Rapport Proj-Juin

ShopIt

Equipe F
Kevin Duglue
Charly Lafon
Antoine Lupiac
Thomas Monzein

19 Juin 2017

Sommaire

Contexte

L'organisation des courses a toujours été compliqué. Pour illustrer cela, nous allons prendre deux exemples.

Exemple 1 : **Henry**, 30 ans, vit dans un quartier paisible de Nice. Chaque année, un barbecue est organisé entre tous les voisins du quartier. Mais à chaque fois, des malentendus sur la liste de courses font qu'il manque toujours quelque-chose. Ce qu'ils leur faudraient est une plateforme en ligne commune à toutes les personnes, afin de pouvoir construire une liste de courses commune. De plus, une répartition automatique leur permettrait de savoir qui achète quoi, en ayant chacun un investissement plus ou moins équivalent.
Exemple 2 : **Laura** est une mère de famille qui aime faire ses courses seule mais avant tout qui aime découvrir de nouveaux plats. Ainsi, elle aimerait avoir une application qui lui permette d'avoir accès à une multitude de recettes en ligne et de pouvoir construire rapidement et simplement une liste de courses en y ajoutant les ingrédients automatiquement. Mais ne possédant pas de téléphone dernier cri, cette application devrait être peu gourmande et pouvoir avoir un mode hors-ligne étant donné que sa connexion est peu stable une fois dans le magasin.

ShopIt vise à simplifier l'organisation des courses, que ce soit seul ou à plusieurs. Notre volonté de nous démarquer des concurrents avec innovation réside dans le partage des listes. Afin d'éviter l'utilisation d'intermédiaire non conçu pour les listes de courses comme Messenger ou Google Doc, nous voulons donner un moyen simple et efficace de faire ses courses à plusieurs. En effet, à l'aide d'une synchronisation en temps réel de la liste, l'ajout d'article se fera de manière la plus fluide possible.
De plus, nous sommes conscient que certaines personnes doivent respecter un budget (colocataire, association, …). C'est pourquoi nous voulons calculer automatiquement le prix de chaque articles afin d'avoir une estimation du prix total de la liste.

Afin d'organiser au mieux les achats, une répartition des articles entre les différents participants nous a paru indispensable. Cette répartition pourra être en fonction du prix, si le but final est une équité au niveau du prix, ou encore par catégorie afin d'accentuer les chances de trouver tous ces articles au même endroit.
Enfin, afin de faire profiter la communauté ou d'avoir de nouvelles idées, nous voulons permettre aux utilisateurs d'accéder aux recettes créé par la communauté. Outre une description, le réel avantage est de pouvoir rajouter directement et simplement l'ensemble des ingrédients de la recette dans une liste.

Etat de l'art

ShopIt est une application mobile pour faire des listes de courses et les partager. Nous avons pensé à ShopIt comme un moyen de faire une liste de course à plusieurs et de faire ses courses en se partageant les achats. Le but de ShopIt est d'être plus simple à utiliser pour faire ses courses que des listes papier, une application de messagerie comme **Messenger** ou de **Google Doc**.
Cependant, il existe déjà sur le marché des applications mobiles de listes de courses partagées. Nous allons comparer notre solution, ShopIt, avec deux autres applications disponibles qui sont **Bring!** et **Lister**. Le tableau ci-dessous fait le récapitulatif des différentes fonctionnalités présentes dans chacune des solutions :

Fonctionnalités	Bring!	Lister	**ShopIt**
Plateforme	Cross-Plateform	Cross-Plateform	Cross-Plateform
Mode hors ligne	Pas de création de liste	Oui	Oui
Liste de courses partagée	Oui	Oui	Oui
Temps réel	Non	Non	**Oui**
Invitations	Oui	Oui	Oui
Prix des articles	Non	Non	**Oui par l'intermédiaire de la communauté**
Catégorie des articles	Oui	Oui	Oui
Recettes	Recette personnelle	Non	**Recette de la communauté**
Répartition des achats	Non	Non	**Manuelle ou automatique (par prix ou catégorie)**
Communication	Messages prédéfinis	Sms/Mail	Non
Choix des articles	Limité aux articles présents dans l'app	Limité aux articles présents dans l'app	**Entrée utilisateur textuelle**
Envoi de la liste	Non	Sms ou mail	Non
Manipulation des articles	Ajouter, supprimer, quantité et détails	Ajouter, supprimer, rayer, quantité	Ajouter, supprimer, rayer, pas trouvé, quantité

