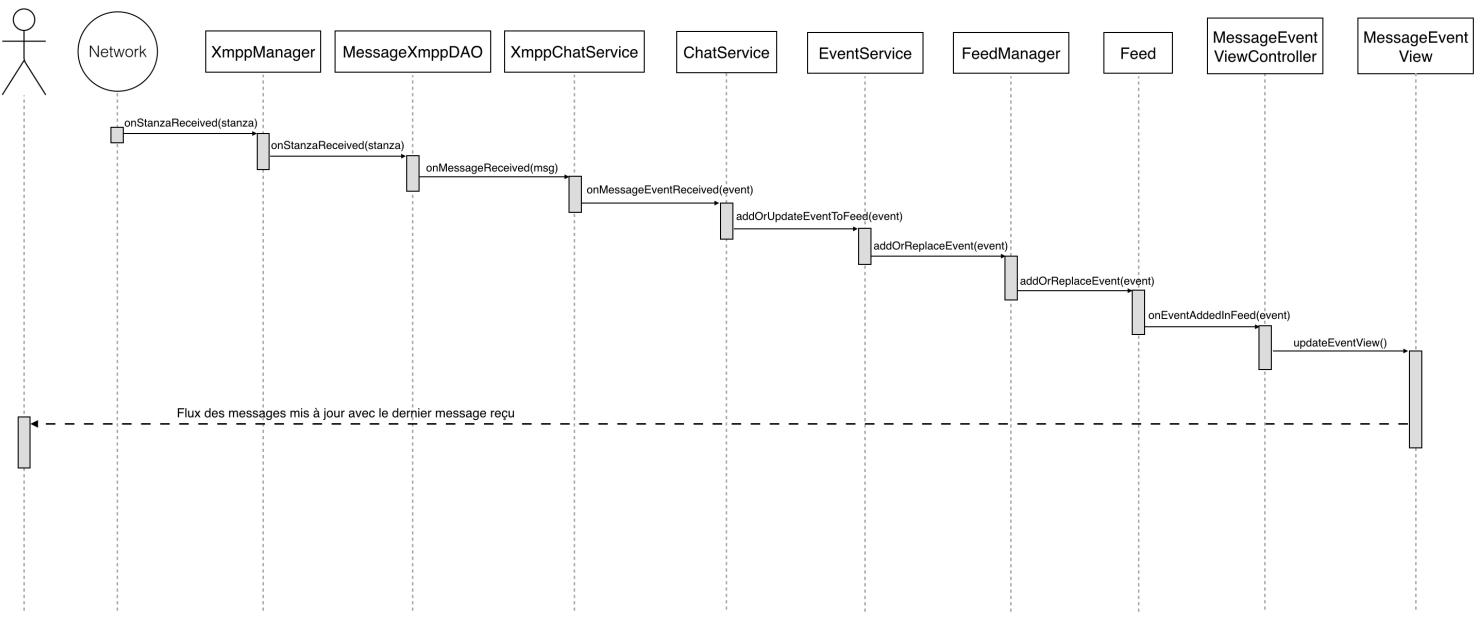
Le diagramme de séquence visible figure 23a détaille de manière simplifiée le fonctionnement des événements *MessageEvent* et *LocationEvent* qui sont utilisés respectivement pour l'émission et la réception de messages de type texte et pour le partage de sa position *GPS*. Dans un premier temps lorsque l'utilisateur va appuyer sur le bouton *Envoyer*, il va exécuter la méthode `sendMessageToUser` du service *ChatService*. Ce dernier va alors effectuer plusieurs traitements en créant notamment un nouvel événement, ici, un *MessageEvent* qui va être rempli avec le texte du message. L'événement est alors transmis au *XmppChatService* via la méthode `sendEventToContact`. Ce service va assurer la conversion de l'objet *MessageEvent* en message *Xmpp*²⁰ pouvant être envoyé sur le réseau, il va également vérifier que le destinataire est bien un utilisateur de *Libon* et auquel cas établir la connexion sinon une invitation sera envoyée au correspondant. Enfin ce service va assurer la persistance du message dans l'application de l'utilisateur en ajoutant une entrée dans la base de données *sqlite* locale de l'application. Une fois que le message *Xmpp*, dont la syntaxe est présentée figure 24, est envoyée par l'intermédiaire du *XmppManager* il va transiter sur le réseau jusqu'à son destinataire.

