

Big Band

DOSSIER DE SYNTHÈSE

Martin Bessey

TITRE PROFESSIONNEL DÉVELOPPEUR WEB /WEB MOBILE 2021

SOMMAIRE

INTRODUCTION.....	4
I. PRÉSENTATION DU PROJET	5
1.1. Genèse du projet.....	5
1.2. Compétences couvertes	5
1.3. Gestion de projet	7
1.4. Cahier des charges	9
2. DESCRIPTION TECHNIQUE.....	11
2.1. Langages et technologies utilisés	11
2.2. Cadre légal.....	12
2.3. Modélisation des données	14
2.4. Maquette et charte graphique	15
2.6. Introduction à Symfony	17
2.7. Doctrine ORM.....	17
2.8. MVC	18
3. SÉCURITÉ.....	22
3.1. Principales failles.....	22
3.2. Authentification & autorisation.....	24
3.3. Captcha.....	28
3.4. Upload de fichiers	30
3.5. Interface d'administration	31
4. FONCTIONNALITÉS MAJEURES	32
4.1. Recherche multi-filtres.....	32
4.2. Gestion d'API et géolocalisation	33
4.3. Système de following	36
4.4. Système de publications et commentaires	39
4.5. Messagerie.....	40
AXES D'AMÉLIORATION.....	42
CONCLUSION	43
BIBLIOGRAPHIE/SOURCES.....	44
ANNEXES	45

INTRODUCTION

Je m'appelle Martin, j'ai 26 ans, titulaire d'un master en design de produits, j'ai choisi d'effectuer ma reconversion en tant que développeur web. On peut prêter certains points communs aux deux disciplines, notamment le fait que ces deux professions sont des métiers de projets, le développeur comme le designer doivent soumettre un certain nombre de projets personnels à un recruteur pour prouver de leurs qualités. De plus, les deux métiers exigent une veille technologique et culturelle dans le but d'actualiser constamment ses connaissances.

Je tiens à remercier l'équipe d'Élan Formation et plus précisément Gilles MUESS et Stéphane SMAIL pour leur aide, leur disponibilité et leur enthousiasme envers le groupe. Enfin, je remercie mes proches pour leurs soutiens ainsi que les membres de ma promotion pour l'ambiance toujours conviviale et l'entraide qui a été établie durant ces mois de formation.

I. PRÉSENTATION DU PROJET

I.1. Genèse du Projet

La genèse du projet provient d'un constat personnel établi lors d'un échange Erasmus en 2017. En tant que musicien découvrant un nouveau pays et une nouvelle ville, il ne m'était pas toujours aisé de rencontrer des personnes avec qui jouer de la musique. J'entends par là, idéalement des musiciens partageant les goûts musicaux et ayant un niveau d'expérience approprié. Mais cela n'était pas toujours le cas. Après recherche j'ai constaté qu'aucune application d'ampleur n'existait dans le domaine. Toutefois nouvelle initiative bulgare nommée Drooble propose aux artistes sous la forme d'un réseau social de partager et commercialiser leurs morceaux, en proposant des outils de promotion payants. Ces morceaux peuvent aussi être notés par les autres membres. En France, deux projets portés par des étudiants, mais toutes deux fortement inspirées de l'application de rencontre Tinder, tant dans la mise en forme que dans les fonctionnalités proposées.

Les musiciens sont présents dans toutes les couches de notre société. C'est pourquoi la création d'un réseau social dédié permettrait d'établir un espace de partage commun. Le principe de Bigband est donc de mettre en relation des musiciens partageant les mêmes goûts musicaux dans un périmètre géographique restreint. Ciblant en priorité des individus jeunes et aguerris aux réseaux sociaux, le but de BigBand est de viser un type précis d'utilisateur, tout en intervenant en complément d'autres réseaux sociaux ou plateformes de partage de musique.

I.2. Compétences couvertes

Un certain nombre de compétences sont requises afin de prétendre à l'obtention du Titre professionnel de niveau III Développeur Web et Web Mobile présentées ci-dessous sont extraites du REAC (Référentiels Emploi Activités Compétences) Développeur Web et Web Mobile. Les compétences travaillées durant le processus de production de l'application présentée sont les suivantes :

Développement de la partie front-end d'une application web ou web mobile en intégrant les recommandations de sécurité.

CP1 : Maquetter une application.

Le maquetage du site illustré dans le chapitre consacré, a été réalisé grâce à Figma qui fut l'outil utilisé pour réaliser l'ensemble de ce travail.

CP2 : Réaliser une interface utilisateur web statique et adaptable.

Pour faciliter la création des pages responsives et adaptables, j'ai privilégié l'utilisation d'une librairie CSS (UIKit) sur la majeure partie des pages. Cette solution permet de faciliter l'adaptation des pages aux formats d'écrans les plus courants.

CP3 : Développer une interface utilisateur web dynamique.

Le **Framework**¹ Symfony et notamment **Twig** son moteur de template offrent une solution web complète par rapport pour la création de pages web dynamique. Presque l'entièreté du site se veut dynamique. Car dès lors que l'utilisateur passe par la phase d'authentification la majeure partie du contenu des pages est extrait de la base de données

CP4 : Réaliser une interface utilisateur avec une solution de gestion de contenu ou e-commerce

Les utilisateurs peuvent gérer les contenus (en téléchargeant des fichiers) ainsi que les informations liées à leur profil.

Développement de la partie back-end d'une application web ou web mobile en intégrant les recommandations de sécurité

CP5 : Créer une base de données

L'ensemble des informations concernant les utilisateurs, ainsi que la majeure partie des entités du site sont stockées dans une base de données relationnelle.

