Colibri

Réseau social orienté autour du partage de fichiers

L. Delafontaine, V. Guidoux, G. Hochet & K. Pradervand

Introduction

Colibri est un petit réseau social orienté autour de la notion de partage de fichiers en permettant aux utilisateurs de liker et commenter l'activité qui s'y déroule et sauvegarder les fichiers qui les intéressent.

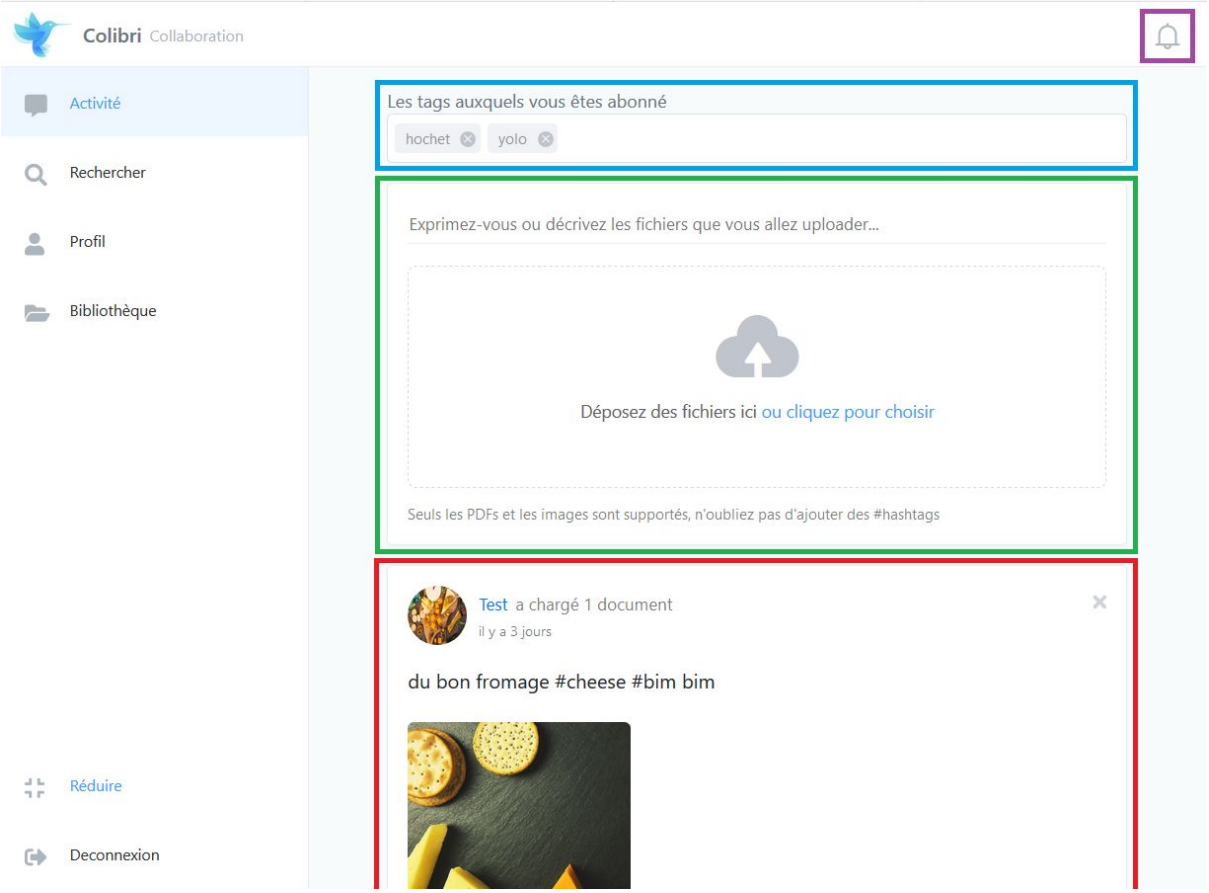
Fonctionnalités

- **Utilisateur** : Inscription et connexion dans l'application, possibilité de mettre à jour ses informations de profil ainsi que sa photo de profil
- **Activité :** Possibilité d'ajouter de l'activité sur le réseau en uploadant des fichiers et en les identifiants par tags. Possibilité de supprimer son activité (pas celle des autres)
- **Fichiers :** Upload de fichiers et identification par tags, possibilité de les visualiser depuis l'application et de les télécharger, support des PDFs et des images. L'affichage des listes de fichiers peut se faire en mode liste ou en mode grille avec prévisualisation
- **Commentaires** : L'utilisateur a la possibilité de commenter l'activité des autres (ou la sienne), de modifier ses commentaires et de les supprimer
