# Rapport de Projet WEB

Les Trilobytes
Alex DANDURAN--LEMBEZAT, Mathias DURAND, Maureen LACHAIZE, Sirine HAMDANA
FacebIIkE - Groupe 15

Mardi 12 mai 2020

## Table des matières

# 1 FacebIIkE, le rézo social du turfu

L'ambition de notre équipe était de créer un réseau social par nous-même et pour nous et notre entourage. Quelque chose de, si ce n'est rustique, familial et chaleureux. Ainsi nous avons développé FacebIIkE. La prétention de ce projet n'est pas d'égaler Twitter ou Facebook mais bien de proposer quelque chose de simple et sans grands artifices.

# 2 Approche et mise en place

## 2.1 Objectifs

Pour réaliser notre "rézo social du turfu", nous avons besoin de plusieurs fonctionnalités classiques d'un réseau social :

- La possibilité de créer un compte utilisateur, de s'authentifier et de se déconnecter
- La possibilité d'avoir des amis et de pouvoir voir le contenu qu'ils partagent dans un fil d'actualité
- La possibilité d'interagir avec ses amis dans l'espace *commentaires* de leurs publications ou bien en "*likant* ou *dislikant*" leurs postes.
- Pouvoir personnaliser son profil et notamment changer son nom d'utilisateur, son adresse mail et son mot de passe.
- La possibilité pour un administrateur de modérer le contenu publié pour chaque utilisateur.

## 2.2 Répartition des rôles au sein de l'équipe

Pour assurer la meilleur cohésion possible au sein de l'équipe, nous nous sommes répartis les tâches en fonction de nos affinités personnelles.

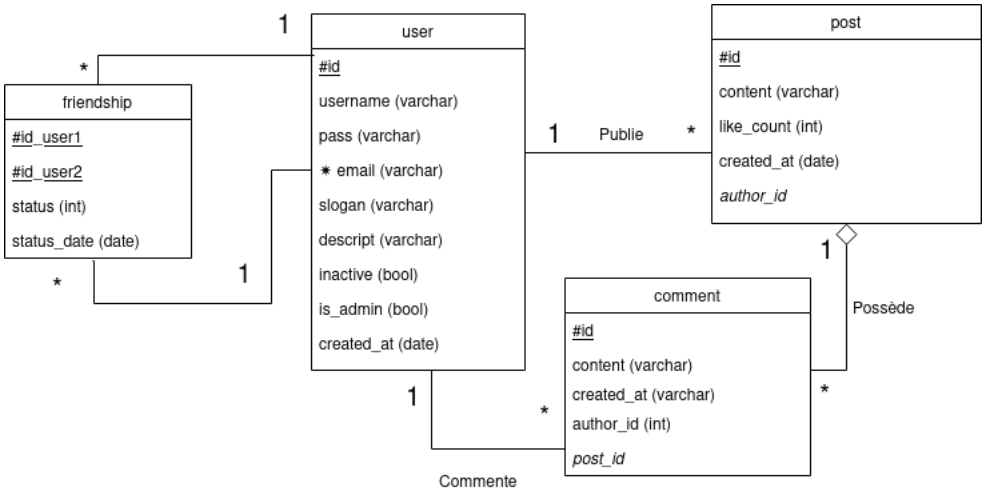
En effet, Maureen Lachaize et Alex Danduran–Lembezat se sont concentrés sur le front-end de FacebIIkE alors que Sirine Hamdana a fait du back-end. Quant à Mathias Durand, grâce à sa polyvalence, il s'est penché sur ces deux aspects.

Du fait de cette répartition et malgré les difficultés liées au confinement, tout le monde a réussi à trouver sa place. La position privilégiée de Mathias a notamment permis une meilleur cohésion et un travail plus efficace.

# 3 Architecture du projet

## 3.1 Base de données

Pour notre base de données, nous avons créé 4 tables comme l'indique le diagramme relationnel suivant :




Grâce à cela, la base de données relationnelle peut ainsi contenir toutes les informations nécessaires pour nous permettre de mener à bien nos objectifs. On remarque que la table *Friendship* est une table de jointure (n...n) symbolisée par la double relation (1...n) entre *Friendship* et *User*.