Nous pouvons voir que les applications couvrent des fonctionnalités similaires, mais se démarquent l'une de l'autre.
Lister permet de communiquer avec les participants à une liste via l'envoi d'email et de SMS. Les fonctionnalités futures se dirigent vers la grande distribution avec un catalogue de produit avec leur prix ou la commande en ligne. Il intègre une facilité d'utilisation par la commande vocale en plus du clavier.
Bring ! contient moins de fonctionnalités, mais se démarque par une utilisation simple et plus design notamment par des articles génériques déjà définis. Les articles à ajouter sont accessible par catégorie et l'utilisateur peut ajouter des articles en l'écrivant puis en ajoutant une photo et une description de l'article.

L'application ShopIt a été conçu pour simplifier les courses faites à plusieurs ou chacun achète une partie des articles de la liste. Ainsi notre solution se démarque par la répartition des achats entre les utilisateurs de la même liste partagée. Nous proposons une répartition manuelle ainsi qu'une répartition automatique par catégorie ou par prix.
De plus, ShopIt veut créer une communauté via les recettes proposées par cette dernière ou l'estimation des prix qui repose sur les photos de tickets de caisses des utilisateurs. Ces fonctionnalités sont soutenues

par de la gamification qui permet de récompenser par des badges l'utilisateur pour contribution à la communauté. Ces derniers sont vues des utilisateurs des listes partagées.
Ces badges sont éphémères et les utilisateurs doivent contribuer constamment pour garder leur titre, ce qui enrichit l'aspect communautaire.

Enfin, notre solution a été développée en trois semaines de développement. Plusieurs nouvelles fonctionnalités sont envisagées pour la suite du développement de ShopIt comme par exemple un système de communication par Chat ou des partenariats avec des commerçants locaux.

Proposition technique

L'application cliente

Notre projet consiste en une liste de courses partagée. Cette seule définition impose la contrainte d'une solution disponible sur un dispositif mobile pour pouvoir utiliser la liste de course lors de nos achats en grande surface ou dans les commerces. Nous voulions toucher le plus grand nombre d'utilisateur possible et ainsi, que notre solution soit disponible sur tous les smartphones du marché.

Pour pouvoir proposer notre solution sur Android, IOS et Windows Phone, nous avons le choix entre un site **Web responsive**, le développement de **plusieurs applications natives** (Android et iOS) ou une **application hybride** cross-platform.
La première est un site responsive, qui s'adapte selon la résolution de l'écran. Il peut alors être accessible, utilisé sur n'importe quelle smartphone, tablette et même depuis un PC. Le développement d'un site Web peut être facile et l'utilisateur n'a pas besoin de mettre à jour son application. Cependant, un site web requiert une connexion internet et nous voulions offrir la possibilité à l'utilisateur d'utiliser un mode hors ligne. De plus, nous envisageons l'utilisation des capteurs disponibles sur un smartphone comme la caméra, ce qui n'est pas faisable avec un site Web.
Notre choix se porte alors sur le développement d'une application mobile.

Le développement natif apporte certains avantages au niveau des performances de l'application et l'utilisation de toutes les capacités du téléphone, notamment en terme d'éléments graphiques. Le prix à payer pour ces avantages est un développement de deux projets pour une application sur les systèmes Android et iOS. Nous n'avions que trois semaines de développement pour une petite équipe ce qui nous éloigne de cette solution par manque de ressource. Même sans prendre en compte cette contrainte, l'application finale ne doit pas effectuer de gros calcul qui seront délégués à un serveur pour gagner en légèreté de l'application. L'un des avantages du développement natif n'est donc pas exploité pour notre projet.