- **Like** : Il est possible de liker une activité ou un fichier donné avec plusieurs niveau de like. Plus on reste longtemps appuyé sur le bouton, plus le niveau augmente, passant de "like" à "star" puis finalement à "savior".
- **Dossiers** : L'utilisateur peut créer des dossiers dans lesquels il peut ajouter des fichiers et les en retirer
- **Notifications** : Quand un utilisateur like ou commente l'activité ou les fichiers d'un autre utilisateur, celui-ci reçoit une notification lui indiquant l'action
- **Tags** : Possibilité de tagger des fichiers et des activités. L'utilisateur peut ensuite suivre des tags qui apparaîtront dans son fil d'actualité
- **Informations supplémentaires :** Quand les utilisateurs passent la souris sur un nom de fichier ou sur un nom d'utilisateur avec une popup contenant des détails
- **Prévisualisation de fichiers :** Ouvrir les fichiers en ligne afin de les consulter avec possibilité d'en ouvrir jusqu'à 3 côte à côte

Fonctionnalités de chaque page

L'application offre quatre pages principales

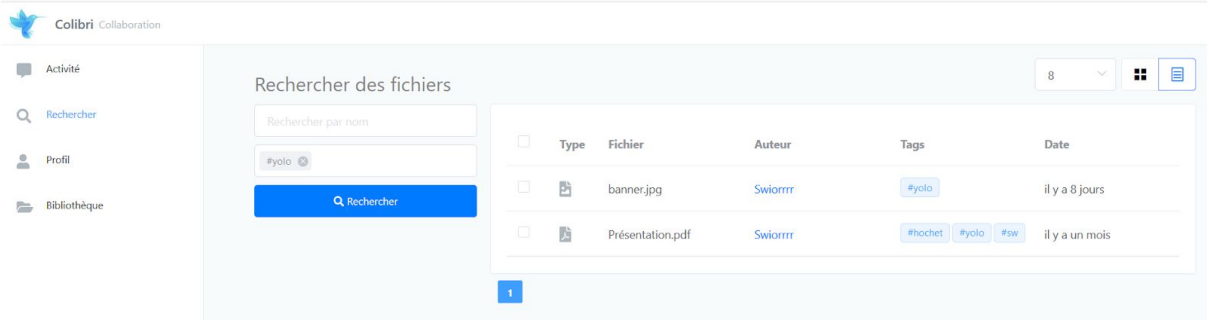
Page activité



La page activité affiche l'activité générale du réseau selon les tags que l'utilisateur suit, c'est là qu'il a la possibilité de les mettre à jour et d'ajouter des fichiers et de l'activité au réseau.