## 3.2 Architecture des pages : patron MV (Modèle Vue)

Dans le code des pages, il est important de ne pas mélanger l'accès aux données d'une part (Modèle) et l'affichage (principalement HTML et Javascript) d'autre part (Vue). Nous avons donc suivi un patron MV. Les fichiers du Modèle permettent d'accéder aux données de la base (lecture et mise à jour). Ils effectuent tous les traitements sur les données. Ceux de la Vue, quant à eux, satisfont l'affichage client et transmettent les données au format client.

Nous avons choisi de ne pas ajouter de pages dédiées uniquement au traitement de données à proprement parler, à savoir de Contrôleur, car notre application n'a qu'une unique page où sont affichés les postes, commentaires, amitiés, etc... Par soucis de simplicité de l'orchestration entre les différentes parties, nous avons directement appliqué la couche "logique" dans la vue. Nous avons donc adapté le patron MVC en MV pour répondre au mieux à notre besoin.

# 4 Solutions techniques

## 4.1 Gestion du compte d'un utilisateur

### Création d'un utilisateur

Pour créer un compte, un formulaire s'affiche dans lequel le futur utilisateur peut renseigner ses informations. La vérification de ce formulaire - notamment pour vérifier que les deux mots de passe demandés sont bien les mêmes - est réalisée en javascript.

Ce formulaire redirige ainsi l'utilisateur vers *CreateUser.php*. Dans cette page, on va établir une connexion à la base de données à l'aide d'un *dbAdaper* et pour lequel on va récupérer le *UserRepository* faisant le lien avec les utilisateurs.

Si l'utilisateur est déjà enregistré dans la base, on le redirige vers *index.php* où lui sera indiqué l'erreur en question. Sinon, on l'ajoute à la base de donnée grâce à la fonction *createUser* de *UserRepository.php* et une requête SQL avec son nom d'utilisateur, son mot de passe (hashé, cf section "Sécurité") et son adresse mail. Dans la base de données, les clés primaires de *User* sont générées en série donc elle n'est pas précisée ici.

Une fois ceci effectué, l'utilisateur est enfin libre de se connecter grâce au formulaire de connexion.

### Connexion d'un utilisateur

En arrivant sur l'index du site, on vérifie si les variables de session *id*, *pseudo* et *email* sont bien initialisées (l'utilisateur est déjà connecté au site). Si c'est le cas, on redirige l'utilisateur vers la page *home* qui affiche dynamiquement les informations (comme les amis, les postes, etc.) en fonction de l'utilisateur connecté.

Si l'utillisateur n'a pas de cookie de session, il lui suffit de se connecter grâce à un formulaire. Alors est appelée la fonction *connectUser* qui compare le mot de passe rentré au mot de passe hashé stocké dans la base de données. Une requête SQL permet ainsi de récupérer toutes les données d'un utilisateur, d'initialiser sa session ainsi que les variables qui lui sont associées.

### Déconnexion d'un utilisateur

Quand un utilisateur demande une déconnexion en appuyant sur le bouton prévu à cet effet, *DeconnectUser.php* s'occupe de supprimer la totalité des variables de session et des cookies de connexion automatique. Il est finalement redigiré vers *index.php*.

### Désactivation d'un compte utilisateur

Un utilisateur peut, s'il le souhaite, désactiver son compte. Ceci aura pour effet de rendre son profil inaccessible (en essayant d'y accéder, il sera vérifié en amont si oui ou non l'utilisateur a désactivé son compte) et il lui sera demandé, dans le cas où il voudrait accéder à n'importe qu'elle page, de réactiver son compte.

Une désactivation de compte consiste à cacher tous les postes d'un utilisateur et d'anonymiser les commentaires dont il est l'auteur. Nous avons préféré cacher les informations plutôt que de les supprimer pour éviter les effets de bords et ne pas avoir à gérer des suppressions en cascade qui peuvent être, si mal gérées, hasardeuses. D'autres réseaux sociaux comme Reddit par exemple choisissent aussi de ne pas supprimer les commentaires des comptes supprimés.

Le choix de désactiver son compte va faire lancer une requête de type *UPDATE* dans la base de données pour passer l'attribut *inactive* à *true*.

