SNAP

Social Network Aggregator Platform

DEHEEGER Pierrick
LAJOUMARD DE BELLABRE Charles
MIGUEL Kim-Adeline
YON Romain

Rapport final de projet

NF28 – P12

utc
Université de Technologie
Compiègne

Sommaire

Introduction

SNAP (Social Network Aggregator Platform) est un projet réalisé dans le cadre de l'UV NF28 à l'Université de Technologie de Compiègne.

L'objectif premier de notre projet est d'étudier les moyens de personnalisation de l'expérience utilisateur offerts par les dernières technologies Web. Pour cela, nous proposons de décliner un site web applicatif sous la forme de plusieurs interfaces, adaptées à différents types de terminaux. En particulier, nous proposons des interfaces dédiées pour les ordinateurs (portables ou fixes, regroupés sous la nomination 'desktop' dans ce rapport), les *smartphones* et les tablettes. Le cas des TV connectées a également été étudié, mais l'implémentation a été abandonnée suite à des problèmes techniques, principalement par manque de temps.

Remarquons d'ailleurs que notre site offre une fonctionnalité de détection du périphérique utilisé par l'utilisateur. Ainsi le type de terminal est automatiquement détecté au chargement de l'application, mais l'utilisateur peut aussi le changer manuellement, ce qui sera mémorisé via un cookie pour ses prochains passages.

Notre site permet à l'utilisateur de faire l'agrégation des fils d'actualités de ses différents réseaux sociaux. En plus de l'agrégation des messages, SNAP permet aussi à l'utilisateur de poster des messages sur plusieurs réseaux sociaux simultanément. Concrètement, nous avons pris en charge *Facebook* et *twitter*.

Tout au long de ce projet, une attention particulière a été portée à l'étude des interactions de l'utilisateur dans notre contexte. En effet, ces dernières tiennent une place centrale dans notre projet, puisque nous cherchons à développer des interfaces le plus adaptées possible aux terminaux utilisés, et donc aux interactions propres à ces terminaux. En particulier, notre application est fortement susceptible d'être utilisée en situation de mobilité, notamment pour la tablette, et surtout le *smartphone*. Aussi, comme nous le verrons, le fait que nous travaillons avec les utilisateurs de réseaux sociaux engendre des comportements spécifiques. Cette étude est synthétisée dans la dernière partie du rapport.

Vous pouvez expérimenter SNAP en ligne à l'URL:
http://snapnf28.herokuapp.com/snapapp/index.html
L'intégralité du code est disponible sur le *GitHub* de SNAP à l'URL:
https://github.com/SNAP-NF28/SNAP
Pour finir, la vitrine NF28 de SNAP est disponible à l'URL:
http://snapnf28.herokuapp.com/website/index.html

I. Organisation

La réalisation d'un tel projet repose non seulement sur la réalisation technique, mais aussi sur l'aspect organisationnel. En effet, les différentes contraintes posées, notamment temporelles, impliquent une distribution des tâches permettant au projet d'avancer de manière homogène. Cette organisation se reflète à travers deux points : une constante organisationnelle, permettant de créer une synergie de groupe, et un point plutôt technique, portant sur les outils utilisés afin de mener à bien notre projet.

1. Organisation projet

L'organisation du projet se base sur un planning prévisionnel des tâches à réaliser. Ce planning définissait en premier lieu les lignes directrices du projet, puis au fur et à mesure de l'avancement, l'étendue des fonctionnalités à implémenter devenait plus précise. Le planning réel se juxtapose donc au planning prévisionnel, ce qui permet de cerner la vitesse de réalisation des différentes fonctionnalités.

a. Réunions hebdomadaires

Par ailleurs, une communication régulière entre les membres du groupe s'avère aussi importante. Au vu du temps et des connaissances disponibles, il est crucial de pouvoir échanger clairement et rapidement, afin de ne pas rester bloquer sur un problème. Par ailleurs, la réflexion collective permet d'aborder les problématiques avec des points de vue différents, ce qui permet d'améliorer le produit.

La séance de TD hebdomadaire est donc consacrée à ce *brainstorming*, qui permet à tout le groupe de se retrouver et de faire un point sur les avancées et soucis de chacun durant la semaine passée. Chacun venant d'exposer son travail, la synthèse à effectuer au chargé de TD est donc étayée directement par les membres du groupe. De plus, cette séance sert aussi de jalon aux étapes indiquées dans le planning prévisionnel.

Cette conjonction entre prévision des tâches et réunion hebdomadaire permet de comparer les objectifs à réaliser à ceux effectivement implémentés. Cela permet de recentrer les priorités en cas de problème.

Par exemple, les technologies utilisées dans ce projet se sont avérées être peu documentées, chacune ayant son propre mode de fonctionnement. Il s'en est donc ensuit quelques difficultés de conception, d'où un ralentissement par rapport au planning prévisionnel. Il a donc été décidé de ne pas réaliser le prototype de la télévision, et de

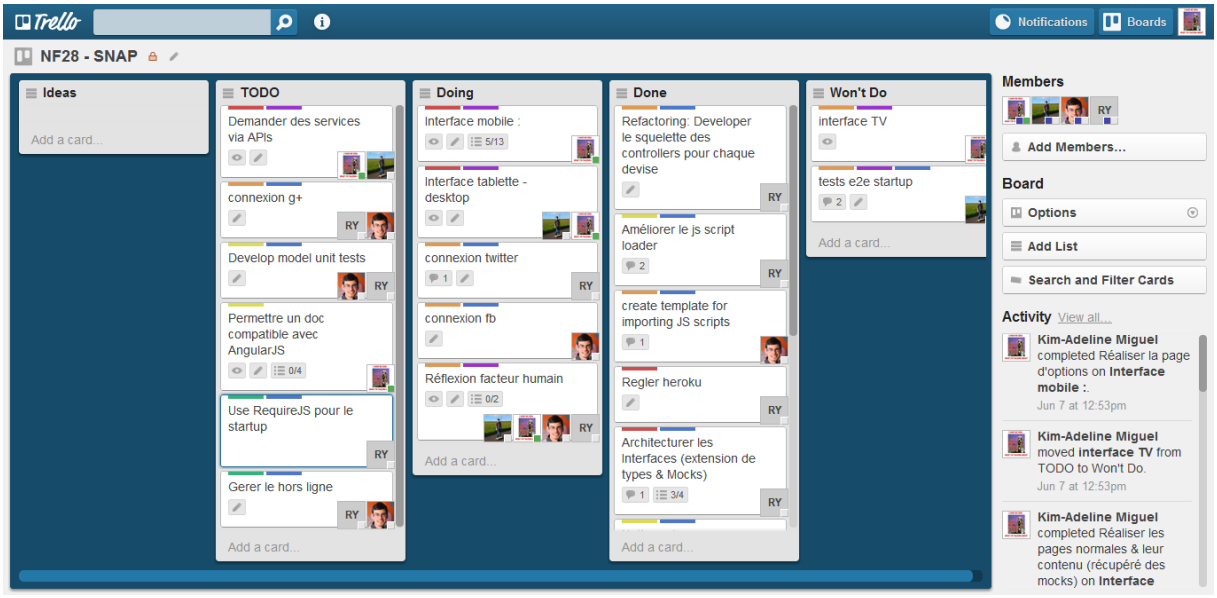
recentrer le projet sur la réalisation des prototypes pour les 3 autres types de terminaux (ordinateur, tablette, *smartphone*), en proposant des interfaces plus abouties.

La méthodologie adoptée pour ce projet se rapproche donc de la méthode *Scrum*, avec un découpage en *sprints* hebdomadaires, une organisation autonome, et un retour en TD sur le travail effectué la semaine passée. L'utilisation de Trello, outil de suivi de progression, permet aussi de visualiser l'avancement du projet, non plus d'après un ordre uniquement chronologique, mais suivant l'état de réalisation des fonctionnalités : concept, réalisation, fini, pas fait, tests et débogage.

b. Outil de suivi de progression: Trello

Notre board pour le projet est disponible à l'URL https://trello.com/board/nf28-snap/

Trello est un site web (trello.com) permettant à ses membres de créer et rejoindre des « *boards* » s'apparentant aux tableaux de bord récapitulatifs des méthodes *Agiles*/*Scrum*.



Board Trello pour le projet SNAP NF28

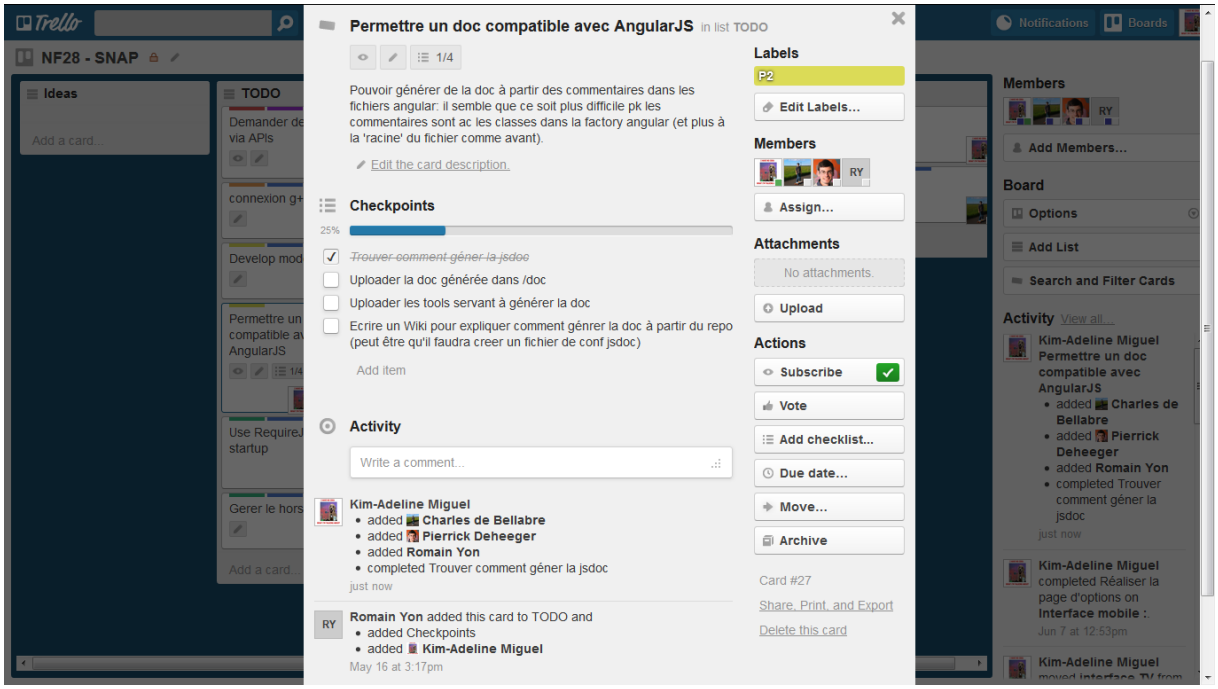
Cette interface offre donc les mêmes fonctionnalités qu'un tableau *Scrum* : ajout de tâches sous forme de « *Cards* », assignation de membres et de catégories, déplacement des tâches suivant leur état d'avancement, archivage des tâches réalisées. Par ailleurs, la transposition de cet outil tangible en logiciel permet d'ajouter des fonctionnalités pour chaque tâche.

On peut ainsi ajouter un descriptif pour chaque tâche, une liste des jalons à remplir associée à une barre d'avancement, des fichiers, et laisser des commentaires. De plus, les activités sur les cartes sont archivées par le système, ce qui permet à tous les membres de voir l'historique des activités pour chaque carte. Les activités récentes sont aussi affichées

sur l'interface principale du *board*, et les membres reçoivent des notifications lorsque des cartes auxquelles ils ont souscrit (par défaut celles où ils font partie des membres) sont modifiées.

Certaines fonctionnalités proposées par *Trello* peuvent toutefois être classées comme anecdotiques, tout du moins dans le cadre de ce projet, dans la mesure où toute l'équipe se trouve dans la même région géographique et peut donc se réunir régulièrement.

Par exemple, le système de vote proposé pour chaque carte, ou encore l'assignation d'une date de livraison de la tâche. En effet, les choix effectués l'ont toujours été en concertation avec les autres membres du groupe, le vote est donc inutile. Concernant la date de livraison, la seule échéance non négociable concerne la date de rendu du projet, les autres fonctionnalités ayant seulement des dates prévisionnelles. Par ailleurs, l'utilisation d'un planning prévisionnel synthétisant les tâches à effectuer rend la fonctionnalité de date de livraison caduque.



Fonctionnalités Trello sur les cartes

Conjointement à l'utilisation de ce logiciel de suivi, il est nécessaire de formaliser les concepts, fonctionnalités, problèmes et solutions évoquées lors des séances de brainstorming. L'utilisation d'outils d'édition collaborative tels que *Google Documents* répond à ce besoin.