CP6 : Développer les composants d'accès aux données.

Symfony intègre par défaut un logiciel (Doctrine) qui a permis au travers de la création des entités et de leurs repositories d'implémenter des méthodes d'accès aux enregistrements en base de données.

CP7 : Développer la partie back-end d'une application Web ou Web mobile.

La partie Back-End comprend plusieurs modules développés sous le Framework Symfony. Ils permettent notamment d'insérer des données dans une base de données. Dans l'autre sens des données sont obtenues à partir de la base de données pour être renvoyé vers la partie front-end.

CP8 : Élaborer et mettre en œuvre des composants dans une application de gestion de contenu ou e-commerce

¹ Un **Framework** est un cadre de travail, une infrastructure qui suit un schéma de fonctionnement et qui souvent inclut des bibliothèques. Son utilisation permet un gain de temps considérable.

I.3. Gestion de projet

La mise en œuvre d'une gestion du projet a permis de mettre en pratiques des notions rencontrées lors de mon stage en entreprise et notamment les **méthodes agiles**. Ce sont des méthodologies de travail regroupant des équipes pluridisciplinaires, elles permettent d'inclure le client dans la prise de décision tout au long de la réalisation du projet.

Pour planifier les tâches à effectuer, et définir clairement leur état d'avancement j'ai privilégié une méthodologie inspirée de la méthode agile **Kanban**. Cette dernière permet de représenter un flux de travail à travers un tableau (**tableau Kanban**) comportant des colonnes correspondant à divers états d'avancement (« à faire », « en cours », « bloqué » et « fait »). Les tâches à effectuer sont représentées par des étiquettes. Ces dernières sont classées dans le tableau en fonction de leur état d'avancement. Cela permet d'avoir une vision d'ensemble instantanée de l'avancée du projet. De plus, il est plus facile d'identifier les tâches à prioriser.

Pour mettre en place cette méthode, l'utilisation de l'outil en ligne **Trello** a été privilégiée. Grâce à son système de tableau, il est possible d'afficher les différentes tâches à réaliser et spécifier leur état d'avancement (« à faire », « en cours », « bloqué » et « terminé »). On déplace la tâche de colonne en colonne au fur et mesure de son avancement.

Ci-dessous : Capture d'écran du tableau du projet dans Trello.



Afin de prioriser les fonctionnalités essentielles et d'assurer dans tous les cas la livraison d'un produit fonctionnel, la solution privilégiée a été celle du **MVP (Minimum Viable Product)**. Elle permet de définir la version la plus minimaliste d'un produit, sans entraver la fonctionnalité. C'est ce minimum acceptable qui est testé auprès d'utilisateurs, appelés clients, avant le lancement d'une version aboutie du produit. Les retours effectués permettront d'améliorer le MVP, de l'enrichir avec de nouvelles fonctionnalités, jusqu'à devenir le produit fini. Le MVP de l'application regroupe ainsi certaines des fonctionnalités décrites dans le cahier des charges.

Ci-dessous : Schéma présentant les avantages de la méthode MVP face à une méthode de conception traditionnelle.

Méthode de conception traditionnelle



Méthode de conception d'un MVP



I.4. Cahier des charges

Problématique :

Il est parfois difficile, de rencontrer des musiciens qui partagent les mêmes goûts musicaux, quand on arrive dans une nouvelle ville. Mettre en relation des musiciens partageant les mêmes goûts musicaux, dans un périmètre géographique prédéfini par l'utilisateur. Le parcours d'un nouvel utilisateur se définirait en quatre étapes :

- **L'enregistrement et la création d'un profil.** L'utilisateur doit renseigner ses informations personnelles (emplacement géographique, instruments et styles joués notamment) pour permettre aux autres d'effectuer une recherche filtrée. Il peut également compléter son profil d'une photo de profil et de liens vers des réseaux sociaux (Instagram, Facebook, Twitter...) ou des plateformes de musique en ligne (Bandcamp, Soundcloud...)
- **La recherche filtrée.** L'utilisateur a la possibilité d'activer des filtres de recherche pour cibler un périmètre géographique, un type d'instrument ou un genre de musique en particulier.
- **Système de suivi.** Une fois le résultat de la recherche examiné, l'utilisateur a le choix parmi une variété de profils et peut décider suivre ces derniers pour consulter tous leurs posts. Et les contacter par messagerie.
- **Le contact par messagerie.** Tout utilisateur peut contacter ou être contacté par un utilisateur qui le suit.

Le site web doit posséder une partie administrateur où celui-ci peut se connecter et se déconnecter, ajouter, modifier et supprimer un post, un message, un commentaire ou un utilisateur. L'administrateur a également accès à tout ce que l'utilisateur a accès. À partir de ces étapes il est donc possible de définir les fonctionnalités générales et avancées pour prioriser les fonctionnalités en vue du processus de développement de l'application. Afin que l'ensemble des fonctionnalités listées soit disponible sur tout type d'appareils, l'application doit être responsive c'est-à-dire avec un contenu qui s'adapte aux différentes tailles d'écrans.

Fonctionnalités générales :

- INSCRIPTION & AUTHENTIFICATION.
- SYSTÈME DE SUIVI.
- MESSAGERIE.
- TÉLÉCHARGEMENT DE FICHIERS.

- SYSTÈME DE PUBLICATION.
- RECHERCHE FILTREE.
- INTERFACE D'ADMINISTRATION

Fonctionnalités avancées :

- GÉOLOCALISATION
- SYSTÈME D'APPRECIATIONS
- COMMENTAIRES SUR LES PUBLICATIONS

2. DESCRIPTION TECHNIQUE

2.1. Langages et technologies utilisés

LANGAGES :



HTML5 (HyperText Markup Language) :

Langage utilisé pour la structure sémantique des pages du site.



