# Projet Web
# Vitz

Thomas Kowalski, Thibaut Milhaud, Florian Barre, Pierrick Barbarroux

ENSIIE S2, Printemps 2018
**Soutenance :** 22 mai 2018

# Contents

# 1 Vitz : pourquoi ?

Notre choix de sujet s'est porté sur le clone de Twitter. En effet, celui-ci nous paraissait accessible de conception et offrait beaucoup de possibilités d'évolution : nous souhaitions faire plus que ce que ce réseau social à a offrir, en se basant sur les goûts des gens afin de les classifier. Dès lors, nous pourrions leur proposer du contenu qui leur plairait afin qu'ils aiment visiter notre site.

**Axiome 1** *La Haine est plus forte que l'Amour*

**Axiome 2** *Les ennemis de mes ennemis sont mes amis*

L'idée motrice était celle de la Haine : l'objectif, contrairement à Facebook et consorts, était de classifier les gens selon ce qu'ils détestent, plutôt que ce qu'ils adorent. En effet, il nous semblait que la haine rassemble bien plus facilement que l'amour, et que l'effet "bulle de confort" serait décuplé si l'utilisateur n'était pas en contact avec des gens aux mêmes goûts que lui, mais plutôt qui détestent les mêmes choses.

Nous souhaitions également qu'à l'instar de la plupart des réseaux sociaux modernes, ce phénomène se fasse naturellement, sans choix de l'utilisateur. Lorsqu'il arriverait sur le site, il pourrait donner son avis (*likes*, *dislikes*) sur un échantillon de publications afin d'obtenir une idée de ce vers quoi il pencherait et vers quelle "zone" de l'espace des utilisateurs il se dirigerait.

Cette classification se ferait ensuite en continu au fur et à mesure qu'il utiliserait le site. En fonction de ses appréciations, on pourrait le rapprocher d'autres utilisateurs qui émettraient le même genre d'appréciations (et le même genre d'idées).

L'avantage d'un réseau social de ce genre et dont le fonctionnement serait celui-ci est évident. En permettant aux gens de se limiter aux idées qu'ils apprécient et soutiennent, on les accoutume à être dans leur bulle de confort. Ils se retrouvent alors, sur les autres sites (et dans d'autres situations sociales) en difficulté, et ont tendance à souhaiter revenir - et à se limiter - au site où ils se sentent bien : le nôtre.

Comme beaucoup des sites actuels, notamment les réseaux sociaux, le *business model* de Vitz se base sur la publicité, l'accoutumance de l'utilisateur à l'utilisation du site nous assurant de bons résultats.

Cependant, nous avons rapidement réalisé que ce concept, bien qu'intéressant, serait difficile à mettre en oeuvre. En effet, il s'agissait plus d'un projet de modélisation mathématique que de Web ; et s'il nous paraissait passionnant à résoudre, le temps dont nous disposions était trop limité devant la charge de travail et le temps à passer pour appréhender totalement le problème.

# 2 *Milestones*

Suite à ces observations, nous avons défini différents objectifs. Dans le cas où un objectif serait rempli, on pourrait alors tenter d'atteindre le suivant, toujours plus ambitieux.

Le premier consistait à simplement respecter le sujet : créer un clone de Twitter.

Le second demandait d'ajouter une fonctionnalité de *dislikes* en plus des *likes*.

Ensuite, venaient des nouveaux types de tendances : plutôt que ne se baser sur les mots les plus publiés sur une période, on se baserait sur les publications les plus et les moins aimées, les plus controversées et enfin les *combats* : les conversations entre des utilisateurs qui deviendraient violentes.

# 3 Organisation

## 3.1 Déroulement du travail

Nous avons jugé utile de découper le travail en plusieurs parties.

D'abord une partie *core* qui permettrait d'avoir des opérations simples sur les objets de notre modèle : Utilisateurs, Publications, Appréciations, Signalements.

Par-dessus cette partie *core* venait l'API. Celle-ci serait utilisée pour récupérer et actualiser du contenu dans des pages déjà chargées : inscription et connexion de l'utilisateur (vérification des données sans rechargement), publication d'un *post*, *like*, signalement, *etc.*

Enfin venait la partie Vue uniquement, qui utiliserait d'une part les fonctions PHP *core* pour charger les pages, puis les fonctions exposées au travers de l'API pour les mettre à jour et permettre des intéractions facilitées pour l'utilisateur.