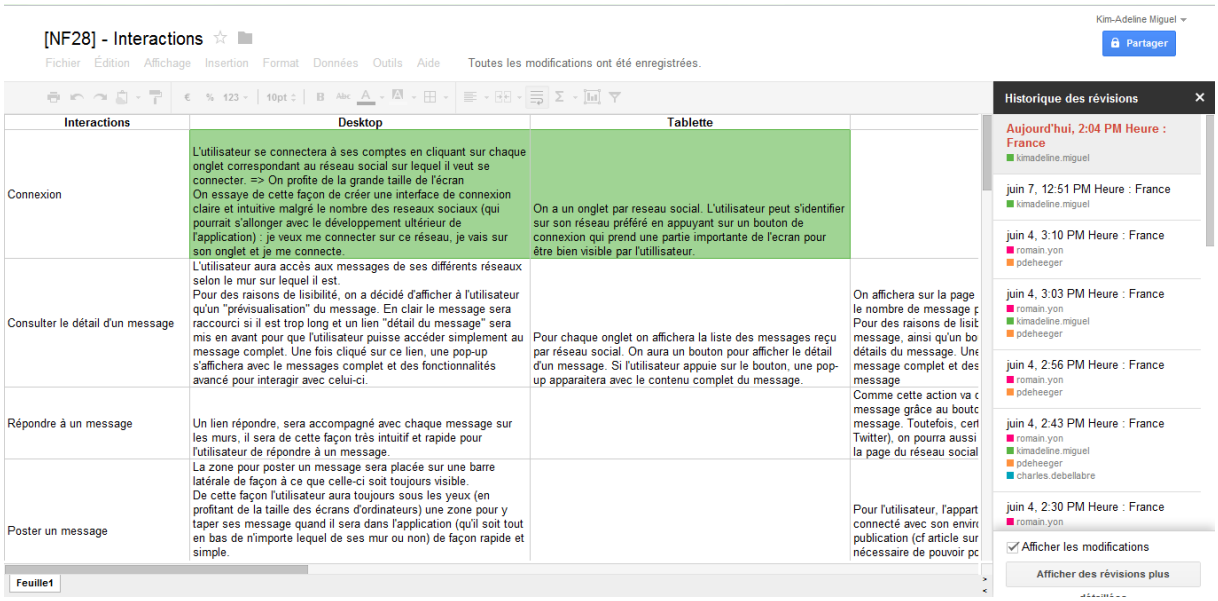
c. Les documents collaboratifs: Google Documents

Google Documents permet de créer des documents de base comme ceux proposés par les suites bureautiques : documents texte, feuilles de calcul et présentations. Ces documents sont stockés sur les serveurs *Google*, et sont accessibles et modifiables par les personnes autorisées par le propriétaire (suivant les permissions du document).

Ces outils permettent de laisser des commentaires sur le document, ce qui permet aux utilisateurs de débattre, et laisser une trace des réflexions engagées, sans modifier le contenu du document.

Un des points critiques du partage de documents est la possibilité que plusieurs utilisateurs se retrouvent à éditer le document simultanément. Ce système d'édition concurrente est intégré dans *Google Docs*, car chaque utilisateur se voit attribuer une couleur pour son curseur, afin que les autres utilisateurs sachent où on se trouve dans le document.

Les utilisateurs peuvent donc modifier les fichiers, mais il est toutefois nécessaire de garder un historique des modifications réalisées, afin qu'un utilisateur ne se sente pas perdu lorsqu'il reprend le document et que sa partie a été modifiée. Il y a donc un système de révisions, qui enregistre ponctuellement les modifications, leur date et les utilisateurs responsables de ces modifications. La fréquence des révisions enregistrées est modulable, avec la possibilité d'afficher des révisions plus détaillées. Il est donc tout à fait possible d'afficher des révisions précédentes, et d'y revenir en cas de problème.



Interactions	Desktop	Tablette	
Connexion	L'utilisateur se connectera à ses comptes en cliquant sur chaque onglet correspondant au réseau social sur lequel il veut se connecter. => On profite de la grande taille de l'écran On essaye de cette façon de créer une interface de connexion claire et intuitive malgré le nombre des reseaux sociaux (qui pourrait s'allonger avec le développement ultérieur de l'application) : je veux me connecter sur ce réseau, je vais sur son onglet et je me connecte.	On a un onglet par reseau social. L'utilisateur peut s'identifier sur son réseau préféré en appuyant sur un bouton de connexion qui prend une partie importante de l'ecran pour être bien visible par l'utillisateur.	
Consulter le détail d'un message	L'utilisateur aura accès aux messages de ses différents réseaux selon le mur sur lequel il est. Pour des raisons de lisibilité, on a décidé d'afficher à l'utilisateur qu'un "prévisualisation" du message. En clair le message sera raccourci si il est trop long et un lien "détail du message" sera mis en avant pour que l'utilisateur puisse accéder simplement au message complet. Une fois cliqué sur ce lien, une pop-up s'affichera avec le messages complet et des fonctionnalités avancé pour interagir avec celui-ci.	Pour chaque onglet on affichera la liste des messages reçu par réseau social. On aura un bouton pour afficher le détail d'un message. Si l'utilisateur appuie sur le bouton, une pop-up apparaitera avec le contenu complet du message.	On affichera sur la page le nombre de message p Pour des raisons de lisib message, ainsi qu'un bo détails du message. Une message complet et des message
Répondre à un message	Un lien répondre, sera accompagné avec chaque message sur les murs, il sera de cette façon très intuitif et rapide pour l'utilisateur de répondre à un message.		Comme cette action va c message grâce au boutc message. Toutefois, cert Twitter), on pourra aussi la page du réseau social
Poster un message	La zone pour poster un message sera placée sur une barre latérale de façon à ce que celle-ci soit toujours visible. De cette façon l'utilisateur aura toujours sous les yeux (en profitant de la taille des écrans d'ordinateurs) une zone pour y taper ses message quand il sera dans l'application (qu'il soit tout en bas de n'importe lequel de ses mur ou non) de façon rapide et simple.		Pour l'utilisateur, l'appart connecté avec son envir publication (cf article sur nécessaire de pouvoir pc

Système de révisions : traçage et affichage des modifications

Enfin, l'interface propose aussi un système de chat sur le côté droit, afin que, si plusieurs utilisateurs sont connectés simultanément et veulent débattre, de ne pas surcharger le document avec des fils de commentaires trop longs.

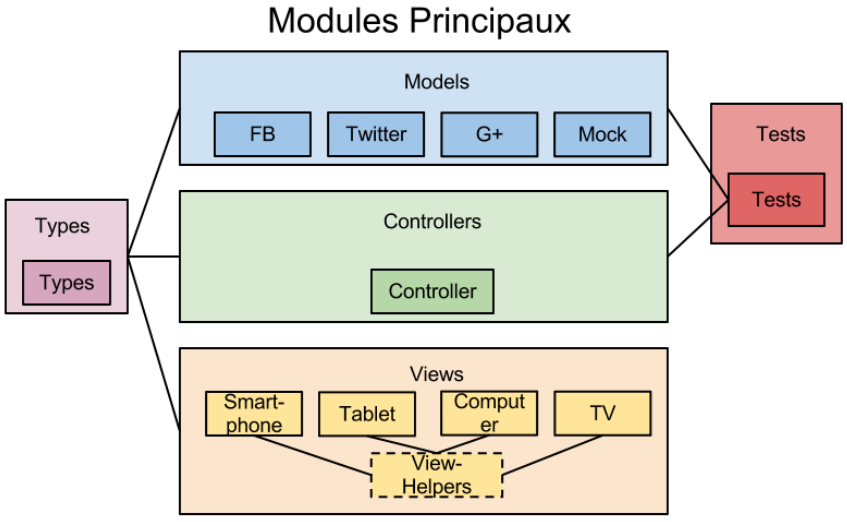
L'ensemble de ces outils participe à un environnement de travail efficace. Toutefois, ce sont seulement des aides permettant de travailler mieux et plus efficacement. L'organisation technique du groupe et la répartition des tâches entre les membres du projet est donc tout aussi cruciale. Il a alors été décidé de catégoriser le travail à réaliser en 2 parties : le *back-end*, pour la connexion avec les APIs des réseaux sociaux, et la réalisation des fonctions et objets qui vont interfacer l'application web SNAP et les APIs ; et le *front-end*, qui va utiliser les fonctions développées par le back-end, et qui va se concentrer sur le message délivré à l'utilisateur à travers l'interface graphique.

2. Organisation technique

a. Deux équipes : Front-end & Back-end

Le groupe a donc été divisé en 2 parties : l'équipe *front-end*, chargée de la réalisation des interfaces graphiques, en prenant en compte les interactions spécifiques à chaque type de périphérique, ainsi que de la définition des méthodes et objets nécessaires ; et le binôme *back-end*, chargé de la mise en place du côté serveur, c'est-à-dire l'hébergement du site, et de la réalisation des méthodes et fonctions demandées par le binôme *front-end*, notamment par la compréhension et la mise en place de la communication entre les APIs natives des réseaux sociaux et le projet SNAP .

L'application a été pensée pour respecter le patron de conception Modèle-Vue-Contrôleur, la séparation des tâches en *front-end*/*back-end* s'est donc faite naturellement.



Architecture initiale du projet

Afin de limiter les dépendances entre les équipes, l'équipe *back-end* a rapidement mis en place des *mockups*, c'est-à-dire des maquettes des objets qui vont être utilisés, mais initialisées avec du contenu statique. Ces maquettes ont permis à l'équipe *front-end* de développer leurs fonctionnalités et tester le rendu visuel du projet, sans dépendre du travail

de l'équipe *back-end* sur les APIs des réseaux sociaux. Au fil de l'avancement de l'équipe *back-end*, les *mockups* ont été progressivement remplacés par des appels aux APIs.

La définition de ces rôles distincts permet une meilleure vue des fonctionnalités à implémenter. En effet, le fait d'être du front-end ou du back-end allège les fonctionnalités à implémenter, car on n'a pas à se soucier du travail de l'autre équipe.

Néanmoins, même si chaque membre du projet s'est vu attribuer un rôle, ce-dernier ne le contraint pas. En effet, les tâches sont flexibles, et si les membres d'une équipe bloquent sur un pont, les membres de l'autre équipe peuvent venir apporter leur point de vue (*brainstorming*).

b. Gestion de code

L'organisation technique du projet passe aussi par la gestion du code. En effet, les deux équipes peuvent développer de front, mais doivent tôt ou tard fusionner leur travail. L'arborescence de l'application a donc été envisagée dès le début, afin de ne pas engendrer de conflits ultérieurs. L'utilisation d'un système de gestion de versions s'est rapidement imposée, et plus particulièrement du logiciel *Git*, en conjonction avec le service web d'hébergement *GitHub*.

c. Un outil de gestion de versions : Git

Il y a deux types de gestion de versions : centralisé (par exemple *Subversion*) et distribué (par exemple *Git*). L'utilisation de *Git* repose plutôt sur un besoin de performances, ainsi que sur une possibilité de décentralisation du travail : chacun peut travailler à son rythme, pour ensuite fusionner les modifications. A contrario, l'utilisation de SVN oblige le projet à avoir un dépôt centralisé et unique.

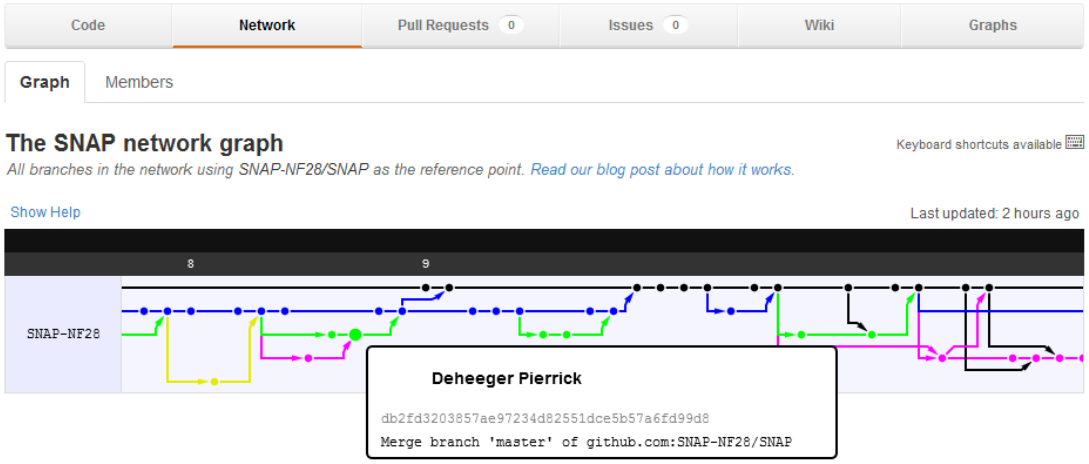
Bien que *Git* soit issu du monde *Linux*, ce logiciel a été porté pour être utilisable Windows et Mac. Nous avons utilisé différentes formes de *Git* : en ligne de commande (*Git Bash*), et avec une interface graphique (*Git GUI*), mais aussi des logiciels tiers, avec des interfaces plus évoluées, comme *TortoiseGit*. Ces logiciels se basent sur *msysgit*, l'implémentation console de Git pour Windows. Le site *GitHub* propose aussi une interface graphique spécialement dédiée aux projets développés sous Windows.

d. Un hébergement du code : GitHub

GitHub est un site web d'hébergement de projets gérés avec *Git*. Ce site offre un dépôt gratuit pour les projets en *open-source*. On peut y héberger son projet, et utiliser *Git* pour cloner le projet en local, et ensuite renvoyer les modifications sur le site.