## 4.2 Système d'amitié

Dans le "turfu" nous pourrons aussi être amis ! C'est pourquoi il nous tenait à coeur de proposer dans notre "rézo social" FacebIIkE un système d'amitié.

Pour cela, nous avons créé dans les *Model* une classe *Friendship*. Le couple formé par les deux clés étrangères *idUser1* et *idUser2* en est la clé primaire dans la modélisation en base de données correspondante.

La variable *status* a deux valeurs possibles : elle est à 0 si l'amitié est en attente de validation et à 1 si elle est validée par l'utilisateur qui a reçu la demande. Pour passer du statut "en attente" (*pending*) au statut "accepté" (*accepted*), on utilise la fonction *update* en passant le statut à 1 et en actualisant la date.

#### Les demandes d'amis

Cette implémentation permet aussi d'orienter les demandes d'amitié : si un utilisateur envoie une demande d'ami à un autre, alors c'est sa clé étrangère qui se trouve en premier dans le couple (clé primaire de la table *friendship*) et au contraire s'il la reçoit, sa clé est donc nécessairement en deuxième position. Grâce à ce système, nous pouvons ainsi faire en sorte que l'utilisateur puisse distinguer s'il est à l'origine ou non de la demande.

Pour effectuer une demande d'ami, il faut que l'on tape un nom d'utilisateur (le début du nom suffit) dans le champ de recherche. Cela va simplement comparer la chaîne de caractère rentrée avec tous les noms de la base de donnée et n'afficher que ceux pour lesquels il y a correspondance.

#### Voir les publications de ses amis

Une fois la relation d'amitié établie, l'utilisateur peut voir le profil de son ami et la liste de ses postes. Pour faire cette vérification, on récupère la liste d'amis de l'utilisateur qui visite le profil d'un autre avec la fonction *fetchFriendsByUserId* qui fait une requête SQL avec une jointure entre la table *User* et la table *Friendship*. Il ne reste plus qu'à vérifier si les deux utilisateurs sont bien amis.

## 4.3 Gestion du profil utilisateur

Chaque utilisateur possède un profil dans lequel on peut y retrouver ses postes, son slogan et une brève description qui sont tous deux personnalisables. Pour récupérer les postes, il suffit de faire une liaison à la base de données avec *dbAdaperFactory()* et de tous les récupérer (récupérer aussi les commentaires) pour ensuite les afficher un par un.

Pour personnaliser son slogan et sa description, l'utilisateur peut accéder à ses paramètres. Dans ces paramètres, on peut également modifier son nom d'utilisateur, son email et son mot de passe. Le système de fonctionnement est le même : on appelle une méthode du *userRepository* qui va réaliser un *UPDATE* dans la base de données.

## 4.4 Les publications et les commentaires

#### Publier un poste et un commentaire

Pour publier un poste, on rentre le contenu dans le formulaire adéquat. Au moment d'appuyer sur le bouton pour publier, on récupère le contenu dans une méthode *POST*. Sera alors créée une nouvelle instance de la classe *Post* dans laquelle on va renseigner l'utilisateur qui a créé le poste, le contenu ainsi que la date et l'heure à laquelle il a été envoyé. Il suffit après de créer ce poste dans le *postRepository* lié à la base de données.

La publication d'un commentaire suit rigoureusement le même principe.

#### Afficher les postes et les commentaires

Dans le fil d'actualité sont affichés les postes des amis ainsi que les commentaires associés. Grâce au *dbAdaper* on crée le lien avec la base de données et on récupère l'ensemble des postes. Pour chacun des postes, on va vérifier si l'auteur n'a pas désactivé son compte et s'il fait partie de nos amis (on utilise la fonction *areFriends* du *friendshipRepository*).

Pour chaque poste à afficher, on va afficher de la même manière les commentaires associés. On prend également en compte si oui ou non l'auteur d'un commentaire a désactivé son compte en affichant "Inconnu" à la place de son nom d'utilisateur.