¹⁹Chaîne de caractères.

²⁰Extensible Messaging and Presence Protocol, est un ensemble de protocoles standards ouverts pour la messagerie instantanée.



(a) Diagramme de séquence pour l'envoi



(b) Diagramme de séquence pour la réception

Fig. 23 : Diagrammes de séquences partiels pour l'event `MessageEvent`

Du coté du correspondant (figure 23b), c'est également le `XmppManager` qui va recevoir en premier le message. Le type du message va alors être analysé afin d'en parser

le contenu correctement. Une fois parsé un signal avec le contenu du message est envoyé. Ce signal, surveillé par le `XmppChatService`, est réceptionné et traité afin de créer l'`Event` approprié. Ce dernier passe alors par le `ChatService` avant d'être transféré à l'`EventService` qui va principalement assurer la synchronisation de ce nouvel `Event` avec les *back-end*²¹ *Libon*. En effet ces *back-end* conservent une copie en ligne de tous les messages envoyés et reçus, permettant en cas de changement de téléphone ou de restauration de retrouver la totalité de ses messages. L'événement est ensuite transmis au `FeedManager` qui se charge de l'ajouter au bon `Feed`. Un `Feed` étant un objet qui représente le flux des messages d'une conversation, il s'agit donc d'une liste d'événements. Enfin lorsqu'il est modifié, le feed envoie le signal `eventAddedInFeed` afin d'avertir toutes les classes qui le souhaitent de l'ajout d'un nouvel event. Ce signal est notamment reçu par la classe `EventViewsController` qui contrôle les vues régissant l'affichage des events dans l'interface utilisateur.

```
<?xml version='1.0' encoding='UTF-8'?>
<stream :stream xmlns :stream='http://etherx.jabber.org/streams' version='1.0'
  xmlns='libon :client' to='libon.com' >
  <message id="1" type="chat" from='alpha@libon.fr' to='tango@libon.fr'>
    <body>
      Lorem ipsum dolor sit amet, consectetur adipiscing elit.
      Quisque tincidunt leo dui, nec dapibus lorem maximus a.
    </body>
  </stream :stream>
```

Fig. 24 : Message Xmpp pour l'événement `MessageEvent`

L'événement `FileTransferEvent` est utilisé pour envoyer une vidéo qu'un utilisateur a filmé sur son téléphone. Cependant contrairement aux deux events présentés dans le paragraphe précédent, le fonctionnement de ce dernier diffère lors de l'envoi. Comme le montre le diagramme de séquence figure 25, l'utilisateur commence par sélectionner une vidéo déjà enregistrée dans sa bibliothèque ou en filme une nouvelle. Lorsqu'il sélectionne la vidéo, la méthode `convertAndSendVideoAtURL` du contrôleur `ChatMoreActionsViewController` est appelée. En substance cette méthode va créer le `FileTransferEvent` et préparer la vidéo, disponible localement, à son upload sur les *back-end* *Libon* dans la mesure où les vidéos partagées sur *Libon* ne sont pas partagées entre utilisateurs, mais sont systématiquement stockées sur les serveurs de *Libon* puis lues en streaming sur les terminaux mobiles. Quand la vidéo est prête à être uploadée la méthode `sendFile` du service `FileTransferService` est exécutée afin d'effectuer la requête `POST` déclenchant l'upload de la vidéo. Une fois l'envoi terminé une url vers la vidéo est retournée sur le terminal de l'émetteur. Les serveurs de *Libon* vont se charger de créer et d'envoyer au destinataire le message `Xmpp` correspondant au `FileTransferEvent` qui le traitera alors de la même manière que les autres événements (figure 23b).

