

Le réseau social de L'INSOLITE

Un lieu pour partager anecdotes
et faits étranges du quotidien

Hubert **Pfersdorff**

Dossier de synthèse
Titre professionnel:
Développeur web et web mobile
Niveau III
2018 – 2019



Table des matières

Introduction.....	3
Qui suis-je ?	3
Le contexte.....	4
L'influence.....	5
Le concept.....	6
Compétences à couvrir	7
Front-end	7
Back-end	7
Cahier des charges	8
Objectifs du site	8
Périmètre.....	8
Objectifs quantitatifs	8
Cible.....	8
Existant.....	8
Charte graphique	8
Design.....	8
Logo	9
Fonctionnalités	9
Déploiement.....	10
Technologies utilisées	11
Symfony 4, PHP 7	11

Bootstrap.....	11
Mapbox GL.....	12
JavaScript / AJAX.....	13
Font Awesome.....	14
Autres outils.....	14
Base de données	15
Symfony	18
Introduction	18
L'ORM Doctrine.....	19
Architecture de Symfony	20
Composer.....	20
Console.....	20
Architecture	21
Organisation	21
Sécurité	22
Jeux d'essais, tests.....	24
Extraits de code	25
Cas pratique	28
English.....	28
Introduction.....	28
The web framework.....	28
Serving HTML	30

Integrating Socket.IO.....	31
Emitting events	32
Broadcasting.....	33
Français	33
Introduction	33
Le framework web	34
Servir du HTML.....	35
Intégrer Socket.IO.....	36
Emettre des évènements	38
Distribuer	38
Aller plus loin	39
Conclusion.....	40
Annexes	41
RGPD	45
Remerciements	45

Introduction

Qui suis-je ?

Je m'appelle Hubert Pfersdorff, j'ai 26 ans. Actuellement en formation au titre RNCP niveau 3 développeur web et web mobile au sein de l'organisme de formation Elan à Strasbourg, j'ai réalisé cette plateforme pour mettre à l'épreuve mes compétences.

Fasciné depuis l'adolescence par la science-fiction, la technologie mais aussi les classiques notamment au cinéma, j'ai naturellement été attiré vers les métiers du web.

J'ai aujourd'hui l'opportunité de conduire un projet du concept à la présentation, m'autorisant une liberté totale sur les moyens et la réalisation. Cette liberté m'a mené à rapidement instaurer une **rigueur** et **un cadre de travail** spécifique afin de ne pas perdre mon fil rouge.

Avant de démarrer la formation, je n'avais pas de notions en développement orienté objet. Mener ce projet à bien m'a permis de découvrir énormément de choses : j'ai conçu chaque aspect **comme si j'étais une petite équipe dans une agence**. J'ai pris du temps et du plaisir pour chaque chose, de la matérialisation de mes idées à la production. Je suis comme ça. J'aime les petits détails, que ce soit sur un outil ou une personne. Le cadre de travail est d'autant plus important !

En abordant un projet, j'avais tendance à **m'engouffrer** dans le développement de détails. Exemple : un carnet de transmission

pour professionnels de la santé, mon meilleur code se situe dans l'enregistrement de chaque action ou exception dans un fichier de logs, consultable dans une zone admin.

Je n'ai pas changé, mais aujourd'hui afin de me préparer au mieux pour mon futur travail, **j'aborde chaque chose de manière professionnelle**. Ce raisonnement a je crois porté ses fruits : dans le cadre de ce projet, beaucoup a été réalisé en amont. Identité visuelle, concept, technologies à utiliser, déploiement, livraison, dossier de synthèse par exemple. C'est un nouvel exercice pour moi que je pense nécessaire peu importe la direction à prendre après la formation.

Cette plateforme est influencée par un certain nombre de choses sur lesquelles nous reviendrons. L'une des influences principales est mon expérience de 3 ans dans le milieu de l'hébergement web, du support technique mais aussi de la **négociation**. J'ai pensé chaque interface avec beaucoup d'empathie, car j'ai appris à me mettre à la place d'une autre personne. Aujourd'hui j'applique cette capacité à **l'ergonomie** de mes interfaces.

Je vous souhaite **bonne lecture**, et espère que les éléments que je vais vous apporter vous aideront à comprendre les **choix** faits ainsi que le produit final. (Une condition de lecture recommandée est l'écoute de *Comfortably Numb* lors du [live de Pink Floyd](#) à Londres en 1994, en 44Khz)

Le contexte



Afin de m'approprier le projet et la volonté d'y investir beaucoup de temps, je dois être **inspiré**. J'ai pris du recul et me suis demandé ce que je recherchais quand arrive le moment d'une découverte. J'ai vais prendre l'exemple du film déjà vu ou non : aujourd'hui, je sais que **l'émotion est le résultat que je recherche**. Vais-je rire, pleurer, m'interroger ?

Moi qui ai maintenant l'opportunité de créer, que vais-je **transmettre à l'utilisateur** ? Ou plutôt, que vais-je proposer comme outil, afin de permettre aux utilisateurs de rechercher à leur tour une émotion. Je vais reprendre cette photo tirée du film *Dikkenek* d'Olivier Van Hoofstadt : pas besoin de chercher l'originalité, il faut simplement que ça me plaise. L'idée m'est venue **par hasard** et j'en suis fier. Attention, je ne prétends pas avoir trouvé l'idée qui révolutionnera notre Internet. C'est une idée qui simplement me plait, mais surtout qui m'inspire. J'y viens.

Nous connaissons tous Facebook, Twitter, Instagram, Reddit qui nous permettent de tout partager. J'ai pensé que me baser sur un réseau social pour mon projet de fin de formation serait une base solide, me donnant assez de matière à défendre. Mais il manque le

grain de sel. Et c'est ça l'idée, **le grain de sel**. De nos vies, mais en particulier le grain de sel de nos interactions avec notre environnement.

Une plateforme de partage de sel. De **l'anecdote originale** aux scènes les plus farfelues, l'outil permet de faire profiter les autres d'un moment **éphémère**. Et comme j'aime les jeux, rendons tout ça **interactif** en permettant aux utilisateurs de jouer avec l'histoire.

On m'interpellera en me demandant « mais pourquoi aller sur **Strapp**, alors que j'ai déjà Facebook ou twitter avec mes amis ? ». Pourquoi utilise-t-on un **four pour faire le gratin**, alors qu'un incendie nous permet l'accès au même résultat ? Le four est adapté, et tous les utilisateurs de four tendent **vers le même but** : manger.

Sur **Strapp** c'est pareil : je suis une plateforme dédiée à l'original, à l'éphémère, **je recueille les histoires et constitue une véritable carte de l'imprévu**. Demain vous visitez une nouvelle ville, pimentez votre séjour en vous basant sur mes données et mes utilisateurs !

L'influence

Le **caractère visuel** est un aspect très important pour moi, j'ai besoin de me sentir bien dans mon projet pour me l'approprier, et ce dès le début. J'ai besoin d'avoir en tête un résultat afin d'être efficace.

Après plusieurs essais, je me suis arrêté sur le raisonnement suivant : événements dans la rue + éphémère + interaction + étrange = Les rues de New York + film éphémère + communication, échange + ville la nuit = **Taxi Driver de Martin Scorsese**. Les couleurs des taxis new-yorkais me plaisent, le film me plait, l'ambiance qu'il dégage me plait.

Sans rentrer dans les détails du film, je m'imagine au volant du taxi de Travis Bickle, assister à une **scène aussi loufoque qu'indescriptible**. Pourquoi ne pas baser la plateforme sur cette ambiance ?

Aussi, n'étant pas graphiste, je ne prétends pas pouvoir créer une identité visuelle de toute pièce, « from scratch », **m'inspirer d'une œuvre** que j'aime m'aide énormément.



Le concept

Le concept est simple : un réseau social où partager un moment original sous la forme d'un **vote, que nous appellerons « story »**. Les autres utilisateurs ont la possibilité de voter s'ils pensent que l'histoire est vraie ou fausse. L'issue du vote détermine le degré de fiabilité de l'auteur, ou sa capacité à raconter une fantaisie. L'issue du vote détermine aussi la capacité des autres à déterminer la **véracité** d'une histoire. Ainsi, chaque utilisateur dresse son profil en fonction de ses actions.

Le partage de l'histoire se fait sous la forme d'un billet comprenant un titre, la description, ainsi qu'une photo de la scène. L'auteur a la possibilité de choisir une durée, pendant laquelle le vote sera visible de tous sans la photo. Cette photo n'est visible par les votants qu'une fois le vote achevé, ainsi que l'issue du vote et la vérité. Le billet de la story s'affiche sous la forme d'un **marqueur sur la carte**.

Ces données serviront à dresser des zones géographiques sur la carte affichant les lieux denses en scènes originales, ou en grand conteurs d'histoires. Je doute que le deuxième filtre soit très pertinent, mais il peut être intéressant de voir se révéler des zones qui inspirent **l'imagination**. Voilà qui couvre la partie carte et story.

Côté réseau social, une fois inscrit, chaque utilisateur a la possibilité de voir le **profil** d'autres utilisateurs et de les ajouter à sa liste d'amis. Une fois amis, deux utilisateurs amis peuvent interagir par le biais des story, des commentaires et plus tard de la messagerie.

Une page de **fil d'actualité** est l'affichage pour l'utilisateur où il peut voir les stories récemment postées par ses amis où les personnes

qu'ils suit, ainsi que les informations basiques telles que le nombre de commentaires, le nombre de votes, et **le temps restant**.

Au clic sur une story nous accédons aux détails de celle-ci, permettant une vue plus statique des différentes informations.

Le cœur de l'application est la carte. Même sans être un utilisateur enregistré, la page d'accueil affiche déjà une carte, avec par exemple les stories populaires accessibles à tous, vue uniquement.

Une fois enregistré, **tout tourne autour de la carte**. On y affiche les stories proches de soi, avec par exemple un filtre sur la ville. Chaque point sur la carte est cliquable pour accéder aux informations basiques de la story, avec un lien menant vers la page des détails.

Quand un utilisateur crée une story, il doit choisir une **durée** qui générera une date de fin. Une fois la date dépassée, l'interaction avec la story n'est plus possible, et les votes sont révélés, ainsi que la véracité de celle-ci, **générant un score** pour l'auteur et les votants en fonction des résultats.

Compétences à couvrir

Le projet étant réalisé seul, il permet d'étendre au maximum la couverture des compétences à couvrir pour le diplôme. Voici les compétences abordées dans mon projet :

Front-end

- Maquetter une application.
 - **Strapp** a été maqueté à l'aide de d'Adobe XD et Adobe Photoshop. Ayant quelques notions sur Photoshop, je n'ai pas eu grand mal à prendre en main XD. Le logiciel m'a permis de dresser un visuel de mes pages **en amont**, avec une gestion des différents chemins et transitions.
- Réaliser une interface utilisateur web statique et adaptable.
 - Pour gérer mon front, j'étais partis pour m'aider du framework CSS Semantic UI, mais je me suis rabattu sur Bootstrap, plus adapté à la **flexibilité** dont j'aurai besoin. L'application étant mobile first, j'ai dû gérer la totalité du responsive moi-même.
- Développer une interface utilisateur web dynamique.
 - L'application étant principalement composée de formulaire, sous la forme de simples boutons jusqu'aux pages entières, l'utilisation de **Javascript et d'Ajax** m'a aidé à rendre le parcours de l'utilisateurs plus fluide et agréable. Aussi, mon système de notifications reflète bien l'aspect dynamique de l'application, permettant la mise à jour de données coté client sans action requise de sa part.

Back-end

- Créer une base de données.
 - Le projet étant réalisé avec Symfony 4, tout tourne autour de la base de données et des relations entre les entités qui y sont stockées. L'appli est basée sur une **base de données MySQL 5.7**, propulsée par le moteur InnoDB.
- Développer les composants d'accès aux données.
 - Symfony ici nous aide en apportant l'implémentation très profonde de **l'ORM Doctrine**, me permettant de manipuler directement des entités générées depuis la base de données. Doctrine permet aussi faire beaucoup à la main, par exemple lorsque je n'ai pas besoin d'une entité entière.
- Développer la partie back-end d'une application web ou web mobile.
 - L'application entière est back-end, puisque **son utilisation requiert un compte**. Une interface d'administration vient compléter cet aspect.

Cahier des charges

Objectifs du site

L'objectif premier de **Strapp** est d'offrir aux utilisateurs un lieu d'échange d'anecdotes vraies ou fausses ayant pris part de préférence en publique. Ce **lieu d'échange** permet aux utilisateurs d'interagir par le biais des « stories », messages, commentaires et votes.

Périmètre

L'application est conçue pour une utilisation mobile avant tout, permettant aux utilisateurs de s'y retrouver facilement sur leurs appareils dans la rue, avec **une interface accessible entièrement d'une main**.

L'application est pour le moment en anglais, afin de permettre à un maximum l'accès au contenu. L'interface est en anglais, le contenu sera dans la langue des utilisateurs. Je suis très curieux de voir où il sera utilisé une fois en production !

Objectifs quantitatifs

Le trafic attendu n'est pas encore connu, étant donné qu'une plateforme semblable n'existe pas encore.

Cible

Le public visé a été pensé comme suit : j'ai la chance d'avoir dans ma famille des proches curieux. **Curieux sur leurs origines**, sur l'histoire de notre région et de nos ancêtres. Je n'ai pas encore développé cette même curiosité, mais leur façon de parler de

petites anecdotes passées m'a inspiré. **Le public visé est donc le curieux**, celui qui s'intéresse aux détails de ce qui l'entoure. C'est un public qui a entre 20 et 70 ans !