Ce site permet de consulter l'historique des révisions précédentes, d'afficher qui a fait quelles modifications, et compare à chaque envoi de modifications les anciens fichiers et les nouveaux, le tout avec une esthétique sobre mais soignée. Il propose aussi une structure de wiki, afin que les développeurs du projet puissent écrire leur documentation.

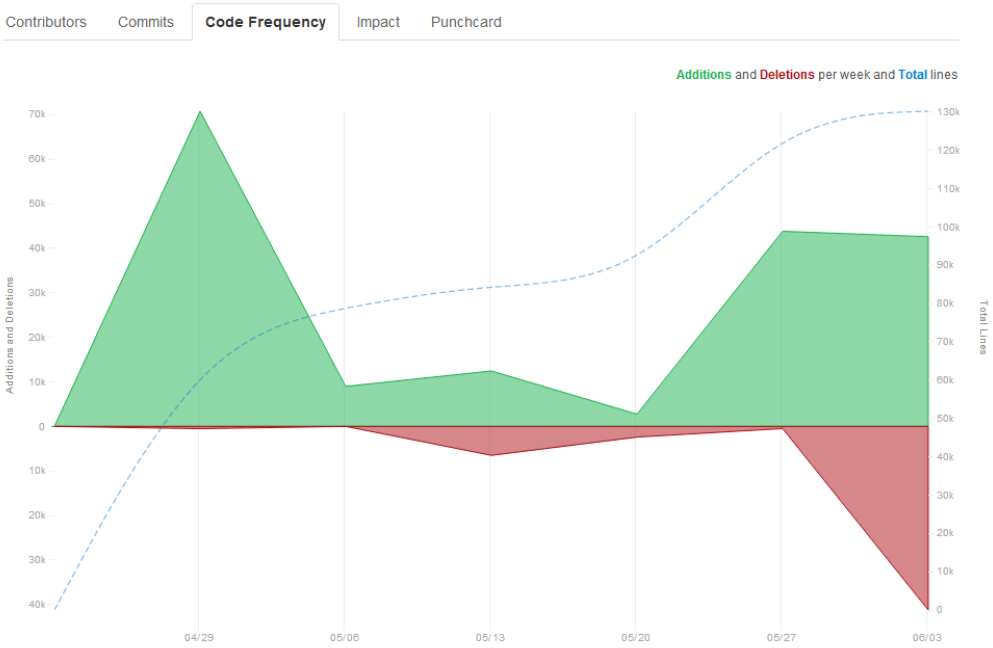
De plus, il offre de nombreuses fonctionnalités statistiques : il possède un onglet « *Network* », qui retrace l'historique des modifications effectuées par les membres sur un projet, et les différentes subdivisions du projet.



Graphe représentant les branchements (*forks*) du projet SNAP

En plus de ce graphe, *GitHub* propose un onglet « *Graphs* » regroupant différents types de graphes mesurant la fréquence, la taille, le nombre et l'impact des contributions suivant les contributeurs.

Ces outils statistiques permettant de suivre l'évolution du projet sont utiles, mais seraient plus significatifs pour des projets de plus grande envergure, excepté l'outil « *Network* », qui permet de retracer l'historique des modifications et les contributeurs.



Graphe synthétisant l'évolution du projet SNAP en termes de lignes de code

e. Un serveur de développement : Heroku

L'hébergement du code sur *GitHub* ne suffit pas à en faire une application exécutable. En effet, il faut voir ce site comme un dépôt, et non pas un serveur capable d'exécuter du code. Il a donc fallu héberger le code du projet SNAP sur un serveur autre qu'en local, sans toutefois devoir passer trop de temps sur l'installation de ce serveur, car notre application est uniquement côté client.

Il a donc été choisi de déployer l'application sur *Heroku*, une plate-forme *cloud* qui gère le déploiement de l'application, et permet donc aux développeurs de se focaliser sur le code de leur application. Cette plate-forme, utilisant Git comme système de gestion de versions, est simple à maîtriser et déployer, comme il sera expliqué dans la partie sur les technologies utilisées.

L'hébergement de notre application sur un serveur tiers permet de s'affranchir des contraintes des tests locaux. En effet, le paramétrage de certaines APIs rentrait en conflit avec les tests en local. Par ailleurs, cet hébergement permet de tester l'accès au site SNAP avec d'autres périphériques, et ainsi vérifier quasi-immédiatement le fonctionnement du site.

II. Aspects fonctionnels et techniques

1. Pourquoi SNAP ?

a. Enjeux

La multiplication des terminaux connectés engendre de nouveaux types de besoins. En effet, un nombre croissent d'utilisateurs souhaitent accéder à leurs services habituels via des périphériques peu conventionnels, qui ne comportent pas encore d'applications adaptées pour accéder à ce contenu. On peut citer par exemple l'exemple des réfrigérateurs connectés.

L'objectif de notre projet est d'expérimenter une approche générique pour adapter l'interface en fonction de l'appareil utilisé, et donc du contexte d'utilisation. Il peut s'agir d'un ordinateur où l'utilisateur est à priori immobile, d'un *smartphone*, ou bien d'une tablette tactile, où l'utilisateur peut être amené à se déplacer. Il s'agit donc d'utiliser les dernières technologies en matière de développement web pour présenter l'interface la mieux adaptée pour l'utilisateur.

b. Services rendus

Aujourd'hui, l'utilisation de réseaux sociaux est devenue commune pour une part importante de la population. L'existence de plus de 900 millions de comptes *Facebook* et de 175 millions de comptes *twitter* en est une preuve. Nous avons aussi remarqué que les utilisateurs ont tendance à être présents sur un nombre croissant de réseaux sociaux. Suivre le fil de tous ces réseaux sociaux peut vite devenir délicat.

C'est pourquoi nous avons décidé de créer un agrégateur de réseaux sociaux. SNAP rend un service adapté aux utilisateurs de ces réseaux, c'est-à-dire la majorité des utilisateurs du web. En effet, le but de l'application est de fusionner différents réseaux sociaux sur un même site. Les utilisateurs pourront notamment consulter leur messages issus de leur comptes *Facebook* et *twitter*. Grâce à l'application ils pourront également publier des messages sur les deux réseaux sociaux en un seul clic. Cela simplifie la vie des utilisateurs qui veulent communiquer des informations sur leurs plateformes sociales préférées.

Remarquons que l'ajout potentiel d'un autre réseau social peut s'effectuer relativement facilement car l'application est générique. Nous avons ainsi défini une entité abstraite, comportant des méthodes génériques et communes à tous réseaux sociaux (envoi de message, récupération des derniers messages, récupération du profil, etc.). Nous avons aussi tiré parti de la composante dynamique de JavaScript pour définir des objets flexibles,

capables de s'adapter aux spécificités des réseaux sociaux. De plus amples explications seront données dans la partie technique.

c. Utilisateurs visés

La problématique d'accessibilité est également un enjeu important de notre application. L'objectif est donc de permettre à un maximum d'utilisateurs d'accéder à un même service par l'intermédiaire du web. Les utilisateurs ciblés sont donc tous les possesseurs de périphériques connectés (que ce soient des ordinateurs, *smartphones*, tablettes, TV ou tout autre objet connecté).

d. Adaptation du site en fonction de l'appareil utilisé

Il y a peu de temps encore, l'ordinateur était de loin l'appareil dominant pour l'accès au web. Maintenant, de nombreux terminaux connectés sont apparus et le *smartphone* est en train de dépasser l'ordinateur. L'enjeu principal de notre application est d'offrir aux utilisateurs une expérience optimale, quel que soit le périphérique utilisé.

Notre application étant destinée à un public très large, le service d'adaptation devient incontournable dû à la démocratisation de différents types d'appareils. Nous avons apporté un soin particulier à l'ergonomie de l'application sur l'ordinateur puisqu'il s'agit du terminal le plus utilisé pour consulter un site web.

Au niveau des périphériques tactiles, on voit aujourd'hui l'expansion des *smartphones* dans la vie quotidienne. Ce type de terminal est aujourd'hui un outil incontournable pour la navigation sur le web. Les tablettes tactiles se développent également au sein du grand public et offrent une expérience de navigation sensiblement différente d'un *smartphone* ou d'un ordinateur.

Notre application permet un ajout relativement aisé de nouveaux types de périphériques, comme par exemple la TV connectée ou d'autres terminaux à venir.

e. Détail des fonctionnalités

Tableau des fonctionnalités proposées par périphérique et comment y accéder
RS = Réseau social

Périphérique / Fonctionnalités	Ordinateur	Smartphone	Tablette
Connexion aux réseaux sociaux	Trois onglets différents	Pages différentes par RS	Trois onglets
Consulter les derniers messages agrégés	onglet All	Page d'accueil	onglet All
Consulter les messages sur un réseau social spécifique	onglet RS spécifique	page RS spécifique	onglet RS spécifique
Consulter le détail d'un message	lien *détail*	bouton *Plus...*	Le message est cliquable
Faire une recherche simple dans le contenu présent	Barre de recherche dans le menu de gauche	Page de recherche	Barre de recherche menu de gauche
Consulter son profil	Icône profil en haut à droite	Cliquer sur l'icône du réseau social	Bouton profil, dans le menu de gauche
Consulter un profil en particulier	Dans la liste des messages, le nom des auteurs sont cliquables	Idem	Idem
Envoyer un message	Formulaire en haut à gauche. Cases à cocher pour chaque RS (par défaut toutes sont cochées), bouton *Envoyer*	Page d'envoi de message, formulaire avec trois boutons activés ou désactivés, bouton *Envoyer*	Pop-up d'envoi de message : formulaire avec des cases à cochés, bouton *Post*
Répondre à un message	Dans le détail du message, lien *répondre*	Dans le détail du message, bouton *répondre*	Dans la liste, à droite de chaque message bouton '*+*'
Se déconnecter	Bouton profil en haut à gauche	Dans les options	Dans les options

2. Aspects techniques

Cette section a pour but de justifier la majorité des choix techniques que nous avons faits pour ce projet. Pour un aperçu rapide des principales technologies utilisées et des raisons pour lesquelles nous avons choisi de les utiliser, nous vous invitons à observer le tableau des technologies par terminal fourni en annexe.

Notons aussi que cette section contient de nombreux termes techniques. Nous avons donc réalisé un glossaire en annexe, pour permettre d'éclaircir d'éventuelles incompréhensions.

a. Serveur

Étant donnés les objectifs de notre projet (avoir un site qui s'adapte fortement au client) nous avons fait le choix d'avoir une partie serveur réduite, et favoriser l'exécution de code côté client. Dans cette partie, nous allons nous pencher sur nos choix techniques en termes de technologies serveur.

Pourquoi un serveur distant ?

Nous aurions pu nous contenter de tester notre projet sur un serveur local (ex: http://localhost:8000/...), mais nous avons fait le choix de déployer notre code sur une plate-forme extérieure, ouverte à l'extérieur. Ce choix comporte plusieurs motivations:

Le site est développé pour être rendu public. Il ne pourra donc pas demeurer pour toujours localement sur nos machines.

Le fait de tester du code localement ne simule pas une situation réelle via l'Internet. Il est donc raisonnable de "pusher" de temps à autre notre code sur un serveur distant dans le but de vérifier que le rendu est bien celui désiré.

La réactivité de la navigation est une dimension primordiale, dans la mesure où nous développons pour des périphériques mobiles. En ce qui concerne la réactivité, le fait que le site soit hébergé localement est un biais considérable, puisque les fichiers sont servis de manière quasi instantanée. On comprend donc bien que la possibilité de tester notre application dans des conditions réelles est très appréciable.

Heroku

L'offre concernant le déploiement de serveurs distant est considérable. Nous avons choisi d'utiliser la plate-forme de *Cloud Computing* Heroku (Site: http://www.heroku.com/).