²¹Serveurs assurant une partie des opérations nécessaires au fonctionnement de *Libon*

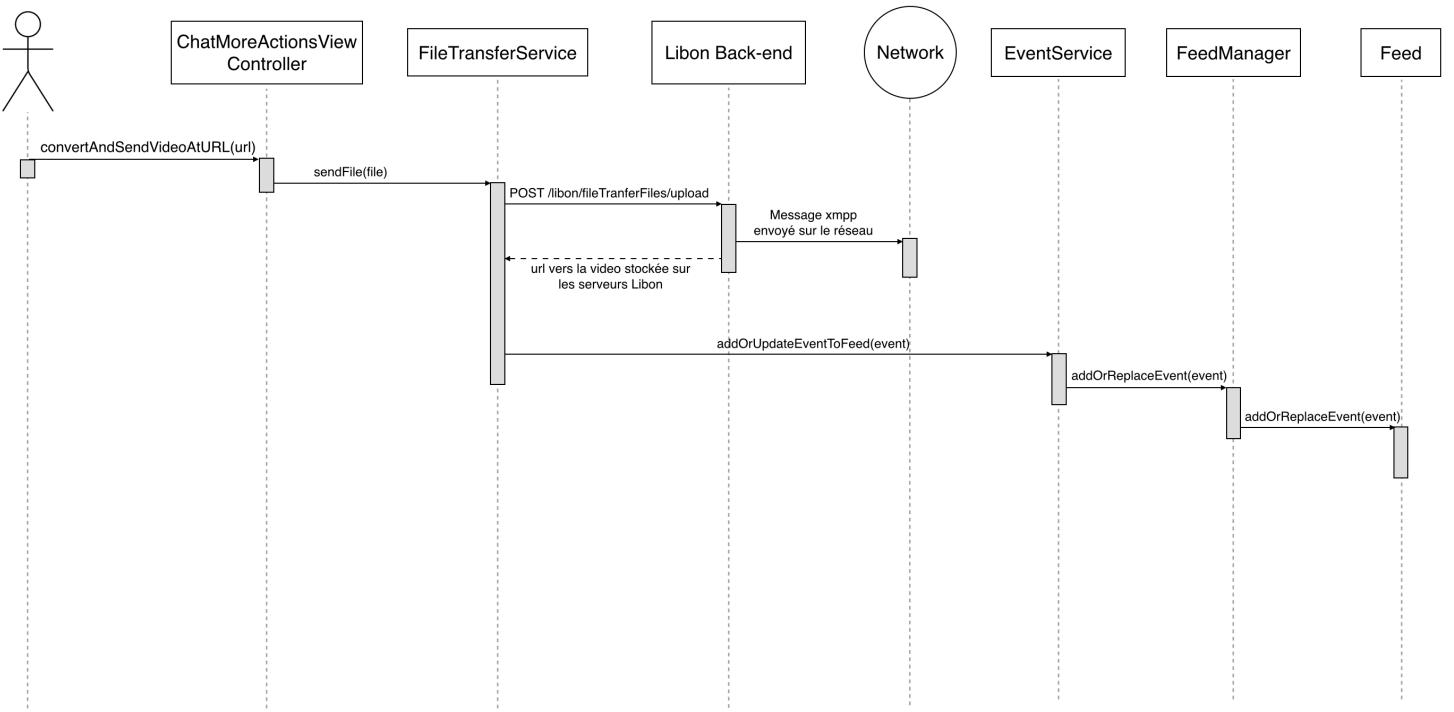


Fig. 25 : Diagramme de séquence pour le *FileTransferEvent*

Dans cette dernière section nous avons détaillé le fonctionnement des *Events* pour l'envoi de messages de différents types. Dans la section suivante les différentes solutions étudiées ainsi que celle retenue pour la transmission d'une vidéo *WittiZ* seront exposées.