Existant

Le nom de domaine choisi est **strapp.xyz**, enregistré chez Online. L'application sera hébergée sur un VPS dans la phase de tests puis serveur dédié chez OVH.

Charte graphique

Les codes couleur ont évolué plusieurs fois avant que je m'arrête sur une combinaison satisfaisante :

#ffda00 (RGB Décimal 255, 218, 0) (Pure (or mostly pure) yellow),

#121212 (RGB Décimal 18, 18, 18) (Very dark gray (mostly black)),

#ffffff (RGB Décimal 255, 255, 255) (white)

Polices utilisées : PORTO, Roboto, Fairplay Display.

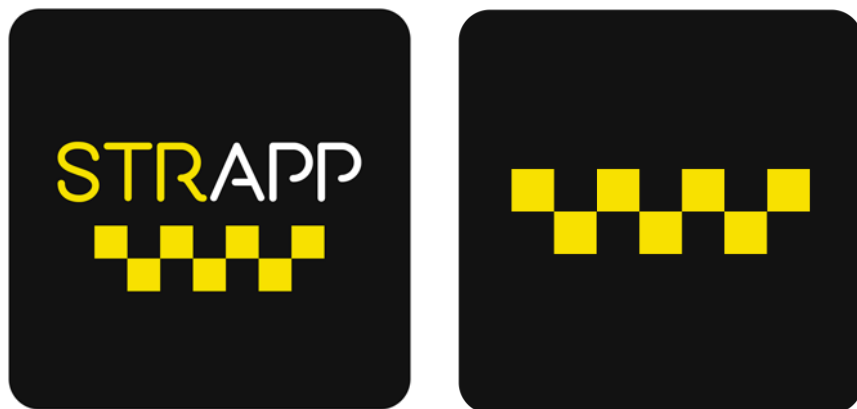
Design

« Good design is as little design as possible » - Dieter Rams

L'objet principal de l'application étant le contenu partagé par les utilisateurs, son design a pour objectif de faciliter l'accès à celui-ci. La majeure partie de l'utilisation de l'application ayant comme media des formulaires et des boutons, ceux-ci doivent **avoir du retour et être satisfaisant à l'utilisation**. Chaque action utilisateur validée est accompagnée d'une notification, qui vient compléter les animations visuelles.

Logo

L'objectif de la charte graphique est de rappeler **les taxis New-yorkais**, à l'aide du jaune et du noir.



A gauche le logo principal, à droite le favicon utilisé pour l'application. Logo basique mais qui remplit son rôle, être **reconnaissable**.

Fonctionnalités

L'application doit dans un premier temps offrir la possibilité d'**inscription** aux utilisateurs non enregistrés.

Une fois en possession d'un compte, l'application dévoile le reste de ses fonctionnalités une fois l'utilisateur **authentifié**, via un formulaire de connexion.

La modification du profil de l'utilisateur se fait sur une page dédiée à la consultation et à la **modification de chaque information**. L'utilisateur peut modifier : nom, prénom, adresse email, mot de passe, humeur et sa courte biographie.

L'application offre ensuite, via le menu principal, l'accès à la page de création d'une story. Une **story** est constituée d'une date de création, d'une date de fin, d'un titre, d'une image, d'une description, d'une position, et d'un auteur.

La page d'accueil, dite « **home** » ou « feed » affiche les stories créées par les amis de l'utilisateur. Chaque story est affichée sous la forme d'un billet contenant ses informations basiques, ainsi que les compteurs de votes et de commentaires.

Au clic sur le billet d'une story, la page des détails de la story permet la consultation des informations relatives à celle-ci. L'image de la story est recadrée afin de ne pas pouvoir la consulter en entier avant la date de fin. L'utilisateur a la possibilité de **voter** et **commenter** tant que la date de fin n'est pas dépassée. Si l'utilisateur est l'auteur de la story, il a la possibilité de la supprimer.

Une fois la date de la story terminée, le résultat est affiché sur la page des détails, et les utilisateurs ayant voté sont notifiés. Les votes sont bloqués et l'image de la story devient accessible dans son intégralité, **révélant ou non la scène atypique**.

Afin de consulter les stories proches de sa position, la page « **map** » affiche sur une carte les stories récemment créées sous la forme de « pins », affichant un pop-up de détails au clic. Chaque pin apparaît à **la position de son auteur** au moment de sa création, permettant aux utilisateurs de se rendre sur place ou encore d'utiliser Google Maps pour se faire une idée de la scène par exemple.

Pour trouver d'autres utilisateurs ou ajouter des amis, une page « **friends** » affiche selon le filtre sélectionné, les amis de l'utilisateur, les utilisateurs proches, ou encore tous les utilisateurs sous la forme d'une liste.

Une **barre de recherche** permet aussi de retrouver un utilisateur en fonction de son nom, prénom ou nom d'utilisateur. Les résultats sont **triés par pertinence**.

Enfin, un système de **notifications** permet à l'application de notifier les amis de l'auteur d'une nouvelle story lors de sa publication ou de sa date de fin. Ce système permet aussi aux utilisateurs d'être informés lorsqu'ils reçoivent une demande d'ajout en ami, ou lorsque l'une de leur demande est acceptée.

Déploiement

L'application sera déployée dans un premier temps sur un **VPS** loué chez OVH. Cette solution me permet de faire participer plusieurs utilisateurs à des tests afin de me rendre compte du comportement de mes fonctionnalités sur une grande variété d'appareils.

Des premiers tests ont tout de suite **révélé des failles** dans mes choix de certaines fonctionnalités :

- Responsive, comportement de la carte, comportement de certains formulaires. **Exemple concret**, je n'ai jamais eu le réflexe en 4 mois de travail, de scroller sur la page de login. Résultat, si on scroll jusqu'à cacher le formulaire, on se retrouve avec la carte en plein écran, sans possibilité de remonter ! Ce sont des **détails** et des **choses simples** à modifier, mais rendre l'application accessible permet de ne plus avoir 1 cerveau et 2 yeux mais bien plus afin de tester tous les petits détails.
- **Comportement de l'utilisateur**, gestion du contenu. Je n'avais par exemple pas prévu le cas où l'auteur « spoil » l'issus de son propre vote dans les commentaires. Vient aussi la question de la censure du contenu. Comment gérer

les photos mises en lignes par les utilisateurs afin d'éviter les abus ? La participation d'autres utilisateurs sera nécessaire. Prenons l'exemple de Facebook encore aujourd'hui, qui fait appel à des prestataires notamment en Allemagne à Berlin pour exercer la **censure** des contenus sensibles. Nous avons aujourd'hui les moyens de demander à une machine si une image contient un certain type de contenu, mais ce genre de scripts se révèlent encore trop lourds pour du simple **contrôle de données**. J'ai pour l'instant choisi de faire confiance à mes utilisateurs.

Technologies utilisées

Symfony 4, PHP 7



Après avoir travaillé avec PHP un certain temps, j'ai voulu aller plus loin. Etant donné la nature du projet, un réseau social, l'utilisation d'un **framework PHP** s'avérait pertinente.

Symfony est le framework PHP développé par l'agence française **SensioLabs**, offrant accès à ses nombreux composants afin d'accélérer le processus de développement d'une application, tout en imposant un cadre de travail précis.

Voici les **fonctionnalités** principales de Symfony m'ayant aidé tout au long du projet :

- Une séparation du code selon le modèle MVC.
- Une gestion des routes
- Gestion native de l'AJAX
- Système de cache
- Prise en charge de bundles

Symfony peut être utilisé de nombreuses manières différentes. De la conception full stack d'un site web vitrine, e-commerce, ou encore application de gestion, à l'API complète. Le framework est utilisé par Askeet, Yahoo, Dailymotion ou encore Drupal. Ses capacités d'**adaptation** et de **flexibilité** en font un très bon outil pour la

réalisation de mon projet. Apprendre à l'utiliser a été pour moi une expérience très agréable et **riche en découvertes**. Je n'ai bien sûr qu'exploité qu'une infime partie de ses composants, ce qui est très réjouissant pour la suite.

Bootstrap



Afin de gagner encore plus de temps pour me concentrer d'avantage sur mes fonctionnalités, j'ai dans un premier temps choisi d'utiliser le framework CSS open source **Semantic UI**, très bel outil permettant de gérer l'UX de mon application. Etant donné la nature du projet, j'ai finalement choisi d'utiliser Bootstrap, framework avec lequel je j'étais déjà familier. Pour être certain de développer les fonctionnalités souhaitées à temps, j'ai préféré ne pas avoir à ajouter un nouvel outil à la liste des choses à **prendre en main**.

La simple utilisation de Bootstrap m'a énormément aidé à arranger les éléments principaux de mon interface: menu, colonnes, boutons, formulaires, responsive. L'**adaptabilité** sur mobile étant un point clé de l'application, une bonne **surcouche** a été nécessaire tout au long de l'avancée du projet. Bien que géré par défaut avec des classes comme "col" ou "row", afin d'arriver au résultat attendu, j'ai dû modifier un grand nombre de comportements de ces dernières. Comme Symfony, Bootstrap m'a fourni un **cadre de travail** dans lequel il ne me restait plus qu'à m'exprimer.

Voici un exemple concret d'une surcouche à Bootstrap: mon menu comportant des icônes, son comportement varie en fonction de la largeur du display. En dessous de 700 pixels, il s'affiche sous la forme d'une liste verticale. Au-dessus de 700 pixels, il s'agit d'une liste horizontale. La hauteur des éléments de la liste variant en fonction de la largeur de la page, afin de garder l'équilibre de largeur

entre chaque élément, le menu étant en position fixe, avait tendance à cacher le haut de mon contenu principal. Pas un gros problème me direz-vous, mais impossible à gérer uniquement avec Bootstrap. C'est là qu'entre en jeu la **media query** ci-dessous, qui vient corriger ce problème. Problème qui est, rappelons-le, empêcher le menu de cacher mon contenu principal lorsque les icônes de chaque élément du menu sont au-dessus du texte, ce qui augmente la hauteur du menu.

```
161 @media screen and (min-width: 700px) and (max-width: 1442px) {
162     .main-content {
163         padding-top: 3.6rem;
164     }
165 }
```

Entre 700 et 1442 pixels de largeur du display, mon contenu principal aura une marge interne de 3.6rem, unité de mesure relative.

Pour résumer, l'utilisation de Bootstrap m'a offert un gros gain de temps, mais n'a pas suffi à elle seule à produire les résultats attendus.

Mapbox GL



Mapbox est une plateforme de gestion de données de localisation pour applications web et mobiles. De nombreux **outils versatiles** comme la localisation, la

carte, la recherche et la navigation offrent une porte d'entrée aux développeurs comme moi qui cherchent une **solution** viable et open source à leurs idées. En plus de la cartographie, l'entreprise Mapbox basée à **San Francisco** développe aussi des composants de réalité augmentée ou encore de gestion de données. De gros acteurs utilisent au quotidien les outils Mapbox : IBM, Tinder, Bosch, Github, National Geographic, Snapchat, ou encore Mastercard.

Pour mon humble projet, Mapbox a été pour moi **l'alternative** parfaite à Google Maps, ne m'obligeant pas à fournir des coordonnées bancaires même pour de simples essais. Avec Mapbox, je suis limité à 50000 appels par mois avec un compte gratuit, et ce sans compter les appels à **l'API Mapbox Places**, me permettant de récupérer les données relatives à des coordonnées.

J'utilise Mapbox de **deux manières** dans mon projet : A l'aide de l'API **Geolocation** d'HTML5, je récupère la longitude et latitude de la position de l'utilisateur, en lui demandant la **permission** au préalable (nous y reviendrons, ceci a été un **point clé** de ma mise en production). Une fois ces coordonnées enregistrées en base de données, je génère la carte Mapbox centrée sur la dernière position connue du client. Une fois la carte générée, sont ensuite générés les pins relatifs aux **stories récemment créées**. Je n'ai pas encore décidé du filtre à appliquer sur la carte afin de ne pas charger tous les pins de la carte. Voici la première utilisation de Mapbox. Je me sers ensuite de l'API Mapbox Places et de son **endpoint** principal :

```
329 /**
330  * set user location with Mapbox API
331  *
332  * @param string|null $coord
333  * @return self
334  */
335 public function setCurrentLocation(?string $coord = null): self
336 {
337     if ($coord != null) {
338         $data = json_decode(file_get_contents("https://api.mapbox.com/geocoding/v5/
mapbox.places/{$coord}.json?
access_token=pk.eyJ1IjoidG9udG9uc2F0IiwiaW5iYSI6ImNqc25jNTIwNjA5bDc0M280dGt4ejt
NXkifQ.h_Ox7WHHtfhpQK9Qr0oTlw"));
339
340         if (!empty($data)) {
341             $this->currentLocation['city'] = $data->features[2]->text;
342             $this->currentLocation['state'] = $data->features[3]->text;
343             $this->currentLocation['country'] = $data->features[4]->text;
344             $this->currentLocation['coord'] = $coord;
345         }
346     }
347     return $this;
348 }
```


Cette fonction me permet de récupérer les informations relatives à une position donnée, afin de plus tard gérer mes filtres d'utilisateurs. Par exemple, afficher uniquement les utilisateurs dont la **dernière position connue** se situe dans la **même ville que vous**.