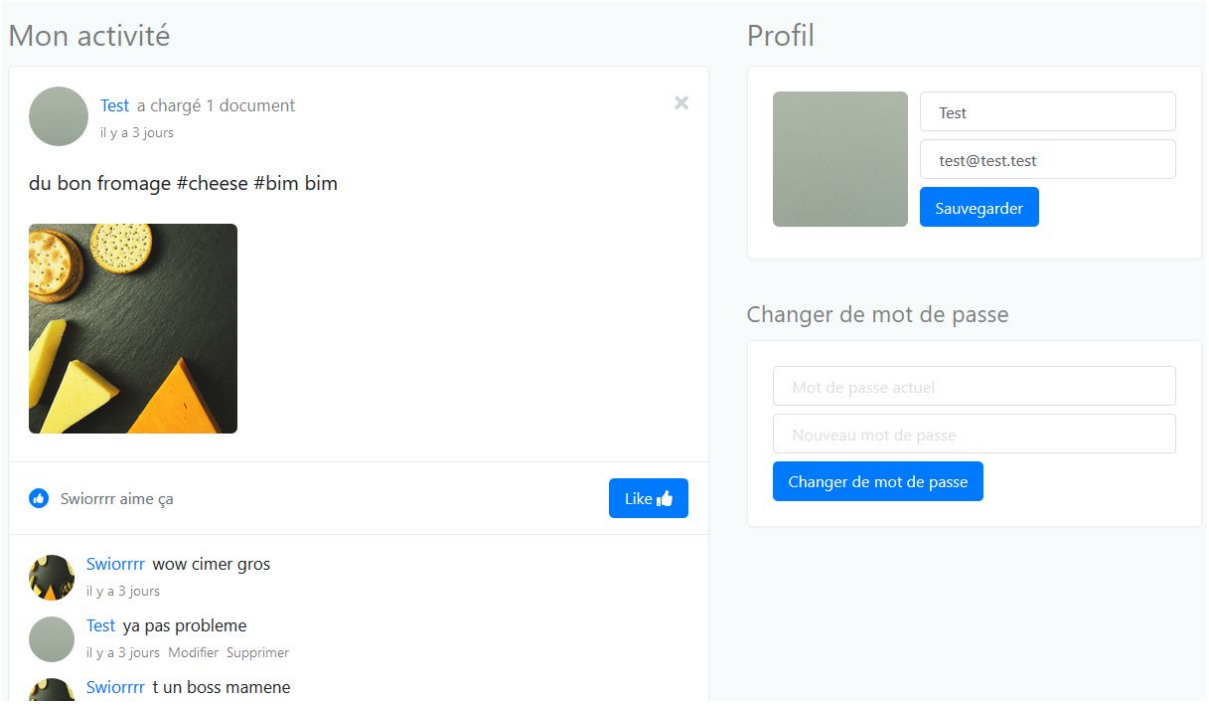
- En bleu ce sont les tags que vous suivez, qui adaptent le contenu du fil d'actualité que vous voyez
- En violet le bouton indiquant si vous avez reçu une notification
- En vert l'espace pour ajouter de l'activité, avec un champ pour décrire l'activité et mettre les tags ainsi que la zone pour déposer ses fichiers
- En rouge le fil d'actualité

Rechercher



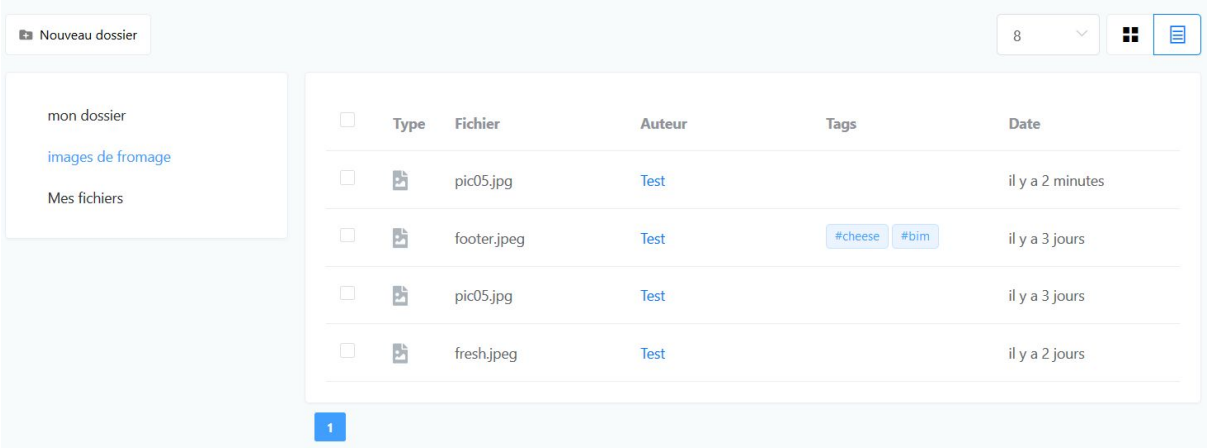
Sur cette page l'utilisateur a la possibilité de rechercher des fichiers selon des tags et le nom du fichier. En haut à droite il y a la possibilité de définir le nombre de fichiers à afficher ainsi que le mode d'affichage, en grille ou en liste, et en bas, les pages de fichiers disponibles.

Profil



Sur cette page l'utilisateur peut voir toute son activité dans la colonne de gauche et interagir avec comme sur son fil d'actualité, et à droite, ses informations de profil et sa photo de profil, qu'il peut changer librement.

