Rapport de projet de session
GLO-2005

Xavier Corbeil
111 184 243

Yan Fortin
111 074 944

William Guimont-Martin
111 175 409

L'énonciation du problème et de ses exigences (2 points)

Memer est la nouvelle façon de consommer des *memes*. Il s'agit d'une application permettant de visionner et de voter pour des *memes* à la façon de Tinder. Chaque utilisateur se crée un compte avec lequel il pourra téléverser ses propres *memes*. Les images seront ensuite présentées au reste de la communauté qui pourra les aimer (*upvote*) ou pas (*downvote*) grâce à une interface similaire à ce qu'on retrouve sur la populaire application de rencontres Tinder. Il sera possible de suivre d'autres utilisateurs afin de consulter leur *memes*.

Liste des exigences :

- Un utilisateur peut :
- Se créer un compte
- Supprimer son compte
- Mettre à jour ses informations (Nom d'utilisateur, courriel, avatar, mot de passe)
- Téléverser des images (*memes*)
- Aimer ou pas des *memes*
- Placer et consulter les commentaires associés à un *meme*
- Consulter les meilleurs *memes*
- Suivre d'autres utilisateurs
- Visionner une suite de *memes*
- Un *meme* est constitué :
    - D'une URL (lien vers une image)
    - Un titre
    - Un nombre de mentions « aime » et « n'aime pas »
- Les *memes* ayant atteints plus de 100 mentions j'aime seront ajoutés au top

Spécifications du système et des responsabilités des trois niveaux

Afin de faciliter le développement, les couches du système sont totalement indépendantes. La base de données, le *back-end* et le *front-end* peuvent donc tous être changés de manière indépendante. Par exemple, la base de données SQL pourrait être remplacée par une base de données MongoDB sans jamais avoir besoin de toucher le *front-end* et en ne modifiant que minimalement le *back-end*. Le même principe s'applique aux autres couches.

Nous avons observé le même principe à l'intérieur de chaque couche. Par exemple, pour le *front-end*, tous les appels au *back-end* sont faits via une seule classe. Ainsi, advenant que le *back-end* modifie ses routes, il serait encore une fois très simple d'ajuster le *front-end* en conséquence. L'interface entre le *front-end* et le *back-end* est bien définie et stable.

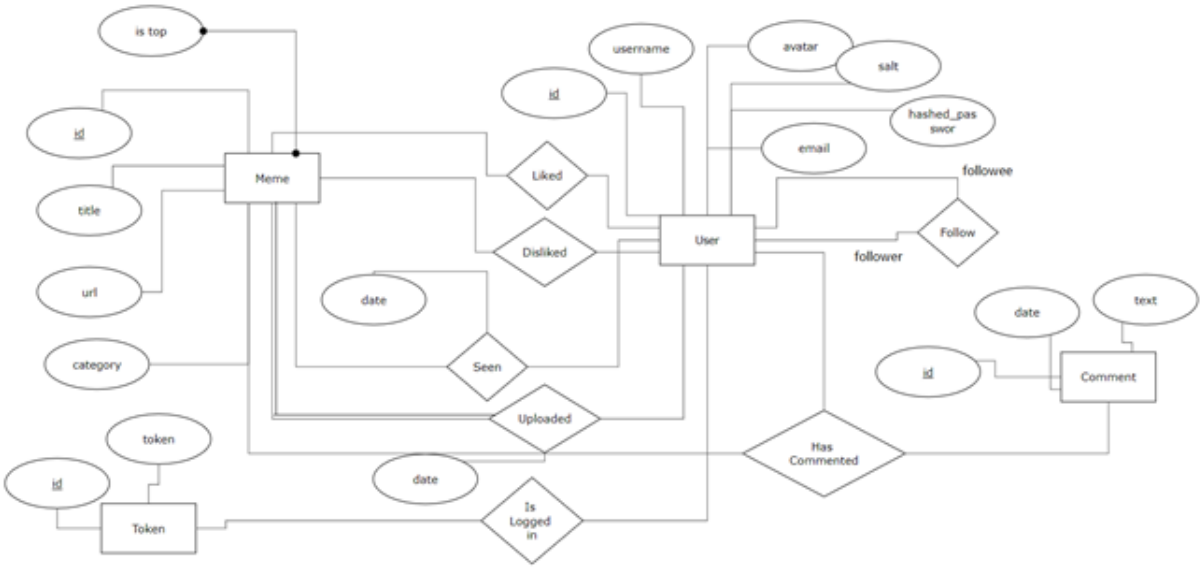
Afin de minimiser les passages des informations d'une couche à l'autre, nous validons autant que possible les informations avant de les transmettre. Par exemple, lors de la connexion d'un utilisateur, nous vérifions que l'adresse courriel entrée en est bien une selon quelques critères. On note que le *back-end* valide aussi la validité des informations.
Concernant la qualité du code, nous nous sommes basés sur les principes de *Clean Code*. Les commentaires sont donc rares, mais nécessaires lorsque présents, les fonctions courtes et les noms de variables explicites.