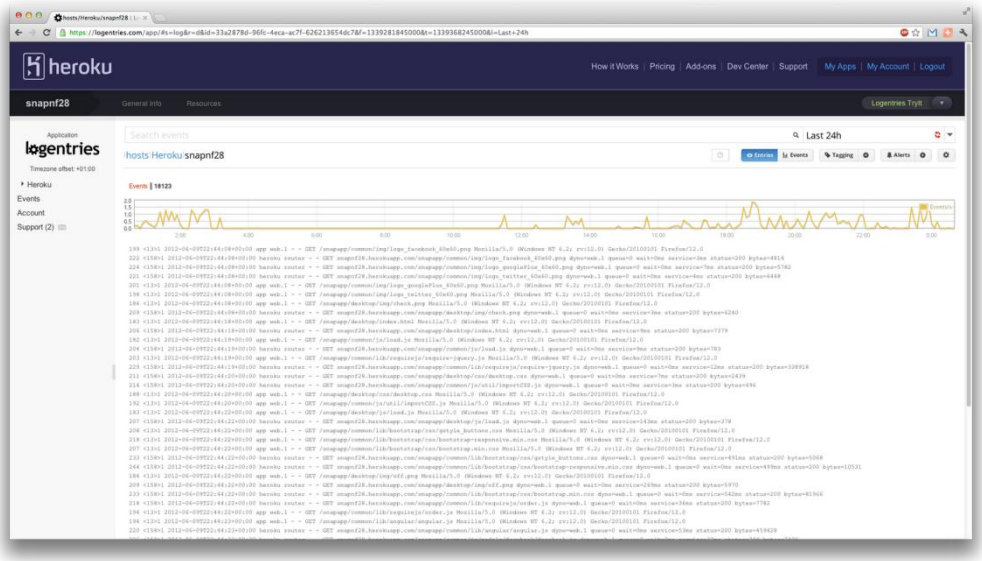
Cette plate-forme propose de nombreux avantages par rapport à notre projet. En voici quelques-uns :

- *Heroku* intègre de manière native les déploiements avec *Git*. C'est donc un allié parfait de *GitHub*. Nous avons ainsi pu avoir un dépôt *Heroku* cloné à partir de notre dépôt principal sur *GitHub*. Lors du développement, il nous a aussi été possible de tester les derniers ajouts de code directement sur *Heroku* puis, une fois les changements validés, de synchroniser les changements avec le dépôt *GitHub*. Cette approche s'est révélée très efficace lors du déploiement, en particulier grâce à *Git*, qui permet une bonne gestion du développement concurrent (il nous est arrivé de développer le projet à 4 simultanément en ne subissant que conflits très réduits).

- *Heroku* est une sur-couche du réseau de *Cloud Computing* proposé par *Amazon* (Le "*Amazon Web Service*" : http://aws.amazon.com/fr/). *Heroku* propose ainsi des performances et une stabilité de tout premier ordre.

- Le premier serveur est gratuit. De plus, l'enregistrement au site est extrêmement rapide et ne nécessite ni carte bleue, ni démarche de validation d'aucune sorte. Cette plate-forme permet ainsi d'expérimenter et de développer de nouveaux services web très facilement. A partir d'un compte gratuit, il est ensuite très simple de passer sur un compte payant, qui offre la possibilité d'augmenter le nombre de serveurs de manière quasi illimitée

- *Heroku* offre de nombreux plugins gratuits (cf. https://addons.heroku.com/) pour ajouter de nouvelles fonctionnalités à notre site. Nous avons ainsi pu bénéficier, de manière intégrée et automatisée, de la fonction de *monitoring*.



Le lancement d'une nouvelle application entraîne la création systématique d'un domaine associé, sans que nous ayons à fournir le nôtre. Nous avons ainsi pu bénéficier du nom de domaine "snapnf28.herokuapp.com" gratuitement et de manière immédiate. Remarquons qu'il est bien sûr très facile de remplacer ce nom de domaine par un autre si on en possède un.

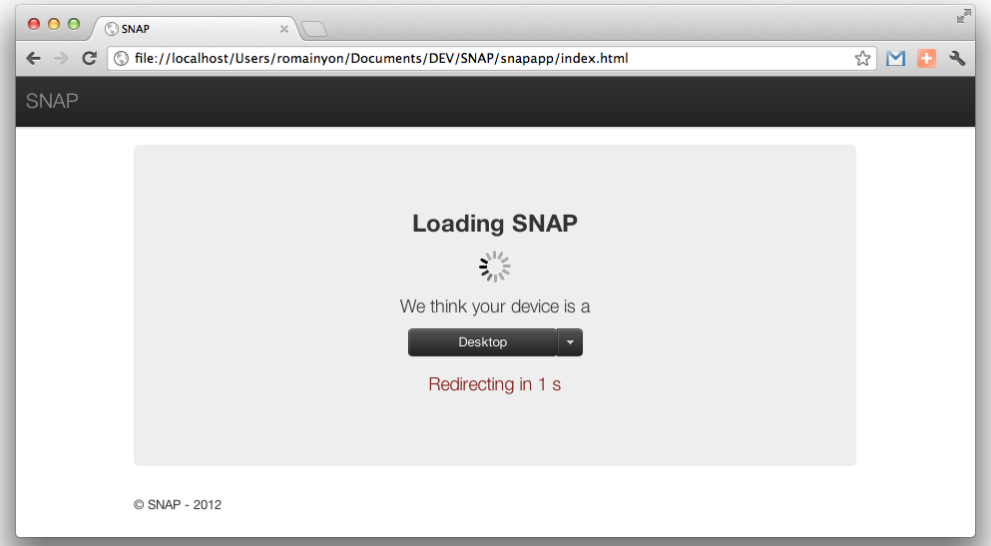
Node.js

En ce qui concerne la technologie côté serveur, nous avons choisi d'utiliser *node.js* (http://nodejs.org/). Le code serveur est disponible sur le *GitHub* de SNAP, à l'url : https://github.com/SNAP-NF28/SNAP/blob/master/scripts/web-server.js.

Comme son nom l'indique, *node.js* est un *framework* basé sur JavaScript. Le JavaScript a la réputation d'être un langage plutôt lent et sa rencontre côté serveur a de quoi surprendre. Il y a en effet un contrecoup de performance, mais ce dernier est limité par le fait que *node.js* repose sur V8, le moteur JavaScript développé par *Google* pour *Chrome*. Ce dernier est un moteur très optimisé, ce qui permet à *node.js* d'avoir des performances honorables (tout en restant en dessous de nombreuses technologies serveur).

En fait l'avantage principal de *node.js* réside dans sa simplicité d'utilisation, ce qui en fait une très bonne technologie pour le développement d'application. Le code est très vite écrit, grâce à un ensemble fonctionnalités intégrées. Il peut être ensuite très simple de passer, pour la production, à une technologie plus éprouvée comme php, ASP, python (via Django) ou encore ruby (via Rails).

Dans le cadre de notre projet SNAP, les besoins en termes de serveurs sont extrêmement limités. En fait nous aurions pu nous contenter d'utiliser les fichiers de manière statique sans utiliser du tout de technologie serveur. Cette approche peut très bien fonctionner, dans la mesure où la seule fonctionnalité de notre code serveur est de délivrer du contenu statique (fichiers).



En fait nous avons choisi d'utiliser du code serveur principalement pour des fonctionnalités de débogage, et pour pouvoir disposer de traces d'accès et de déploiement. Il nous a ainsi été possible d'enregistrer les logs d'accès et d'y afficher différentes granularités d'information en fonction de besoins. C'est d'ailleurs ces mêmes logs qui sont visibles dans la capture d'écran de logentries.

Ajoutons pour finir que le déploiement d'applications en *node.js* est intégré nativement par *Heroku*. Nous avons ainsi pu déployer très facilement notre code, avec seulement deux courts fichiers de configuration nécessaires :

- le Procfile (https://github.com/SNAP-NF28/SNAP/blob/master/Procfile
- le package.json (https://github.com/SNAP-NF28/SNAP/blob/master/package.json)

b. Client

Étant donné que la logique de notre application n'est pas du tout présente côté serveur, elle se trouve entièrement du côté client. Le code s'exécutant du côté client, nous avons légitimement choisi JavaScript comme langage de base. Le JavaScript étant un langage simple et dénué de fonctionnalités avancées (comme l'héritage, l'encapsulation, l'utilisation de classes templates, etc.), il aurait été imprudent d'envisager un tel projet sans s'appuyer sur des *frameworks* existants pour nous aider à structurer notre code. Cette partie sera structurée autour de trois axes. Tout d'abord *angular.js*, la librairie principale que nous avons utilisée pour organiser notre code (L'organisation du code client est visible sur *GitHub*: https://github.com/SNAP-NF28/SNAP/tree/master/snapapp). Ensuite nous nous intéresserons aux librairies que nous avons utilisées pour améliorer l'interface utilisateur. Pour finir, nous parlerons des librairies "utilitaires" auxquelles nous avons fait appel.

Angular JS

Qu'est-ce qu'angular.js?

Dans le but de nous aider à structurer notre projet, nous avons choisi d'utiliser angular.js (cf. http://angularjs.org/), un *framework* JavaScript très complet développé par une équipe de *Google* et mis à disposition sous forme de code *open-source* (*GitHub* de *Angular.js*: https://github.com/angular/angular.js).

Nous avons fait ce choix, séduits par l'argumentation des créateurs du *framework* : Combler les lacunes du HTML, originellement créé pour gérer du contenu exclusivement statique, en s'affranchissant de la gestion des appels asynchrones (via les fonctions de "*callback*"). De plus, *Angular.js* fournit un ensemble intégré de mécanismes pour faciliter la gestion du code via le *design pattern* MVC, et permet de produire du code particulièrement bien testable.

Principe de fonctionnement

Angular.js pourrait faire l'œuvre d'un ouvrage dédié, nous ne citerons donc ici que quelques principes simples.

Scope

Un problème majeur souvent associé à JavaScript est le "*scope*" (espace des noms) partagé. C'est à dire que les variables et fonctions déclarées simplement (sans les encapsuler dans des fonctions anonymes) sont accessibles en tout endroit de l'application (une fois le fichier contenant le code chargé). C'est un lourd problème pour les grosses applications pour plusieurs raisons:

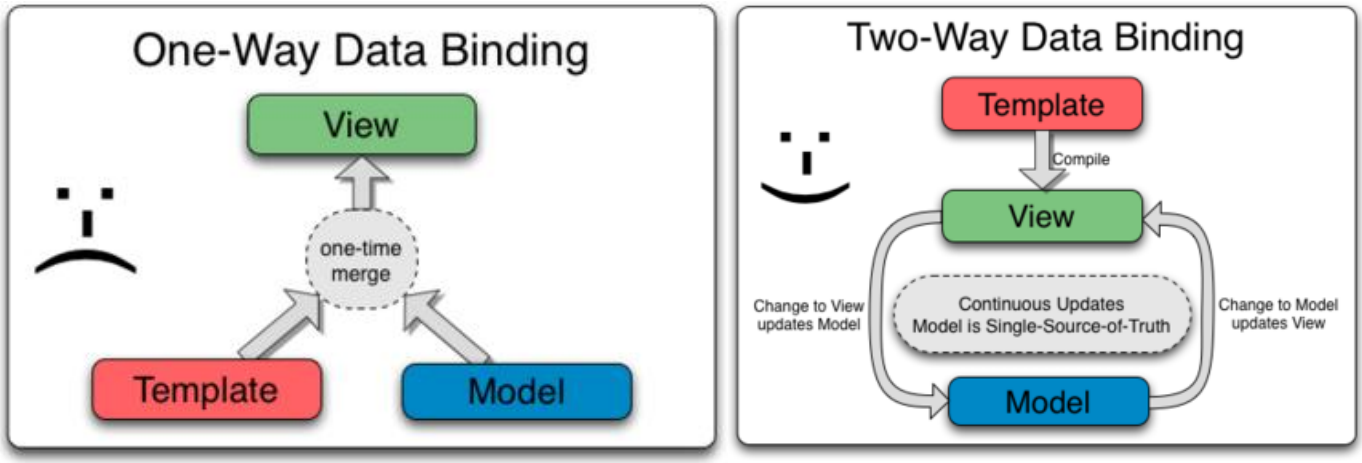
- Des noms peuvent être réutilisés par mégarde, écrasant l'ancienne valeur des variables et provoquant des comportements imprévisibles et difficiles à débugger.
- Comme toutes les variables JavaScript sont obligatoirement publiques, il est impossible de masquer des attributs ou des méthodes. Cette caractéristique est bien évidemment problématique puisque l'utilisateur a ainsi accès aux rouages internes de l'objet et peux ainsi les utiliser de manière imprévue.

Angular.js propose une solution efficace à ce problème. Une variable nommé '*$scope*' est définie pour chaque contrôleur et n'est visible (par défaut) que pour ce contrôleur. Le résultat final est ainsi beaucoup plus propre puisque chaque contrôleur dispose de son propre scope. Une fonctionnalité d'héritage de *scope* permet en plus de factoriser facilement du code.

c. Two Way Data Binding

Comme nous l'avons dit précédemment, *Angular.js* incite (pour ne pas dire oblige) le développeur à adopter le *design pattern* MVC. Ainsi, le modèle contient les données et la vue met les données à disposition de l'utilisateur.