Mon expérience avec Mapbox ne s'arrête pas là. J'ai eu aussi le plaisir de travailler avec leur interface de modification de « couches », pour personnaliser ma carte. Intuitive bien que très poussée, j'ai pu obtenir un **résultat satisfaisant** afin de présenter aux utilisateurs une **carte claire**, dans le thème de l'application.

JavaScript / AJAX

JavaScript a aussi été utilisé à deux fins. La première étant la gestion de mes **animations** à l'aide de la librairie **jQuery**, et la seconde la gestion de nombreuses actions utilisateurs, permettant des **pages dynamiques**, à l'aide **d'AJAX**. Ajax est aussi utilisé pour générer mes stories sur la carte.

L'application de comportant aucune animation « pour faire joli », JQuery est **couplé** à mes appels AJAX afin que l'utilisateur ai un **retour sur une action effectuée**. Exemple : la pagination, ou **le lazy**



loading. Sur la Home notamment, qui affiche un nombre donné d'éléments récents, une fois arrivé en bas de page, l'utilisateur voit apparaître une petite icône lui indiquant que la suite est sur le point de s'afficher.

J'étais dans un premier temps parti sur une pagination classique, mais le lazy loading est à mon goût bien plus agréable pour une **application récréative**. Nous reviendrons sur le code plus tard quand

nous aborderons la structure du projet, mais je vais vous narrer le déroulement de ce simple appel Ajax afin que vous puissiez vous faire une idée. Il faut savoir que Symfony gère **nativement** très bien les requêtes AJAX et l'implémentation de ses différents procédés est relativement simple.

- 1- En JavaScript, nous détectons quand l'utilisateur scroll jusqu'en bas de la page. Exactement comme ceci : Si la somme de **ScrollTop** (valeur du curseur de déroulement via JQuery) et de la hauteur de la fenêtre est égale à la hauteur du document, j'exécute ma requête AJAX.
- 2- Ma requête utilise **plusieurs paramètres** : la route Symfony et l'offset (décalage utilisé plus tard en SQL, afin de ne récupérer que la suite du contenu). Si ces paramètres sont réunis, j'affiche à l'aide de JQuery une icône de chargement. J'exécute en suite la requête.
- 3- La requête est récupérée par mon **contrôleur Symfony** (gérant uniquement mes requêtes AJAX). Si aucune erreur n'est détectée, une requête SQL est exécutée. Le résultat est retourné au client sous la forme d'une collection d'objet passée à un template Twig, ou un tableau vide si aucun résultat n'est trouvé.
- 4- Quand le client récupère une réponse, la suite se déroule via le **callback** de ma requête : Si la réponse est vide, j'affiche un message l'indiquant à l'utilisateur, sinon, j'affiche le résultat à la suite du contenu.

Ainsi, l'utilisateur récupère le contenu sans même l'avoir demandé, **fluidifiant** ainsi son expérience. J'utilise le même procédé pour les notifications, l'affichage de stories sur la carte, ou encore la mise à jour de la position de l'utilisateur dans certains cas.

Font Awesome



Font Awesome est une boîte à outils rendu accessible depuis 2017 par **Dave Gandy** sous licence Freemium pour une utilisation intégrée à Bootstrap. Basé sur CSS et LESS, cet outil permet de générer une grande variété d'icône de la manière suivante :

```
<a class="nav-link new-pin-link" href="{{ path('home_vote') }}"><i class="fas fa-pen-nib"></i> New</a>
```

L'utilisation de Font Awesome m'a permis d'agrémenter mes menus, formulaire et boutons d'icône afin d'approcher de manière **sémantique** l'expérience utilisateur. Ces icônes sous la forme de vecteurs CSS permettent d'obtenir systématiquement le meilleur rendu peu importe la condition d'affichage. Si l'icône doit être affichée en 3000x3000px, la qualité sera la même qu'en 25x25px, sans avoir à charger un fichier, sans se soucier des temps de chargement. Ajoutons à cela la **cache** navigateur et le cache de Symfony, j'ai obtenu un résultat très satisfaisant en ce qui concerne mes temps de chargement.

Autres outils

- **Git**, pour ma **gestion de versions**. Tout au long du développement du projet, et même lors de la rédaction de cette synthèse, j'ai conservé mes fichiers et contrôlé mes versions à l'aide de l'émulation Bash proposée par Git. Je retrouve mes fichiers aussi bien chez moi qu'au travail, et je peux aussi partager mon code sur **Github** par exemple.
- **Visual Studio Code** pour l'édition du code. J'ai commencé cette formation sur NetBeans, puis basculé sur Atom, mon choix s'est enfin porté sur VS Code, développé et très bien

maintenu par Microsoft. J'aurai je pense par la suite besoin de travailler sur de véritables **IDE** comme PHP Storm, mais n'étant pas gratuit, VS Code est pour moi la meilleure alternative gratuite, **personnalisable** à 100%, intégrant Git nativement, ainsi que de bons outils de debugs et un bon support des différents langages utilisés à l'aide des extensions.

- **Adobe XD**, pour la réalisation des maquettes.
- **Adobe Photoshop** pour toute réalisation visuelle, logo, Powerpoint ou autres retouches.
- **Postman**, pour apprendre à travailler avec les endpoints de Mapbox. Outil gratuit et intuitif que j'ai découvert en stage.
- **Penzu**, en guise de carnet de bord, afin de ne pas perdre mon fil rouge.
- **Trello**, avec extension ElleganTT, pour la gestion de mes tâches. Très utile surtout en amont afin de me faire une idée du travail à effectuer. J'ai ensuite travaillé selon la méthode la méthode **Agile**, très simplifiée, ce qui m'a amené à aborder presque tous les jours des fonctionnalités différentes.
- **Wamp**, pour faire tourner le projet en local. L'utilisation de Composer et Symfony requiert des petits paramétrages de Windows et Wamp, mais l'environnement de travail résultant est très agréable.

Base de données

Rentrons dans le **vif du sujet**. Pour être honnête, j'ai réalisé un modèle de ma base de données avant de coder la moindre ligne de code, mais je n'ai pas su m'y tenir. N'ayant à ce moment-là aucune expérience avec Symfony, j'ai réalisé mon **modèle conceptuel de données** avec une direction générale à suivre, mais je n'avais pris en compte ni les contraintes de mon framework, ni les différents bundles que j'ai eu l'occasion d'utiliser.

Pour rédiger ce dossier de synthèse, j'ai eu l'occasion de le refaire en m'aidant du projet réalisé, et j'ai pu constater finalement que la différence avant/après n'était pas si importante. Vous trouverez en fin de section une copie de mon modèle **conceptuel** et **logique** de données.

Vous pourrez revenir ici afin d'avoir une explication. Je précise que je n'ai pas intégré les tables générées par le **notification-bundle** développé par **Maximilien Gilet**, les tables n'entretenant aucune relation.

Je vais tâcher d'être clair :

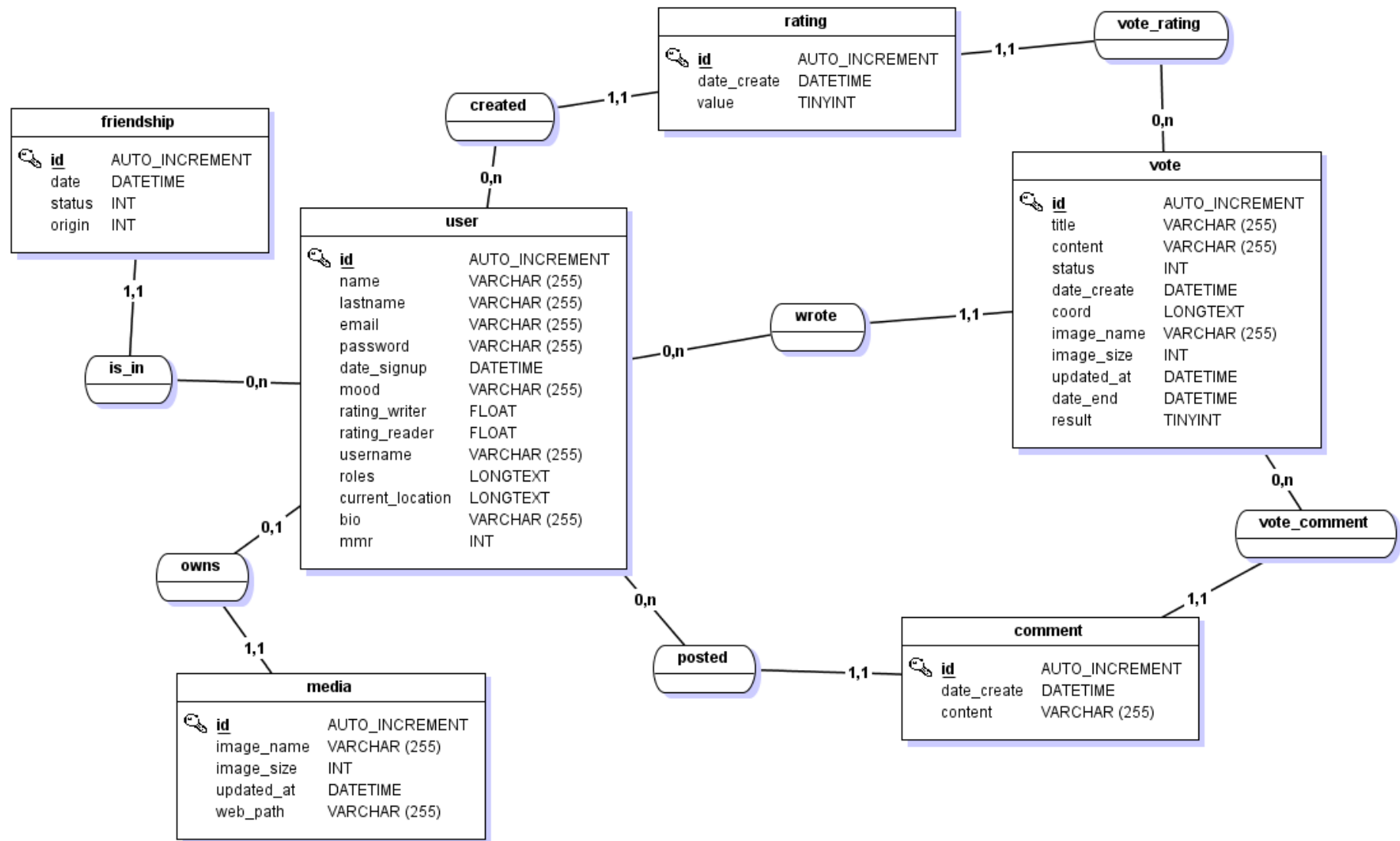
- Un utilisateur peut avoir aucune ou plusieurs amitiés. Une amitié représente la relation entre deux utilisateurs. Une amitié a forcément **un utilisateur et un seul**. Chaque utilisateur présent dans une amitié possède sa propre copie de l'amitié. Donc pour 2 utilisateurs amis, nous avons deux entités « amitié », ou les rôles sont échangés, mais l'utilisateur **origine** de l'amitié est conservé. J'aurai préféré faire sans duplicata de l'amitié mais j'ai dû faire ce choix afin

de gérer correctement le principe d'ajout en ami, où une **invitation** envoyée doit être acceptée par la cible.