#### *Liker* et *Disliker* des postes

FacebIIkE permet aux utilisateurs de *liker* et même *disliker* des postes. Le principe est simple : sur n'importe quel poste s'affiche un bouton *Like* et un bouton *Dislike*. Appuyer dessus va lancer le script *Like.php* qui va se connecter à la base de donner et incrémenter de *1* ou *-1* l'attribut *like_count* de la base de données grâce à une requête SQL de type *UPDATE*. Ainsi, nous avons laissé le choix d'avoir un *like_count* négatif, chose qui, nous trouvons, manque cruellement sur bien des réseaux sociaux.

### 4.5 Gestion des administrateurs

Un administrateur est un utilisateur dont l'attribut booléen *is_admin* dans la base de données est à *true*. Nous avons choisi cette implémentation plutôt qu'une autre car nous n'avons pas la nécessité d'avoir, par exemple, un rôle modérateur ou d'autres rôles étant donné la taille de ce projet.

#### Supervision de tous les postes

Un administrateur a la possibilité de voir la totalité des postes de chaque utilisateur mais aussi d'avoir une vision plus globale sur son profil où il peut voir tous les postes publiés jusque là (qu'il soit ou non ami avec celui-ci).

Pour réaliser cela, on récupère tous les postes de la base de données de la même manière que pour un utilisateur normal sauf qu'ils ne sont pas filtrés de la même manière : là où pour un utilisateur lambda on n'affiche que les publications de ses amis, pour un administrateur on ne s'embarrasse pas de cette condition.

On utilise alors la variable de session *$_SESSION['admin']* pour déterminer si l'utilisateur est un administrateur ou non.

#### Suppression des postes

Les administrateurs ont également le privilège de pouvoir supprimer des postes. Pour ce faire, un bouton "*Supprimer le post*" qui redirige vers *DeletePost.php* est présent pour les administrateurs dans la publication.

*DeletePost.php* établie une connexion à la base de données avec le *dbAdaper* et, si l'administrateur a bien appuyé sur le bouton, lance la fonction *deleteByID*. Dans cette dernière, une requête *DELETE* est envoyée à la base de données pour le poste dont on a renseigné l'identifiant. Ce même processus est alors appliqué aux commentaires du poste en question.

### 4.6 Sécurité

#### Formulaire d'inscription

Dans le formulaire d'inscription d'un nouvel utilisateur se trouvant dans *index.php* est utilisé du Javascript pour s'assurer que le mot de passe et le champ de vérification de celui-ci sont identiques. Cependant, certains navigateurs et utilisateurs peuvent désactiver Javascript. Le risque serait donc d'entrer des mots de

passe différents.

Pour prendre en compte ceci, nous effectuons une seconde vérification dans *CreateUser.php* dans le langage php pour s'assurer de la validité des champs renseignés.

**Stockage et vérification du mot de passe**

Le mot de passe est la donnée la plus sensible que puisse avoir un utilisateur. S'il était stocké en clair dans la base de données et qu'un individu mal intentionné parvenait à y accéder (même les plus grands sites peuvent voir ce genre de données fuiter, personne n'est à l'abri de cette éventualité), alors ce pourrait être un désastre pour l'utilisateur.

Ainsi, il en revient de la responsabilité des gestionnaires du site de s'assurer que les mots de passe sont stockés de la manière la plus sécurisée possible. Une solution pour s'en assurer est de "hasher" un mot de passe et stocker cette version hashée dans la base de données. En effet, une fonction de hashage est une fonction bijective (pour une entrée n'existe qu'une sortie possible et pour une sortie donnée, il n'existe qu'une seule entrée qui puisse la produire) qui permet d'associer à une chaîne de caractère une autre chaîne de caractère très longue, inintelligible et surtout unique. Ainsi, à un mot de passe donné n'est associé qu'un seul mot de passe hashé.

Dans la fonction *createUser* de *UserRepository.php*, le mot de passe de l'utilisateur est donc hashé grâce à la fonction *password_hash* avant d'être stockée. La fonction *password_hash* utilise un algorithme de hashage fort et irréversible.

Pour vérifier à la connexion que le mot de passe renseigné par l'utilisateur correspond bien au mot de passe hashé stocké dans la base de données, il suffit d'utiliser la fonction *password_verify* qui prend le mot de passe rentré par l'utilisateur, le hash et compare cette version hashée avec celle enregistrée dans la base de donnée.