Fichiers



Cette page affiche l'ensemble des dossiers créés par l'utilisateur, où il peut en créer de nouveaux et consulter les fichiers qui s'y trouvent, ainsi que tous les fichiers qu'il a lui-même ajouté et rendu disponibles sur le réseau.

Backend

Choix des technologies

- **Node.js :** Nous avons opté pour un serveur Node.js car nous souhaitions mettre en place une API GraphQL dès le début, et Apollo étant présenté en cours et étant l'une des librairies les plus évoluée, le choix s'imposait
- **ArangoDB** : Une base de données relativement récente, nous ne la connaissions absolument pas avant le projet mais elle s'est avérée très performante et adaptée dans notre cas
- **Typescript :** Le driver JS d'Arango est très bas niveau et n'offre aucune sorte de vérification ou quoi que ce soit, et aucune librairie wrapper comme Mongoose pour MongoDB n'existe. Nous avons donc décidé de coder le backend en Typescript afin de bénéficier des vérifications de type ainsi que de la généricité
- **Joi.JS** : Petite librairie permettant de vérifier nos données efficacement avant chaque écriture dans la base de données
- **Apollo Server** : Librairie qui s'est naturellement imposée pour gérer notre API GraphQL efficacement
- **ExpressJS** : Nous avons greffé le serveur Apollo sur une instance d'Express, en effet nous avions besoin d'une et une seule route Express, pour retourner nos fichiers
- **JWT et bcrypt** : Pour les aspects de sécurité

ArangoDB

ArangoDB est une base de données relativement récente supportant les trois modèles répandus en NoSQL: document, clé valeur et graphe. Pour le projet nous avons tenté d'utiliser efficacement l'approche graphe pour structurer nos données mais la documentation officielle est parfois imprécise, et nous ne découvrions des fonctionnalités de la base de données qu'après avoir fait des choix d'implémentation.
L'un des principal défaut que nous avons à lui reprocher est le fait qu'en dehors de la documentation officielle, très peu d'information et d'exemples sont disponibles en ligne. Néanmoins elle est d'une excellente qualité et explique relativement bien les différents cas d'utilisation et les différentes APIs disponibles. Nous aurions dû la lire intégralement une fois avant de commencer à coder, tant son utilisation est différente de MongoDB.

Arango Query Language AQL

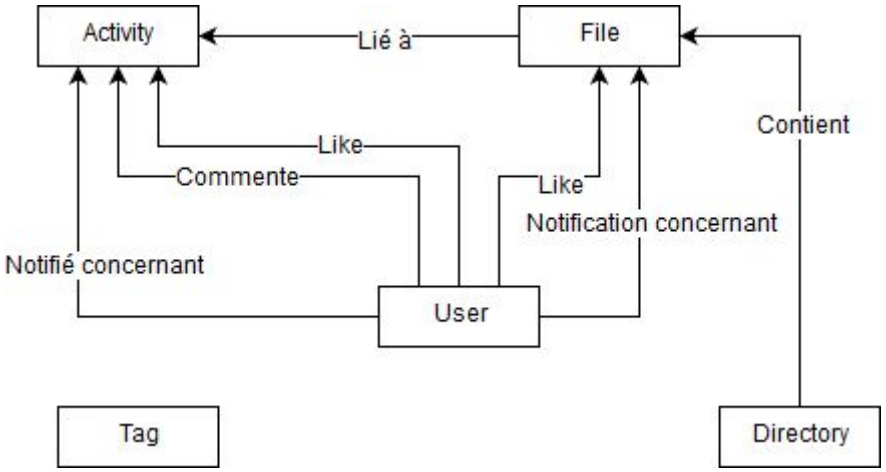
Le langage de requête d'ArangoDB s'appelle AQL et nous pensons qu'il mérite une petite rubrique dans cette documentation tant il est différent des autres langages de requête observés jusqu'ici.
Sa structure ressemble énormément à du code procédural, par exemple récupérer les fichiers ayant le tag "*heig-vd*" se fait ainsi

```
FOR file IN files
     FILTER POSITION(file.tags, “heig-vd”) == true
          RETURN file
```

Qui nous retourne un tableau de fichiers. Sa prise en main a donc été très simple, mais il est plus difficile de réaliser des requêtes plus complexes (comme paginer un sous-ensemble d'une collection mais en même temps retourner le nombre total d'éléments). Néanmoins AQL supporte les sous-requêtes, les jointures et la déclaration de variables dans nos requêtes ainsi que la manière dont sont structurées les valeurs retournées.
Le langage permet donc des requêtes très complexes, beaucoup plus qu'avec MongoDB par exemple, mais elles doivent être optimisées, et sur ce point, la documentation n'est pas très claire.