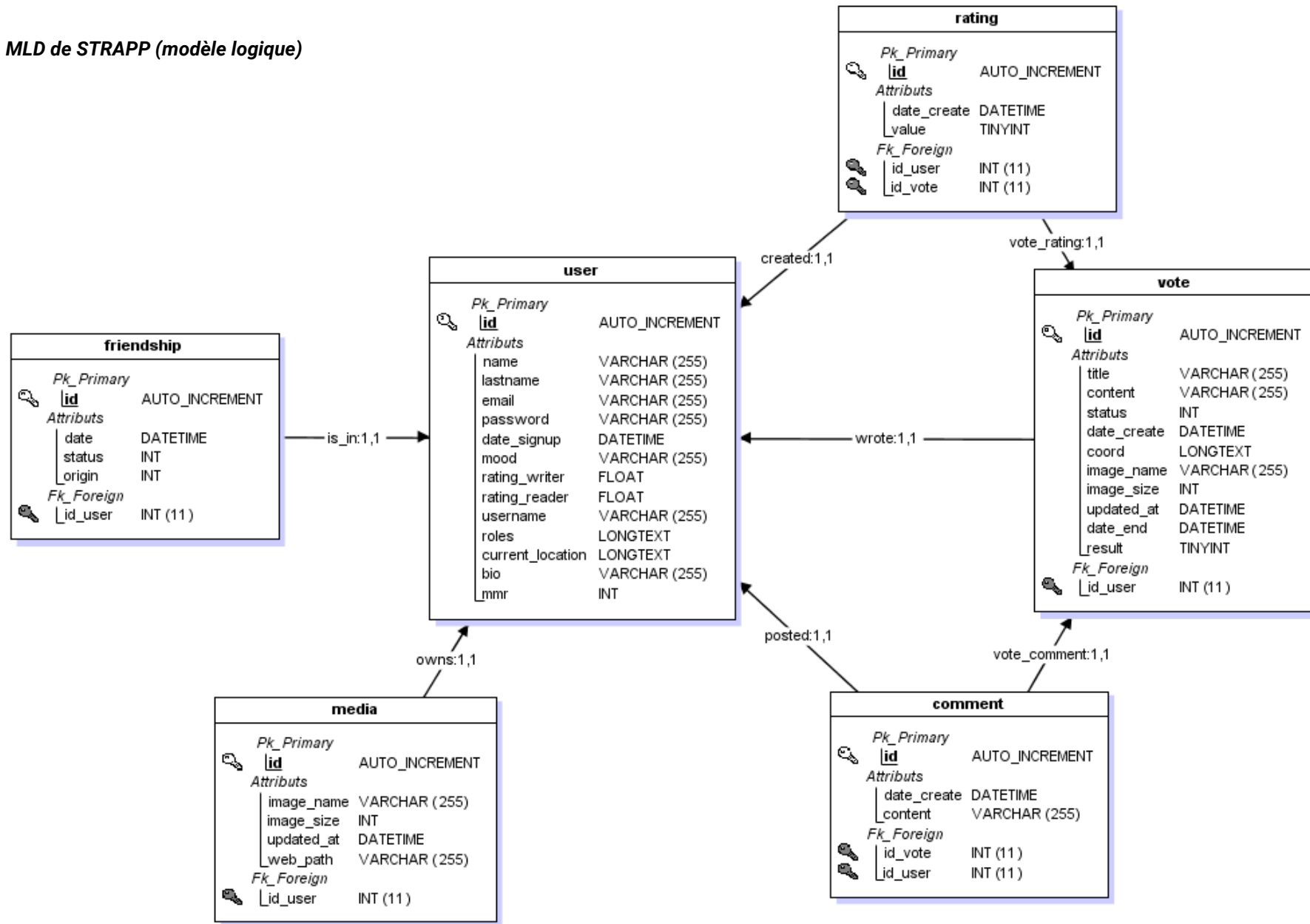
- Un utilisateur ne possède aucun ou un seul media, qui représente sa photo de profile, ou avatar. Chaque nouvel utilisateur se voit **attribué** une image dont les propriétés ne sont pas en base de données. C'est lorsqu'il en choisit une qu'une nouvelle **entité** est créée, qui ne peut exister sans l'utilisateur.
- Un utilisateur peut créer des stories, appelés votes en base de données. Un utilisateur ne possède aucun, ou plusieurs votes, et un vote **ne peut exister** sans son utilisateur, dit auteur.
- Un utilisateur peut commenter une story. Donc un utilisateur ne possède aucun ou plusieurs **commentaires**. Un commentaire ne peut exister sans son propriétaire, ni le vote auquel il est rattaché. Un vote ne possède donc aucun ou plusieurs votes.
- Enfin, un utilisateur peut voter sur une story. Les termes se mélangent un peu, mais on va y arriver : un utilisateur possède un ou plusieurs « ratings », un **rating** ne peut exister sans le vote (story) auquel il est rattaché. Un vote (story), peut n'avoir aucun ou plusieurs votes.

Voilà qui résume les relations entre mes entités (nous allons en parler en abordant Symfony). Ses relations se matérialisent par la présence de **clés étrangères** dans les tables, grâce au moteur InnoDB. Par exemple, On retrouve dans la table user une clé étrangère media_id, ou dans la table vote, une clé étrangère author_id. Ces relations seront très utiles plus tard pour Symfony, qui à l'aide de **l'ORM Doctrine** pourra par exemple, en récupérant un utilisateur donné, afficher ses stories sans requête supplémentaire. J'y viens.

MCD de STRAPP (modèle conceptuel)



MLD de STRAPP (modèle logique)



Symfony

Introduction

Pour bien cerner l'état d'esprit pour travailler avec Symfony, il ne faut plus raisonner par relation entre le code PHP et la base de données. Ce qui amène à ne plus travailler avec des tables, mais avec des **entités**. Une entité Utilisateur, une entité Vote, une entité Media, et ainsi de suite. J'ai eu l'occasion à plusieurs reprises de développer des petits sites / applications en structurant mon code en suivant le motif d'architecture **MVC** (Model, View, Controller), inventé par **Trygve Reenskaug** en 1978.

Bien qu'il existe de très nombreuses variantes de motifs, même au sein de MVC, respecter ces architectures permettent **d'organiser** son code pour mieux suivre un gros projet, travailler en groupe, et produire un code plus propre.

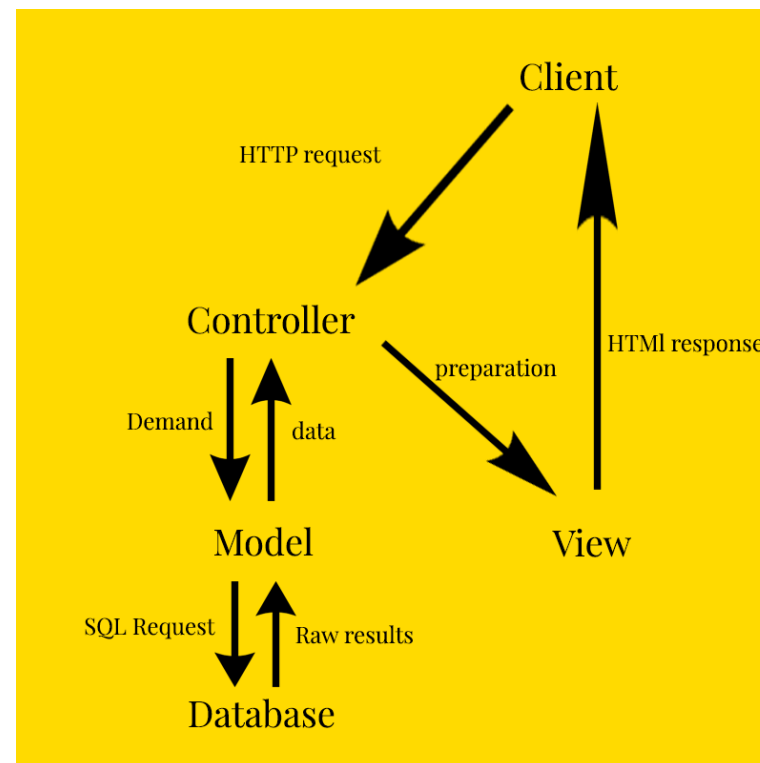
La couche **View**, ou vue, est la partie visible de l'application, affichée coté client. Avec Symfony, j'utilise le moteur de template **Twig** pour générer le rendu HTML, sans avoir à faire appel à PHP pour afficher le contenu de mes variables. Code plus propre, qui permet par exemple de dédier toute la partie View à un designer pas forcément compétent en PHP. Des extraits de code vous permettront de vous faire une idée par la suite.

La couche **Controller**, couplée au routeur (qui gère si une route donnée existe et comporte des paramètres légitimes, pour ensuite appeler la fonction correspondante), permet de faire la **passerelle** entre la couche View et la couche Model. C'est ici que le cœur du

code et du raisonnement se tient. Par exemple, si dans une vue j'ai un lien vers /home, le router va demander au controller la fonction correspondante à la **route** /home. Dans cette fonction, je vais demander à ma couche Model d'aller récupérer en **base de données** les stories récentes par exemple. Une fois les données récupérées, le controlleur les envoie à la vue afin de les afficher, après par exemple des traitements, tris ou autres actions.

La couche **Model** est la couche d'abstraction de la base de données. Quand elle est appelée par le controller, son seul rôle est d'exécuter la requête SQL demandée, de contrôler **l'intégrité** du résultat, puis de le transformer en objet, en respectant un schéma donné par nos **classes PHP** d'entités.

Un schéma vaut mieux que des explications :



L'ORM Doctrine

Doctrine est l'élément clé permettant de ne plus traiter directement avec la base de données. Il s'agit de la **couche d'abstraction** de notre base de données. Récupérer des données, puis les transformer en **objets PHP**. Doctrine est supporté et installé par défaut avec Symfony, mais son utilisation n'est pas obligatoire. Doctrine est indépendant, et est composé de deux grandes parties :

L'ORM, et la DBAL.

Commençons par la DBAL, ou **Database Abstraction Layer**. S'appuyant directement sur PDO, l'interface objet maintenant natif à PHP permettant l'accès à la base de données, DBAL rajoute une couche d'abstraction en nous permettant l'accès à des fonctions de tris, filtres, ce qui nous offre une **manipulation** de la base de données plus complète.

L'ORM, ou **Object Relational Mapping**, permet le lien entre DBAL et nos objets PHP. C'est lui qui va **transformer nos données brutes** en collections d'objets et inversement, c'est lui qui les persistera afin de les sauvegarder en base de données. Revenons un moment sur les entités : Symfony propose une gestion des relations qui requiert une certaine rigueur mais aussi une bonne anticipation du modèle à respecter. Lors de l'édition d'une entité, à l'ajout d'un nouvel attribut, nous pouvons choisir d'en faire une relation. Les relations utilisées ici sont de type **OneToOne** et **OneToMany**, Symfony mettant automatiquement à jours les fichiers de classes respectifs. Ces relations se réfèrent directement au modèle de donnée établi au préalable.

Pour résumer, c'est Doctrine qui transforme une table en instance de classe PHP. Afin de travailler avec Doctrine, une légère adaptation au **DQL** est requise. De nombreuses fonctions sont proposées, mais dès que vous souhaitez aller plus loin et exécuter des requêtes plus spécifiques, le Doctrine Query Language vous aidera à obtenir vos résultats. Le raisonnement est similaire au SQL, mais la syntaxe fournit directement les fonctionnalités de Doctrine. Exemple :

```
38  /**
39   * @return Vote[] Returns an array of Vote objects
40   * finds user votes and friends votes
41   * used for ajax pagination
42   */
43   public function findByUserIdAndFriendsOffset($id, $friends, $offset)
44   {
45       return $this->createQueryBuilder('v')
46           ->andWhere('v.author = :id OR v.author IN (:friends)')
47           ->setParameter('id', $id)
48           ->setParameter('friends', $friends)
49           ->orderBy('v.id', 'DESC')
50           ->setFirstResult($offset)
51           ->setMaxResults(3)
52           ->getQuery()
53           ->getResult()
54       ;
55   }
```

Dans l'exemple ci-dessus, nous nous trouvons dans le **repository** de nos votes, dans la couche Model. C'est-à-dire le fichier où nous pouvons écrire nos fonctions dont le seul but sera d'exécuter des requêtes. Il faut savoir que Doctrine permet l'écriture de requêtes SQL directement, mais que l'utilisation de DQL permet une meilleure **lisibilité du code**. Le repository est le seul endroit dans notre code où nous parlerons de requêtes SQL. Premier avantage de la séparation du code, pas de mélange entre accès à la base de données et fonctionnalités.

Architecture de Symfony

Composer

Composer est le manager de **dépendances** PHP, permettant l'installation, gestion et mise à jour d'un projet Symfony par exemple. C'est avec lui que j'installe mes bundles Symfony ou que je tiens à jours mes librairies. Au long du développement du projet, son utilisation n'a pas été quotidienne, surtout au début du développement ou lors de l'installation d'un nouveau bundle.

Petite précision, Composer gère les dépendances au sein du projet, contrairement à NPM par exemple. Très utile pour moi pour tester une nouvelle fonctionnalité dans une copie indépendante du projet.

Console

L'utilisation de la console a quant à elle été quotidienne. Entre Git, Composer, Doctrine, les **commandes** PHP Bin et JavaScript, mon espace de travail a toujours été partagé entre code et console Bash.

Pour le développement d'un projet Symfony, les commandes récurrentes sont celles de **lancement du serveur** propre à Symfony, de mise à jour de la base de données, d'installation de bundle et de suppression du cache de développement.

Voici quelques exemples :

```
Stagiaire@shuttle18-76 MINGW64 /c/wamp/www/strapp (dev)
$ php bin/console server:run
```

```
[OK] Server listening on http://127.0.0.1:8000
```

```
// Quit the server with CONTROL-C.
```

```
Stagiaire@shuttle18-76 MINGW64 /c/wamp/www/strapp (dev)
$ php bin/console doctrine:database:update --dumpsql
```

Ou encore l'utilisation quotidienne de Git :

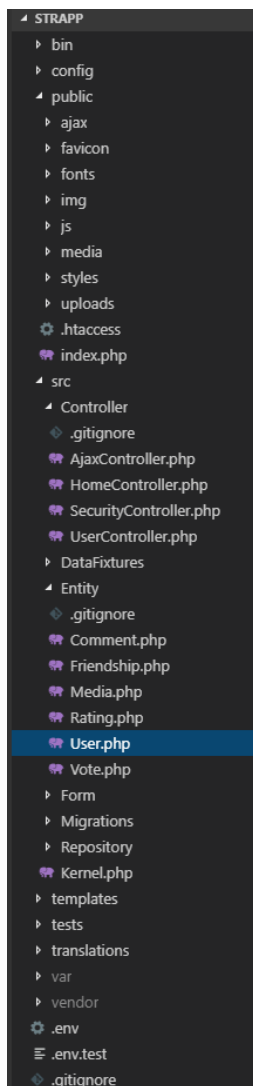
```
Stagiaire@shuttle18-76 MINGW64 /c/wamp/www/strapp (dev)
$ git add .
```

```
Stagiaire@shuttle18-76 MINGW64 /c/wamp/www/strapp (dev)
$ git commit -m "minor changes, dossier de synthèse"
[dev 14ba054] minor changes, dossier de synthèse
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 ~$ssiersynthese.docx
```

```
Stagiaire@shuttle18-76 MINGW64 /c/wamp/www/strapp (dev)
$ git push
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 4 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 218.68 KiB | 6.83 MiB/s, done.
Total 4 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/tontonsat/strapp.git
 9b86dd4..14ba054 dev -> dev
```

Architecture

Voici à quoi ressemble un projet Symfony :



Le dossier /config contient les **configurations** relatives au projet et aux bundles au format yaml.