CSS3 (Cascading Style Sheet) :

Langage utilisé pour mettre en forme des éléments contenus dans les fichiers HTML.



PHP (Hypertext Preprocessor) :

PHP est un langage de programmation libre coté serveur, utilisé pour la création de pages web dynamiques.



JavaScript :

Langage utilisé pour l'animation d'éléments des pages HTML.

TECHNOLOGIES :



Symfony 5 : Symfony est un framework orienté objet écrit en PHP, qui permet de travailler plus rapidement et de gérer des failles de sécurité plus simplement. Composer lui est associé afin de gérer les bibliothèques (entités, contrôleurs).



Twig : Moteur de template en PHP qui permet de faciliter l'affichage de données du back-office sur les pages HTML.



Doctrine ORM : Logiciel présent par défaut dans Symfony. Permet de mettre en relation les objets PHP avec la base de données.



Composer : Logiciel gestionnaire de dépendances libre écrit en PHP.



Looping : Logiciel de modélisation de base donnée relationnelles SQL.



Laragon + HeidiSQL : Interface de gestion de bases de données SQL.



Visual Studio Code : IDE sur lequel le projet a été réalisé.



Figma : éditeur de graphiques vectoriels et outil utilisé ici pour le maquettage de l'application.



UIKit : Librairie CSS accessible à partir d'un **CDN**², permettent d'utiliser des classes CSS présentent dans une feuille de style préétablie.



Mapquest Geocode API : API en partie OpenSource permettant d'établir une géolocalisation.

2.2. Cadre légal

Le projet consistant à mettre en relation des utilisateurs entre eux, ces derniers doivent renseigner un certain nombre de données personnelles et peuvent fournir s'ils le souhaitent, une photo de profil ainsi que des photos personnelles ou des liens vers leur page personnelle sur d'autres réseaux sociaux. Toutefois, aucun visiteur externe ne possédant pas de compte utilisateur ne pourra avoir accès au site. Concernant les communications avec d'autres utilisateurs, il est nécessaire d'envoyer une invitation à un utilisateur tiers avant de pouvoir lui adresser un message. Enfin, un utilisateur doit avoir la possibilité à tout moment de modifier ou supprimer ses informations personnelles ainsi que son compte. Conformément à la législation en vigueur et notamment au **RGPD**.

² **CDN** (Content Delivery Network) est constitué d'ordinateurs reliés en réseau à travers Internet et qui coopèrent afin de mettre à disposition du contenu ou des données à des utilisateurs. Accessibles via des liens.

Le **RGPD** « **Règlement Général sur la Protection des Données** » (en anglais « General Data Protection Regulation » ou GDPR). Le RGPD régit le traitement des données personnelles au sein de l'Union européenne. Le contexte juridique s'adapte pour suivre les évolutions des technologies et de nos sociétés (usages accrus du numérique, développement du commerce en ligne...). Ce règlement européen s'inscrit dans la lignée d'une loi française (Informatique et Libertés) de 1978 et renforce le contrôle par les citoyens de l'utilisation qui peut être faite de toutes les données les concernant. Il harmonise également la législation européenne en offrant un cadre juridique aux professionnels.

Tout organisme, quel que soit son échelle, son pays d'implantation et son secteur d'activité, peut être potentiellement concerné. Car toute organisation (qu'elle soit publique ou privée), qui traite des données personnelles pour son compte ou celui d'un tiers est concernée, dès lors :

- Qu'elle est établie sur le territoire de l'Union européenne.
- Que son activité cible directement des résidents de l'Union européenne.

Afin de respecter les principes du RGPD dans le projet, les principes suivants ont été adoptés :

- La modélisation des données s'est faite en respectant le principe de **minimisation des données** recueillies sur l'utilisateur.
- **Aucune donnée sensible** ou considérée comme tel par la **CNIL**³ ne sera collectée.
- Tout utilisateur a la possibilité s'il le souhaite, de supprimer son compte et les données associées dans le cadre du **droit à l'oubli**.
- Lors de l'inscription, l'utilisateur doit **consentir** aux **conditions générales d'utilisation** de manière explicite. Ces conditions rappellent les raisons de la collecte des données, ainsi que leurs fins dans le cadre du **devoir de transparence**. Mais permettent également un rappel des droits et des devoirs de l'utilisateur. Une case à cocher permet de s'assurer du consentement **libre spécifique et éclairé**.

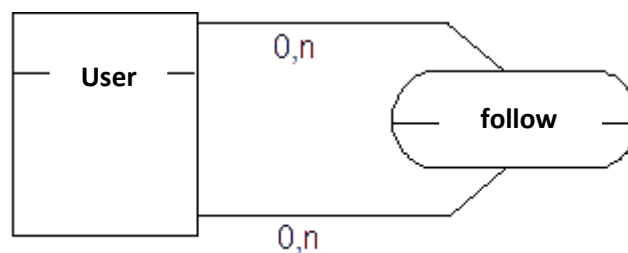
³ La **CNIL** (Commission Nationale Informatique et Liberté) est un organisme indépendant français, en charge de veiller à la protection des données personnelles des citoyens. Elle possède le pouvoir de contrôle et de régulation, mais également de sanction en cas de non-respect de la loi.

2.3. Modélisation des données

Afin d'établir les fondements du fonctionnement de l'application, l'étape de la modélisation des données et notamment la méthodologie **Merise** ont permis d'établir un point de départ. Cette méthode vise à déterminer le type de données requises ainsi que les relations entre ces dernières dans le but de guider le processus développement. Cette analyse, a ensuite eu pour résultat une représentation abstraite, de ce qui par la suite allait être la base de données sous la forme d'un **MCD**⁴ et d'un **MLD**⁵ (voir en annexe).