Enfin, le développement cross-platform répond aux désavantages exposés par le développement d'un site web et n'augmente pas les ressources nécessaires aux développements natifs sur les trois différents systèmes. Une **application hybride cross-plateforme** est moins efficace qu'une application native, mais comme exposé précédemment, cet inconvénient n'en est pas un pour nous. Ainsi notre choix s'est porté sur cette dernière solution.

Une application hybride cross-platform

De nombreuses technologies permettent de développer une application cross-plateforme. Nous nous sommes avant tout dirigés vers **Xamarin** puisque c'est une technologie qui semble être utilisé par les entreprises et qui présente l'avantage d'offrir des performances proche d'une application développée en natif. Cependant, bien que le cœur de l'application soit à développer une seule fois, il faut développer

séparément la couche IHM de chacun des systèmes. De plus, nous avons eu beaucoup de mal à mettre en place un projet sous Xamarin.
Notre seconde option est Ionic 2 qui nous a été conseillé et qui présente de nombreux avantages pour le développement. Le langage utilisé est Angular 2, qui a déjà été utilisé par tous les membres de notre équipe. La mise en place d'un projet est bien plus facile comme le développement pour toutes les plateformes puisqu'il suffit d'un seul et même code pour tous les systèmes. La communauté autour d'Ionic 2 est un bon atout car nous avons trouvé à notre disposition de nombreux guides et plugins. Enfin, des étudiants de notre promotion nous ont proposés leur aide si on rencontrait un problème technique sur Ionic.

Ionic 2 ne permet pas d'obtenir ni les performances, ni la finesse de construction d'éléments graphiques qu'offre une application native ou développée sous Xamarin. Cependant en ce qui concerne les performances, nous comptions déléguer le plus gros des opérations aux serveurs pour alléger le plus possible l'application. Ainsi la perte de performance en utilisant Ionic 2 a très peu d'importance pour notre application. Cependant, le **gain de temps** pour sa prise en main et le développement sur tous les systèmes est déterminant.
C'est pour cela que nous avons choisis de développer l'application sous Ionic 2.

Le serveur

Nous avions besoin d'un serveur Web afin de stocker les données des listes partagées, des utilisateurs et des recettes. Nous en avions aussi besoin afin d'effectuer des opérations lourdes qui ne devaient pas s'effectuer sur les smartphones des utilisateurs car cela ralentirait fortement l'application.

Ce serveur avait plusieurs besoins :

- Un accès efficace à une base de données quelconque
- Un déploiement à grande échelle au cas où l'application aurait du succès
- Des opérations asynchrones (que nous verrons plus bas pour FCM)
- Une exposition de Web Services efficace

Intégré dans un environnement Java EE, nous avons choisi **TomEE** car il répond à tout ces besoins. Nous avons choisi un environnement Java car nous voulions un langage que nous maitrisons bien afin de ne pas perdre du temps sur la formation d'un langage que notre équipe ne connait pas comme le PHP.

TomEE répond à nos besoins d'accès à une base de données car il possède une couche **OpenJPA** qui permet d'y accéder efficacement.

En ce qui concerne le déploiement à grande échelle, la couche **OpenEJB** de TomEE remplit le besoin en s'occupant de gérer le déploiement des composants sur le serveur.

Les opérations asynchrones comme nous le verrons plus tard ont été d'une important cruciale. Il nous fallait un serveur supportant ce type d'opérations. Bonne nouvelle, TomEE contient un composant **Apache ActiveMQ** qui contient un service de messaging (**JMS)** permettant d'envoyer des messages à un composant. JMS permet à ce composant de traiter ces messages de façon asynchrone.

Enfin, TomEE contient un composant **Apache CXF** qui permet de construire des Web Services RESTful et SOAP ce qui répond au besoin du serveur.

Nous aurions très bien pu nous servir par exemple d'un serveur Glassfish plutot que TomEE, seulement TomEE était le seul que nous maitrisions et nous ne voulions pas nous imposer une perte de temps sur la formation.