Le dossier /public sera la seule partie accessible de l'application, contenant toutes nos **ressources**, images, css, JavaScript entre autres, ainsi que notre index.php, unique interface entre le client et le serveur.

Le dossier /src contient nos controllers, nos repositories, et nos entities. Il s'agit du **cœur** du projet, constitué aussi du kernel de Symfony, assurant l'intégrité du code. Ce dossier contient aussi nos classes de formulaires.

Le dossier /templates contient nos fichiers twig, nous n'y rédigeons que du HTML.

Vous pouvez constater aussi la présence de nombreux autres dossiers / fichiers, bien qu'étant **essentiels** au bon fonctionnement du projet, ils n'ont pas constitué un espace de travail essentiel pour moi. Par exemple, le dossier var contient le cache généré par Symfony pour le développement et la production.

Organisation

Je vous propose un **exemple** concret pour nous aider à s'y retrouver dans le projet.

Imaginons que je souhaite créer une nouvelle fonctionnalité, permettant à un administrateur de créer un nouvel utilisateur.

Nous admettons que l'administrateur est authentifié et qu'il possède le rôle (droit) adéquat.

Je commence par mon controller, où je crée une fonction dont le rôle est d'envoyer le template twig correspondant à un formulaire de création d'utilisateur. On se trouve dans le dossier /src/Controller, dans le fichier UserController.php. Une fois la fonction préparée, je **génère un nouveau formulaire** basé sur l'entité User à l'aide la console. Une fois le formulaire créé, je demande à ma fonction du controller de le traiter si une requête est reçue.

Si le formulaire est traité et valide, je demande à ma couche Model, donc à mon repository, de **persister** la nouvelle entité User afin de l'enregistrer en base de données. Si cette action se déroule avec succès, la fonction du controller se charge de rediriger l'administrateur vers une autre route correspondant à la liste des utilisateurs, avec une notification lui indiquant le succès de son action par exemple.

Voici dans les grandes lignes le déroulement de l'ajout d'une **nouvelle fonctionnalité**. Ajoutez à cela l'édition de la vue twig, les vérifications de sécurité et les tests.

Sécurité

Symfony dispose de **nombreux composants** pour gérer la sécurité de l'application. De la configuration aux templates twig, en passant par les controllers et les fichiers de classe PHP, les **couches** de sécurité se superposent.

Commençons par l'élément majeur de la sécurité que j'ai utilisé : le bundle natif **Security**, permettant de gérer les droits et restreindre les accès. Security utilise le principe de firewall, qui vient réguler l'accès global aux fonctionnalités, ou **routes**. Son rôle démarre dès l'inscription, où chaque utilisateur sera authentifié à l'aide de son username, identifiant unique.

Voici à droite le fichier de configuration de Security, afin de se faire une meilleure idée :

J'y définis d'abord l'algorithme bcrypt pour **encoder** mes mots de passe, puis le **provider**, c'est-à-dire l'entité Symfony qui servira de **référence** pour les vérifications de sécurité.

Ensuite, pour configurer les firewalls, nous lui donnons un provider, la route de connexion, de déconnexion et leur cible. Ainsi, toute tentative d'accès non authentifiée via ces routes sera bloquée.

Vient enfin la gestion des droits : tout utilisateur possède par défaut le rôle « ROLE_USER », qui lui permet d'accéder aux routes une fois **authentifié**.

Security fonctionne avec son propre **contrôleur**, qui gère par exemple l'inscription, connexion et déconnexion.

```
config > packages > ! security.yaml
1 security:
2   encoders:
3     App\Entity\User:
4       algorithm: bcrypt
5     # https://symfony.com/doc/current/security.html#where-do-users-come-from-user-providers
6   providers:
7     in_database:
8       entity:
9         class: App\Entity\User
10        property: email
11    in_memory: { memory: ~ }
12   firewalls:
13     dev:
14       pattern: ^/(_(profiler|wdt)|css|images|js)/
15       security: false
16     main:
17       anonymous: true
18
19       switch_user: true
20
21       provider: in_database
22
23       form_login:
24         login_path: security_login
25         check_path: security_login
26         default_target_path: home_home
27
28       logout:
29         path: security_logout
30         target: home_root
31
32       # http_basic: true
33       # https://symfony.com/doc/current/
34       # security.html#a-configuring-how-your-users-will-authenticate
35
36       # form_login: true
37       # https://symfony.com/doc/current/security/form_login_setup.html
38
39       # Easy way to control access for large sections of your site
40       # Note: Only the *first* access control that matches will be used
41       access_control:
42         #- { path: ^/admin, roles: ROLE_ADMIN }
43         - { path: ^/home, roles: ROLE_USER }
```


La couche suivante de sécurité vient s'ancrer directement dans les entités : Symfony propose un système d'annotations, notamment les « asserts », afin d'appliquer une contrainte à chaque attribut d'une classe PHP. Ces contraintes seront retranscrites dans la base de données grâce à Doctrine.

Voici un exemple :

```
16 /**
17  * @ORM\Entity(repositoryClass="App\Repository\UserRepository")
18  * @UniqueEntity(
19  *   fields = {"email"},
20  *   errorPath="email",
21  *   message = "email already in use.",
22  * )
23  * @UniqueEntity(
24  *   fields = {"username"},
25  *   errorPath="username",
26  *   message = "username already in use.",
27  * )
28  * @Notifiable(name="User")
29  */
30 class User implements UserInterface, NotifiableInterface
31 {
```

Vous pouvez voir ci-dessus les annotations relatives à ma classe User. Nous y spécifions que chaque utilisateur doit être unique, en se basant sur l'adresse email, agrémenté d'un message d'erreur qui sera automatiquement récupéré par notre formulaire d'inscription. Il en est de même pour le username. Ligne 28 vous pouvez aussi voir une annotation obligatoire pour un bundle utilisé pour les notifications, nous y spécifions que l'entité User est capable d'être notifiée.

Deuxième exemple, plus parlant cette fois-ci, qui concerne les asserts :

```
55 /**
56  * @ORM\Column(type="string", length=255)
57  * @Assert\Length(min = 8, minMessage="mdp trop court")
58  * @Assert\EqualTo(propertyPath="confirmPassword", message="must be equal to confirm")
59  */
60 private $password;
```

Ce qui se traduit par : l'attribut « password » de chaque utilisateur doit faire au moins 8 caractères, et doit être égal à l'attribut « confirmPassword », attribut temporaire lors de la création de compte, sinon message d'erreur.

Notez aussi à la ligne 56 un détail, bien que vital, où on spécifie que cet attribut est persisté en base de données. Donc si j'effectue une modification sur une instance de la classe User que je demande à Doctrine de le mettre à jour, le champs password le sera automatiquement.

Voilà comment nous protégeons l'application contre les intrusions. Sécurité basique mais concrète.

Viennent ensuite le contrôle des actions utilisateurs. Par exemple, comment empêcher un utilisateur de supprimer une story dont il n'est pas l'auteur. Ce sont des vérifications très basiques, mais indispensables pour un résultat cohérent. Exemple :

```
253 if($this->getUser()->getId() == $vote->getAuthor()->getId()) {
254     $manager->remove($vote);
```

Dernier composant de sécurité que j'aborderai avec vous, les annotations de routes, pour les fonctions de controller.

```
203  /**
204      * @route("/ajaxlistUserScroll/{slug}/{offset}",
        name="ajax_ajaxlistuserscroll")
205      */
206  public function ajaxListUserScroll(Request $request, $slug = null, $offset =
        null)
207  {
```

Les annotations ci-dessus permettent de bloquer brutalement l'accès à la route si les paramètres attendus ne sont pas respectés. Dans le cas présent, la fonction ajaxListUserScroll attend un premier paramètre « slug » (filtre pour la recherche), et « offset », permettant le lazy loading.

Enfin, nous spécifions le nom de la route, qui sera utilisé soit dans les fichiers de configuration, d'autres controllers, ou encore dans des templates twig pour générer des liens.

Jeux d'essais, tests

Afin de tester mes différentes fonctionnalités, j'ai dans un premier temps créé plusieurs comptes à la main pour développer par exemple l'ajout d'amis, profiles ou encore liste d'utilisateur. Mais je me suis vite retrouvé limité dans la variété de données de tests.

Symfony propose un système de DataFixtures : des classes PHP dédiées à l'ajout de données dans la base de données. Par exemple, si je souhaite ajouter 4000 utilisateurs fictifs à mon application, sans le faire à la main, je vais y instancier un objet User dans une boucle, la datafixture se chargera de les enregistrer en base de données à l'aide de Doctrine.

Mais cela ne suffit pas. Je suis allé plus loin en utilisant le bundle Faker, permettant de générer une immense variété de données fictives. Adresses postales, mail, nom d'utilisateur, images et j'en passe. Voici un extrait du résultat :

```
3  namespace App\DataFixtures;
4
5  use Doctrine\Bundle\FixturesBundle\Fixture;
6  use Doctrine\Common\Persistence\ObjectManager;
7
8  use App\Entity\User;
9  use Faker;
10
11  class UserFixtures extends Fixture
12  {
13      public function load(ObjectManager $manager)
14      {
15          $faker = Faker\Factory::create('fr_FR');
16
17          for ($i=0; $i < 100; $i++) {
18
19              $user = new User();
20              $user->setPassword('dontTell')
21                  ->setDateSignup(new \Datetime())
22                  ->setCurrentLocation('2.3480874152280933,48.87062501694089')
23                  ->setEmail($faker->email)
24                  ->setName($faker->firstNameMale)
25                  ->setLastName($faker->lastName)
26                  ->setUsername($faker->username)
27                  ->setMood($faker->sentence($nbWords = 4, $variableNbWords = true))
28                  ->setBio($faker->sentence($nbWords = 10, $variableNbWords = true))
29                  ->setRatingWriter(0)
30                  ->setRatingReader(0);
31
32              $manager->persist($user);
33          }
```

Les données générées aléatoirement m'ont permis de corriger un grand nombre de mauvais comportements dans mes vues, mais aussi dans mes formulaires. Exemple, contrôler la taille des entrées proposées aux utilisateurs. Un nom trop long, une description trop longue, bien que géré en base de données, une fois affiché j'ai eu à faire à de nombreux bugs visuels.

La phase de test la plus enrichissante, efficace et divertissante a été le **déploiement** de l'application sur serveur virtuel. J'ai pu donner faire tester plusieurs personnes, et surtout tester l'application directement sur smartphone, en conditions réelles.

Ce fût un **désastre**, mais s'il existe, dans le bon sens du terme. Dans un premier temps, j'ai été extrêmement surpris par les temps de chargement offerts par Symfony en production, contrairement à l'environnement de développement, très chargé.

Viennent ensuite les comportements des testeurs, qui sont allés chercher le moindre défaut. Ils **ont révélé des problèmes** auxquels je n'avais pas pensé, ou qui étaient littéralement sous mes yeux pendant tout le développement, sans que je m'en rende compte.

Le cas le plus important a été un souci de sécurité, mais pas du côté de l'application, mais côté serveur et navigateur : J'utilise la librairie **Geolocation** d'HTML5, mais depuis un certain temps, la plupart des navigateurs bloquent complètement l'accès si le serveur et le nom de domaine ne présente pas un **certificat SSL** à jour.

Chose à laquelle je n'avais pas pensé, mais surtout qu'il m'était impossible de tester en local. Résultat, mes utilisateurs étaient localisés dans le département de Padéma au Burkina Faso.

J'y ai remédié en installant un certificat SSL dans un premier temps gratuit, avec une redirection forçant l'HTTPS. Ainsi, j'ai la possibilité de demander l'autorisation à l'utilisateur pour récupérer sa **position**.

L'autre avantage de la mise en production a été de réellement tester l'aspect **responsive** de l'application, point clé du projet. Un certain nombre de corrections ont été requises afin d'arriver à un résultat concluant.

Extraits de code

Nous allons nous pencher sur quelques extraits de code pour se faire une idée finale du raisonnement derrière une fonctionnalité qui utilise un **grand éventail de technologies** présentes dans le projet : la barre de recherche

Nous partons du principe que l'utilisateur a déjà effectué sa recherche, je ne vous partage donc pas le formulaire en lui-même, mais le **traitement** et la **réponse**.

```
119  /**
120   * @route("/ajaxSearch/{query}", name="ajax_ajaxsearch")
121   */
122  public function ajaxSearch($query = null)
123  {
124      $em = $this->getDoctrine()->getManager();
125      $userRepo = $em->getRepository(User::class);
126
127      $queryData = explode(' ', $query);
128      $result = array();
129      $result['byName'] = array();
130      $result['byLastname'] = array();
131      $result['byUsername'] = array();
132      $result['bestResults'] = array();
133
134      foreach ($queryData as $value) {
135          if(strlen($value) >= 3) {
136              $byName = $userRepo->createQueryBuilder('u')
137                  ->where('u.name LIKE :query ')
138                  ->setParameter('query', '%' . $value . '%')
139                  ->setMaxResults(10)
140                  ->getQuery()
141                  ->getResult();
142              $result['byName'] = $byName;
143              if (empty($result['bestResult'])) {
144                  $result['bestResult'] = $byName;
145              } else {
146                  $result['bestResult'] = array_merge($result['bestResult'],
147                  $byName);
148              }
149          }
150      }
151  }
```

```

183 $result['byAll'] = array();
184 foreach ($queryData as $value) {
185     $byAll = $userRepo->createQueryBuilder('u')
186         ->where('u.name LIKE :query AND u.username LIKE :query')
187         ->setParameter('query', '%' . $value . '%')
188         ->setMaxResults(10)
189         ->getQuery()
190         ->getResult();
191     $result['byAll'][] = array_unique($byAll);
192 }
193
194 $result['bestResult'] = array_unique(array_intersect($result['bestResult'],
195     $result['byAll'], function ($res1, $res2) {
196         return spl_object_hash($res1) <=> spl_object_hash($res2);
197     }));
198
199 return $this->render('ajax/ajaxListSearch.html.twig', ['result' => $result,
200     'query' => $query]);
201 }

```

J'ai segmenté la fonction afin de vous épargner les **réurrences**, vu que la fonction va effectuer des recherches en fonction de ce qui est donné comme requête.