**Protection contre les injections SQL**

Une première protection contre les injections SQL est la préparation de requêtes. Nous l'avons fait en utilisant la fonction *prepare* pour l'ensemble des requêtes qui le nécessitaient. Préparer une requête permet de vérifier la syntaxe, et initialiser les ressources internes du serveur pour une utilisation ultérieure de cette requête.

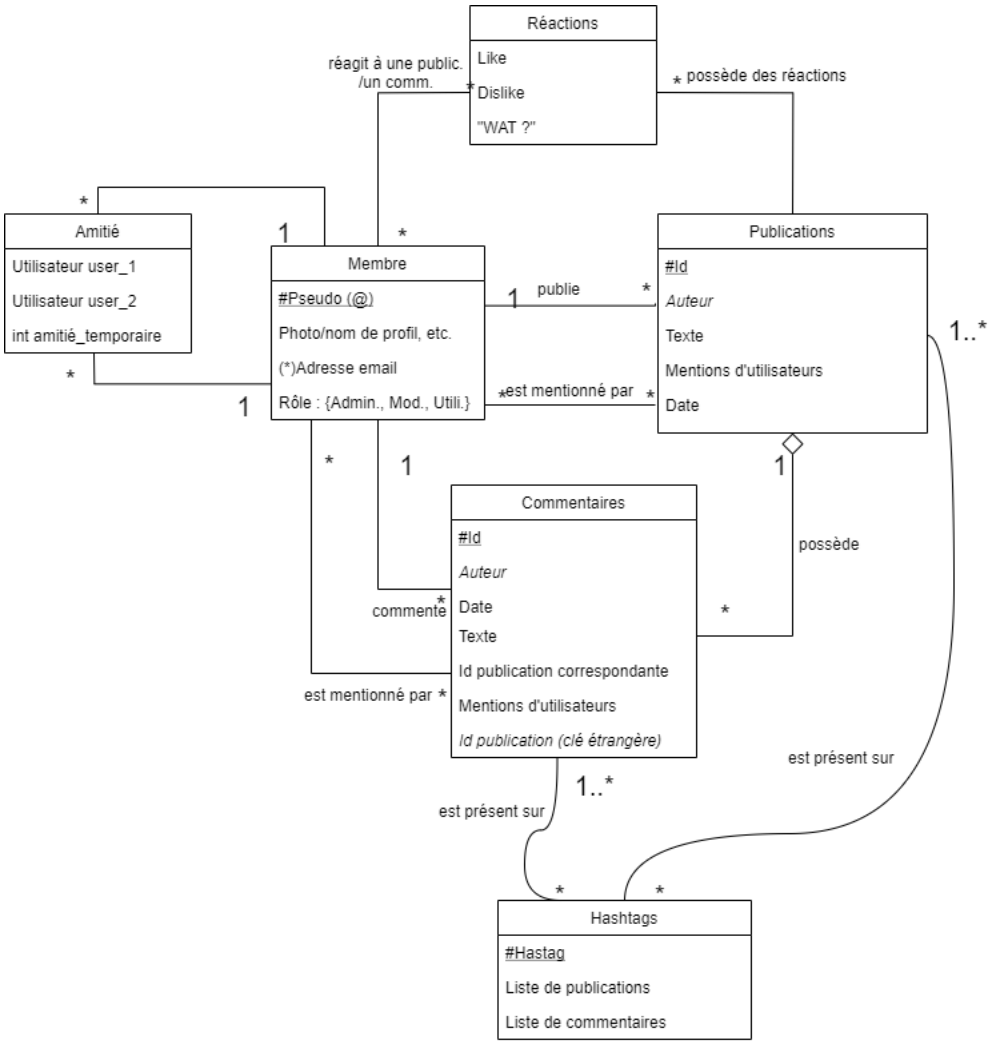
Même si l'avantage principal de la préparation de requêtes reste de ne pas avoir à ré-appliquer cette requête à chaque appel mais de l'exécuter en mettant seulement à jour les variables concernées, la vérification de syntaxe qu'elle applique permet de formater les requêtes et d'empêcher ainsi l'injection d'un autre requête possiblement malicieuse.