Persistence

Le serveur doit enregistrer les listes ainsi que les utilisateurs qui y sont liés. C'est pour cela que nous avons choisi une base de données relationnelles. Nous avons choisis SQL pour modéliser la base de données. Notre équipe a des connaissance en SQL et en NOSQL, mais ce dernier est plus adapté au big data avec énormément de données en sortie.

Nous utilisons JPA pour faire le lien entre le serveur J2E et le système de base de données. JPA permet d'utiliser plusieurs systèmes de base de données sans avoir à changer les appels dans le code.

Pour la base de données elle-même, nous avons choisi HSQL qui est réputé pour être léger et performant. Nous l'avons déjà utilisé lors du projet d'ISA. Nous avons regardé du côté de MySQL, mais il est dit qu'il n'est pas adapté pour un système qui va beaucoup modifier ses données. Cependant, les listes et leurs éléments sont voués à être modifié régulièrement, donc MySQL ne semblait pas être un choix optimal pour une réel mise à l'échelle.

Communication entre une application et le serveur

Afin d'exécuter des fonctions du serveur sur évènement utilisateur ou non, il faut qu'une application puisse à tout moment communiquer avec le serveur Web. Pour ça, nous avons fait exposer au serveur des Web Services.

Nous avions 2 choix : faire des Web Services RESTful ou suivant le protocole SOAP. Partant du fait que l'application Ionic est décrite en Angular 2 qui lui même est basé sur **JavaScript**, les objets décrits et envoyés par le Front-End sont en JSON. De ce constat, nous avons écarté les Web Services de type SOAP car étant entièrement batis sur XML, l'utilisation de ces WS aurait provoqué un parsing implicite de JSON vers XML, ou même de XML vers JSON pour les réponses du serveur. Ce parsing peut être coûteux en temps si les données à envoyer sont lourdes.

C'est pourquoi nous avons opté pour des Web Services REST, communiquant du JSON.

Problèmes et défis

Temps réel

Une des grandes difficultés de notre projet a été de proposer un temps réel pour l'utilisateur afin d'avoir un confort d'utilisation maximum. En effet, **contrairement aux principaux concurrents** qui proposent une synchronisation manuelle (appui sur un bouton pour rafraîchir), une de nos innovations voulait se porter sur la modification en direct des articles visibles pour tous les autres participants de la liste (à la manière d'un Google Doc) dans le but d'éviter les conflits d'écriture (doublons d'articles, …) et apporter une innovation.
Pour le bien de la première démonstration, nous avions fait une première ébauche de cette partie en élaborant un **auto-refresh** de l'application sous un intervalle de temps régulier. Cependant, nous étions clairement conscient que cette solution n'était **pas optimale**, tant par sa consommation d'énergie que sa consommation de données.

Firebase Cloud Messaging (FCM)

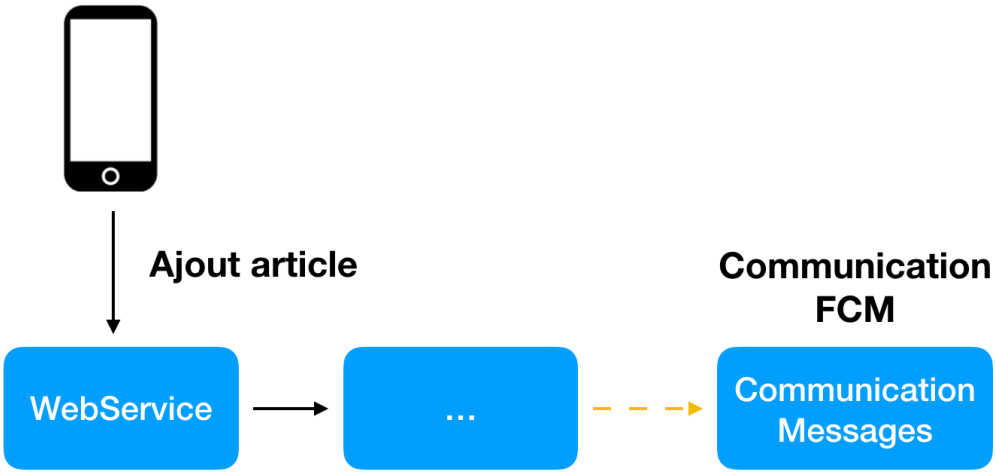
Le besoin du temps réel et le rejet de l'auto-refresh a amené à une problématique. Nous devions faire en sorte que le serveur Web communique directement avec un appareil mobile, sans que l'appareil mobile ait demandé quoi que ce soit. Comment le serveur Web pouvait t-il communiquer avec un appareil mobile spécifique ? Sachant que notre application mobile n'expose pas de Web Service ou autre mécanisme de ce genre. La solution optimale est **Firebase Cloud Messaging**. Ce service exposé par Google permet à n'importe qui d'envoyer un message à une instance d'application mobile bien spécifique pour peu qu'on en connaisse l'identifiant unique : son token.
Nous parlons de solution optimale car à notre connaissance, FCM est le service qui livre un message le plus rapidement (95% des messages sont livrés en dessous de 250 ms après la requête). De plus, FCM propose un Payload (charge) maximal d'envoi de 4 Ko, ce qui est plus que suffisant pour notre utilisation qui ne dépasse pas 1 Ko.