Le niveau client est responsable d'afficher les informations et de gérer les interactions avec l'utilisateur. Il s'agit d'un client léger qui ne réalise que des validations sommaires sur les données avant de les transmettre au serveur d'application. L'API ne demande que les informations nécessaires afin de réduire les transferts de données inutiles.

Le serveur d'application implémente la logique d'affaires et fait le lien entre l'interface et la base de données. Le serveur d'application prend la forme d'une API REST. Le serveur déclare plusieurs routes HTTP qui permettent d'obtenir, modifier et supprimer les données. Il existe une route pour toutes les actions possibles.

Le serveur de BD s'occupe de la persistance des données et de vérifier les contraintes d'intégrité critiques telles que les références non nulles et les noms d'utilisateur n'est pas vide. Le serveur de base de données implémente quelques gâchettes pour vérifier les contraintes critiques et ajouter, par exemple, des *memes* au top lorsqu'un *meme* a atteint un certain nombre de mentions j'aime. On note aussi que le serveur de base de données s'occupe de supprimer les *token* d'authentification expirée à l'aide d'une procédure qui est appelée périodiquement par le serveur d'application.

Le modèle entité-relation du système (3 points)



Meme : représente une image que l'interface devra afficher.
User : représente un utilisateur qui utilise l'application. On conserve son nom, son avatar, son courriel et les informations pour le mot de passe.
Comment : Commentaire sur un *Meme*. On conserve le titre, son URL, sa catégorie et s'il se trouve dans le top.
Token: il s'agit du token que l'interface utilisateur utilise pour s'identifier avec le serveur d'application. Chaque utilisateur connecté a un *token* qui est transmis à chaque appel au serveur nécessitant des autorisations.
Liked: représente l'action d'aimer un *meme*.
Disliked: représente l'action de ne pas aimer une *meme*.
Seen: représente l'action d'avoir vu un *meme*. La date est enregistrée aussi.
Uploaded: permets de conserver l'auteur d'un *meme*, permettant ainsi de voir les *memes* d'un utilisateur en particulier.
Follow: représente le fait qu'un utilisateur en suit un autre.
Has commented: lie les commentaires avec les *memes*.
Is logged in: lie un token avec l'utilisateur

Le diagramme d'entité-relation représente le domaine de l'application et les besoins techniques tels que les *Tokens*.

On note que *Meme* est en relation totale avec *User* puisque chaque *meme* se doit d'être téléversé par un utilisateur.

Le modèle relationnel du système (3 points)

Le modèle relationnel reprend l'essentiel du modèle entité-relation. Le modèle est détaillé dans *bd_init/init.sql*.

On note que les noms d'utilisateur et les courriels doivent être uniques. L'id, le nom d'utilisateur et le courriel sont uniques, ceci permet de changer le nom d'utilisateur et le courriel sans affecter les relations.

La relation *Follow* fait référence aux *id* des utilisateurs concernés.

Seen, Liked et *Disliked* font référence à l'id du *meme* et de l'utilisateur.

La majeure différence avec le modèle entité-relation est dans la gestion du *Top*. Il s'agit d'une relation qui ne conserve que les id des *memes* qui font partie du top.

Le nom de l'utilisateur ne doit pas être vide puisqu'il s'agit de la façon dont les autres utilisateurs interagissent avec un utilisateur donné.

Les références sont faites de sorte que si l'on supprime ou modifie un *meme* ou un utilisateur, les changements et la suppression se cascadent dans les relations. Ceci permet de garder la base de données cohérente.

L'implémentation et les fonctionnalités du niveau serveur de BD (4 points)

On génère des requêtes MySQL à l'aide d'un script Python. Ces requêtes sont alors copiées dans init.sql ce qui permet de remplir la base de données.

La base de données valide que le nom d'utilisateur n'est pas vide à l'aide d'une gâchette.

La base de données supprime aussi les *token* qui ont expiré avec une procédure qui est appelée périodiquement par le serveur d'application.

L’indexation des données, normalisation des relations, et l’optimisation des requêtes (4 points)

Puisque l’appel le plus fréquent au serveur est la validation de *token* (on valide le token a chaque changement de page de l’utilisateur), un index a été mis en place pour accélérer cette validation. L’index est sur la valeur *token* de la relation *Token* qui est unique et puisqu’il s’agit d’une recherche par égalité, on utilise un index par fonction de hachage.

Pour l’autocomplétion du nom, beaucoup d’appels au serveur sont faits pour avoir les noms qui contiennent la recherche (un appel par touche appuyée pour permettre une recherche interactive). Un index sur du *FULLTEXT* est mis en place sur le *username* de *Users* afin d’accélérer les recherches.

