

# SOMMAIRE

## TABLES 🙌

<b>USER</b>	<b>2</b>
<b>TWEET</b>	<b>6</b>
<b>RETWEET</b>	<b>8</b>
<b>HASHTAG</b>	<b>8</b>
<b>MESSAGE</b>	<b>9</b>
<b>FOLLOW</b>	<b>11</b>
<b>TABLES BONUS</b>	<b>12</b>
BLOCK_USER	12
REPORT	14
IMPRESSION	15
LIKES	16
BOOKMARK	17
COMMUNITY	18
USER_COMMUNITY	19

## USER

- **id** integer [auto\_increment, not null] :  
Identifiant unique de l'utilisateur
- **role** varchar(255) [default: 'user', null] :  
Rôle de l'utilisateur (Admin/user/ect..)
- **firstname** varchar(255) [not null]
- **lastname** varchar(255) [not null]
- **username** varchar(255) [not null, unique] :  
nom du compte avec un "@" au début
- **display\_name** varchar (255) [null] :  
nom affiché sur le profil non unique
- **email** varchar(255) [not null, unique]
- **password** varchar(255) [not null] :  
mot de passe du compte à enregistrer haché au format ripemd160
- **birthdate** date [not null] :  
date de naissance avec ou sans l'heure
- **phone** varchar(255) [null]
- **url** varchar(255) [null] :  
potentielle url du site que la personne veut avoir sur son profile
- **biography** varchar(255) [null]:  
Description présente sur le profil.

- **city** varchar(255) [null]
- **country** varchar(255) [null]
- **genre** varchar(255) [null]
- **picture** varchar(255) [null] :

Chemin vers la photo de profil (de préférence stockée localement).  
A l'aide de la superglobale \$\_FILES récupérer le nom du fichier uploadé, enregistrer une copie dans un dossier (cf php), stocker le chemin dans la db.

ex: [www.localhost:8000/app/download/picture.jpg](http://www.localhost:8000/app/download/picture.jpg)

Attention à récupérer le bon chemin sinon ça ne fonctionnera pas !  
cf chemin absolu/relatif.

- **header** varchar(255) [null] :

Chemin vers la bannière (de préférence stockée localement).  
(explication cf picture)

- **NSFW** boolean [Par défaut: false, null] :

Un boolean pour dire si l'utilisateur veut ou non voir des contenus sensibles.

- **is\_active** boolean [Par défaut: true, not null] :

Permet de rendre un compte inactif, avant de le supprimer complètement de la db.


Par défaut lors de la création d'un utilisateur la valeur sera vraie.  
On peut la modifier en false si l'utilisateur veut désactiver son compte.

Pensez à modifier inactive\_date par la date du jour

ex: Lors de la connexion vérifier si le compte est inactif, si `is_active == true`, rediriger vers une page pour réactiver le compte (UPDATE) puis connexion.

- **is\_verified** boolean [Par défaut: false, not null] :

Permet de vérifier si un compte est certifié ou non.

ex: en PHP afficher une  si le résultat de la requête pour `is_verified == true`.

- **ban** varchar(255) [Par défaut: null] :

Identifiant du ban appliqué à l'utilisateur.

Permet de vérifier si l'utilisateur est banni de la plateforme (si `!= null` alors banni).

`user.id_ban` est lié à `ban.id`, pour récupérer le motif de ban associé:

```
SELECT ban.name AS 'Motif' from user INNER JOIN ban ON
user.id_ban = ban.id WHERE user.id = ...;
```

Pensez à ajouter une condition à toutes vos requêtes utilisateur pour éviter d'afficher les utilisateurs ayant un ban:

Exemple : Toto est banni, il aura donc un id dans la colonne `id_ban` de la table `user` correspondant à sa ligne.

Si on souhaite afficher tous les users sont Toto (qui est ban) on rajouter cette condition à notre requête:

```
WHERE user.id_ban = null;
```

- **creation\_date** date [default: current\_timestamp; null] :

Date de création du compte.

- **verified\_date** date [default: null] :

Date de vérification du compte. (  )

- **inactive\_date** date [default: null] :

Date de désactivation du compte. Permet de vérifier si au bout de 7/20/30/... jours le compte peut être supprimé de la base de donnée définitivement (DELETE).

- **verification\_code** varchar(6) [null]:

Code de vérification à envoyer à l'utilisateur par mail pour valider la récupération du message.

## TWEET

- **id** int [primary key, increment] :
- **id\_user** integer [not null] :

Identifiant de l'utilisateur qui poste le tweet.

`tweet.id_user` est lié à `user.id` .

Récupérer l'id/nom/prénom/username de l'utilisateur via une jointure ou récupérer tous les tweet de cet utilisateur avec une condition WHERE user.id = ...;

- **reply\_to** integer [null] :

Identifiant du tweet auquel l'utilisateur répond (ce tweet n'est pas une réponse si l'id est null).