On constate la présence d'une entité prédominante **User**, qui représente un utilisateur. Comme toute entité, elle est constituée d'une **propriété identifiante** (id) et de **propriétés quelconques**. **User** est liée à différentes entités qui lui permettent de compléter les informations définissant le profil d'un utilisateur comme entre autres **Instrument**, **Style**, **SocialMedia** ou **Image**. Ces entités correspondent respectivement à un instrument, un style de musique, des liens vers des réseaux sociaux et des images.

L'association **follow** permet aux utilisateurs de s'inviter entre eux. Et de ce fait, de communiquer par message. Elle représente une **association réflexive** car elle relie l'entité **User** à elle-même.



Ci-dessus : Schéma de l'association réflexive.

L'entité **Post** est liée par deux associations à l'entité **User**. Les associations **publish** et **comment**. On remarquera dans le MCD que l'association **comment** possède deux cardinalités maximales de type N elle se transforme donc en une nouvelle table dite **table d'association** dans le MLD. Cette dernière contiendra deux nouvelles propriétés qui correspondent aux **clés primaires** (ou propriétés identifiantes) des deux entités qu'elle associe. On parle de **clés étrangères**. Au passage l'entité **Post** contient aussi une clé étrangère correspondant à la valeur de clé primaire de l'entité **User**.

⁴ Le **MCD** ou **modèle conceptuel des données** est une représentation graphique des données utilisées par le système d'information à l'aide d'entités, d'associations et leurs cardinalités respectives.

⁵ Le **MLD** ou **modèle logique des données** représente graphiquement la base de données sans définir les cardinalités entre les associations. Il consiste à transformer toute entité en table et montrer les clés étrangères pour chaque table ou association.

2.4. Maquette et choix graphiques

La partie front-end est déterminé durant le maquetage de l'application, ce processus est divisé en plusieurs phases. Le zoning constitue la première phase de ce processus, il consiste à diviser une page en plusieurs zones afin de créer un premier niveau d'agencement du site. La phase suivante, le wireframe, permet de définir les différentes zones de contenus d'une page web en délimitant de simples blocs et en établissant une logique de navigation entre chacune des pages. Enfin la maquette proprement dite vient préciser le contenu visuel du wireframe en établissant une gamme colorée un choix de typographie dans le but de s'approcher de la version finale du site.

La **page d'accueil** du site pour un utilisateur non connecté, consiste en un contenu principal présentant le site ainsi que des liens qui renvoient vers la page de connexion ou d'inscription. Après connexion, l'utilisateur a accès à sa page d'accueil personnelle. Cette dernière contient les posts des personnes en contact avec l'utilisateur en session. Ce dernier peut commenter un post grâce au lien en dessous de celui-ci.

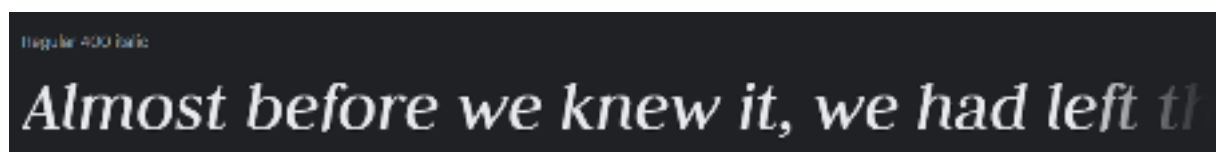
Le **header** comporte un lien vers un bouton, qui représente un lien la page d'accueil. Ainsi qu'un menu de navigation, comprenant un lien vers le profil personnel de l'utilisateur en session, représenté par le nom et la photo de profil de celui-ci. Un lien vers la page d'accueil de l'utilisateur, pour la messagerie ainsi que le lien de déconnexion.

Lors d'une recherche de musiciens, l'utilisateur en session accède à une liste de profils. Chacun est représenté par son nom, sa photo de profil, sa localisation, ainsi que ses instruments représentés par une série d'icônes. Un fichier audio personnel (si ce dernier en possède un). En cliquant sur un profil, une vue présentant les informations du musicien sélectionné en détail apparaît. D'ici l'utilisateur en session a la possibilité de suivre ou non le musicien, ainsi que d'accéder à la liste des autres musiciens suivis ou qui suivent ce dernier.

Le logo est décliné en or et blanc ou or et noir en fonction de la couleur de fond (voir ci-dessous).



La police de caractères utilisée pour certains contenus du site est Judson, une police disponible gratuitement via Google Fonts (voir ci-dessous).



Au niveau de la charte graphique du site (voir ci-dessous), la gamme colorée utilisée joue sur trois couleurs présentes en fond, un gris foncé (#2A2824), un gris moyen (#BFBFBF) et un gris clair (#FAFAFA). Ainsi que deux couleurs de premier plan : un or foncé (#2A2824) présent dans le logo, ainsi que sa complémentaire (#CEF2EF) un bleu pastel qui permet un contraste fort.



Adaptabilité

Outre l'identité visuelle du site, le travail de maquettage a aussi permis de penser l'aspect **responsive** c'est-à-dire l'adaptabilité du contenu aux différentes tailles d'écrans (voir annexes). Pour ce faire le code CSS contient notamment des **Media Queries**. Ces dernières permettent l'attribution de propriétés CSS uniquement pour certaines tailles d'écrans. Des choix en vue d'adapter l'interface utilisateur ont été opérés comme l'apparition d'un **menu burger** en lieu et place de la barre de navigation. Dans un premier temps l'interface utilisateur a été privilégiée mais dans une prochaine version, l'adaptabilité de l'interface d'administration devra être modifiée pour être entièrement responsive.