3.3.4.3 Un Event adapté à la transmission des vidéos *WittiZ*

Lors de la réalisation des spécifications, la solution à adopter pour l'envoi des vidéos avait été partiellement étudiée. L'approche semblant la plus logique était d'envoyer, sous une forme masquée pour l'utilisateur, le lien vers la vidéo que l'on souhaite partager. De fait seul du texte transite sur le réseau. En effet télécharger la vidéo pour la faire retransiter sur le réseau est contre productif.

Au vu du fonctionnement de *Libon* détaillé en section 3.3.4.2, plusieurs solutions ont été étudiées afin de coller au plus proche du comportement souhaité dans les spécifications.

- La première solution, en apparence la plus évidente, était de réutiliser ce qui se faisait pour le partage des vidéos dans *Libon*. Cependant comme nous l'avons vu ce système de partage a été conçu pour les vidéos présentes localement dans la mémoire du téléphone. Ainsi pour utiliser le *FileTransferEvent* il faut nécessairement avoir une vidéo à uploader ce qui implique dans notre cas de télécharger auparavant la vidéo des serveurs *WittiZ*; solution contre productive en l'état. Une modification de cet événement a été envisagée afin d'autoriser en entrée une url

vers une vidéo. Cependant dans la mesure où la majorité des traitements se font du côté des serveurs *Libon* toute modification était impossible sans intégrer de nouvelles personnes dans la boucle.

- Une autre solution était de passer par un *MessageEvent*. En effet dans la mesure où l'on souhaite avant tout transmettre du texte, utiliser l'événement fait pour envoyer du texte n'est pas si illogique. Le seul problème étant d'arriver à masquer habilement le texte et l'url *WittiZ* en les remplaçant, dès que détectés, par un lecteur vidéo.

D'un point de vu purement technique, les modifications à apporter étaient relativement limitées et consistaient principalement à modifier l'*EventViewController* (qui se charge de l'affichage des *EventView*) afin que ce dernier affiche une vue spécifique pour les vidéos *WittiZ* lorsqu'il en détecte une. Toute la subtilité de cette solution est justement de détecter la syntaxe de ces vidéos. Il s'agit donc d'ajouter dans ce contrôleur un système de *parsing* qui en cas de résultat positif affichera la vue appropriée. En parcourant le code j'ai ainsi découvert que cette solution avait déjà été utilisée et implémentée par les développeurs de *Libon*. Dans le cas où un lien *Youtube* est détecté, le lien est remplacé par une vue personnalisée avec une image d'aperçu et la description de la vidéo. L'inconvénient principal du *parsing* est qu'il ne passe pas facilement à l'échelle et si l'on souhaite avoir le même comportement que les liens *Youtube* avec toutes les plates-formes de partage de vidéo, les performances et la lisibilité du code vont rapidement se dégrader.

- La dernière solution était de mettre en place un nouveau type d'événement qui permettrait de facilement envoyer des vidéos disponibles sur des serveurs distants. Le fonctionnement de cet event nommé *RemoteVideoEvent* va se calquer sur celui du *MessageEvent*, les différences vont se situer au niveau du format du message *Xmpp* envoyé (figure 26) ainsi que de l'*EventView* associée qui se compose d'un lecteur vidéo embarqué et du titre de la vidéo. Un des inconvénients de cette solution est la charge de travail impliquée. En effet même si le fonctionnement de cet événement est similaire à l'existant, un grand nombre de lignes de code va devoir être modifié, ce qui nécessite d'effectuer des tests afin d'assurer la non régression du code. Ces tests sont facilement exécutable quand ils ont été écrits et implémentés au fur et à mesure du développement, voire en amont dans le cas du développement dirigé par les tests. Cependant ce n'est pas le cas pour *Libon* où aucun test n'existe, les seules procédures de test se font de visu et implémenter des tests maintenant représenterait une énorme charge de travail que le *PO* n'est prêt à déployer.