Modèle

Notre modèle de données se compose ainsi, le schéma est grossier mais permet de représenter la structure en graphe pour laquelle nous avons opté.



Les flèches représentent les arcs de notre graphe ainsi que leur signification, tandis que les rectangles représentent les entités enregistrées dans des collections propres.
L'ensemble des arcs sont eux enregistrés dans une seule grosse collection Edges afin de pouvoir traverser le graphe.
En effet, le langage de requête d'Arango se base sur les jointures pour lier des collections, nous étions donc obligés de mettre tous nos arcs dans une seule collection. Ceux-ci sont ensuite identifiés dans la collection à l'aide d'un attribut spécial "_qualifier" que nous avons défini.
Ce modèle peut être largement amélioré notamment en intégrant le fait qu'un fichier ou qu'une activité soit taggé comme un arc du graphe. Actuellement, les tags sont enregistrés en tant qu'attribut dans les fichiers, les activités et les utilisateurs (pour les tags suivis).

Les avantages de Typescript

Selon une approche multi tiers, nous en avons deux bien distincts. Notre API graphQL, resolvers, mutations et subscriptions sont comprises dans le tier de présentation. Pour le tier d'intégration, nous avons mis en place une structure du type entité-manager en se basant sur les fonctionnalités POO de Typescript. Le tiers business est relativement petit mais nous avons quand même extrait de la logique dans certains services, comme l'Uploader.

Les managers

Chaque entité de notre application (ainsi que chaque type d'arc, like, commentaire…) possède un manager afin de gérer le lien avec le driver Arango. C'est dans ces managers que nous placons nos requêtes spécifiques. Ceux-ci étendent tous de l'AbstractManager qui offre des méthodes CRUD utiles et complètement génériques.

Les entités

Chaque entité et chaque type d'arc est représenté par une interface et une classe afin de les manipuler, nous travaillons au minimum avec des objets javascript génériques.

API

Pour notre API nous avons opté pour GraphQL, tout d'abord parce qu'on avait envie d'expérimenter avec mais également parce que ce modèle se prête très bien aux structures en graphes comme la nôtre.

Structure du schéma

Pour implémenter notre schéma, nous l'avons déclaré dans des fichiers Typescript à l'aide du helper GQL ce qui nous permet de déclarer directement en dessous l'implémentation de nos resolvers.

Notre API

Nous avons fait en sorte que notre API GraphQL soit la plus logique possible et qu'elle retourne des ressources de manière naturelle, comme par exemple récupérer la réputation d'un utilisateur donné, qui représente l'ensemble des likes qu'il a reçu de la part d'autres utilisateurs sur ses fichiers ou activités.

Directives personnalisées

Nous avons implémenté deux directives personnalisées pour notre schéma:

- Capitalize qui permet de mettre la première lettre de chaque mot en majuscule, surtout pour expérimenter avec l'API d'Apollo
- AQL qui nous permet de déclarer des requêtes en AQL directement depuis notre schéma, violant violemment la "separation of concerns" mais qui nous a été très utile pour des resolvers simples. Celle-ci supporte trois variables intelligentes, @current qui représente l'objet courant, @args qui représente les variables d'argument et @context qui représente le contexte donné.

Authentification et contexte

Nous nous sommes appuyés sur le concept de contexte d'Apollo, autant dans les requêtes HTTP standard que dans le cas des websockets et des subscriptions. Pour l'authentification nous nous sommes appuyés sur les JWT afin d'identifier nos utilisateurs simplement. Lors de la réception d'une requête nous récupérons le JWT s'il existe et intégrons l'utilisateur lié dans le contexte Apollo afin de pouvoir l'utiliser dans nos resolvers.

Upload de fichiers

Apollo offre une fonctionnalité spéciale et en plus par rapport à la spécification GraphQL, c'est la capacité d'uploader des fichiers. Nous l'avons donc utilisée afin de supporter l'upload et pouvons confirmer que ça marche très bien. Le problème était ensuite de savoir comment streamer ce fichier au navigateur.