UI/UX

La phase de maquettage permet également d'avoir une première idée de **l'interface utilisateur (UI)** et de **l'expérience utilisateur (UX)**. Concernant l'ergonomie différentes pratiques ont été instaurées dans le but de se conformer aux recommandations de la **W3C**⁶. L'UI et l'UX ont fait l'objet d'une attention particulière dans le processus de maquettage en vue de respecter notamment **la règle des trois clics**⁷.

Référencement

Afin de prendre en compte le **SEO** (Search Engine Optimisation). Ce terme définit un ensemble de techniques mises en œuvre pour à mettre en avant un site web sur les pages de résultats des moteurs de recherche. On parle aussi de **référencement naturel**. Pour favoriser ce dernier une attention particulière a été portée au **balisage sémantique** dans le code HTML.

- Des balises **strong** permettent de mettre en valeur des mots clés.
- Les images contiennent des attributs **alt**.
- Les pages contiennent des balises **meta**.

⁶ **W3C : Le World Wide Web Consortium**, abrégé par le sigle W3C, est un organisme de standardisation à but non lucratif, chargé de promouvoir la compatibilité des technologies du World Wide Web .

⁷ **La règle des trois clics** est une règle de conception non officielle, stipulant qu'un internaute ne puisse accéder à l'information recherchée en ne cliquant pas plus de trois fois sur des liens hypertextes depuis la page d'accueil.

2.5. Introduction à Symfony

Symfony est un framework en langage PHP. Un framework représente un cadre de travail qui suit un schéma de fonctionnement défini. Il présente un certain nombre d'avantages par rapport à une application développée en PHP sans framework. Concernant le développement, Symfony permet d'intégrer des solutions de sécurité de manière systématique sans les implémenter à chaque fois. De plus, un certain nombre d'extensions permettent au framework de proposer des solutions rapides et sécurisées à des fonctions essentielles (login, interface d'administration, etc.). L'utilisation de certaines extensions ou leur utilisation à outrance, peut toutefois venir alourdir considérablement l'application. Il est donc nécessaire d'en mesurer le besoin réel avant installation. La création d'un réseau social à l'aide de Symfony permet de bénéficier de l'adaptabilité et de la flexibilité d'un des framework PHP les plus répandus, ainsi que son excellente documentation technique.

En tant que framework PHP, Symfony dispose de technologies intégrées assurant son bon fonctionnement. **Twig**, son moteur de template associé PHP et HTML afin de créer des pages web dynamiques. **Composer** a été choisi comme gestionnaire de dépendances et permet d'importer et de gérer les différentes extensions liées à Symfony via l'invite de commande directement depuis **Visual Studio Code**.

2.6. Doctrine ORM

Concernant la relation avec la base de données, Symfony dispose par défaut de **Doctrine** qui est un **ORM** libre sous licence. Un **ORM** (Object Relational Mapping) est un type de programme informatique qui se place en interface entre un programme applicatif et une base de données relationnelle pour simuler une base de données orientée objet. Ce programme définit des correspondances entre les schémas de la base de données et les classes du programme applicatif. Si son utilisation dans le cadre d'un projet Symfony est optionnelle, il permet ici la persistance des objets PHP, en créant un lien ou "mapping" entre les objets PHP et les éléments de la base de données.

Doctrine permet au début du projet de créer les tables correspondantes aux entités grâce à un système de migration activable via l'invite de commandes. Ce système permet également de maintenir à jour la base de données.

```
PS C:\Users\marti\Documents\GitHub\BigBand> doctrine make:migration migrate
```

Ci-dessus : la commande permettant d'effectuer une migration en base de données.

Doctrine fait appel à des repository (un par entité) ces fichiers permettent de créer des requêtes **DQL** (Doctrine Query Language), un langage de requête spécifique à Doctrine. Ces requêtes

permettent d'obtenir les informations désirées puis de les transmettre au contrôleur. Pour prendre exemple dans le projet on retiendra l'entité Post, qui correspond au fichier **Post.php**. Grâce à **Doctrine** il est possible de relier l'entité à un repository qui exécutera les requêtes en base de données nécessaires (ici le fichier **PostRepository.php**). À l'aide de l'annotation **@ORM\Entity(repositoryClass= ...)** on renseigne la classe du repository correspondant.

Ci-contre : extrait de l'entité Post issue du fichier Post.php.

```
<?php

namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;
use Doctrine\Common\Collections\Collection;
use Doctrine\Common\Collections\ArrayCollection;

/**
 * @ORM\Entity(repositoryClass=PostRepository::class)
 */
class Post
```

2.5. M.V.C.

Symfony est un framework basé sur un **Design Pattern** (ou patron de conception). Les **design patterns** représentent l'architecture de l'application en différents composants selon un arrangement défini. Ils constituent un ensemble de bonnes pratiques basées sur l'expérience pour répondre aux problèmes d'architectures.

Symfony utilise un design pattern de type **MVC** (Model View Controller). C'est un motif d'architecture logicielle destiné aux interfaces graphiques des applications web. Le motif est composé de trois types de modules ayant trois responsabilités différentes : les modèles, les vues et les contrôleurs.

- Une couche modèle (**Model**) contient les données à afficher.
- Des vues (**View**) contiennent la présentation de l'interface graphique.
- Une couche (**Controller**) contient la logique concernant les actions effectuées par l'utilisateur.



Ci-dessus : Schéma du MVC.