# 5 Difficultés rencontrées

## 5.1 De trop grandes ambitions

En commençant à réfléchir sur le projet, nous avions beaucoup d'ambitions vis-à-vis de ce que l'on souhaitait réaliser. Assez rapidement, nous sommes tombés d'accord sur cette idée de réseau social "fait maison" avec le petit jeu de mot ("bIIkE") faisant référence à la fois à l'ENSIIE et à nos parents qui, vous avez déjà dû l'entendre, parlent de Facebook en ces termes : "Face de bouc".

Ainsi, au commencement nous avions bien des idées et des fonctionnalités à ajouter en tête. Pour l'illustrer, voici le premier diagramme UML que nous avions créé :




Malheureusement, nous n'avons pas pu réaliser toutes ces idées comme les mentions ou la table de réaction dans le temps imparti et le contexte de vie particulier qu'est le confinement.

## 5.2 Installation et configuration de pgsql selon les architectures

Deux membres du groupe travaillent sous Arch Linux, un sous Ubuntu et l'autre sous Windows avec WSL. Nous avons mis beaucoup de temps au tout début du projet pour réussir à faire fonctionner pgsql sur tous nos systèmes. Sur Arch, notamment, la procédure est assez complexe car certaines configurations ne se font pas automatiquement. Enfin, sur WSL, la configuration des ports et la communication du serveur avec Windows était à réaliser manuellement en naviguant dans les fichiers de configuration.

Récupération des données utilisateur

Une difficulté qui s'est posée était notamment pour récupérer les données des utilisateurs. En effet, de la trop faible communication entre les *Repositories* découle la nécessité d'effectuer des appels en chaîne sur des *Repositories* dont les formats de paramètres diffèrent souvent. De ce fait, nous avons eu beaucoup d'erreurs et quelques difficultés à interagir avec eux.

### 5.3 Gestion des administrateurs

L'implémentation que nous avons choisie pour les utilisateurs administrateurs s'est trouvée être en réalité très peu optimisée. En effet, la qualité d'administrateur est renseignée dans un attribut booléen de la table *User*. Pour permettre aux fonctionnalités d'administration de fonctionner, il était donc nécessaire de modifier toutes les pages dans lesquelles il devait y avoir du contenu particulier pour l'administrateur. Ainsi, ce choix a rendu le développement des outils d'administration très laborieux.

### 5.4 La fonction *fetchByName*

Le fonction *fetchByName* avait initialement pour ambition de récupérer la donnée d'un utilisateur grâce à son nom d'utilisateur. Pour protéger et optimiser la requête SQL, nous avons utilisé la méthode *prepare*. Nous avons alors remarqué que pour rechercher une chaîne de caractères (ici un nom d'utilisateur) dans la base de données, elle doit être entourée de *simple quote* (’). Cependant la fonction *prepare* en formattant retirait les apostrophes et la requête était donc inutilisable car elle générait une erreur.

Pour résoudre ce problème, nous avons choisi de récupérer tous les utilisateurs et de comparer chacun de leur nom avec celui recherché grâce à la fonction *strncmp*. Si les chaînes sont les mêmes, on récupère toutes les informations sur l'utilisateur.

### 5.5 Ajouter un panel de sélection d'émoticon

Nous avons essayé d'ajouter un panel de sélection d'émoticon pour que les utilisateurs puissent réagir à leur convenance aux publications des autres. Cependant, cela nécessitait l'utilisation de Javascript. Le panel s'affichait bien mais on ne pouvait pas sélectionner d'émoticon. Aucune solution que nous avons essayé d'implémenter n'a fonctionné.

## 6 Conclusion

Ce projet a permis à notre équipe de travailler ensemble sur un second projet de grande envergure et de développer ainsi un certain savoir-faire. En effet, nous comprenons mieux les enjeux des projets en équipe et les dynamiques qui se créent au sein d'un groupe de collaborateurs.

De plus, nous avons pu approfondir nos connaissance en développement WEB en réussissant à créer de toutes pièces un site dynamique s'appuyant sur une base de données. C'est, pour nous, une grande percée qui nous a permis de devenir meilleurs développeurs.