Utilisation d'express

Nous avons été obligé d'utiliser Express pour implémenter une et une seule route, celle permettant de streamer un fichier.

Frontend

Choix des technologies

- **VueJS :** Nous avons opté pour Vue pour la gestion de l'interface parce qu'on a une bonne expérience avec
- **Vue-router :** Pour la gestion des routes
- **VueX :** Pour le cache persisté en local storage
- **Vue-apollo :** Pour l'intégration avec la librairie apollo, qui offre toutes les fonctionnalités nécessaires (composants query, fetch more, smart queries…)
- **ElementUI :** Une librairie de composants très pratique mais avec quelques défauts qui nous ont obligé à travailler le cache apollo plus intensément
- Quelques composants de **Bootstrap** comme la grille ou le layout (padding, flex, margin…) qui ont été intégré dans notre style en utilisant **Sass** dans nos composants

Evolution du développement

Nous avons commencé à coder directement sans avoir trop réfléchi à la structure globale de l'application comme les bons débutants que nous sommes, mais on a eu de la chance dans le sens où nous avions directement bien sub-divisés nos composants. Nous en avons donc finalement un total de 49 qui composent notre application.

Vue-Apollo

Un défaut de vue-apollo c'est que quelqu'un n'ayant jamais fait d'API GraphQL avant et qui souhaite l'utiliser avec cette librairie fera face à un manque de précision et de clarté dans la documentation. Nous n'avons pu exploiter toute la puissance de la libraire que dans les deux dernières semaines, après avoir parcouru la documentation officielle d'Apollo dans tous les sens. Elle offre notamment (au travers de vue-cli, un petit aide en ligne de commande pour gérer ses projets) son installation facilitée, qui nous a fait perdre deux bonnes journées en nous imposant des choix (stockage du JWT en localstorage directement sans influence dessus par exemple). Nous avons fini par la réinstaller manuellement afin qu'elle soit configurée pour nos besoins.
Néanmoins ce qui était à l'origine un fardeau obligatoire s'est transformé en allié de taille en nous donnant accès à des fonctionnalités très pratiques et en s'intégrant parfaitement à Vue.

Gestion du cache Apollo

Nous avons attaqué un peu trop tard la mise à jour du cache et n'avons ainsi pas utilisé apollo-link-state dans notre application, mais nous avons à la place utilisé la manière dont le cache structure ses données. Celui-ci aplatit son arbre en donnant un identifiant unique à chaque ressource qu'il reçoit. Nous avons donc surchargé la manière de récupérer cet ID (notre id de base de donnée qui contient le nom de la collection et la clé unique à l'intérieur). Nos requêtes Apollo nous retournant à chaque fois les identifiants nécessaires, il est devenu simple de mettre à jour le cache là où c'était nécessaire.

Utilisation de Vuex

Vuex est une implémentation à l'image de redux pour Vue. Nous l'avons utilisé afin de manipuler des détails d'UI (minimiser le menu par exemple), ainsi que des données comme le JWT et les persister dans le local storage. Son utilisation n'interfère donc jamais avec le cache Apollo et nous avons pu utiliser les deux côte à côte efficacement.

Structure de nos requêtes

Nos requêtes GraphQL sont stockées dans des fichiers Javascript afin de pouvoir les rassembler selon les ressources et les inclure facilement avec la syntaxe de déstructuration. C'était à la base surtout parce qu'on savait pas comment intégrer des fragments entre fichiers `.gql`, mais finalement notre structure s'est montrée bien pratique alors on l'a gardée.

Déploiement

Il faut savoir qu'Arango étant jeune, le déploiement en ligne a été relativement pénible. Aucun service gratuit n'offre d'hébergement de la base de données. Heureusement pour nous une image docker officielle existe, nous avons donc pu la déployer sur les serveurs du groupe scout à Guillaume, mais elle a tendance à consommer pas mal de RAM du coup dès la fin du semestre elle sera supprimée.