Son fonctionnement dans Symfony est le suivant. Sur demande de l'utilisateur, le client effectue une requête **HTTP**⁸, il demande une **URL**⁹. Sa demande est traitée par le **contrôleur frontal**¹⁰ avant d'être transmise au **noyau** de Symfony. Le noyau appelle à son tour le **contrôleur** correspondant grâce au système de **routing** de Symfony. Le contrôleur appellera la **couche modèle** pour obtenir les données voulues. Une fois ces données obtenues, le contrôleur génère la vue. Une fois la vue générée, le contrôleur retourne le résultat au noyau. Ce dernier va envoyer une réponse **HTML** au client.

Dans le cas de Symfony on peut aussi parler de design pattern de type **MVP** (Model View Presenter) car la couche modèle n'est pas directement en relation avec la vue.

Pour illustrer ce propos par l'exemple, on retiendra l'affichage des posts pour un utilisateur. Le client envoie une requête HTTP qui fait référence à la méthode **post_index** du contrôleur (PostController.php). Afin d'obtenir les données, le contrôleur fait appel au repository présent dans la couche Modèle.

⁸ **HTTP** (Hypertext Transfer Protocol) est un protocole de communication client-serveur utilisé dans le Web.

⁹ **URL** (Uniform resource Locator) est un ensemble de chaîne de caractères qui permet d'identifier une ressource sur le Web (une page web par exemple).

¹⁰ Le **contrôleur frontal** permet à partir d'une requête, d'appeler la méthode correspondante grâce au système de routing de Symfony. Il correspond au fichier index.php.

Ci-dessous : la méthode « `post_index` » qui permet l’affichage de tous les posts créés par les utilisateurs suivis par l’utilisateur en session issue du fichier `PostController.php`.

```
/**
 * @Route("/post", name="post_index")
 * @IsGranted("ROLE_USER")
 */
public function index(TokenStorageInterface $tokenStorage, PostRepository $repository): Response
{
    // On récupère l'utilisateur en session
    $userConnected = $tokenStorage->getToken()->getUser();
    //On affiche la liste de tous les Post par ordre chronologique
    //Grâce à findAllByUsers présente dans le repository
    $posts = $repository->findAllByUsers($userConnected->getFollowing());
    //On renvoie vers la vue correspondante avec post en paramètre
    return $this->render('post/index.html.twig', [
        'posts' => $posts
    ]);
}
```

Le repository dans la couche modèle effectue ensuite une requête **DQL** auprès de la base de données et renvoie les données au contrôleur.

Ci-contre : la méthode `findAllByUsers` qui effectue la requête **DQL** en base de données

```
/**
 * @return Post[] Returns an array of Post objects
 */
public function findAllByUsers(Collection $users)
{
    $query = $this->createQueryBuilder('p');

    return
        $query->select('p')
            ->where('p.creator IN (:following)')
            ->setParameter('following', $users)
            ->orderBy('p.releaseDate', 'DESC')
            ->getQuery()
            ->getResult();
}
```

Le Contrôleur renvoie alors une vue au format **Twig** présente dans le dossier **Templates** de Symfony.

Ci-dessous : la vue au format Twig permettant d'afficher la liste des post issue du fichier index.html.twig.

```
{% extends 'base.html.twig' %}

{% block title %}
    Feed
{% endblock %}

{% block content %}
    <main class="white-background uk-flex uk-flex-center uk-flex-column fullwidth">
        <div class="uk-flex uk-flex-row fullwidth uk-margin-top">
            <a href="{{path('post_new')}}" class="newpost-btn">
                <i class="fas fa-plus"></i>
                New Post
            </a>
            <a href="{{path('post_show')}}" class="post-btn">
                <i class="fas fa-paper-plane"></i>
                My posts
            </a>
        </div>
        {% if posts %}
            {% for post in posts %}
                {% include 'post/_post.html.twig' with { post: post } only %}
            {% endfor %}
        {% else %}
            <section class="uk-inline">
                <div class="empty-background"></div>
                <div class="uk-overlay uk-overlay-default uk-position-top uk-text-center">
                    <h1 class="principal-text">Follow other musicians and see their posts...</h1>
                </div>
            </section>
        {% endif %}
    </main>
{% endblock %}
```

L'intérêt de **Twig** réside dans le fait de pouvoir intégrer des objets PHP à une page HTML pour créer un contenu dynamique.

- On peut utiliser des conditions et des boucles (**if**, **for**).
- Il est possible de subdiviser la page en incluant d'autres pages HTML avec **include**.
- Enfin on peut filtrer l'affichage avec la fonction **is_granted** pour définir les rôles utilisateur autorisés à accéder à une partie d'une vue.

Ci-contre : l'accès à l'interface d'administration dans le menu principal.

```
{% if is_granted('ROLE_ADMIN') %}
    <li>
        <a class="uk-active uk-link-heading" href="{{path('admin')}}">
            <i class="fas fa-user-lock"></i>
        </a>
    </li>
{% endif %}
```

3. SÉCURITÉ

3.2. Principales failles

Faillle XSS (Cross Site Scripting)

C'est une faille de sécurité qui consiste pour un utilisateur malveillant, à implémenter du code dans un langage pris compte par le navigateur (Javascript par exemple). Le tout au travers d'un site ou une application web (dans un champ de formulaire par exemple). Le but étant de faire exécuter ce script sur le navigateur d'une victime, un fois cette étape franchi l'utilisateur malveillant peut :

- Récupérer des données personnelles (cookies, session).
- Rediriger la victime vers un autre site.
- Usurper l'identité de la victime sur le site et accéder à toute ses données.
- Bloquer l'accès à un utilisateur.