Il ne semble pas y avoir de dépendances fonctionnelles dans les relations puisqu’il n’existe pas de tuple (t1, t2) dans une relation R telle que t1.X == t2.X implique t1.Y == t2.Y. Il n’aurait alors pas de redondance dans la base de données.

L’implémentation et les fonctionnalités de la logique d’affaires (3 points)

Le serveur d’application valide les informations qu’il reçoit de l’interface et indique l’erreur au *front-end*. Comme spécifiés dans *backend/handler/meme_handler.py* et *backend/handler/user_handler.py*, l’application définie plusieurs routes permettant de d’effectuer les actions. Certaines routes demandent d’être authentifiées. L’authentification est réalisée avec un en-tête qui contient le *token*. Ce *token* est ensuite validé et permet d’identifier l’utilisateur.

Ce niveau s’occupe de faire le pont entre l’interface et la base de données. Ce niveau exécute les commandes SQL sur la base de données.

L’API est organisé de façon à suivre les principes REST. La logique d’affaires est séparée dans un service, ce qui permettrait éventuellement de changer la façon d’offrir l’application (par GraphQL plutôt que par REST par exemple). De plus, les services n’ont aucune connaissance de la base de données MySQL.

L'implémentation et les fonctionnalités de l'interface utilisateur (3 points)

L'interface a été réalisée entièrement en Vue.js ainsi qu'avec la librairie de composants Vuetify. Le choix de ce *Framework* s'est principalement fait sur sa facilité de prise en main ainsi que l'expérience des membres de l'équipe. Considérant les contraintes de temps, il était impératif de pouvoir développer un *front-end* rapidement.

Concernant l'interface, nous avons mis l'emphase sur la facilité d'utilisation. L'interface est donc très similaire à celle de *Tinder*. Tout d'abord, l'utilisateur est accueilli par un écran de *Login*. S'il n'a pas de compte, l'inscription se fait via un formulaire facilement accessible.

Une fois connecté, il est directement apporté sur la page permettant d'aimer (ou pas), ainsi que commenter des *memes*. C'est aussi sur cette page que seront présenté les *memes* ayant le plus de mentions « aime » grâce à une barre de menu situé au sommet de la page, il peut naviguer vers une page permettant de modifier ses informations (*My Account*), comme son avatar, son nom d'utilisateur, son courriel ainsi que son mot de passe. C'est à cet endroit que ce trouver l'option pour supprimer son compte.

L'utilisateur peut aussi faire une recherche afin de trouver un autre utilisateur. Une fois un utilisateur sélectionné, la page de son profil sera affichée. Il sera alors possible de le suivre (*follow*), consulter son avatar ainsi que les *memes* qu'il a téléversés.

Finalement, un simple bouton de déconnexion (*Sign out*) interdira de faire des requêtes demandant des autorisations vers le serveur et redirigera l'utilisateur vers la page de connexion.

La sécurité du système (2 points)

Le système suppose que les communications sont réalisées en HTTPS. Considérant que la certification HTTPS est hors de la portée du projet, le système opère pour le moment en HTTP.

Les mots de passe ne sont pas gardés en texte clair pour des raisons de sécurités. Les mots de passe sont plutôt hachés avec du sel qui est différent pour chaque utilisateur. Le sel permet de ralentir le travail de craquage de mot de passe et rend impossibles les attaques par table arc-en-ciel.

Les entrées utilisateur sont nettoyées lors des appels à la base de données pour éviter les injections SQL par la bibliothèque `mysql.connector`. L'interface s'assure d'être protégée contre les injections de JavaScript.

L'application utilise un compte différent de *root* qui permet de changer ses permissions facilement. On laisse à l'application les droits de lecture, d'insertion, de modification, et de suppression des tuples puisque l'application doit effectuer ces opérations.

L’organisation et la gestion de l’équipe, et division des tâches (1 point)

Avant de commencer quoi que ce soit, nous avons fait un *brainstorm* pour trouver un concept, puis nous avons schématisé grossièrement un modèle entité-relation pour s’assurer que nous allions être en mesure de répondre aux critères de correction.

Une fois l’idée choisie, nous avons mis un membre par couche de système à développer (la base de données, le *back-end*, le *front-end*). En procédant à un découpage minutieux de ces couches et des classes les constituant, nous avons pu mettre sur pied les bases du système de manière très rapide. Par la suite, chaque membre a manipulé chacune des couches afin de tout connecter ensemble et repoudre les bogues.

L’équipe se rencontre régulièrement pour se mettre à jour sur l’avancement du projet.

La mise en commun du travail sur le logiciel de gestion de version git sur la plateforme GitHub.

La gestion des tâches est réalisée à l’intérieur de GitHub où l’on s’assigne les tâches.

Voici le lien du GitHub : https://github.com/xaviercorbeil1/BD_Projet.

Pour exécuter le projet, il suffit d’exécuter la commande *docker-compose build && docker-compose up* dans le répertoire de base