### 3.2 Répartition des tâches

**Conception du modèle** Thomas et Thibaut

**Programmation de la partie contrôleur et modèle** Thomas (en utilisant PHP (orienté objet) / PDO / pgsql comme demandé)

**Programmation de l'API REST** Thomas

**Conception de l'interface** Thomas, Thibaut et Florian

**Création de la charte graphique** Thomas

**Programmation des vues** Toute l'équipe

## 4 Structure générale du code

Lors de la première revue de code, Un3x nous a présenté une version détaillée du modèle MVC que celle que nous connaissions, en nous expliquant ce qu'étaient un routeur, une entité, un *hydrator*, un service, *etc.*

Suite à son conseil, nous avons décidé d'utiliser un seul fichier `index` qui construirait des instances de classes, chaque classe étant à créer et représentant un type de page. Il en allait ainsi pour les vues mais également pour l'API. *Nous avons cependant préféré garder ce travail pour plus tard, dans la mesure où les fonctionnalités du site nous paraissaient plus importantes que la façon dont il était fait, tout en ayant conscience qu'un tel changement de structure serait d'autant plus complexe qu'il serait effectué tard.*

Nous n'avons finalement pas eu le temps d'utiliser ce modèle, et restons donc avec beaucoup de fichiers PHP (un par vue / *endpoint* REST) :

- assets
  - styles
    - compte.css
    - feed.css
    - general.css
    - menu.css
    - trends.css
    - ...
  - js
    - compte.js
    - forms.js
    - general.js
    - post.js
    - trends.js
    - ...
- Classes
  - Appreciation.php
  - Post.php
  - Report.php
  - SearchHelper.php
  - Trend.php
  - User.php
- view
  - compte.php
  - feed.php
  - hashtag.php
  - index.php
  - login.php
  - logout.php
  - profile.php
  - redirect.php
  - signup.php
  - trends.php
  - tweet.php
  - ...
- api
  - post
    - get.php
    - edit.php
    - ...
  - user
    - follow.php
    - report.php
    - ...
  - trends
    - fromtag.php
    - toprt.php
    - ...
  - etc.

Nous tirons ensuite profit de la réécriture d'URL proposée par nginx. Pour éviter de se retrouver avec des URL du type `/public/src/view/profile.php?username=realDonaldTrump`, nous créons les règles suivantes :

- `/(.*)` donne `/public/src/view/$1` (les vues sont appelées depuis la racine du site) ;
- `/profile/username` donne `/public/src/view/profile.php?user=username` (l'URL est plus propre, les utilisateurs sont plus habitués à une adresse du type `twitter.com/LaLaLand` qu'à `twitter.com/profile?user=LaLa` ; `/api/(.*)` donne `/public/src/api/$1.php` (très peu d'API exposent leurs endpoints sous formes de fichiers PHP).

Enfin, l'utilisation de la réécriture d'URL nous permet également d'apporter une couche supérieure de sécurité au site, puisqu'elle cache à l'utilisateur final la véritable structure du code du site.

# 5 *Stack* du site

Vitz n'utilise qu'un nombre très limité de bibliothèques *externes*. Le site ne dépend que de PDO (et de son *driver* postgreSQL), il utilise PHP 7.0, CSS3 et JavaScript.

Nous n'utilisons aucune bibliothèque JavaScript (comme JQuery) et la grande majorité des fonctionnalités JavaScript que nous implémentons ne sont que des appels en utilisant AJAX (`XMLHttpRequest`) et des affichages et masquages d'éléments DOM, ainsi que de l'ajout / suppression de classes sur des éléments.

# 6 Modèle / Contrôleur

Les parties Modèle et Contrôleur du projet sont gérées par les mêmes fichiers. En effet, le fichier `/classes/User.php` permet de modéliser les entités Utilisateur, d'instancier des Utilisateurs à partir d'un tuple (*Hydrator*), de créer des utilisateurs et de les mettre à jour (Service / Contrôleur) ainsi que d'effectuer des tests (tests d'existence, de correction du mot de passe, *etc.*).

## 6.1 Méthodes standard

Chaque classe que nous utilisons expose des méthodes que nous jugeons standard pour le projet.