Une des fonctionnalités les plus puissantes d'anglar.js est sans aucun doute le "*two way data binding*". Comme son nom l'indique, cette fonctionnalité permet de relier directement le modèle et la vue: si une modification est apportée sur le modèle, la vue en est immédiatement notifiée, et vice versa. Cette caractéristique peut sembler minime, mais c'est en fait un changement majeur par rapport au développement classique JavaScript, puisque ce système de notification/mise à jour n'est pas implémenté nativement par les navigateurs.



Voici un exemple d'utilisation simple: Lors du développement d'applications complexes en JavaScript, il est quasi systématique de devoir gérer les évènements asynchrones. Ce fut le cas lors de notre projet. Par exemple, nous avons dû aller récupérer de façon asynchrone les messages depuis les réseaux sociaux auxquels l'utilisateur s'est connecté. Le "*two way data binding*" permet en fait de gommer la complexité de la gestion de l'asynchrone puisque les données (ici les messages) sont automatiquement intégrées à la vue au fur et à mesure de leur arrivée.

Pour donner une idée concrète de la syntaxe d'a*ngular.js*, voici une utilisation pour la vue (dans tablet/index.html : ce code prend en charge l'affichage de la liste de messages sur le mur "*All*" de la tablette) :

```
64              <ul ngm-if="anyConnected()" data-role="listview" data-split-icon="plus" data-split-theme="d" >
65                  <li ng-repeat="msg in getAllMsg()">
66                      <a ngm-click="onMsgDetail(msg)" href="msgdetail.html">
67                      <img src="{{getMsgImg(msg)}}" height="90px" width="90px"/>
68                      <h5>{{msg.authorName}}</h5>
69                      <p>Posted on <em>{{formatDate(msg.msgDate)}}</em></p>
70                      <p>{{msg.msgContent}}</p>
71                      </a><a href="newmsg.html" data-rel="dialog" data-transition="slidedown">Reply</a>
72                  </li>
73              </ul>
```

Notons que le code situé entre {{...}} fait appel au *scope* du contrôleur parent (ici: “tabletAppCtrl”, spécifié ligne 25, invisible dans cette captuere d’écran).

d. Abstraction

Le code produit par *angular.js*, fortement influencé par MVC, est modulaire. En plus, *angular.js* favorise un couplage faible entre les éléments. Cette caractéristique est intéressante pour de nombreuses raisons.

Le code produit est ainsi facilement testable : Comme le code est très fragmenté en sous fonctions, il est facile de tester de comportements indépendamment. Pour les parties complexes du code, il peut être très utile d’écrire des tests pour être sûr que le code fait bien ce pour quoi il a été prévu. Aussi, ces tests pourront être utilisés pour tester la “non-régression” du code : On s’assure ainsi que des fonctionnalités qui étaient effectives, n’ont pas été cassées par un *refactoring* du code. Étant donné que la plupart des fonctions de notre code ont des fonctionnalités basiques, il n’a pas été très utile pour nous d’écrire de nombreux tests.

Par contre, le découplage du code nous a été très utile car il nous a permit de mettre en place des *mock-ups*.

En effet, dans notre projet, les modèles occupent une place déterminante, puisque notre projet vise à agréger les informations en provenance de différents réseaux sociaux. Les *mock-ups* nous ont ainsi permis de pouvoir travailler sur l’interface graphique, alors même que les modèles n’étaient pas encore implémentés. En effet, un *mock-up* est un code statique simple, censé simuler grossièrement le comportement d’une entité du programme, dans notre cas les modèles.

Nous avons pu ainsi développer chaque partie indépendamment, sans nous soucier des blocages et interdépendances entre la vue et le modèle.

e. Quelques commentaires

Maintenant que sous avons achevé notre projet, nous pouvons dire que la réalité fut moins idéale. *Angular* est une technologie très récente : même si le *framework* est en développement depuis 2009, la version 1.0.0 (toujours en vigueur aujourd’hui) est sortie officiellement le 14 mars 2012. Par conséquent, le *framework* n’est pas encore répandu, et la documentation n’est pas aussi abondante qu’on l'aurait souhaité.

La difficulté principale de ce *framework* réside dans sa plus grande force : de nombreuses actions sont prises en charges automatiquement par *Angular.js*, laissant paraître que les

choses fonctionnent "par magie". Or, quand il faut plonger plus en détail dans le fonctionnement d'*Angular.js*, les choses se compliquent : *Angular.js* comporte plus de 13000 lignes de code, et leurs fonctionnements sont souvent complexes.

En fait, nous pensons qu'*Angular.js* permet effectivement de gagner du temps de développement. Cependant, ce *framework* nécessite un temps d'apprentissage considérable, d'autant plus que par manque d'aide disponible sur Internet, on peut potentiellement rester bloqué longtemps sur certains problèmes.

3. Technologies pour l'interface utilisateur

Dans cette sous section, nous allons lister les technologies que nous avons utilisé pour la partie interface utilisateur de notre programme.

a. HTML 5 / CSS 3

Bien sûr, le support principal de notre site web est HTML. Précisons que nous avons tiré parti de certaines des avancées les plus récentes de ce langage.

En particulier, comme nous n'avons aucune possibilité de mémorisation sur le serveur, nous avons utilisé les APIs de *sessionStorage* fournies par HTML 5 (cf. http://www.w3schools.com/html5/html5_webstorage.asp). Ainsi, il nous a été possible de stocker certaines informations de l'utilisateur directement dans son navigateur, de manière à les mémoriser pour son prochain passage. Aussi, nous avons essayé d'utiliser les fonctions du '*manifest*' fourni par HTML5
(cf. http://www.w3schools.com/html5/att_html_manifest.asp). Cette fonctionnalité permet de spécifier dans un '*manifest*' les ressources que nous utilisons pour l'application, le navigateur peut donc les mettre en cache. Cette fonctionnalité est particulièrement utile car elle permet de faire fonctionner l'application hors ligne (le navigateur garde en mémoire les ressources listées dans le manifeste, même une fois le navigateur fermé). En pratique, ce *manifest* s'est montré pénible à utiliser lors du développement car le navigateur gardait systématiquement les fichiers en cache, et comme aucun système de versionnement n'a été prévu, nous avions du mal à actualiser les ressources sans réinitialiser la mémoire du navigateur. Nous avons ainsi abandonné l'utilisation du *manifest*, plus compatible avec une application au stade de production.

Même si nous avons utilisé les fonctionnalités de HTML5 de manière marginale, nous avons largement utilisé les dernières fonctionnalités de CSS3, comme les coins arrondis (*border-radius*), les ombres (*box-shadow*, *text-shadow*) ou les transitions (*transition*).

Remarquons d'ailleurs que de la plupart des librairies graphiques que nous allons évoquer dans la suite dans ce rapport font un usage intensif de CSS3.

b. Bootstrap

Bootstrap est un *framework* graphique développé par *twitter* (cf. http://twitter.github.com/bootstrap/). C'est en fait leur propre *framework* qu'ils ont offert au monde du libre. Ce *framework* jouit ainsi d'une qualité et d'une robustesse sans égales (du moins dans les *framework* libres) puisque largement éprouvées par *twitter*. Nous avons utilisé *Bootstrap* principalement pour l'interface pour ordinateur (toute la partie graphique est prise en charge par *Bootstrap*), et pour l'écran de chargement et de détection du terminal.

Bootstrap se compose principalement d'un fichier de style très complet. En effet, *Bootstrap* comporte un affichage amélioré pour la plupart des éléments standards de HTML (boutons, formulaires, conteneurs, etc.). De plus, *Bootstrap* est particulièrement robuste et supporte différents types de terminaux : la taille du *viewport* est prise en compte et l'affichage est adapté à la taille de l'écran de l'utilisateur. *Bootstrap* est conçu pour que chaque propriété puisse être facilement surchargée, afin de personnaliser très facilement le rendu.

Bootstrap peut être vu comme un couteau suisse pour le design web puisqu'il regroupe bien plus de fonctionnalités qu'un simple fichier de styles. En effet, de nombreux plugins *JavaScript* sont disponibles pour rajouter des fonctionnalités visuelle. Par exemple, on peut citer les *modaux* (cf. http://twitter.github.com/bootstrap/JavaScript.html#modals), qui peuvent faire office de *pop-ups* (on pourra voir leur utilisation pour l'affichage du détail des messages et des profils sur l'interface ordinateur).

Remarquons pour finir qu'une des critiques principales que l'on peut faire à *Bootstrap* est d'être victime de son succès. En effet *Bootstrap* est présent sur un nombre grandissant de sites web, et laisse donc une impression de déjà vu à de nombreux utilisateur pratiquant intensément le Web. Il s'agit d'un point d'autant plus important que l'identification graphique est devenue très importante pour le web, où de très nombreux sites se font concurrence pour fournir des services souvent très proches.

c. JQuery Mobile

JQuery Mobile est un *framework* graphique libre porté par communauté JQuery (cf. http://JQuerymobile.com/). Comme son nom l'indique, ce *framework* est orienté vers les terminaux mobiles, et notamment les périphériques tactiles. C'est la raison pour laquelle nous avons fait le choix de l'utiliser pour les terminaux *smartphone* et tablette. On remarque

d'ailleurs que *JQuery* fournit une interface dont la ressemblance et frappante avec les applications natives des systèmes mobiles les plus répandus (i.e. iOS et android).

Ce *framework* est compatible avec la plupart des navigateurs mobiles du marché. C'est l'une de ses plus grandes forces. Il est connu que l'homogénéisation du rendu entre les différents navigateurs est une contrainte particulièrement pénible à gérer pour les développeurs web. On comprend donc bien pourquoi s'en affranchir est un avantage de taille.

En fait, le plus grand problème que nous avons rencontré avec *JQuery Mobile* est son manque d'interopérabilité avec *Angular*. En fait nous avions prévu ce comportement dans la mesure où *JQuery Mobile* et *Angular* procèdent tous deux à des modifications automatiques du DOM. Nous avons donc trouvé un adaptateur *JQuery Mobile/Angular* fourni par un développeur indépendant et mis au profit de la communauté du logiciel libre (cf. https://github.com/tigbro/JQuery-mobile-angular-adapter). L'adoption de *JQuery Mobile* a donc été d'une difficulté limitée (hormis le fait que cette technologie vienne s'ajouter à la longue liste des technologies nouvelles à appréhender)

4. Librairies Utilitaires

Étant donné que *JavaScript* et un langage relativement limité en fonctionnalités natives, nous avons dû utiliser de nombreuses librairies utilitaires pour nous aider à réaliser les opérations les plus communes de façon la plus simple possible.

5. Les librairies génériques

Dans cette section, nous détaillerons les librairies utilitaires génériques que nous avons le plus utilisé.

a. JQuery

JQuery (cf. http://JQuery.com/) est une librairie très utilisée en *JavaScript* (probablement la plus répandue d'ailleurs). En fait très rares sont les applications *JavaScript* qui ne font pas usage de *JQuery*.

JQuery est la librairie utilitaire par excellence : elle fournit de nombreux outils pour faciliter la plupart des opérations communes, et pourtant fastidieuses. On peut citer par exemple la manipulation du *DOM*, ou encore l'implémentation de systèmes *AJAX*.

b. RequireJS

RequireJS (cf. http://requirejs.org/) est une librairie *JavaScript* majoritairement utilisée pour le chargement asynchrone de scripts. En fait, depuis les premières versions de *RequireJS*, le nombre de fonctionnalités offertes par cette librairie a beaucoup augmenté. Nous nous en sommes toutefois tenus à sa fonction première.

Le chargement de script représente un point primordial dans notre projet, dans la mesure où nous avons développé de nombreux scripts, parfois lourds, alors que les performances de notre application sont critiques.

Le chargement standard de scripts (lorsqu'ils sont inclus dans la balise <*head*> du document HTML) se fait de manière linéaire : le téléchargement d'un script ne commence que lorsque le téléchargement du précédent est terminé. Sachant que le temps d'attente est principalement lié à de l'entrée/sortie, on a tout intérêt à paralléliser le téléchargement des scripts pour avoir un gain significatif.

Remarquons que dans notre projet l'ordre de chargement est, comme souvent, significatif. Nous avons ainsi utilisé un plugin "*order*" fourni par la communauté *RequireJS* pour garantir que les scripts seront exécutés dans un ordre précis (alors que leur chargement reste parallèle, et donc dans un ordre pseudo-aléatoire).

c. Formfactor

Formfactor (cf. https://github.com/PaulKinlan/formfactor) est une petite librairie *JavaScript* développée par un ingénieur de *Google* à titre expérimental. Cette librairie permet de tester certaines caractéristiques du browser (principalement à l'aide du *DOM* et de librairies tierces comme *modernizr.js*) et de charger certains scripts en fonction du résultat de ces tests.