Le navigateur de la victime considérant le site script du site ou de l'application visé comme sûr, ce il fournira les informations demandées.

Pour ce faire, Symfony permet le filtrage des entrées en spécifiant le type de données attendue pour un champ lors de la création d'un formulaire. On peut aussi ajouter manuellement des contraintes sur ces mêmes champs. Enfin Twig le moteur de template de Symfony, permet **l'échappement automatique des caractères** en sortie. En dehors du framework, avec PHP il est possible de filtrer les caractères spéciaux grâce aux la fonctions native `htmlspecialchars()` et les entités HTML avec `htmlspecialchars()`.

Ci-contre : Schéma d'un exemple action utilisant la faille XSS.



Faible CSRF (Cross Site Request Forgery)

Consiste à exécuter des actions malveillantes à l'aide des identifiants d'un utilisateur en session sans son consentement. L'utilisateur malveillant fait exécuter une requête HTTP qui pointe sur une action interne au site. Elle peut être réalisée en ajoutant des paramètres dans une URL pour faire exécuter une requête sans que la victime ne le sache. L'utilisateur en session utilisant ses propres droits de connexion devient le déclencheur d'une action malveillante sur le site en question.

Symfony fournit une protection CSRF par défaut. Pour chaque formulaire créé à l'aide du composant Symfony Form un token c'est-à-dire un jeton de validité est généré. Il est hashé et ensuite stocké en session puis comparé avec le token contenu dans le formulaire.

Ci-contre : Schéma d'un exemple action utilisant la faille CSRF.



Injection SQL

Cette dernière consiste en l'injection de code SQL par un utilisateur malveillant (dans un champ de formulaire par exemple). Le but étant d'effectuer des requêtes afin de récupérer des données, ou d'affecter directement l'intégrité de la base de données. L'utilisateur malveillant peut :

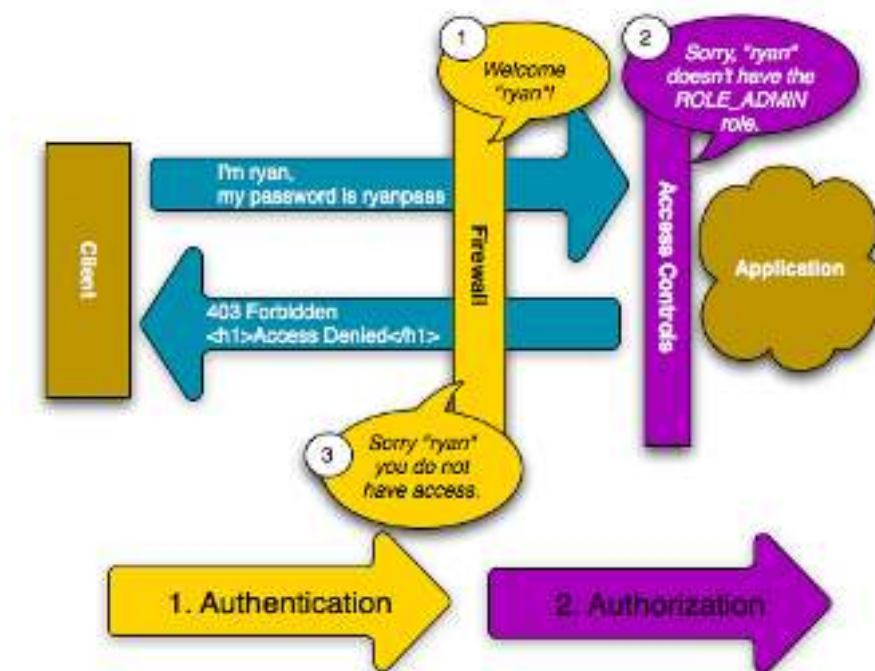
- Usurper l'identité d'un utilisateur ou créer un compte.
- Accéder aux données du serveur
- Modifier ou supprimer tout ou une partie de la base de données.

Cette vulnérabilité est présente quand la saisie de l'utilisateur est directement transmise à une requête SQL. Ce dernier qui peut alors modifier le sens de la requête en renseignant d'autres données que celles attendues. Symfony permet de se prémunir de cette faille, grâce aux mesures de sécurisation des champs et à l'échappement des caractères dans Twig vus précédemment. Doctrine ORM permet également d'utiliser la méthode **setParameter()** dans la création de requête. Cela permet d'effectuer une **requête préparée**, c'est-à-dire définir des paramètres qui seront interprétés indépendamment de la requête elle-même.

3.2. Authentification et autorisation

L'authentification et l'autorisation sont deux niveaux de sécurité qui permettent la gestion des utilisateurs dans l'ensemble de l'application. Le rôle de l'authentification est de déterminer les connexions autorisées en vérifiant un nom d'utilisateur et un mot de passe. Ensuite l'autorisation autorise ou non l'accès à certaines parties de l'application en fonction du rôle de l'utilisateur en session.

Ci-contre : schéma de résumant les étapes d'authentification et d'autorisation tiré de la documentation officielle de Symfony.