La fonction se charge dans un premier temps de récupérer le **repository** Utilisateur, nous permettant de demander à Doctrine d'aller faire des recherches en base de données.

Nous transformons la requête en **tableau**, et pour chaque élément de celui-ci, nous effectuons une requête qui ira chercher un résultat correspondant à un nom, prénom, et nom d'utilisateur.

Les résultats obtenus sont triés, les doublons retirés, comme on peut le voir entre la ligne 194 et 196. J'ordonne les résultats par **pertinence** : un utilisateur dont le nom est similaire à la recherche est moins pertinent qu'un autre dont le nom et le nom d'utilisateur sont tous les deux similaires à la recherche.

Le rendu est sous la forme d'un template twig, qui n'affiche qu'une liste. En voici un extrait :

```

5      {% if result['bestResult'] is not empty %}
6          {% for user in result['bestResult'] %}
7              <a href="{{ path('home_user', {'slug': user.id}) }}">
8                  <li>{% if user.media is null %}
9                      
11                      {% else %}
12                          
15                      {% endif %}
16                      <span class="search-user-info">
17                          <b>{{ user.name }} {{ user.lastname }} @{{
18                             user.username }}</b>
19                      </span>
20                  </li>
21          </a>
22      {% endfor %}
23      <div class="dropdown-divider"></div>
24  {% endif %}

```

Ce template est affiché à l'aide de Javascript, le temps rendu dynamique par AJAX :

```
public ▾ ajax ▾ JS search.js ▾ ...
1  var searchType = () => {
2      let query = $('#search-bar').val()
3      $.ajax({
4          url: '/ajaxSearch/' + query,
5          type: 'GET',
6          success: (data) => {
7              $('#search-result-container').html(data)
8          }
9      })
10 }
```

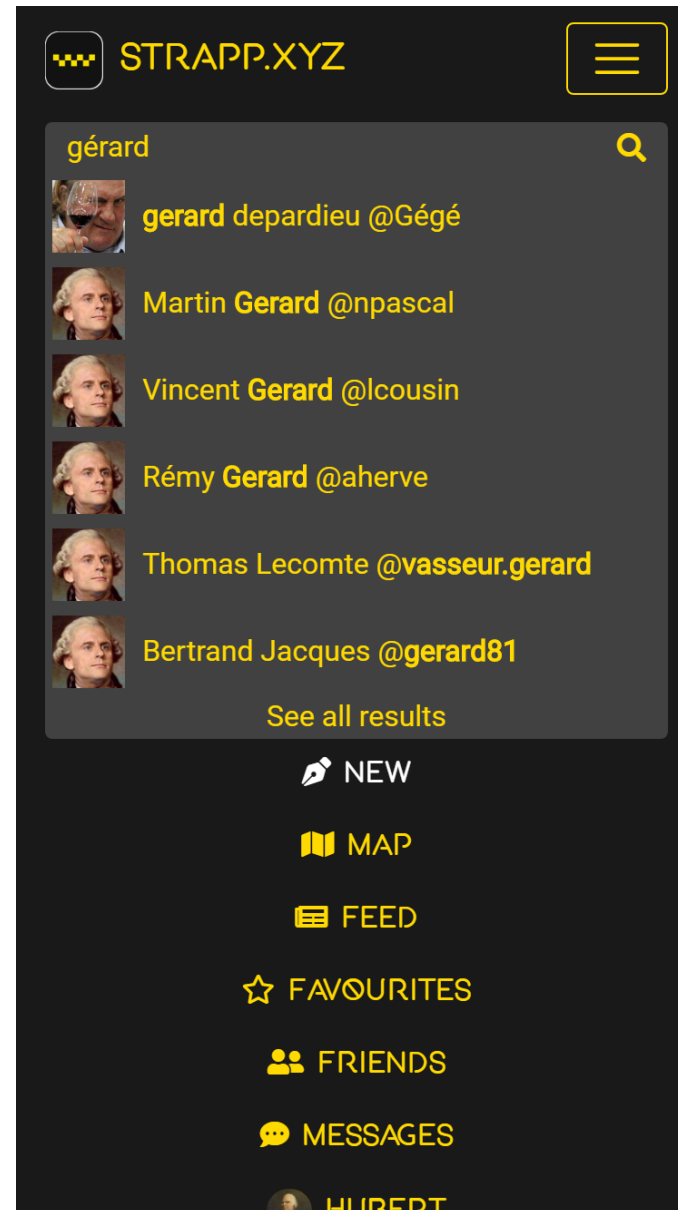
Ci-dessus, la **requête AJAX** dont le seul rôle est d'envoyer l'entrée utilisateur au controller, puis d'insérer le résultat pour l'afficher.

Il faut ensuite gérer le comportement de l'utilisateur quand il interagit avec le formulaire de recherche :

```
12  /* here the use of setTimeout and clearTimeout is essential to have a throttle for
13  ajax requests */
14  var searchBar = $('#search-bar')
15  var typingTimer
16  var doneTypingInterval = 800
17
18  searchBar.on('click paste keyup', () => {
19    clearTimeout(typingTimer)
20    if (searchBar.val().length >= 3) {
21      $('.search-bar-icon').css({
22        'border-bottom-right-radius': '0',
23        'border-bottom-left-radius': '0',
24        'border-top-left-radius': '0'
25      })
26      $('.search-bar').css({
27        'border-bottom-left-radius': '0',
28      })
29      $('.search-result-container').html('<i class="fas fa-compass fa-spin"></i>')
30      typingTimer = setTimeout(searchType, doneTypingInterval)
31    }
32  })
```

Afin d'empêcher le « spam » de requêtes, j'ai utilisé les fonctionnalités de JavaScript pour **limiter** l'appel de la requête AJAX dans le temps. Elle ne s'exécute qu'après 800ms après la dernière touche tapée, avec un minimum de trois caractères.

Vous pouvez constater le résultat à droite.



Cas pratique

J'aimerais partager avec vous le début de la suite du projet, et ma première interaction avec la technologie utilisée : Node.JS et Socket.IO. La nouvelle fonctionnalité visée est l'ajout d'une messagerie, permettant aux utilisateurs de communiquer en temps réel avec leurs amis. Cette fonctionnalité est une plateforme d'apprentissage pour moi, je vous partage mes premiers pas.

English

In order to implement a way for users to communicate, I had to look for a new technology allowing me to handle real time requests. I have already worked with AJAX (PHP and jQuery) but it seemed too heavy to operate a chat server. I knew I had to use Node.js, which is the perfect platform to do such things, but it does not allow real time queries by itself (update one client after another triggers a listener).

Socket.IO is my solution (see <https://socket.io/>). Here I have for you the documentation I followed in order to setup my first chat that would later become my chatroom server used in Strapp.xyz:

In this guide we'll create a basic chat application. It requires almost no basic prior knowledge of Node.JS or Socket.IO, so it's ideal for users of all knowledge levels.

Introduction

Writing a chat application with popular web applications stacks like LAMP (PHP) has traditionally been very hard. It involves polling the server for changes, keeping track of timestamps, and it's a lot slower than it should be.

Sockets have traditionally been the solution around which most real-time chat systems are architected, providing a bi-directional communication channel between a client and a server.

This means that the server can push messages to clients. Whenever you write a chat message, the idea is that the server will get it and push it to all other connected clients.

The web framework

The first goal is to setup a simple HTML webpage that serves out a form and a list of messages. We're going to use the Node.JS web framework express to this end. Make sure Node.JS is installed.

First let's create a package.json manifest file that describes our project. I recommend you place it in a dedicated empty directory (I'll call mine chat-example).

```
{
  "name": "socket-chat-example",
  "version": "0.0.1",
  "description": "my first socket.io app",
  "dependencies": {}
}
```

Now, in order to easily populate the dependencies with the things we need, we'll use npm install --save:

```
npm install --save express@4.15.2
```

Now that express is installed we can create an index.js file that will setup our application.

```
var app = require('express')();  
var http = require('http').Server(app);
```

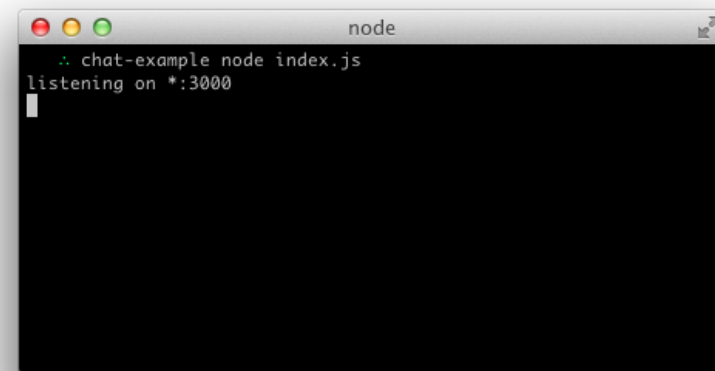
```
app.get('/', function(req, res){  
  res.send('<h1>Hello world</h1>');  
});
```

```
http.listen(3000, function(){  
  console.log('listening on *:3000');  
});
```

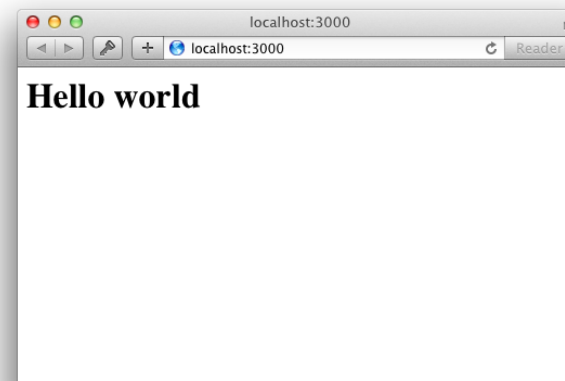
This translates into the following:

- Express initializes app to be a function handler that you can supply to an HTTP server (as seen in line 2).
- We define a route handler / that gets called when we hit our website home.
- We make the http server listen on port 3000.

If you run node index.js you should see the following:



And if you point your browser to <http://localhost:3000>:



Serving HTML

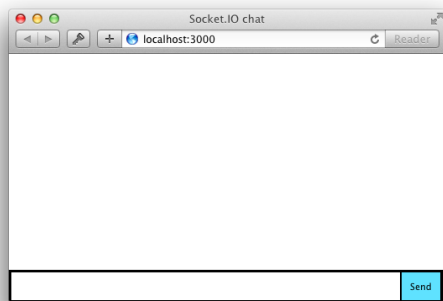
So far in index.js we're calling res.send and pass it a HTML string. Our code would look very confusing if we just placed our entire application's HTML there. Instead, we're going to create a index.html file and serve it.

Let's refactor our route handler to use `sendFile` instead:

```
app.get('/', function(req, res){
  res.sendFile(__dirname + '/index.html');
});
```

And populate index.html with the following

If you restart the process (by hitting Control+C and running node index again) and refresh the page it should look like this:



```
<!doctype html>
<html>
  <head>
```

```
    <title>Socket.IO chat</title>
    <style>
      * { margin: 0; padding: 0; box-sizing: border-box; }
      body { font: 13px Helvetica, Arial; }
      form { background: #000; padding: 3px; position: fixed; bottom:
0; width: 100%; }
      form input { border: 0; padding: 10px; width: 90%; margin-right:
.5%; }
      form button { width: 9%; background: rgb(130, 224, 255); border:
none; padding: 10px; }
      #messages { list-style-type: none; margin: 0; padding: 0; }
      #messages li { padding: 5px 10px; }
      #messages li:nth-child(odd) { background: #eee; }
    </style>
  </head>
  <body>
    <ul id="messages"></ul>
    <form action="">
      <input id="m" autocomplete="off" /><button>Send</button>
    </form>
  </body>
</html>
```

Integrating Socket.IO

Socket.IO is composed of two parts:

- A **server** that integrates with (or mounts on) the Node.JS HTTP Server: socket.io
- A **client library** that loads on the browser side: socket.io-client

During development, socket.io serves the client automatically for us, as we'll see, so for now we only have to install one module:

```
npm install --save socket.io
```

That will install the module and add the dependency to package.json. Now let's edit index.js to add it:

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.get('/', function(req, res){
  res.sendFile(__dirname + '/index.html');
});
io.on('connection', function(socket){
  console.log('a user connected');
});
http.listen(3000, function(){console.log('listening on *:3000')});
```

Notice that I initialize a new instance of socket.io by passing the **http** (the HTTP server) object. Then I listen on the connection event for incoming sockets, and I log it to the console.

Now in index.html I add the following snippet before the </body>:

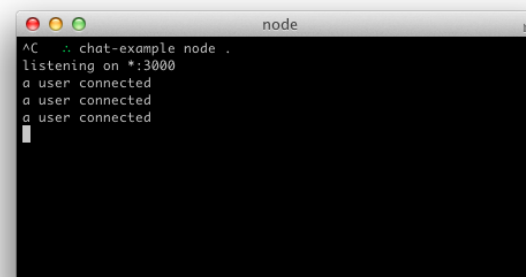
```
<script src="/socket.io/socket.io.js"></script>
<script>
  var socket = io();
</script>
```

That's all it takes to load the socket.io-client, which exposes a io global, and then connect.

Notice that I'm not specifying any URL when I call io(), since it defaults to trying to connect to the host that serves the page.

If you now reload the server and the website you should see the console print "a user connected".

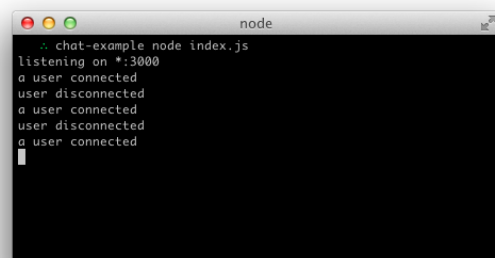
Try opening several tabs, and you'll see several messages:



Each socket also fires a special disconnect event:

```
io.on('connection', function(socket){
  console.log('a user connected');
  socket.on('disconnect', function(){
    console.log('user disconnected');
  });
});
```

Then if you refresh a tab several times you can see it in action:



Emitting events

The main idea behind Socket.IO is that **you can send and receive any events you want**, with any data you want. Any objects that can be encoded as JSON will do, and binary data is supported too.

Let's make it so that when the user types in a message, the server gets it as a chat message event. The scripts section in index.html should now look as follows:

```
<script src="/socket.io/socket.io.js"></script>
<script src="https://code.jquery.com/jquery-1.11.1.js"></script>
<script>
  $(function () {
    var socket = io();
    $('form').submit(function(e){
      e.preventDefault(); // prevents page reloading
      socket.emit('chat message', $('#m').val());
      $('#m').val('');
      return false;
    });
  });
</script>
```

And in index.js we print out the chat message event:

```
io.on('connection', function(socket){
  socket.on('chat message', function(msg){
    console.log('message: ' + msg);
  });
});
```

Broadcasting

The next goal is for us to emit the event from the server to the rest of the users.

In order to send an event to everyone, Socket.IO gives us the `io.emit`:

```
io.emit('some event', { for: 'everyone' });
```

If you want to send a message to everyone except for a certain socket, we have the broadcast flag:

```
io.on('connection', function(socket){
  socket.broadcast.emit('hi');
});
```

In this case, for the sake of simplicity we'll send the message to everyone, including the sender.

```
io.on('connection', function(socket){
  socket.on('chat message', function(msg){
    io.emit('chat message', msg);
  });
});
```

And on the client side when we capture a chat message event we'll include it in the page. The total client-side JavaScript code now amounts to:

```
<script>
$(function () {
  var socket = io();
  $('form').submit(function(e){
    e.preventDefault(); // prevents page reloading
    socket.emit('chat message', $('#m').val());
    $('#m').val("");
    return false;
  });
  socket.on('chat message', function(msg){
    $('#messages').append($('- ').text(msg));
  });
});
</script>

```

And that completes our chat application, in about 20 lines of code!

Français

Dans ce guide nous allons créer une application de chat basique. Ce guide n'ayant presque aucun prérequis sur Node.JS ou Socket.IO, il est idéal pour les utilisateurs **de tous niveaux**.

Introduction

Il est admis que développer une application de chat avec des technologies comme L.A.M.P. peut s'avérer fastidieux. Ce qui inclus des modifications coté serveur ou encore prendre en compte les horodatages, il en résulte **des temps d'exécution trop longs**.

Les sockets ont traditionnellement été le cœur des solutions de chat en temps réel, fournissant un canal de discussion bidirectionnel entre le client et le serveur. Cela signifie que le serveur peut envoyer des messages au client. Dès que vous écrivez un message, le serveur le récupère et le distribue à tous les autres clients connectés.

Le framework web

Le premier objectif est de mettre en place une simple page web HTML qui délivre un formulaire et une liste de messages.

Nous allons utiliser le framework de **Node.JS Express** pour y parvenir. Faites attention à ce que Node.JS soit installé.

Créons d'abord un fichier manifeste package.json qui décrira notre projet. Je vous recommande de la placer dans un dossier vide (je vais appeler le miens chat-example).

```
{
  "name": "socket-chat-example",
  "version": "0.0.1",
  "description": "my first socket.io app",
  "dependencies": {}
}
```

Pour maintenant peupler les dépendances avec ce dont nous avons besoin, nous allons utiliser npm install --save.

```
npm install --save express@4.15.2
```

Maintenant qu'Express est installé nous pouvons créer un fichier index.js qui se chargera de notre application.

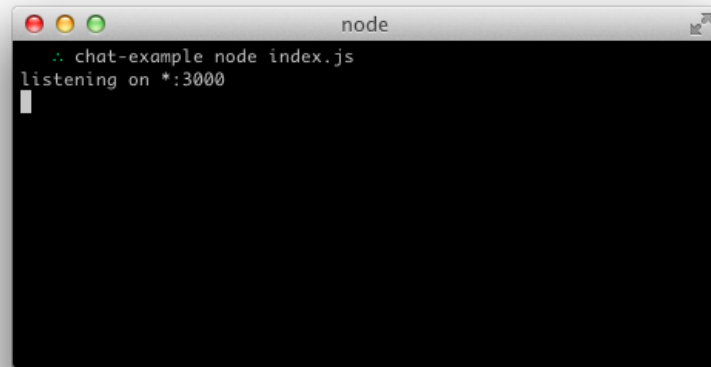
```
var app = require('express')();
var http = require('http').Server(app);
```

```
app.get('/', function(req, res){
  res.send('<h1>Hello world</h1>');
});
http.listen(3000, function(){
  console.log('listening on *:3000');
});
```

Ce qui se traduit de la manière suivante :

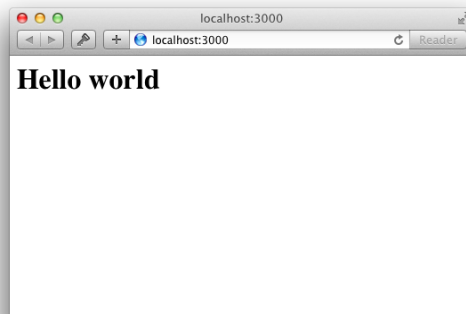
- Express initialise app pour être notre gestionnaire de fonctions, soumissible à un serveur HTTP.
- Nous définissons notre gestionnaire de route / qui sera appelé à l'accueil du site.

En exécutant node index.js nous devrions avoir le résultat suivant :

A terminal window titled 'node' with a dark background. It shows the command 'chat-example node index.js' and the output 'listening on *:3000'.

```
node
chat-example node index.js
listening on *:3000
```

Dans notre navigateur sur <http://localhost:3000>:



Servir du HTML

Jusqu'ici dans notre index.js nous appelons `res.send` en lui fournissant une chaîne de caractères HTML. Notre code serait illisible si nous y mettions tout notre HTML. Nous allons plutôt lui servir un fichier `index.html`.

Corrigeons notre gestionnaire de route :

```
app.get('/', function(req, res){
  res.sendFile(__dirname + '/index.html');
});
```

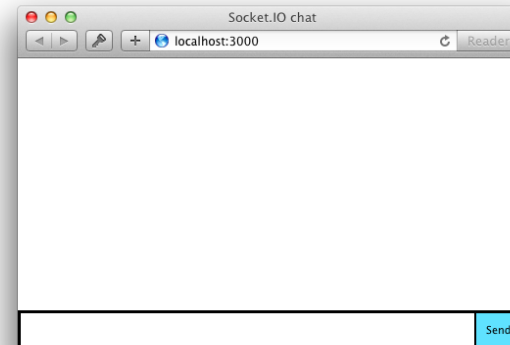
Notre `index.html` ressemble à ce qui suit :

```

<!doctype html>
<html>
  <head>
    <title>Socket.IO chat</title>
    <style>
      * { margin: 0; padding: 0; box-sizing: border-box; }
      body { font: 13px Helvetica, Arial; }
      form { background: #000; padding: 3px; position: fixed; bottom:
0; width: 100%; }
      form input { border: 0; padding: 10px; width: 90%; margin-right:
.5%; }
      form button { width: 9%; background: rgb(130, 224, 255); border:
none; padding: 10px; }
      #messages { list-style-type: none; margin: 0; padding: 0; }
      #messages li { padding: 5px 10px; }
      #messages li:nth-child(odd) { background: #eee; }
    </style>
  </head>
  <body>
    <ul id="messages"></ul>
    <form action="">
      <input id="m" autocomplete="off" /><button>Send</button>
    </form>
  </body>
</html>

```

En redémarrant le serveur, voici le résultat :



Intégrer Socket.IO

Socket.IO est composé de :

- Une **surcouche du serveur http de Node.JS** : socket.io
- Une **bibliothèque client** chargée côté navigateur : socket.io-client

Pendant le développement, socket.io sert automatiquement le client pour nous, comme nous allons le voir, donc pour le moment nous n'avons besoin que d'un seul module :

```
npm install --save socket.io
```

Cette commande va installer le module et ajouter la dépendance dans package.json.

Editons maintenant index.js :

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.get('/', function(req, res){
  res.sendFile(__dirname + '/index.html');
});

io.on('connection', function(socket){
  console.log('a user connected');
});

http.listen(3000, function(){
  console.log('listening on *:3000');
});
```

Notez que j'initialise une nouvelle instance de socket.io en lui passant l'objet http (serveur http). J'écoute ensuite l'évènement connexion pour des sockets entrants, que je log en console.

J'ajoute maintenant dans mon index.html avant </body> :

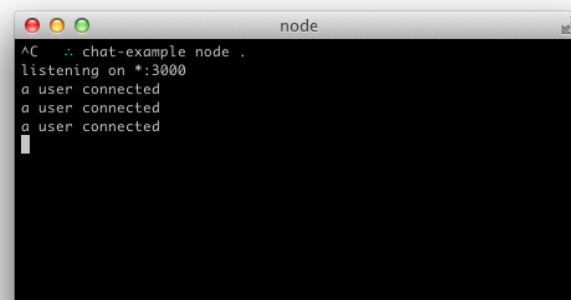
```
<script src="/socket.io/socket.io.js"></script>
<script>
  var socket = io();
</script>
```

Voici tout ce qui est nécessaire coté client pour charger socket.io-client, ce qui expose la globale io et connecter.

Notez que je ne spécifie pas d'url quand j'appelle io(), qui par défaut tentera de se connecter à l'hôte qui sert la page.

Si vous redémarrez le serveur et le site vous devriez voir en console « a user connected ».

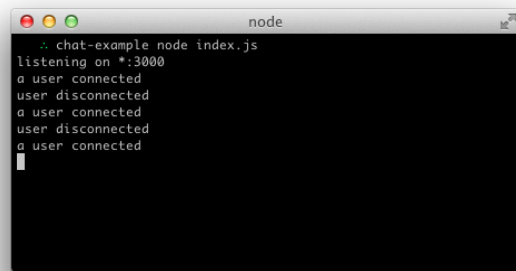
Essayez sur plusieurs onglets et vous verrez plusieurs messages :



Chaque socket lance aussi un évènement disconnect :

```
io.on('connection', function(socket){
  console.log('a user connected');
  socket.on('disconnect', function(){
    console.log('user disconnected');
  });
});
```

Si vous rafraichissez plusieurs fois la page vous pouvez le voir en action :

A terminal window titled 'node' showing the output of a chat server. The text inside the terminal is:

```
chat-example node index.js
listening on *:3000
a user connected
user disconnected
a user connected
user disconnected
a user connected
```

Emettre des évènements

L'idée principale derrière Socket.IO est de **pouvoir envoyer et recevoir n'importe quel évènement**, avec n'importe quelle donnée. N'importe quel objet encodable en JSON ou donnée binaire sera supporté.

Faisons-en sorte que lorsque l'utilisateur tape un message, le serveur l'interprète comme un évènement chat message. La section script dans index.html devrait ressembler à ceci :

```
<script src="/socket.io/socket.io.js"></script>
<script src="https://code.jquery.com/jquery-1.11.1.js"></script>
<script>
$(function () {
  var socket = io();
  $('form').submit(function(e){
    e.preventDefault(); // prevents page reloading
    socket.emit('chat message', $('#m').val());
    $('#m').val("");
    return false;
  });
});
```

```
});
});
</script>
```

Nous affichons ensuite l'évènement chat message dans index.js.

Distribuer

L'objectif suivant pour nous est d'émettre l'évènement depuis le serveur aux autres utilisateurs. Afin d'envoyer un évènement à tout le monde, Socket.IO nous fournit io.emit :

```
io.emit('some event', { for: 'everyone' });
```

Si vous souhaitez envoyer un message à tout le monde excepté un socket en particulier, nous avons le filtre broadcast :

```
io.on('connection', function(socket){
  socket.broadcast.emit('hi');
});
```

Dans notre cas, par soucis de simplicité, nous allons envoyer le message à tout le monde, y compris l'expéditeur.

```
io.on('connection', function(socket){
  socket.on('chat message', function(msg){
    io.emit('chat message', msg);
  });
});
```

```
});  
});
```

Coté client, il faut ensuite capturer l'évènement chat message et l'intégrer à la page. Le JavaScript final coté client est le suivant :

```
<script>  
$(function () {  
    var socket = io();  
    $('form').submit(function(e){  
        e.preventDefault(); // prevents page reloading  
        socket.emit('chat message', $('#m').val());  
        $('#m').val('');  
        return false;  
    });  
    socket.on('chat message', function(msg){  
        $('#messages').append($('- ').text(msg));  
    });  
});  
</script>

```

Voilà qui complète notre application de chat, en environ 20 lignes de code !