`fromRow` construit une entité à partir d'un tuple de base de données correspondant ;

`fromAttribute` construit une entité à partir d'un attribut `Attribute` donné (qui doit identifier l'entité de façon unique : par exemple le nom d'utilisateur, l'ID d'une publication, etc.) en récupérant les informations depuis la base de données ;

`setAttribute` met à jour l'attribut `Attribute` d'une entité (acesseur classique) et met à jour la représentation de l'entité dans la base de données ;

`create` (*uniquement pour les entités "indépendantes" : par exemple, un utilisateur peut être créé ; un like ne peut pas l'être car c'est à un Post de le produire*) créé une instance de cette classe et l'insère dans la base de données ;

`getConnexEntities` permet de récupérer une liste d'entités connexes à l'objet : par exemple, on peut souhaiter récupérer les publications d'un utilisateur, les *likes* d'une publication, *etc.*

`jsonSerialize` produit une sérialisation JSON de l'objet. Utilisé par l'API REST pour envoyer ses réponses au client.

De manière générale, nous essayons au maximum d'implémenter les méthodes là où il est naturel de le faire, cependant il peut parfois y avoir un doute : est ce que `report` (signalement d'une publication) devrait être membre de `User` (`$user->report($post, $reason)`) ou plutôt de `Post` (`$post->createReport($user, $reason)`) ?

## 6.2 Utilisateur

Nous représentons les utilisateurs par la classe *User*.

Celui-ci contient les méthodes classiques du CRUD (création d'un nouvel utilisateur, ie inscription, lecture d'un utilisateur pour l'affichage de son profil ou pour l'authentification, mise à jour d'un utilisateur lorsqu'il choisit de modifier son nom, son adresse e-mail ou son mot de passe et enfin suppression si un modérateur décide de désactiver son compte).

Cette classe expose également des méthodes telles que `follow` (qui permet de faire suivre un utilisateur à un autre utilisateur), ou `findPosts` (qui permet de trouver les publications d'un utilisateur).

## 6.3 Appréciation

La classe *Appreciation* représente une appréciation donnée par un utilisateur sur une publication. Dans la version actuelle du site, il s'agit d'un *Like* ou d'un *Dislike*.

Cette classe ne sert qu'à contenir des informations (elle n'a aucune méthode non standard) sur les appréciations pour que celles-ci puissent être manipulées plus facilement dans les vues.

## 6.4 Tendances

Cette classe abstraite expose une seule fonction : `getTrends` qui renvoie les sujets de conversation (*hashtags*) les plus utilisés dans une fenêtre de temps (entre il y a un certain temps et maintenant).

## 6.5 SearchHelper

La classe permettant d'effectuer les recherches. Son but est avant tout de réunir différentes méthodes de recherche pour pouvoir les appeler depuis un seul et même module.

Cependant, la version actuelle du site ne proposant pas d'effectuer une recherche (autre que sur les hashtags), cette classe est inutile.

## 6.6 Signalement

Enfin, notre classe *Report* (dont héritent deux classes *UserReport* et *PostReport*) représente un signalement effectué par un utilisateur envers un autre utilisateur ou une publication.

Cette classe se résume principalement à la représentation d'un signalement, dans la mesure où toutes les actions prises par un modérateur (suppression d'une publication, désactivation d'un compte, etc.) s'effectuent directement dans les classes *User* et *Post*.

Les seules méthodes *non-standard* qu'expose *Report* sont `getSameEntityReports` (qui permet de récupérer tous les signalements associés à une même entité Utilisateur ou Publication) et `resolveAll` qui permet de résoudre en une fois tous les signalements associés à un utilisateur ou une publication).

## 6.7 Exceptions

Avec l'API et les entrées difficilement vérifiables avant construction des entités sont naturellement apparues des exceptions.

Afin de gérer plus proprement les différentes erreurs qui pourraient être commises par un utilisateur de l'API (par exemple demander les dernières publications d'un utilisateur inexistant) ou tout simplement les erreurs 404 du type `/profile/unUtilisateurQuiNexistePas`, nous avons créé différentes exceptions très simples.

`UnknownAppreciationException` levée lorsqu'un appel à l'API demande de créer une appréciation de type inconnu (non Like / Dislike) ;

`AppreciationExistsException` levée lorsqu'un appel à l'API tente de créer une appréciation qui existe déjà ;

`PostNotFoundException` levée lorsqu'un appel à l'API tente de construire une entité Utilisateur qui n'existe pas (par exemple, on donne un identifiant de publication inexistant) ;

`UserNotFoundException` de même, levée lorsqu'on tente de construire un utilisateur qui n'existe pas (par exemple, on donne un nom d'utilisateur inexistant) ;

`UserExistsException` levée lorsqu'on essaie de créer un nouvel utilisateur avec une adresse e-mail ou un nom d'utilisateur déjà enregistré dans la base de données.