En accord avec Fabien Crapiz, c'est la dernière solution qui a été retenue. Cette implémentation c'est fait sans trop de problème dans la mesure où une grosse partie du code était similaire à celui des autres events. Une attention particulière a toutefois été portée sur la rétrocompatibilité de cette fonction avec les versions de *Libon* plus anciennes qui risquaient de ne pas recevoir ce type de vidéo dans la mesure où n'ayant pas connaissance de ce nouvel événement, serait simplement ignoré par l'application. Heureusement à chaque utilisateur *Libon* est rattachée la version utilisée, ainsi avant d'envoyer un *RemoteVideoEvent*, on s'assure que le destinataire sera en mesure de le reconnaître, sinon on se rabat sur un simple message texte avec un lien vers l'url

publique²² de la vidéo.

```
<?xml version='1.0' encoding='UTF-8'?>
<stream :stream xmlns :stream='http ://etherx.jabber.org/streams' version='1.0'
xmlns='libon :client' to='libon.com' >
  <message id="1" type="chat" from='alpha@libon.fr' to='tango@libon.fr'>
    <x type="remote_video" >
      <video-url>http ://p.com/vid.mp4</video-url>
      <thumbnail-url>http ://p.com/img.jpg</thumbnail-url>
      <public-url>http ://p.com/public</public-url>
      <label>My Label</label>
    </message>
</stream :stream>
```

Fig. 26 : Message Xmpp pour l'événement *RemoteVideoEvent*

Cette dernière étape d'intégration permet désormais d'exploiter pleinement les fonctionnalités du module *WittiZ*, développé durant le stage, au sein de l'application *Libon*.

²²Chacune des vidéos *WittiZ* possède une url publique pointant vers un site internet aux couleurs de la société *WittiZ* où il est possible de lire la vidéo.

4 Conclusion

Six mois de stage, finalement, cela passe très vite. Ainsi un des plus gros enjeux est d'arriver au terme de son projet avec la livraison d'un produit abouti correspondant au cahier des charges.

Ainsi, malgré mon manque d'expérience, j'ai réussi à fournir dans les temps un livrable qui répond aux besoins exprimés par le *PO*. Pour cela, j'ai dû composer avec quelques réalités du terrain telles que la communication avec les différentes équipes, rendue compliquée par la distance. Ce qui m'a parfois amené à prendre les devants dans la prise de décision et à aller chercher l'information par moi-même.

Le seul élément manquant est la partie authentification à l'API *WittiZ*, qui n'a pas été réalisée par manque de ressources. En effet des personnes travaillant chez *WittiZ* et sur les back-end de *Libon* auraient dû travailler avec moi, or personne n'était disponible pour travailler sur cette fonctionnalité, non prioritaire, au moment de mon stage.

Toutefois, l'intérêt du *PO* suscité par le module que j'ai réalisé laisse à penser que ce dernier sera prochainement intégré dans la version en production de *Libon*.

5 Bilan

Ces six mois de stage, ont avant tout été l'occasion d'acquérir de bonnes bases de développement pour la plate-forme *iOS*. J'ai également eu l'occasion de travailler avec de nouveaux outils (*Mercurial*, *JIRA*), de mettre en pratique les méthodes de travail *Agiles* et d'en découvrir de nouvelles : les *DevOPS*. Par ailleurs, en seulement 6 mois l'équipe à laquelle j'appartenais a connu de nombreux bouleversements (changement de manager, restructuration de l'équipe, abandon de projet, ...). Tous ces chamboulements sont bien heureusement plutôt rares mais ont permis de rendre cette expérience au sein d'un grand groupe très enrichissante pour ma carrière professionnelle.

Toutefois ce stage ne m'a pas fait changer d'avis concernant l'environnement et le type de sociétés dans lesquelles je souhaite travailler, du moins pour débuter ma carrière. Ainsi ma recherche d'emploi concerne majoritairement des *start-up* ou *PME*, dans le domaine de l'internet des objets. Je cherche dans un premier temps un poste d'ingénieur *iOS* afin de consolider mes acquis et me forger une expérience. Ce qui me permettra d'évoluer par la suite vers un poste avec plus de responsabilités comme, par exemple, la supervision d'une petite équipe de développeurs.