- **quote\_to** integer [null] :

id du tweet que ce tweet cite ( ce tweet n'est pas une citation si l'id est null)

- **NSFW** boolean [default: false, not null] :

Définit si le tweet contient du contenu sensible ou non avec un boolean. (modifier après plusieurs signalements par exemple)

- **content** varchar(140) [not null] :

Le contenu du tweet/réponse/commentaire (limité à 140 caractères, ajouter une limitation au front de votre app aussi pour que l'utilisateur sache quand il a dépassé le nombre de caractères autorisé).

- **creation\_date** datetime [default: current\_timestamp; null] :

Date de création du tweet.

- **is\_pinned** boolean [not null, default: false] :

Définit si le poste est épinglé au profile. Par défaut aucun post n'est épinglé, un seul poste peut être épinglé par utilisateur.

S'il existe déjà un utilisateur qui a épinglé un tweet dans la table tweet, il faudra modifier is pinned de ce tweet avant de modifier le post suivant.

Exemple :

TWEET id 1 est épinglé, on souhaite épingler TWEET id 2 pour l'user à l'id 5:

On vérifie si un post est épinglé pour l'user 5 dans la table :

```
SELECT tweet.id, tweet.is_pinned FROM tweet WHERE id_user = 5;
```

Si un résultat > 0 alors on modifie le tweet:

```
UPDATE tweet SET is_pinned = false WHERE tweet.id = 1 and id_user = 5;
```

Puis on update le POST 2:

```
UPDATE tweet SET is_pinned = true WHERE tweet.id = 2 and id_user = 5;
```

- **is\_community** boolean [not null, default: false] :

Détermine si un tweet appartient au sein d'une communauté. Par défaut false = visible en dehors de la communauté.

- **media** (1-4) varchar(255) [null] :

Chemin vers l'image. Cf explication user.picture = même principe mais on associe un chemin à un tweet. Il peut y avoir plusieurs images qui auront le même id\_tweet (4 images max/tweet).

## RETWEET

- **id\_tweet** integer [primary key] (clé composite)
- **id\_user** integer [primary key] (clé composite)
- **creation\_date** datetime [default: current\_timestamp; not null]

## HASHTAG

- **id** integer [primary key, auto increment] :

Identifiant unique d'un hashtag.

- **name** varchar(255) [not null] :

Nom d'un hashtag. Utile pour l'auto-complétion, à faire à partir de 3 caractères.

On entoure les #... par une balise <a href=""> en JS et lorsqu'on clique sur un # on récupère tous les tweets et on procède à une sélection avec une condition (cf regex) pour garder que les tweets qui ont ce hashtag spécifique.



## MESSAGE

- **id** integer [primary key] :

Identifiant unique pour chaque message.

- **content** varchar(255) [not null] :

Le contenu du message.

- **id\_sender** integer [primary key] : (clé composite)

Identifiant de l'utilisateur qui envoie le message.

message.id\_sender est lié à user.id .

- **id\_receiver** integer [primary key] : (clé composite)

Identifiant de l'utilisateur qui reçoit le message.

message.id\_receiver est lié à user.id .

- **date** datetime [default: current\_timestamp; null] :

Date et heure de l'envoi du message.

- **is\_hidden** boolean [default: false, not null] :

Boolean qui permet de définir si le message doit être caché ou non. Par défaut = false, si on souhaite effacer le message il faudra faire un update de is\_hidden = true et ajouter une condition en back où si is\_hidden == true alors afficher 'message indisponible' par exemple à la place du contenu.

- **is\_viewed** boolean [default: false, not null] :

Boolean qui vérifie si le message a été vu. Par défaut = false.

Si l'on souhaite afficher les messages entre Toto (id 1 de la table user) et Nana (id 2 de la table user) :

```
SELECT * FROM message INNER JOIN user ON  
message.id_sender = user.id INNER JOIN user ON  
message.id_receiver = user.id WHERE (message.id_sender = 1  
OR message.id_sender = 2) AND (message.id_receiver = 2 OR  
message.id_receiver = 1) ORDER BY date ...;
```

- **media** varchar(255) [null] :

Chemin vers une image. Cf explication user.picture = même principe.

## FOLLOW

On récupère les abonnements et les abonnés :

- **id\_user\_follower** integer [primary key]: (clé composite)

Identifiant de l'utilisateur qui suit un autre utilisateur.

`follow.id_user_follow` est lié à `user.id` .

- **id\_user\_followed** integer [primary key]: (clé composite)

Identifiant de l'utilisateur qui est suivi par `id_user_follower`.

`follow.id_user_followed` est lié à `user.id` .