Il nous a été ainsi possible de tester la résolution de l'utilisateur, le *user-agent*, la présence de certaines fonctionnalités (comme le tactile) pour détecter le type de terminal utilisé par l'utilisateur et rediriger ce dernier vers la spécialisation du site le plus adapté à son terminal (tablette, *smartphone*, desktop ou TV).

d. SDK développeurs

En ce qui concerne les modèles dans notre projet, nous avons eu recours à des SDK nous aidant à interagir avec les différents réseaux sociaux.

Notons d'abord une difficulté : Nous avons remarqué au fil de nos recherches que les libraires fournies par les réseaux sociaux sont beaucoup plus fréquentes pour les langages serveurs que pour les langages clients. Nous avons trouvé deux explications principales à ce constat.

La plupart des développeurs Web favorisent le code serveur par rapport au code client. C'est donc naturel que les réseaux sociaux développent en priorité des librairies pour les langages serveur.

L'authentification est un critère délicat d'un point de vue sécurité. Et la sécurité est, en toute logique, beaucoup plus délicate à gérer côté client que côté serveur (c'est logique car tout le code exécuté côté client est directement accessible et visible en clair par ce même client). Ainsi le développement de SDK sécurisés côté client requiert un surcoût en travail pour les réseaux sociaux, qui sont donc plus réticents à développer ce type de solution.

Pour nous aider dans notre projet, nous avons utilisé les *frameworks* directement fournis par *Facebook* (cf. http://developers.facebook.com/docs/reference/JavaScript/) et *twitter* (nommé @*Anywhere*: https://dev.twitter.com/docs/anywhere/welcome). Notons que si le SDK fourni par *Facebook* est d'une grande qualité et est très bien documenté, il n'en n'est pas de même pour le SDK *twitter*. En effet, ce dernier semble être tombé en désuétude, et une simple tâche comme envoyer un *tweet* s'est avérée compliquée à implémenter (principalement à cause du manque de support et de la volonté de *twitter* d'obscurcir le fonctionnement du script, pour les raisons de sécurité mentionnées ci dessus).

Notons que nous n'avons pas pu nous procurer un SDK *JavaScript* pour *Google*+. C'est d'ailleurs la raison pour laquelle nous n'avons pas implémenté ce réseau social dans notre projet. Cette absence est principalement due au fait que *Google*+ est un des derniers acteurs arrivé sur le marché des réseaux sociaux de grande envergure.

III. Difficultés rencontrées

Pour mener à bien notre projet, nous avons été amenés à utiliser de nombreuses technologies différentes. En fait, la majorité des difficultés que nous avons rencontrées ont été liées à ces technologies.

La majorité de ces technologies sont récentes, et les communautés qui les supportent sont encore peu développées. De ce fait, la documentation accessible est souvent réduite et il est difficile de trouver du support sur internet. Il nous est ainsi arrivé de rester bloqués de nombreuses heures sur certaines fonctionnalités de notre application, simplement par manque de support. Nous avons donc parfois eu recours à des solutions considérées comme dépréciées (redondance, variables codées en dur, etc.) pour régler ces problèmes, ce qui a dégradé la facilité de lecture et la maintenabilité du code.

Nous avons rencontrés un problème lors de l'utilisation de *frameworks* graphiques prêts à l'emploi (principalement *Bootstrap* et *JQuery Mobile*) : Commencer une interface est très facile, par contre dès lors que l'on veut modifier et adapter des éléments, les choses se compliquent. En fait, étant donné la complexité de ces *frameworks*, il est inenvisageable de les modifier pour les adapter à nos besoins : c'est en quelque sorte « à prendre ou à laisser ». Ce manque d'adaptabilité fut particulièrement gênant lorsque nous avons voulu adapter les interfaces pour améliorer l'utilisabilité (par exemple quand nous avons voulu augmenter la taille des boutons du smartphone par rapport à la tablette). C'est un contrecoup intrinsèque à l'utilisation de fonctionnalités prêtes à l'emploi. Remarquons tout de même qu'étant donné le temps qui nous a été imparti pour réaliser se projet, il aurait été impensable de nous passer de tels *frameworks*.

De plus, nous avons aussi eu des soucis de compatibilité entre les technologies que nous utilisons, principalement entre *JQuery Mobile* et *Angular*. Ces deux technologies modifient le *DOM* de la page, et il en a résulté plusieurs conflits. La présence d'un adaptateur *JQuery Mobile*/*Angular* a permis d'améliorer la situation mais des problèmes subsistent. Par exemple, dès que *JQuery Mobile* opère une modification sur le *DOM*, *Angular* en est notifié et cherche à rafraîchir le modèle (cf. *Two-way binding*). Il en résulte de nombreux appels inutiles aux méthodes du modèle (et donc des appels en rafale aux APIs des réseaux sociaux). Nous avons dû parer à ce problème en sauvegardant les états du modèle sous forme de cache.

Le facteur temps nous a aussi posé problème. La structure ayant été relativement complexe à mettre en place (pour factoriser la gestion des les réseaux sociaux), nous avons dût mettre en place un système de "*mock-up*" (cf. glossaire) pour pouvoir commencer à coder les interfaces en évitant d'attendre que les modèles soient fonctionnels. Nous avons donc passé un temps non négligeable à installer cette structure, même si elle nous a facilité la tâche par la suite. Le problème est en fait que le développement des interfaces est la partie

visible de notre projet, mais une grande partie de notre travail a en fait porté sur des parties invisibles (comme la structure du projet, les mocks-up, la détection du périphérique, etc.).

Le nombre d'interfaces à considérer nous a aussi posé des problèmes au niveau du temps. Effectivement, dès la présentation des maquettes, nous avions proposé quatre interfaces différentes selon les périphériques : desktop (ordinateur), tablette, *smartphone* et TV. Nous avions déjà décidé dès la présentation des maquettes de ne pas réaliser l'interface TV pour nous concentrer sur les trois autres. Ce fait est d'autant plus regrettable pour nous que la TV présente des interactions particulières qu'il aurait été très intéressant d'étudier. Remarquons que même si toutes les plateformes partagent beaucoup de code, les interfaces sont toutes totalement différentes les unes des autres pour prendre en compte les spécificités de chaque plateforme.

Nous avons aussi rencontré des difficultés avec les différents navigateurs internet du marché : leurs moteurs *JavaScript*, CSS, HTML, etc. étant différents, beaucoup de fonctionnalités fonctionnent sur certains navigateurs et pas sur d'autres. Nous avons donc essayé de nous concentrer sur quelques navigateurs (*Chrome* et *Firefox*) pour avoir une application présentable à la fin du projet. Bien évidemment, la situation s'empire dès lors que l'on considère la diversité des navigateurs mobiles en supplément.

Enfin, nous avons eu des difficultés pour satisfaire les attentes des utilisateurs. En effet, ceux-ci ont l'habitude d'utiliser des applications natives, qui sont très performantes en termes de fluidité et de fonctionnalités. Par contre, il est très difficile d'atteindre de telles performances avec une application en *JavaScript*. C'est en fait un contrecoup inévitable au fait de développer une seule application générique : on perd les spécificités et l'optimisation de chaque plateforme.

De plus, comme nous le préciserons dans la partie consacrée aux interactions, la plupart de nos utilisateurs auront tendance à utiliser notre application de manière occasionnelle et rapide. Nous nous devions d'avoir une application très réactive pour ne pas perdre cet intérêt du point de vue de l'utilisateur. C'est en fait une des contraintes les plus difficiles à satisfaire des lors que l'on cible des périphériques mobiles, susceptibles d'avoir des vitesses de connexion dégradées.

IV. Etude des interactions

1. L'utilisateur typique de SNAP

Notre application cible principalement les personnes souhaitant obtenir dans un temps limité, un aperçu des fils d'actualité de leurs réseaux sociaux. On a donc clairement

une contrainte temporelle. En effet, une personne ayant du temps libre favorisera les sites de ses réseaux sociaux (plus complet que SNAP en terme de fonctionnalités) ou encore une application native dédiée (plus efficace en terme de réactivité). Nous devons donc concevoir nos interfaces en prenant en compte ce facteur et mettre en avant principalement les informations auxquelles l'utilisateur souhaite accéder lors d'une utilisation 'express'.

2. Les interactions spécifiques à chaque périphérique

Pour que nos interfaces soient le mieux adaptées aux différentes plateformes, nous avons fait la liste des interactions spécifiques à chaque périphérique.

a. Desktop

Le desktop (ordinateur fixe ou portable) est une plateforme qui exploite très souvent des écrans relativement grands avec une résolution de taille variable. Nous prenons donc en compte ces deux aspects. Il faut que l'interface puisse s'adapter à différentes résolutions ainsi qu'au redimensionnement, tout en restant utilisable.

De plus on y utilise le plus souvent un pointeur très précis (manipulé par souris ou *trackpad*). Nous pourrons donc utiliser les liens et les petites zones cliquables pour accéder aux différentes fonctionnalités ou aux différents éléments de l'interface.

Enfin l'utilisateur est statique (immobile derrière son écran), il est donc normalement concentré sur l'interface et ne souffre que peu de distractions extérieures. On peut donc lui afficher des informations denses, grâce à la taille et à la résolution des écrans utilisés avec ces plateformes.

b. Tablette

La tablette est une plateforme qui se manipule principalement avec le bout des doigts. Nous devons donc favoriser les grandes zones cliquables (comme les boutons) contrairement aux liens, et nous pouvons utiliser les interactions tactiles (*swipe*).

Les écrans des tablettes étant de taille moyenne, nous ne pouvons y afficher qu'un nombre d'informations réduit par rapport au desktop.

De plus les utilisateurs peuvent être mobiles avec une tablette et peuvent donc être soumis à des distractions. On évitera donc d'y mettre trop d'informations et de détails, afin de se concentrer sur du contenu pertinent pour l'utilisateur. La mobilité influe aussi sur la façon dont les utilisateurs vont utiliser notre application. Un utilisateur devra pouvoir accéder très vite au contenu de l'application (par exemple au cours d'un transport) et ce contenu devra être rapidement lisible. On mettra donc l'accent sur une interface épurée pour que l'utilisateur ne puisse pas être déconcentré et pour qu'il ne perde pas de temps à

cause de certains détails dans l'interface ou du trop grand nombre d'informations non pertinentes.

De plus la tablette est un périphérique destiné à une très large audience, notamment pour des personnes qui ne sont pas forcément habituées à manipuler un ordinateur (comme des jeunes enfants ou des personnes âgées). Nous devons donc en plus avoir une application simple à utiliser et la plus intuitive possible en ayant recours par exemple à des indices visuels.

c. Smartphone

Le *smartphone* est une plateforme avec beaucoup de contraintes. Tout d'abord, l'écran utilisé par ces périphériques est généralement de petite taille. On doit donc favoriser la catégorisation des fonctionnalités dans des espaces séparés, en affichant dans chaque catégorie les informations les plus pertinentes possibles.

De plus, ces plateformes sont tactiles : nous pouvons donc exploiter les interactions tactiles (*swipe :* mouvement horizontaux). Remarquons que l'écran d'un smartphone est de taille limitée et que de nombreux utilisateurs l'utilisent seulement avec les pouces. Les zones hautes de l'écran sont par conséquent difficilement accessibles. Nous ferons donc en sorte d'éviter de placer des liens cliquables tout en haut de l'écran.

L'utilisateur est aussi très mobile dans le contexte d'utilisation d'un *smartphone,* et est donc très sensible aux distractions extérieures. Pour la même raison, l'utilisateur va utiliser l'application souvent mais pour des durées très limitées. Nous devons donc nous concentrer en priorité, dans chaque catégorie, aux informations auxquelles l'utilisateur souhaite accéder, tout en ayant une interface très épurée pour éviter toute distraction supplémentaire à l'utilisateur. Nous devons aussi faire en sorte que l'application soit la plus réactive possible pour correspondre à cette utilisation occasionnelle. Aussi, les boutons doivent être plus grands que pour les tablettes, car si l'utilisateur est potentiellement plus mobile, il sera plus difficile de cliquer sur une zone précise.

d. TV

La télévision ne propose que très peu de boutons pour interagir avec l'interface : 4 flèches de direction et un bouton "ok". Nous devons donc simplifier les interactions à leur plus simple expression pour avoir une interface utilisable.