Une alternative à FCM était **Oracle Cloud Messaging**. Nous l'avons vite éliminé car ce service est payant. De même pour le service **Pushy** qui nous avait l'air très intéressant cependant il est aussi payant. Il existe

une version gratuite mais nous n'aurions pas pu excéder 100 applications clientes, ce qui n'est pas suffisant pour notre projet qui se veut d'une ampleur relativement bonne.

Java Message Service (JMS)

Un des grands défis de ce projet était l'asynchronisme. Nous avions besoin que le composant qui s'occupe de communiquer avec FCM soit asynchrone. En effet, une action comme l'ajout d'un article se résume par ceci :



Lors de l'ajout d'un article, on notifie tous les contributeurs de la liste afin que leur liste soit à jour. S'ils sont nombreux et que l'appel de la flèche jaune est synchrone, alors l'appareil mobile ayant ajouté un article devra attendre qu'on ait notifié tous les contributeurs avant d'avoir une réponse du WebService (OK j'ai réussi à ajouter ton article ou FAIL pour X raison). Ceci n'est pas envisageable. Il faut que l'appel de la flèche jaune soit **asynchrone** afin que l'appareil mobile n'attende pas que les notifications soient finies avant de pouvoir continuer son exécution (qui peut dépendre du message de retour du WebService).

L'utilisation de JMS, un système de messaging asynchrone qui fait partie de Apache ActiveMQ contenu dans TomEE était donc nécessaire. Nous avons choisi cette solution car nous l'avions déjà à disposition puisque le serveur Web tournait grâce à TomEE.
Cette solution est très efficace, cependant nous avons rencontré un problème qui pour le moment est toujours non résolu. Il nous était impossible de passer directement un objet au composant CommunicationMessages. Il a fallu parser cet objet en format JSON pour ensuite le récupérer dans le composant de communication et le reconstruire en l'objet d'origine. Ce contournement n'est pas propre, néanmoins d'un point de vue fonctionnel il marche tout aussi bien.

Ionic Push

Côté Front-end, pour recevoir les **Push messages**, nous avons utilisé Ionic Push. L'utilisation de ce plugin nous a procuré plusieurs avantages. Le premier, qui est le plus important, est d'assurer le "réel" temps réel. Ainsi, à la réception d'un Push message, une fonction spécifique du plugin est appelée. Enfin, à partir de cette fonction, on met à jour la vue correspondant au message automatiquement à l'aide de call-backs.
Le deuxième avantage de ce plugin et de l'utilisation de FCM est que l'on peut gerer en même temps le système de notification pour l'application. En effet, FCM proposant deux types d'envoi (notifications ou données), nous pouvons avec Ionic Push alerter l'utilisateur avec une notification, même si l'application est en arrière-plan.