Exemple :

Toto (`user.id = 1`) suit (follow) Nana (`user.id = 2`), dans ses abonnements (following) il y a le compte de Nana et inversement Nana peut voir le compte de toto cité dans ses followers.

Pour récupérer **les followers** de Nana avec des jointures:

```
SELECT user.username FROM follow INNER JOIN follow ON
user.id = follow. id_user_follow INNER JOIN follow ON user.id =
follow. id_user_followed WHERE id_user_followed (la personne suivie) = 2 (id de Nana) ; // → retourne Toto
```

Inversement pour récupérer **les following** de Nana:

```
SELECT user.username FROM follow INNER JOIN follow ON
user.id = follow. id_user_follow INNER JOIN follow ON user.id =
follow. id_user_followed WHERE id_user_follow (la personne qui suit) = 2 (id de Nana) ; // → retourne tous les comptes suivis par Nana
```

La même chose peut être fait pour Toto en changeant l'id de l'utilisateur après le WHERE.

## TABLES BONUS

### BLOCK\_USER

- **id\_user** integer [primary key]: (clé composite)

Identifiant de l'utilisateur qui bloque un autre utilisateur.

`block_user.id_user` est lié à `user.id` .

- **id\_blocked\_user** integer [primary key]: (clé composite)

Identifiant de l'utilisateur qui est bloqué par `id_user_follower`.

`block_user.id_blocked_user` est lié à `user.id` .

Maintenant on souhaite cacher un profil à un autre utilisateur.

Exemple : Toto a bloqué Nana (`user.id = 2`).

Du côté de nana on va vérifier si elle est bloqué par des utilisateurs :

```
SELECT id_user FROM block_user WHERE id_blocked_user = 2;
```

On récupère un tableau contenant tous les id des utilisateurs qui ont bloqué Nana ( donc Toto le rageux).

La même chose peut être fait pour récupérer les comptes à cacher à Nana car elle les a bloqué :

```
SELECT id_user FROM block_user WHERE  
block_user.id_blocked_user = 2 OR block_user.id_user = 2;
```

Si on souhaite afficher le profil de tous les users sauf Toto (qui a bloqué Nana) on rajoute cette condition à notre requête:

```
SELECT * FROM user WHERE user.id NOT IN (1 pour Toto, 3 pour  
Chichi par exemple etc... tous les id des users qui ont bloqué nana  
ET/OU que nana a bloqué);
```

## REPORT

- **id\_tweet** int [primary key] : (clé composite)

Identifiant du tweet signalé.

- **id\_user** integer [primary key] : (clé composite)

Id de l'utilisateur qui a signalé.

- **description** varchar(255) [null] :

Description du signalement.

- **creation\_date** datetime [default: current\_timestamp; null] :

Date de création du signalement.

## IMPRESSION

- **id\_tweet** int [primary key, not null] :

Identifiant du tweet.

- **id\_user** integer [primary key, not null]

## LIKES (Bonus plus compliqué, à faire en dernier)

- **id\_tweet** int [primary key, not null] :

Identifiant du tweet aimé par l'utilisateur.

`likes.id_tweet` est lié à `tweet.id`.

- **id\_user** integer [primary key, not null] :

Identifiant de l'utilisateur qui a aimé un tweet.

`likes.id_user` est lié à `user.id`.

## BOOKMARK

- **id\_tweet** int [primary key] : (clé composite)

Identifiant du tweet enregistré par l'utilisateur.

`bookmark.id_tweet` est lié à `tweet.id`.

- **id\_user** integer [primary key] : (clé composite)

Identifiant de l'utilisateur qui a enregistré le tweet.

`bookmark.id_user` est lié à `user.id`.

## COMMUNITY

- **id** int [primary key, not null]
- **name** varchar(255) [null]
- **biography** varchar(255) [null] :

Courte description de la communauté.

- **id\_creator** integer [not null] :

Identifiant du créateur de la communauté.

- **cover** int [primary key, not null] :

Photo de couverture (bannière).

Chemin vers l'image. Cf explication user.picture = même principe.

- **creation\_date** datetime [primary key, default: current\_timestamp, null] :

Date de création de la communauté par défaut current\_timestamp.

## USER\_COMMUNITY

- **id\_community** int [primary key] : (clé composite)

Identifiant de la communauté.

`user_community.id_community` est lié à `community.id`.

- **id\_user** integer [primary key] : (clé composite)

`user_community.id_user` est lié à `user.id`.

- **role** varchar(255) [Default: user, null] :

Rôle attribué à chaque membre de la communauté par défaut un simple utilisateur (user), lors de la création d'une communauté faire un insert avec un rôle 'admin' pour identifier l'utilisateur qui a créé la communauté; ce dernier pourra ajouter des rôles mods aux autres membres par exemple.