Déploiement en local

1. Téléchargez et installez ArangoDB (arangodb.com), connectez-vous au dashboard et créez une nouvelle base de données
2. Clonez le projet se trouvant à l'adresse https://github.com/sysmoh/mac-tweb
3. Rendez-vous dans `/client` et `/server` et y exécuter "`npm install`"
4. Dans `/server`
    a. Créez un fichier `.env` à l'image du `.env.dist` et mettez-y les valeurs que vous souhaitez
    b. Exécutez "`npm run setup`" qui configurera la base de données
    c. Exécutez "`npm run dev`" qui lancera nodemon et le serveur

5. dans /client
    a. Créez un fichier `env.json` basé sur le `env.dist.json`
    b. exécutez "`npm run serve`" qui lancera le client normalement sur `localhost:8080`
6. rendez-vous sur `localhost:8080` pour commencer à utiliser l'application!

Auto-évaluation

Ce projet était vraiment intéressant car il nous a permis de découvrir et d'expérimenter avec beaucoup de technologies complètement nouvelles, comme Apollo, Typescript et Arango. Etant donné la nature du projet, nous en avons donc profité pour les découvrir et nous faire nos armes dessus tout en utilisant des linters pour garder une certaine clarté dans notre code.

Apprentissage des technologies

Nous avons visé très haut avec ce projet, en voulant y inclure beaucoup de fonctionnalités. Le résultat est à la hauteur de nos attentes, mais nous a obligé à commencer à coder très tôt. Nous avons donc expérimenté avec les technologies au fur et à mesure que le projet avançait, se heurtant à des murs, devant faire marche arrière, refactoriser du code par gros paquets parfois, mais cela nous a permis de bien les maîtriser et nous sommes très confiants dans la réalisation de projets futurs basés dessus.

Fonctionnalités manquantes

Il y a quand même certaines fonctionnalités que nous aurions souhaité implémenter mais que nous n'avons pas fait par manque de temps. Elles ne sont pas compliquées, nous en avons codé des similaires, mais nous avons préféré nous focaliser sur celles difficiles afin d'en apprendre un maximum.

- **Suppression d'un fichier :** On ne peut pour l'instant pas supprimer de fichier qu'on a uploadé
- **Suppression de compte :** À la base nous souhaitions permettre à l'utilisateur de supprimer son compte et de lui laisser le choix de laisser ses fichiers en ligne ou pas, nous n'avons pas eu le temps de le mettre en place
- **Algorithme du "mur" plus développé :** La mise en place du graphe Arango sur nos données qui nous a pris du temps devait permettre ceci, en puissant utiliser les capacités de la base de données à traverser le graphe efficacement, mais AQL n'est parfois pas du tout naturel et nous avons préféré développer d'autres fonctionnalités après un moment à essayer
- **Gestion plus efficace des tags** actuellement les tags que vous mettez dans la description de votre activité sont appliqués aux fichiers, il serait plus logique de pouvoir tagger les fichiers séparément avant upload, et de pouvoir en ajouter ou supprimer après coup, cela n'est pas encore possible
- **Recoder Vue-at** on utilise une petite librairie vue-at qui permet d'afficher des tags possibles dès qu'on entre un "#" dans une activité ou un commentaire.

Cette librairie est terriblement buggée mais il n'existe aucune alternative, on a donc fait au mieux pour l'intégrer mais pas le temps de la recoder

- **Plus de retour utilisateur** il n'y a pas beaucoup d'indications pour l'utilisateur que quelque chose s'est mal passé, on a mis beaucoup de chose et pas eu le temps de toutes bien les finaliser proprement comme on aurait voulu

Tests

On va pas se mentir, il n'y a aucun test unitaire dans notre programme. Mais au final nous n'avons pas beaucoup d'endroits où en mettre, beaucoup de la logique se fait au niveau de la base de données, chaque requête réalisée a donc été testée en utilisation du système et vérifiée (notamment au niveau de la pagination).

Conclusion

Ce projet a vraiment été une chance pour nous de tester plein de technologies dont tout le monde parle et de se les approprier dans un travail d'envergure. Ca a été un peu un projet bac à sable, dans le sens où nous avons appris durant l'implémentation, heureusement pour nous le choix d'une structure établie en avance nous a permis d'éviter d'avoir à trop revenir sur nos pas et de garder un code de qualité. Nous sommes vraiment satisfait du résultat final, autant sur l'esthétique que sur les possibilités qu'offre notre application !

Sources

Le logo utilisé dans le cadre de ce projet est l'oeuvre de Ivan Bobrov dont nous n'avons pas obtenu les droits d'utilisation. De ce fait, ce logo ne sera pas utilisé en dehors de ce projet de cours. Le logo peut être trouvé sur dribbble.com.