Récupérer la catégorie d'un article

Pour le confort de l'utilisateur, les articles de la liste de courses doivent être séparés, selon leur catégorie. Ainsi l'utilisateur peut gagner du temps en faisant ses courses en lui permettant de récupérer tous les articles disponibles dans un **même rayon** ou un même commerce en même temps. De plus, nous voulions proposer une répartition des achats par catégorie. Nous avions donc besoin de trouver la catégorie correspondant à un article saisie par un utilisateur.

Nous avons identifié trois moyens. Le premier est le parsing d'un ou de plusieurs sites de grande distribution comme Carrefour ou Casino. Un problème majeur se pose alors si le code du site est modifié. Dans cette situation, notre parsing devrait être mis à jour.
Le second moyen est d'utiliser une API qui peut nous fournir les catégories du produit recherché. Par rapport à la solution précédente, on gagne en stabilité puisqu'une API est moins sujette à modification comme le peut-être un site internet. Cependant, nous resterions dépendant d'un service extérieur.
Enfin une dernière solution est de laisser l'utilisateur choisir sa catégorie. Cette solution nous défait de la responsabilité de trouver les catégories. Cependant, l'utilisateur peut se retrouver à entrer de mauvaises catégories même s'il doit choisir parmi des catégories valides. D'autre part, nous avons jugé que cette utilisation n'était pas satisfaisante pour un utilisateur qui voudrait ajouter rapidement des articles.

Nous avons ainsi choisi d'utiliser la deuxième solution : l'API **Open Food Facts** qui nous permet de récupérer la catégorie d'un article saisit. Cette API open-source et totalement gratuite permet d'avoir accès à quelques 320 000 produits du monde entier. Un des avantage de cette API est qu'en plus de la catégorie pour un articles, nous avons accès à plein d'autres informations concernant un produit (allergènes, nutrition, …). Cela pourra éventuellement servir dans le futur d'élément de filtre lors de la recherche d'articles.

Bien que nous soyons dépendant d'un service externe, nous avons veillé à ce que l'application puisse fonctionner (du moins pour les listes personnelles) sans connexion internet. Ainsi, lors d'un ajout d'un article il se verra positionné dans une catégorie temporaire. Dès qu'une connexion internet est repéré, l'application cherchera automatiquement la catégorie de tous les articles ajouté hors-connexion.

Notre solution actuelle peut encore être améliorée en affinant les catégories retournées à l'utilisateur. Open Food Facts dispose de plusieurs niveaux de catégories pour chaque article. On peut imaginer une amélioration pour récupérer la catégorie la plus pertinente à renvoyer à l'utilisateur en sélectionnant parmi un ensemble de catégorie qui ne sont ni trop générales ni trop spécifiques.

Déterminer les prix des articles

Le dernier gros problème rencontré fut la manière de connaître le prix (moyen ou réel) des articles. Nous avions trouvé un peu avant le début du projet une API qui correspondait parfaitement à nos attentes : MasterCourses. Cependant, après les première utilisation, il s'est avéré que son utilisation restait instable (2 fois sur 3 les requête renvoyait un tableau vide).
Nous avions également pensé à parser les différents site de grandes surfaces, mais par manque de temps et pour les même raison que la récupération des catégorie présenté ci-dessus, nous avons évincé cette option.
Ainsi, nous avons choisis de rendre l'utilisateur au coeur du processus de détermination de prix en lui permettant de prendre en photo ses tickets de caisse et ainsi de nous envoyer les prix des différents articles.

Extraire les prix des tickets de caisse

La première solution que nous avons trouvé est une librairie utilisable sous Ionic : **Ocrad JS**. Cependant l'énorme désavantage était le temps de calcul. Il était beaucoup trop long, ce qui se comprends car le calcul s'effectuait sur un périphérique mobile. Nous avons laissé tomber cette option en saisissant bien que les calculs de reconnaissance devront se faire sur le serveur Web et non sur le Front-End afin d'alléger le travail de l'application.
Nous avons cherché une librairie en Java pour récupérer le texte contenu dans une photo. Nous avons trouvé **Asprise OCR** qui est une librairie gratuite et qui permet de récupérer un résultat acceptable en peu

de temps. Nous l'avons utilisé pour travailler sur l'extraction des produits et des prix et pour les enregistrer dans notre base de données.
Néanmoins, le résultat fourni étant peu précis, nous sommes conscients que la reconnaissance de texte peut être améliorée par l'utilisation d'outils plus performants. L'avantage est que si nous trouvons dans le futur un outil plus performant que **Asprise OCR**, il sera très facile de le changer car ce n'est qu'une dépendance Maven.