# 7 API REST

Vitz fonctionne en se basant fortement sur une API REST. Celle-ci se décompose en quatre principaux noeuds :

**user** permet d'effectuer des tâches classiques sur l'utilisateur courant ou de récupérer des informations sur un utilisateur

**post** permet de créer, d'obtenir des informations et de modifier des publications

**trends** permet d'obtenir les informations sur les tendances en cours ainsi que les publications associées

**auth** gère l'authentification de l'utilisateur

**moderation** contient les *endpoints* nécessaires à la résolution des signalements

## 7.1 Développement de l'API REST

L'API REST a totalement été programmée par Thomas, à un moment où aucune vue n'était encore implémentée.

Pour vérifier son fonctionnement, il a fallu trouver une solution temporaire. Nous avons alors utilisé Postman.

Postman est un outil pratique qui permet d'envoyer des requêtes HTTP (`GET`, `POST`, `PATCH`, `DELETE`) à une URL afin de vérifier son fonctionnement et son comportement. Cela évite d'avoir à attendre le développement de la vue, qui n'aurait de toute façon pas pu être prêt tout au début du travail.

N'étant pas des professionnels de l'analyse préalable, il nous fallu par la suite et à plusieurs reprises modifier cette API : changer ce qu'elle renvoie (au coût de plus de requêtes parfois).

# 8 Fonctionnement du Site

Arrivé sur le site, l'utilisateur a le choix entre deux actions : se connecter ou créer un compte. Dans le cas où il créé un compte, il va devoir ensuite se connecter. Comme sur de plus en plus de sites, nous ne demandons pas de validation par e-mail du nouveau compte.

Une fois connecté, l'utilisateur arrive sur la page des Dernières publications de son réseau.

Pour chaque publication, il peut choisir de *like*, de *dislike*, de *recycler* (comme *retweeter* sur Twitter), riposter (répondre) ou la signaler à un modérateur.

En cliquant sur le nom de l'auteur d'un post, l'utilisateur peut accéder à son profil et s'abonner à lui, pour recevoir ses statuts à l'avenir.

En cliquant sur la date, on peut voir la publication et toutes ses réponses.

En haut, il a plusieurs onglets à sa disposition pour naviguer sur le site :

- Derniers : La page affichant les derniers posts. Il y est déjà d'origine.
- Fil : La page affichant les posts des gens auxquelles l'utilisateur est abonné.

- Tendances : La page affichant (selon la sélection de l'utilisateur) les posts les plus aimés, les plus détestés, les plus partagés et les hashtags les plus populaires.
- Profil : La page affichant les abonnements et les abonnées de l'utilisateur, ainsi que ses dernières publications.
- Compte : La page permettant de publier un nouveau post et changer les informations de son compte.
- Notifications : La page affichant à l'utilisateur toutes ses dernières notifications : Like/ Dislike/ Recyclage/ réponses à ses publications, ainsi que les activités de ses amis.
- Modération : La page affichant les Signalements récents, ainsi que la possibilité pour les modérateurs de bannir des utilisateurs malveillants et de supprimer des messages inadaptés.
- Déconnexion : Déconnecte l'utilisateur et le renvoi sur la page de connexion/ création de compte.

# 9 Vues

## 9.1 Fonctionnement général d'une page

Une page Vitz lambda fonctionne de la manière suivante : dans un premier temps, on vérifie que l'utilisateur est connecté. Ensuite, le chargement du contenu se fait directement en PHP et la page s'affiche.

Lorsque l'utilisateur interagit avec le site, ses actions sont traitées en JavaScript. JavaScript utilise alors des requêtes asynchrones pour faire des appels à l'API (pour aimer une publication par exemple) puis mettre à jour la vue pour indiquer que l'action s'est correctement déroulée (changement de couleur temporaire, par exemple).

Sur la plupart des pages, il y a une liste de publications, qui est mise à jour automatiquement (toutes les cinq secondes) grâce à une fonction JavaScript qui récupère les dernières publications et qui les affiche, en utilisant `setTimeout` avec elle même comme *callback*.

Certaines pages sont particulières (comme la page Tendances) et fonctionnent directement. Dans le cas de cette dernière, il n'y aucun contenu chargé en PHP et c'est lorsque l'utilisateur choisit le type de Tendances puis la fenêtre de temps qu'il souhaite que le site récupère les informations demandées (en AJAX) puis les affiche sur la page.