Aller plus loin

- Avertir les utilisateurs lors de la connexion d'un nouvel utilisateur
- Gérer les pseudos
- Ajouter une fonctionnalité « un utilisateur est en train d'écrire »
- Montrer la liste des utilisateurs connectés
- Messagerie privée

Conclusion

Ce projet étant loin d'être terminé, il a été pour moi ma **première interaction** avec Symfony, mais surtout mon premier véritable travail avec un framework.

Comme vous avez pu le constater, j'ai tenté de m'imposer une ligne de conduite tout au long du développement, mais la tâche n'a pas été aisée. Sujet aux changements d'avis, découvertes qui mènent à **changer de direction**, l'idée de départ est loin du résultat actuel.

J'ai eu l'occasion d'utiliser un certain nombre de composants de **Symfony**, me permettant d'avoir une certaine compréhension du framework actuellement.

Mais je n'ai que gratté la surface.

Beaucoup reste à faire : Administration, messagerie, suppression de JQuery, refonte de l'UX.

Je vais maintenant m'atteler au **référencement**, pour voir ce que l'application vaut dans un réel environnement de production, tout en affinant les fonctionnalités. Si même un petit succès est rencontré, j'envisage de **refondre** totalement le projet en le basant sur un serveur Node, avec une API Symfony gérant la base de données. Avec par exemple un front géré par React, ce qui me permettrait de continuer mon apprentissage.

Vous pouvez retrouver une version de test ici :

<https://v1.strapp.xyz>

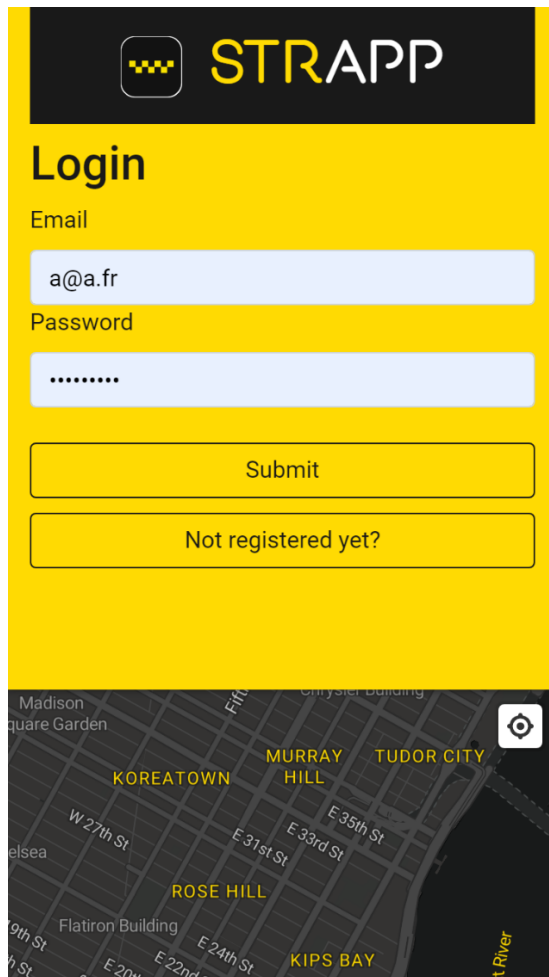
Le code est disponible ici :

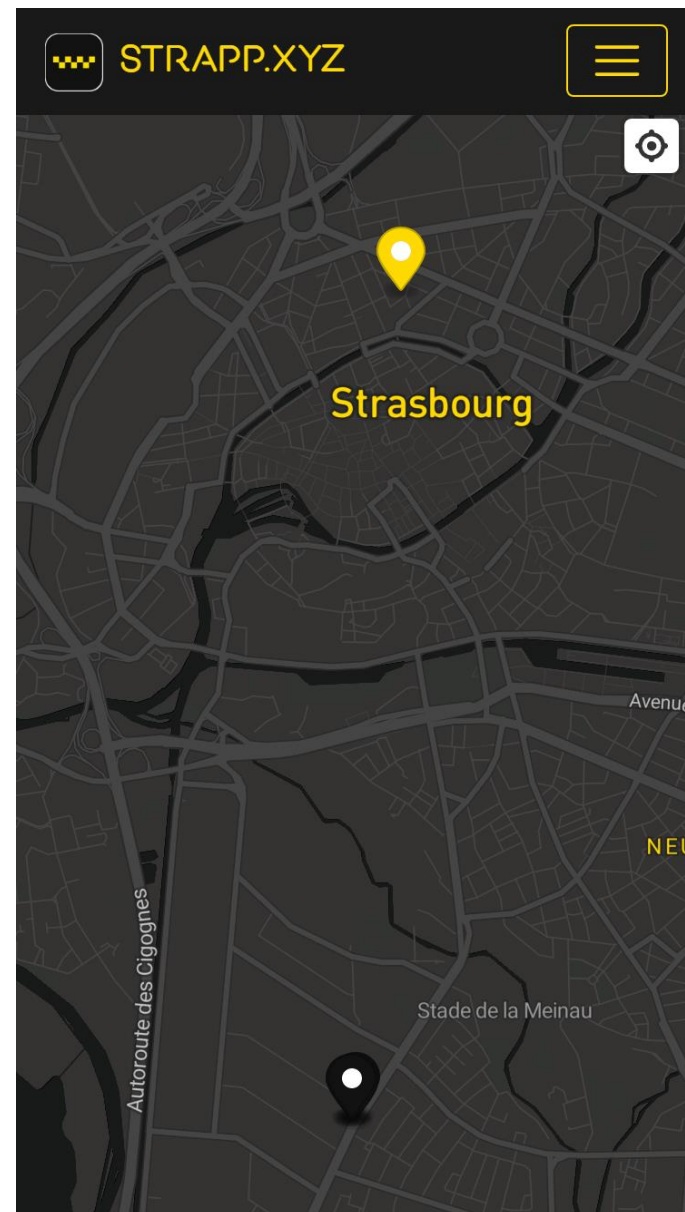
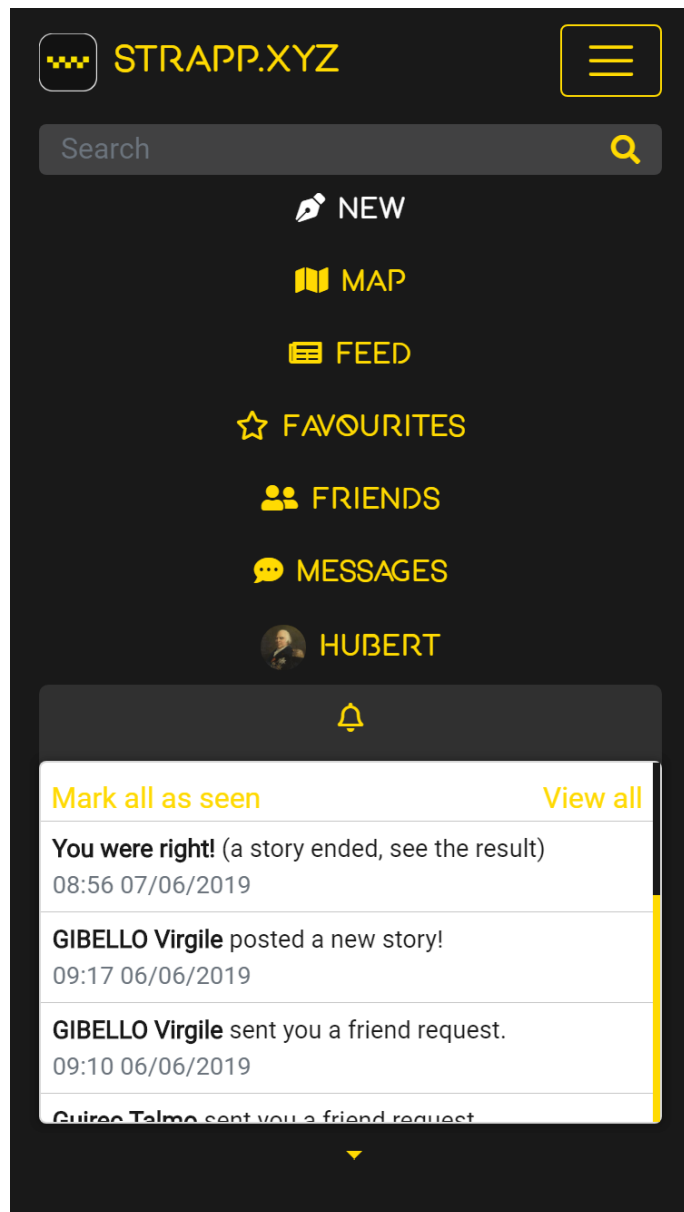
<https://github.com/tontonsat/strapp>



J'espère avoir su garder une certaine **clarté** tout au long de ce document. Ce que j'ai retenu lors de l'utilisation de Symfony, c'est que c'est un outil formidable pour développer une application qui comprendra des relations. Tout est centré autour de la gestion de ces **relations**, l'utilisation et le développement de fonctionnalités en sont fortement améliorés.


Annexes

Vous pouvez trouver ici des captures d'écran de l'application, si vous souhaitez vous en servir de **référence** lors de votre lecture.








STRAPP.XYZ





Jean-Luc Momo
@Miniz (0/0)
Strasbourg - France
2 be 3 or not to be !!!




Dudu Dada
@coco (0/0)
Strasbourg - France
I like trains





Jacques Chirac
@JacquesChirac (0/0)
Strasbourg - France
I like trains




gerard depardieu
@Gégé (0/0)
Undefined - Padéma
I like trains



Donald Trump
@WeHaveToBuildAWall (0/0)
Strasbourg - France
CLICK OFF


STRAPP.XYZ


My profile



hubert pfersdorff (0)
Strasbourg - France
Mood:hello there
30 characters max.



MY NAME IS JEEefghghghgh

Personnal information

Name

hubert

Lastname

 STRAPP.XYZ

Story

Main information

Title

Content

Options



Duration


1 hour ▼


Result


real ▼


Submit story

 STRAPP.XYZ


hubert


**hubert** pfersdorff @tontonsat


**Anouk Rodrigues** @hubert.suzanne


**Charles Colas** @hubert.pauline


See all results


 NEW


 MAP


 FEED

 FAVOURITES

 FRIENDS

 MESSAGES

 HUBERT



▼

RGPD

Je terminerai en abordant les données des utilisateurs stockées par l'application.

A l'inscription, je demande **explicitement** nom, prénom, adresse email, nom d'utilisateur et mot de passe, l'utilisateur aura ensuite l'option de personnaliser son avatar, par le biais d'un fichier stocké. Je demande aussi **implicitement** les positions géographiques de l'utilisateur. Sujet à risque en ce moment avec les récents événement notamment chez Facebook, où une fuite de données peut entraver la vie privée des utilisateurs. Les données sont pour le moment uniquement stockées en base de données, et utilisées dans l'application, du simple affichage.

Chaque utilisateur a la possibilité de **modifier** toutes ses données personnelles excepté son nom d'utilisateur, qui doit rester unique, et qui permet de conserver l'intégrité de ses actions. Une fonction de **suppression de compte** est aussi disponible. Supprimer son compte efface toute donnée relative à l'utilisateur, à l'exception des votes réalisés, afin de ne pas **pénaliser** les autres utilisateurs. Les votes étant anonymes.

L'application étant basée sur les données de l'utilisateur, il sera averti de leur utilisation.

Remerciements

Un grand merci à l'organisme de formation Elan à Strasbourg, plus particulièrement mes deux formateurs **Mickaël** et **Virgile**, qui de part leur patience, bien vaillance et curiosité ont su m'inspirer tout au long du projet.

Je tiens également à remercier toute l'équipe chez Stack Overflow, Google, SensioLabs et GitHub, ainsi que tous les anonymes m'ayant aidé à résoudre mes problèmes. Parfois même une suggestion de fonction lors de la lecture de **commentaire** a suffi à me diriger vers la bonne voie.

How almost every Stack Overflow question looks to me I agree ✕

How do I do this thing?

43 coding question share | improve this question edited Apr 2 '12 at 8:13 Grammar Nazi 2.5M 7 39 70 asked Feb 1 '10 at 16:27 1337z0r 2 1 3 6 coding × 155474 question × 37256 asked 3 years ago viewed 43962 times active 1 month ago

5 Answers active oldest votes

12 share | improve this answer answered Feb 1 '10 at 16:30 Joe the Coder 1,230 3 14 25

That's perfect! I'm never checking back here again! – 1337z0r Feb 1 '10 at 16:42

248 share | improve this answer answered Feb 1 '10 at 16:30 WTF look at my points 34M 400 150 60

18 share | improve this answer answered Feb 1 '10 at 16:29 Professional Coder 5,241 2 24 63

Uh... No thanks, this is too hard. Can you give me an example of how my code should look when complete? – 1337z0r Feb 1 '10 at 16:41

Community Bulletin

event Microsoft can help you port your app to Windows 8 - win prizes with Appviate! – now through June 7

Work. From Home.

If you are a dog...

CAREERS 2.0

There really are jobs for coders available! Not for you though. Kiersted Systems Houston, TX / relocation

Highly paid, competitive benefits; that one language you didn't learn.