Nous sommes conscients que notre solution conserve sa faiblesse de dépendance envers nos utilisateurs. Cependant cela nous permet d'introduire un système intéressant que nous allons voir tout de suite.

Pousser l'utilisateur à prendre des photos

Un challenge rencontré très vite après l'instauration du système de photos était la récompense utilisateur. Que gagnait un utilisateur à prendre son ticket de caisse en photo ?
La première chose à laquelle nous avons pensé, et qui est pour nous la bonne car elle introduit un aspect communautaire intéressant, est un système de **gamification** (ludification en français). Ce système récompensera par un titre/badge un utilisateur qui prendra ses tickets de caisse en photo. Le système actuel pousse à prendre des photos des tickets de caisse pour conserver son badge mois après mois.
De plus, nous avons étendu ce système de gamification. Les badges ne se résument pas qu'à la prise de photo, mais aussi à l'achat d'articles d'une certaine catégorie, à la création de liste partagée ou au partage de recette. Avec ce système, nous voulons pousser au développement de la communauté autour de l'utilisation de ShopIt.

Organisation

Organisation au sein du groupe

Nous jugeons que chaque personne du groupe s'est autant impliqué dans le projet que les autres. Nous attribuons donc 100 points à chaque personne :

Kevin DUGLUE 100 points
Charly LAFON : 100 points
Antoine LUPIAC : 100 points
Thomas MONZEIN : 100 points

Organisation technique

Exploitation du versioning

Lors de l'entretien technique, nous avons clairement vu que nous ne profitions pas au maximum des fonctionnalités de Git pour l'avancé de notre projet et que nous n'utilisions pas assez d'outils pour assurer la bonne avancé de notre projet. En effet, nous avons pas utilisé de tags afin d'expliciter les versions de fin de sprint. Nous n'y avions pas pensé mais c'est vrai qu'avec le recul cela nous aurait aidé à revenir à une version sûre en cas de problème. De même pour les branches, nous ne les avons pas utilisé. Cependant nous n'en avions pas ressenti le besoin car chaque membre de l'équipe travaillait sur des parties bien distinctes en terme de fichiers, il n'y avait alors que très peu de merge.

Exploitation des tests

En ce qui concerne les tests, ils ont essentiellement été effectué côté serveur avec **JUnit**. C'est ici que le code est le plus critique et que les tests apporte une grande valeurs. En effet, c'est côté serveur que la plupart des opérations de calculs (répartition d'une liste entre les participants, calculs des articles récurrents, …) sont implémenté.
Le front-end étant principalement de l'affichage, nous n'avons pas effectué énormément de tests. Néanmoins, certaine partie du front-end sont testé, essentiellement celles qui effectue des opérations sur des objets importants (ajout/suppression d'un article, .classement des articles par catégorie, …). Cependant, nous sommes conscient que ca ne reste pas suffisant et que c'est également une amélioration à faire pour la partie Ionic. Une des pistes à entreprendre est celle de la mise en place de test E2E (End-to-End), plus communément appelé tests fonctionnels. Nous avions déjà essayé de les mettre en place durant le projet (à l'aide de **Protractor** sous Angular 2), mais c'est un outil assez dur à prendre en main et nous aurions passé beaucoup trop de temps à tester nos vues.

En revanche, nous avons quand même mis en place un outil d'intégration continue pour le front-End. En effet, nous avons utilisé **NeverCode** qui propose une solution rapide et simple à mettre en place. A l'aide de hooks Git, on peut lancer build, lancer les tests de notre application et également la push directement sur les Stores (sous condition d'avoir des comptes développeur payant sur les différentes plateformes).