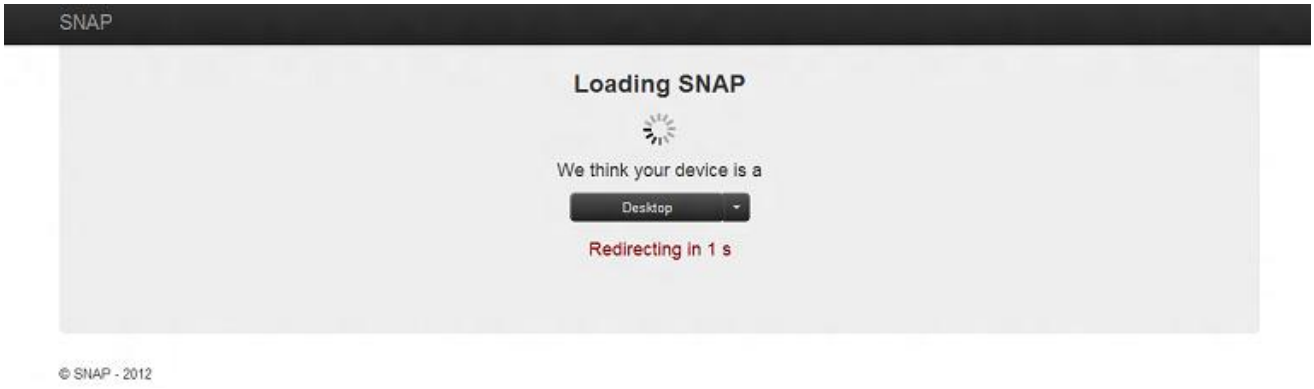
De plus les télévisons ont principalement de grands écrans, ce qui nous permet de favoriser le confort d'affichage et les détails dans l'interface, l'utilisateur n'étant que peu soumis aux perturbations extérieures dans son salon. Cela nous permet de favoriser le côté "entertainment" de l'application.

Pour que l'application soit la plus naturelle possible à utiliser avec 5 boutons, nous devons aussi mettre le plus possible d'indices visuels qui guideront l'utilisateur dans son utilisation. Aussi, nous devrons faire attention à tenir une démarche systématique de manière à ce que l'utilisateur puisse trouver facilement ses repères en tout endroit de l'application.

3. Les solutions utilisées

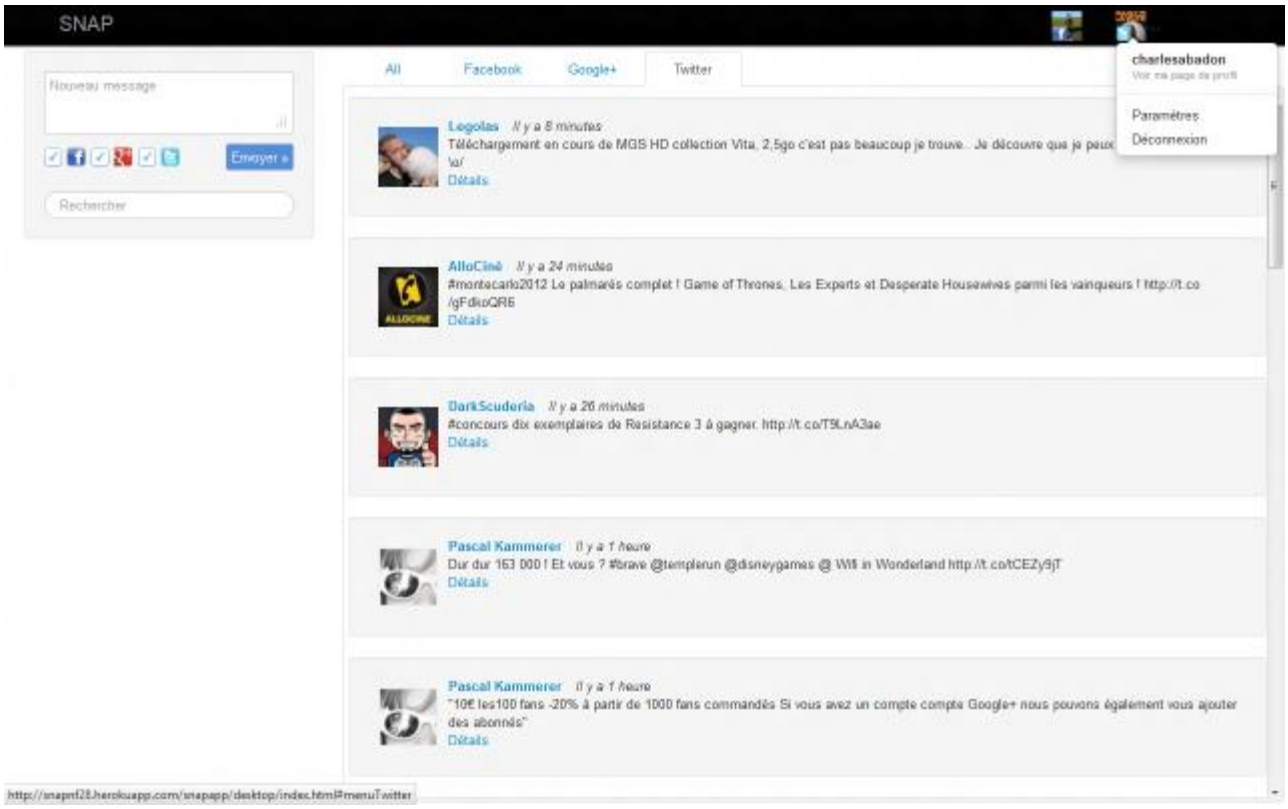
Pour que l'interface utilisée corresponde à la plateforme de l'utilisateur nous avons mis en place une fonctionnalité de détection automatique du terminal. L'utilisateur peut malgré tout décider d'en choisir une autre s'il le souhaite (et ce choix sera mémorisé pour ses prochain passages). Cet écran de chargement nous permet aussi de charger les scripts au démarrage de SNAP pour accélérer la navigation sur les différents périphériques.

De cette façon, on effectue un chargement à l'ouverture de l'application, plutôt qu'en cours d'utilisation de l'application. Cet étape sert donc un double intérêt : choix de l'interface, bien sur, mais aussi charger les scripts de manière silencieuse, sans que l'utilisateur aie l'impression de patienter (son attention est attirée par la détection de son terminal). On réduit donc ici une des grandes nuisances de la navigation sur internet mobile : les temps d'attente.



Ecran de chargement de SNAP

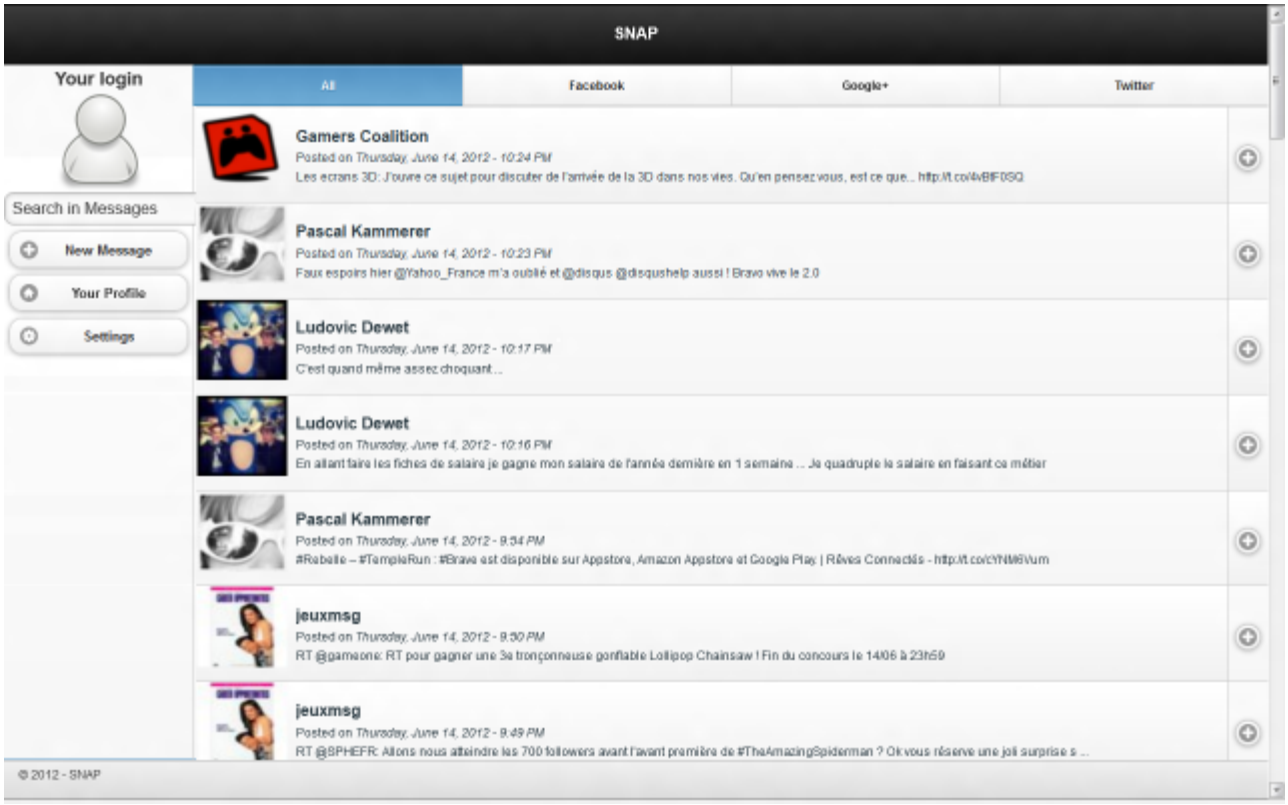
Pour le desktop nous avons favorisé une interface de grande taille, avec beaucoup de détails et d'informations, et qui utilise la largeur de l'écran pour afficher plusieurs fonctionnalités à l'utilisateur (nouveau message, recherche...) en plus de l'affichage des messages. Du fait de l'utilisation d'un pointeur (souris, *trackpad*), nous avons aussi favorisé les liens ou les zones cliquables de petite taille. De plus nous avons fait attention à ce que l'interface reste utilisable si l'utilisateur redimensionne sa fenêtre.



Interface Desktop

Concernant la tablette, nous avons essayé de faire en sorte qu'elle soit la plus utilisable possible avec les doigts, avec des zones cliquables et des boutons relativement larges. Nous avons aussi fait en sorte de mettre les messages des réseaux sociaux en avant (les informations qui intéressent en premier l'utilisateur), en consacrant moins d'espace aux autres fonctionnalités qui seront appelées grâce à des zones cliquables. Cela permettra à ce que l'utilisateur ait en priorité les informations qui l'intéressent.

Nous avons aussi implémenté les interactions tactiles (*swipe*) pour passer d'un onglet à l'autre. De plus nous avons fait en sorte d'avoir une interface épurée et facilement lisible (avec des textes de plus grande taille).



Interface tablette

Enfin, le *smartphone* étant tactile, nous avons également implémenté les interactions tactiles (*swipe*), mais cette fois ci pour passer d'un panneau à l'autre. De plus nous avons utilisé des boutons et des zones cliquables de grandes tailles et placés prioritairement en bas d'écran pour qu'ils soient utilisables au pouce.

Chaque catégorie ne contient que des informations pertinentes pour celle-ci. Par exemple, sur l'image ci-dessous nous pouvons voir la liste des messages agrégés, qui ne contient que les messages des différents réseaux sociaux. Pour envoyer un nouveau message, il faudra cliquer sur le bouton de nouveau message en bas de l'écran pour afficher un nouveau panneau consacré à l'écriture d'un nouveau message.

Ce choix de la pertinence du contenu ajouté à la sobriété de l'interface permet à l'application d'être particulièrement lisible dans des situations de mobilité.

Interface smartphone

V. Protocole de test

1. Etablissement d'un protocole de test

En ce qui concerne l'évaluation logicielle, plusieurs critères peuvent être pris en compte. On peut citer par exemple les 10 critères de Nielsen. Comparons succinctement quelques-uns de ces critères et mettons les en relation avec notre application :

- Adéquation du système au monde réel: Notre application utilise un langage naturel simple et concis que l'utilisateur peut comprendre facilement.
- Cohérence et respect des standards : Nous avons cherché a respecter les codes typiques des périphérique (comme la barre d'action en bas de l'interface *smartphone*).
- Design minimaliste et esthétique : Les interfaces de notre application sont simples, quel que soit le périphérique utilisé nous affichons uniquement l'information strictement nécessaire au sein des pages. En cas de doutes, nous choisis d'éliminer des informations plutôt que d'en rajouter.
- Reconnaissance plutôt que rappel : Prenons par exemple le cas de l'interface ordinateur, l'utilisateur qui souhaite envoyer un message voit directement comment effectuer cette action, grâce au formulaire d'envoi de message.

- Flexibilité d'utilisation: Certaines options, comme la sélection réseaux sociaux par défaut, peuvent permettre d'adapter l'usage de SNAP à l'expérience de l'utilisateur.
- Aide et documentation : notre application possède un site explicatif disponible à l'adresse : http://snapnf28.herokuapp.com/website/index.html

Pour résumer, notre application est en adéquation avec la plupart des critères de Nielsen. Remarquons tout de même que SNAP est au stade de prototype et que l'ergonomie n'est pas optimale. Il existe bien sûr de nombreux critères ergonomiques permettant d'évaluer un logiciel. On peut également citer les critères de Bastien et Scapin que nous ne détaillerons pas ici pour éviter une certaine redondance avec ceux décrits ci dessus.