## 9.2 Interface du site

Notre idée de base était une page unique, comme sur TweetDeck, où l'utilisateur pourrait voir les dernières publications, ses notifications, son compte, etc.

Cependant, nous avons développé chaque fonctionnalité dans une page différente (pour pouvoir développer chacune tranquillement au début) et avons été finalement convaincus par une interface composée d'un menu (où l'on peut accéder aux différentes pages) et du contenu de la page juste en-dessous.

# 10 Améliorations possibles

## 10.1 Informations du profil

Le profil public Vitz ne contient qu'une information : le nom d'utilisateur. Le compte comporte deux informations supplémentaires : l'adresse e-mail de l'utilisateur (inutilisée actuellement) ainsi que son mot de passe (chiffré).

La plupart des réseaux sociaux modernes proposent beaucoup plus d'informations : une photo de profil, une biographie, un nom d'affichage différent du nom d'utilisateur, etc.

## 10.2 Profil privé

Il serait intéressant d'ajouter une fonctionnalité de profil privé, ainsi l'utilisateur pourrait limiter l'accès à son profil à une liste d'utilisateurs agréés.

## 10.3 Personnalisation utilisateur

Actuellement, Vitz ne permet que d'utiliser une charte graphique par défaut. Sur Twitter, par exemple, l'utilisateur peut changer l'apparence de son profil (code couleur, photo de couverture, *etc.*). Cependant nous n'avons pas souhaité mettre en place cette fonctionnalité, puisque nous n'avions pas assez de temps mais également par doutes sur la façon de la mettre en place.

## 10.4 Messages privés

Une autre fonctionnalité classique est la possibilité d'envoyer des *messages privés* (ou *direct messages* en anglais) entre utilisateurs. Cela se fait relativement facilement avec une base de données du type `Auteur, Destintaire, Timestamp, Contenu`, mais nous n'avons simplement pas eu le temps de la mettre en place.

## 10.5 Recherche

La version actuelle de Vitz ne comporte pas de vue permettant d'effectuer des recherches.

Cependant, l'API REST contient les *endpoints* nécessaires à une telle fonctionnalité (recherche d'utilisateurs et de publications).

## 10.6 Anti-spam

Une fonctionnalité très importante des réseaux sociaux à grande échelle est la présence d'un anti-spam. Aucune mesure de ce genre n'a été prise dans la création de ce site, cependant il serait possible facilement d'en ajouter une de la façon suivante :

- Pour les actions telles que le signalement, les "J'aime", *etc.*, ajouter avant tout un "limiteur" en JavaScript qui compte le nombre d'actions effectuées lors des dernières quinze secondes et qui empêche toute action à partir d'un certain nombre. *Cela ne résout pas tout mais économise de la bande passante côté serveur et des ressources de recherche en base de données.*
- Si le limiteur JavaScript a laissé passer les requêtes (c'est à dire que l'utilisateur ne produit pas de spam ou alors il passe directement par l'API), on garde côté serveur un historique des actions par IP. Avant d'effectuer toute action, on regarde le nombre d'actions effectuées lors des dernières cinq minutes par une adresse IP, et on vérifie qu'il est inférieur à un certain nombre. *Cette méthode est relativement infaillible : c'est le serveur qui effectue les contrôles. Cependant, elle coûte beaucoup plus cher au serveur (car il faut stocker les données, ne serait-ce que pour dix minutes, mais surtout il faut faire beaucoup plus de requêtes SQL). De plus, ce procédé peut présenter un risque si on n'applique par (directement sur le serveur HTTP) un limiteur de requêtes par client par seconde : en envoyant des requêtes dont on sait qu'elles vont être filtrées par le serveur directement à l'API, on provoque l'exécution d'un grand nombre de requêtes de contrôle anti-spam, ce qui cause une utilisation intensive des ressources.*

## 10.7 Récupération périodique des notifications

Beaucoup de réseaux sociaux utilisent des technologies avancées afin de pouvoir notifier en direct les utilisateurs s'ils ont une nouvelle notification. Nous aurions pu "imiter" cette fonctionnalité avec un script qui vérifie périodiquement s'il y a une nouvelle notification, cependant cela implique d'ajouter un caractère *lu / non lu* aux notifications, ce que nous n'avions pas prévu de faire.