Partage dans l'équipe

Au niveau du partage dans l'équipe, nous nous sommes dès le début séparé de manière à ce qu'on soit chacun le plus indépendant dans notre travail. Ainsi, deux personnes se sont occupé du back-end et deux du front-end. De plus, même à l'intérieur des deux groupes, les personnes travaillaient sur des choses bien distinctes (mais pour autant ayant un lien). Par exemple, pour le back-end une personne était plutôt orienté sur la communication avec l'application tandis que l'autre personne travaillait essentiellement sur la base de données et le stockage.

Après quelques conseils que l'on a eu pendant l'entretien technique, nous avons mis un place un README complet à la racine du projet afin de principalement spécifier les modalités d'installation/lancement de notre application. Cela sert à donner des indications précises, notamment dans le cadre d'une nouvelle personne qui intégrerait l'équipe.

Conclusion

Pour conclure, le résultat obtenu à la fin du projet plein était celui qu'on s'était fixé. En effet, au bout de ces 3 semaines à temps plein, nous somme parvenus à mettre en oeuvre les fonctionnalités les plus importantes.

En effet, nous proposons à ce stade un certain nombre de fonctionnalité. Evidemment la création de liste est au coeur des fonctionnalité, avec la création de liste personnelle (avec possibilité de mode hors-ligne), de liste partagée (nécessitant un compte). Un ensemble d'action est disponible sur les article d'une liste comme indiquer qu'on a pris/trouvé un article, qu'on veut réserver ce produit (pour les listes partagées) ou encore qu'on a pas trouvé un article. Ce qui nous démarque de nos concurrent c'est la notion de temps réel qui assure, comme un Google Docs, que chaque participant voient en temps réels les modification dans la liste. Nous proposons également lors de la création d'une liste des articles récurrents de l'utilisateur en se basant sur l'ensemble de ses listes. Un outil de création/recherche de recette est également présent afin de pouvoir ajouter rapidement tous les ingrédients d'une recette à une liste ou encore rechercher une recette à faire en fonction d'un ou plusieurs ingrédients. Une autre innovation, qui certes doit être amélioré pour les prochains sprint, concerne l'estimation des prix pour chaque articles. Enfin, afin de rendre une meilleur expérience à l'utilisateur et l'éviter à avoir une quantité de liste importante sur la page d'accueil, nous avons mise en place un système d'archivage de liste, afin de pouvoir garder des anciennes listes sans pour autant polluer la page principale de l'application.

Un autre résultat, autre que purement fonctionnel, concerne notre apprentissage. En effet, durant ses 3 semaines de développement, nous avons acquis énormément d'expérience que ce soit par des erreurs et prises de conscience notamment lors de l'entretien technique, que par la découverte de nouveau outils et technologie comme le développement d'applications hybrides.

Nous avons quelques pistes pour améliorer les fonctionnalités déjà présente de notre application mais également pour en rajouter d'autres.
Concernant les améliorations des fonctionnalités déjà présente, la principale concerne la détermination du prix et le système de prise de photo du ticket de caisse. En effet c'était une première ébauche pour mettre en place l'architecture et ainsi pouvoir dans le futur changer facilement pour une librairie de reconnaissance de caractère plus performantes que celle actuelle. Une autre amélioration concerne l'estimation des prix. A l'heure actuelle, on se base sur les photos de tickets de caisse. Mais on pourrait rajouter de l'informations en plus des tickets de caisses, comme par exemple sa géolocalisation, afin d'avoir une estimation du prix plus juste qui dépendrait des régions.
Concernant les nouveautés à apporter au projet, nous avons pensé à créer une messagerie interne (comme Messenger) propre à chaque liste afin d'avoir un endroit de discussion commun à tous les participants d'une liste. Nous voudrons également proposé les différentes promotions des magasins afin par exemple de notifier un utilisateur qu'une promotion est disponible sur un de ses articles récurrent. Mais cette fonctionnalité implique un partenariat avec des enseignes de grande distribution.