Nous avons décidé de procéder à des tests utilisateurs dans le but d'avoir un regard critique sur notre application et en vue d'une amélioration future. Nous avons listé une série de fonctionnalités à tester pour chaque type de périphérique :

- Se connecter aux réseaux sociaux
- Consulter les derniers messages (Pour tous les réseaux sociaux et pour un réseau social en particulier)
- Consulter le détail d'un message
- Consulter son profil
- Effectuer une recherche de messages
- Poster un message.
- Se déconnecter

Nous avons également listé un certains nombre de variables à prendre en compte pour l'évaluation des tests:

- Expérience de l'utilisateur (novice, généraliste, expert)
- Taux de succès pour la réalisation d'une tâche
- Temps pour exécuter une tâche
- Nombre d'étapes nécessaire pour effectuer une tâche
- Le ressenti de l'utilisateur (satisfait, perdu, énervé...)

Pour réaliser ces tests, nous avons sélectionné un panel de 6 utilisateurs, avec différentes expériences, soit deux utilisateurs par type d'appareil.

2. Compte-rendu

a. Résultats des tests

- Tests effectués sur l'interface ordinateur (par Pierre-Yves Gicquel et Clément Gaudin.)

Nous avons réalisé en moyenne les résultats suivants :

	Description de la tâche	Temps	Nombre de clics	Succès	Ressenti sur 5
Q1	Connexion à Facebook	5 sec	3	Oui	4
Q2	Consulter sa liste de messages aggrégés	1 sec	1	Oui	4
Q3	Consulter ses messages twitter ou Facebook	1 sec	1	Oui	4
Q4	Afficher les détails d'un message	1 sec	1	Oui	4
Q5	Afficher un profil	1 sec	1	Oui	4
Q6	Afficher son profil	> 10 sec		Non	2
Q7	Poster un message sur tous les RS	1 sec	1	Oui	4
Q8	Poster un message sur un RS spécifique	2 sec	3	Oui	5
Q9	Faire une recherche	2 sec	0	Oui	4
Q10	Se déconnecter	2 sec	2	Oui	4

- Tests effectués sur l'interface tablette (par Jalal Miftah et Adrien Volokhoff)

Notons que nous avons enlevé certaines tâches car nous n'avons pas développé toutes les fonctionnalités sur ce périphérique.

	Description de la tâche	Temps	Nombre de clics	Succès	Ressenti sur 5
Q1	Connexion à Facebook	10 sec	3-4	Oui	3
Q2	Consulter sa liste de messages aggrégés	3 sec	1	Oui	3
Q3	Consulter ses messages twitter ou Facebook	3 sec	1	Oui	4
Q4	Afficher les détails d'un message	1 sec	1	Oui	4.5
Q5	Poster un message sur tous les RS	2 sec	2	Oui/Non	2.5
Q6	Poster un message sur un RS spécifique	1 sec	2	Oui	4
Q7	Faire une recherche	1 sec	1	Oui	4

- Tests effectués sur l'interface *smartphone* (par Jalal Miftah et Alice Portalier)

	Description de la tâche	Temps	Nombre de clics	Succès	Ressenti sur 5
Q1	Connexion à Facebook	10 sec	3	Oui	4
Q2	Consulter sa liste de messages agrégés	1 sec	1	Oui	4
Q3	Consulter ses messages twitter ou Facebook	5 sec	2-3	Oui	4
Q4	Afficher les détails d'un message	5 sec	1	Oui	3
Q5	Afficher un profil	2 sec	1	Oui	3
Q6	Poster un message sur tous les RS	30 sec	4-5	Oui	2
Q7	Poster un message sur un RS spécifique	5 sec	1	Oui	5
Q8	Faire une recherche	2 sec	2	Oui	3

b. Critiques de SNAP

Remarquons que dans l'ensemble les utilisateurs ont été satisfaits de SNAP, ce qui est gratifiant, d'autant plus que l'application en est encore au stade de prototype.

Nous allons nous attarder sur leurs éventuelles critiques et suggestions d'amélioration :

- Dans le cas général, nous observons un temps de chargement relativement long, surtout sur les appareils mobiles.

- Nous utilisons un modèle unique de réseaux sociaux : on perd ainsi les spécificités de chaque réseau social.

- La fonctionnalité de recherche n'est pas très claire quel que soit le périphérique. En effet, rien n'est spécifié, l'utilisateur ne connaît pas toutes les possibilités.

- Dans le cas de l'interface ordinateur, la consultation du profil pose problème pour la majorité des utilisateurs. Les icônes en haut à droite ne sont pas assez grandes et pas assez explicites pour l'utilisateur. Nous pourrions ajouter un sous-titre noté *Profil utilisateur* sous les icônes.

- Concernant la navigation sur *smartphone*, les flèches de navigation sont ergonomiques et rappellent celle du téléphone, mais la barre d'outil de SNAP et accolée à celle du téléphone. Il est ainsi facile de presser la mauvaise touche.

- Autre critique sur le *smartphone*, les images de la barre de navigation ne sont pas assez explicites, notamment l'image permettant de poster un nouveau message, que l'on pourrait confondre avec la consultation de ses messages.

- Lorsque l'on consulte le détail d'un message sur tablette, l'utilisateur n'a pas accès à un bouton qui lui permettrait de répondre au message.

- Sur la tablette, la date des messages ne s'affiche pas de la même manière que sur le *smartphone* et l'ordinateur, nous avons ici un problème de cohérence.

Conclusion

SNAP a été un projet très enrichissant pour tous les membres du groupe à de nombreux égards :

Tout d'abord, nous avons pu améliorer nos compétences organisationnelles, en ayant l'occasion de coordonner un travail en groupe relativement grand (pour un projet scolaire). C'est ainsi que nous avons utilisé des outils de gestions de projet innovant comme *Trello*. En plus, nous avons expérimenté des outils de travail collaboratif comme *Github*.

Ce projet nous a également permis de nous perfectionner largement sur le plan technique puisque nous avons pu expérimenter de nombreuses technologies, dont la plupart sont récentes et amenées à se développer fortement dans les années futures. Nous avons ainsi gagné de précieuses compétences dans le domaine du développement web, qui seront probablement très utiles pour notre insertion sur le marché de l'emploi.

Les difficultés que nous a posées ce projet, surtout sur le plan technique, nous ont permises de développer notre capacité d'adaptation et notre indépendance, compétences essentielles pour faire de nous des ingénieurs compétents.

Ce projet s'est concrétisé par un sujet passionnant pour étudier en détail les interactions et les interfaces hommes / machines. En effet, nous avons eu la chance de pouvoir réfléchir en détail sur des types d'interactions spécifique à plusieurs terminaux pour pourvoir développer des interfaces réellement adaptées aux périphériques cibles. Nous avons ainsi pu élaborer trois interfaces distinctes dédiées aux ordinateurs, tablettes et *smartphones*.

Enfin, nous avons pu bénéficier de retours utilisateurs par le biais des tests que nous avons élaborés. Cette expérience a été très bénéfique car elle a permis de révéler de nombreux défauts que nous n'avions pas vus lors du développement et nous avons ainsi pu apprendre de nos erreurs.

ANNEXE A : Glossaire

- **AngularJS** : Framework JavaScript open source développé par Google. Il incite le développeur à utiliser le design patern MVC pour produire du code élégant et testable.
- **API (Application Programming Interface)** : Interface (signatures) fournie par un programme informatique, permettant à une entité tierce au programme de faire appel à ses services.
- **AWS (Amazon Web Services)** : Collection de services informatiques «Cloud Computing » fournis par Amazon.
- **Bootstrap** : Framework graphique développé (et utilisé) par twitter.
- **callback** : Fonction de rappel passée en argument à une autre fonction. Particulièrement dans les contextes asynchrones.
- **commit** : Commande utilisée dans les systèmes de gestion de version correspondant à une validation d'une transaction.
- **cookie** : Fichier hébergé côté client afin de stocker des informations utiles à ce dernier.
- **CSS3 (Cascading Style Sheets version 3)** : Feuille de style utilisée pour mettre en forme le rendu d'une page HTML.
- **Desktop** : Interface de l'application sur ordinateur, qu'il soit fixe ou portable.
- **DOM (Document Object Model)** : Interface indépendante de tout langage et de toute plate-forme, permettant à des programmes informatiques et à des scripts d'accéder ou de mettre à jour le contenu.
- **formfactor** : librairie JavaScript permettant de déterminer le type de périphérique utilisé et à charger des scripts en fonction de ce périphérique.
- **framework** : Ensemble autosuffisant et complet de librairies.
- **gestures** : Mouvements de main prisent en charge par certains terminaux tactiles.
- **git** : Système de gestion de version.
- **Github** : Service web d'hébergement et de gestion de développement de logiciel utilisant le logiciel de gestion de versions Git.
- **Heroku** : Platerforme de cloud computing orientée services.

- **JQUERY** : Framework JavaScript permettant de réaliser plus simplement des taches communes en JavaScript.
- **JQUERY MOBILE** : Framework JavaScript utilisé pour de générer une interface graphique complète et de gérer les interactions utilisateurs sur les périphériques tactiles.
- **JSON (JAVASCRIPT OBJECT NATION)** : Format de données textuel et générique permettant de représenter des informations de manière structurée.
- **LIBRAIRIES** : Anglicisme du mot « library » (le mot « bibliothèques » serait plus approprié), regroupement de fonctions pouvant être réutilisé par un développeur.
- **LOCALHOST** : Nom utilisé pour se référer à une interface logique de l'ordinateur local.
- **MANIFEST (HTML)** : Document décrivant les informations stockées en mémoire dans le navigateur. Il est notamment utilisé pour gérer le "hors ligne".
- **MOCK-UP** : Code statique simple, sensé simuler grossièrement le comportement d'une entité du programme
- **MODERNIZR** : Bibliothèque JavaScript détectant l'implémentation native par le navigateur des technologies les plus récentes.
- **NODE.JS** : Framework JavaScript exécuté le plus souvent côté serveur
- **PLUGIN** : Extension d'un logiciel pour lui apporter de nouvelles fonctionnalités
- **PUSH** : Commande Git pour déployer le code sur un dépôt distant après un commit
- **REPOSITORY** : « dépôt », arborescence des fichiers utilisé notamment dans Github
- **REQUIREJS** : Framework JavaScript permettant, entre autres, de charger des scripts de manière asynchrone.
- **SDK (SOFTWARE DEVELOPMENT KIT)** : « kit de développement », ensemble de composants permettant aux développeurs de créer des applications de type définis
- **SNAP : (SOCIAL NETWORK AGGREGATOR PLATFORM)** : Agrégateur évolutif de réseaux sociaux, notre projet ☺
- **SWIPE** : geste particulier sur les périphériques tactiles tel que le smartphone ou la tablette : glissement horizontal du doigt
- **TRACKPAD** : Pavé tactile, faisant office de souris sur les ordinateurs portables.

- **TRELLO** : Site internet permettant de faire de la gestion de projet se rapprochant de la méthodologie Scrum.
- **TWO WAY DATA BINDING** : fonctionnalité mise à disposition par Angular permettant de relier directement le modèle et la vue.
- **VIEWPORT** : Dimension d'une page web, détecté par le navigateur.

ANNEXE B : Raisons pour l'utilisation des technologies

A: accessibilité
E: Esthétique
Q: quantité du code (concision, factorisation)
R: Réactivité de l'application
S: Simplification et facilitation du développement

	Desktop	Smartphone	Tablette
AngularJS	QRS	QRS	QRS
Bootstrap	AERS		
CSS3	AE	AE	AE
SDKs des réseaux sociaux	QS	QS	QS
Formfactor	A	A	A
git	S	S	S
Github	S	S	S
Heroku	AS	AS	AS
HTML5	AQR	AQR	AQR
JQuery	S	S	S
JQuery Mobile		AERS	AERS
Mock Ups	S	S	S
Node.js	S	S	S

RequireJS	R	R	R

ANNEXE C : Références et bibliographie

Liens concernant SNAP :

Application http://snapnf28.herokuapp.com/snapapp/index.html
Site web explicatif http://snapnf28.herokuapp.com/website/index.html
Dépôt GitHub de l'application : https://github.com/SNAP-NF28/SNAP
Trello https://trello.com/board/nf28-snap/4fa79b4d7d4c9c2d24375d9f

Quelques Livres :

HTML5 : Une référence pour le développeur web - Rodolphe Rimelé
JavaScript & Jquery: The Missing Manual – McFarland & David Sawyer
Tout sur le code : Pour concevoir du logiciel de qualité - Steve McConnell

AngularJS:

Site officiel http://angularjs.org/#/list
Guide du développeur http://docs.angularjs.org/guide/

Node.js : http://nodejs.org

Heroku : http://www.heroku.com

Documentation JQuery : http://docs.JQuery.com/Main_Page
Documentation JQuery Mobile : http://JQuerymobile.com/demos/1.1.0/

Adaptateur Angular/JQuery Mobile : https://github.com/tigbro/JQuery-mobile-angular-adapter