L'authentification des utilisateurs durant les phases d'inscription et connexion est gérée par des formulaires contenant les champs suivants :

Connexion (login) :

- email
- mot de passe
- token CSRF (caché)

Inscription (register) :

- email
- mot de passe
- confirmation du mot de passe
- conditions d'utilisation
- captcha

Lors de l'inscription, l'utilisateur se voit attribuer une date d'inscription, ainsi qu'un rôle par défaut (**USER**). L'email et le pseudo renseignés doivent être uniques. Le champ du mot de passe de type « password » masque les caractères tapés par l'utilisateur. Pour l'inscription, un champ de type

« checkbox » demande également l'acceptation des conditions d'utilisation avant la soumission du formulaire.

```
<form method="post" class="login-form">
  <input type="email" value="test@mail.fr" name="email" placeholder="your email..." class="inputEmailLogin"
    autocomplete="email" required autofocus>
  <input type="password" name="password" placeholder="your password..." class="inputPasswordLogin" autocomplete="current-password" required>
  <input type="hidden" name="_csrf_token" value="95a206ce06b6.C3McmgX0vPDc8f-hJE85ZhVeLuB0jLWt048XeqULsY.WTdQ7zf73p
    m8xZiDaShyIENHtBYI4u3u8TiaMU7vyf7J_KnaFQ4SFx0ilvg">
  <input type="submit" class="inputSubmit" value="login">
</form>
```

Ci-dessus : vue du navigateur avec le formulaire contenant le token CSRF.

Lors de la connexion, un token temporaire (jeton de validité) hashé est contenu dans le formulaire dans un champ caché. Il est ensuite vérifié et stocké côté serveur pendant toute la durée de la **session**¹¹ de l'utilisateur. Cette pratique permet de s'assurer que l'utilisateur est bien sur le site lorsqu'il fournit ses informations de connexion. Et donc ainsi limiter la possibilité de récupération de la session par un utilisateur malveillant via l'exploitation de failles comme la **faille CSRF**.

```
/**
 * @Route("/register", name="app_register")
 */
public function register(Request $request, UserPasswordHasherInterface $passwordEncoder): Response
{
    $user = new User();
    $form = $this->createForm(RegistrationFormType::class, $user);
    $form->handleRequest($request);


    if ($form->isSubmitted() && $form->isValid()) {
        // encode the plain password
        $user->setPassword(
            $passwordEncoder->hashPassword([
                $user,
                $form->get('plainPassword')->getData()
            ])
        );

        $entityManager = $this->getDoctrine()->getManager();
        $entityManager->persist($user);
        $entityManager->flush();
    }
}
```

Ci-dessus : La méthode `app_register` issue du fichier `Registration.php`.

¹¹ Une **session** correspond à un groupe d'interactions effectuées par un utilisateur sur un site Web au pendant une période donnée. La session contient l'ensemble des événements enregistrés depuis la connexion.

À la soumission du formulaire d'inscription le champ email subi une vérification du format avant d'être envoyé en base de données (Le mot de passe renseigné doit comporter au moins huit caractères, une majuscule et un caractère spécial). Une fois son format validé, ce dernier est ensuite hashé à l'aide de l'algorithme **Argon2i**¹² avant d'être stocké en base de données. Cette opération est gérée automatiquement par le composant **UserPasswordHasherInterface** de Symfony. Puis il est transmis en base de données par **Doctrine ORM** grâce aux fonctions **Persist** qui signale qu'un nouvel objet doit être enregistré. Puis **Flush** : qui permet d'exécuter la requête et d'envoyer tout ce qui a été persisté préalablement en base de données.

 id	roles	password
3	["ROLE_ADMIN"]	\$2y\$13\$Q6QSS9M.BycBbQ0W8Pnec.wDqymxH...

Ci-dessus : Le mot de passe hashé enregistré en base de données

Dans Symfony la gestion des formulaires est opérée grâce à des **FormTypes** qui permettent de définir une classe de formulaire qui peut ensuite être appelée dans une vue grâce à Twig. Chaque FormType permet de définir les différents champs du formulaire et leurs propriétés. Ces champs peuvent être soumis à des contraintes par exemple ici, l'attribut '**constraints**' contenu dans le champ password. Des contraintes ont été ainsi ajoutées pour définir la longueur minimum du mot de passe à huit caractères. De la même manière, une **Regex**¹³ est implémentée pour faire en sorte que le mot de passe ne contienne que des lettres majuscules et minuscules et au moins un chiffre et un caractère spécial. Cela permet de respecter les recommandations de l'**OWASP**¹⁴ en matière de mot de passe fort et de se prémunir de la **faible XSS**.

Ci-dessous : les contraintes contenues dans le champ password du formulaire d'inscription provenant du fichier RegistrationFormType.php.

¹² **Argon2i** est une fonction de dérivation de clé, En 2017, pour la version 7.2 de PHP, Argon2i est maintenant intégré dans le cœur de PHP sur la fonction password_hash () avec la constante PASSWORD_ARGON2I.

¹³ Les **Regex** ou regular expression en anglais sont des outils permettant de représenter des chaînes de caractères pour examiner, changer ou manier du texte.

¹⁴ **OWASP** (Open Web Application Security Project) est une communauté internationale ayant pour mission de publier des recommandations concernant la sécurité des applications web, ainsi que de proposer des solutions pour contrôler leur niveau de sécurité.

```
'constraints' => [
    new NotBlank([
        'message' => 'Please enter a password',
    ]),
    new Length([
        'min' => 8,
        'minMessage' => 'Your password should contain at least {{ limit }} characters',
        // max length allowed by Symfony for security reasons
        'max' => 32,
        'minMessage' => 'Your password should contain less than {{ limit }} characters',
    ]),
    new Regex([
        'pattern' => "/^(?=.*\d)(?=.*[A-Z])(?=.*[@#$%&'-])(?!.*(\.|\1{2}).*[a-z])/m",
        'match' => true,
        'message' => "Warning : Your password must be at least eight characters long, inc
    ])
```

Que ce soit pour assurer la sécurité de l'application ou celle des utilisateurs, une interface d'administration représente une fonctionnalité incontournable dans un réseau social. Elle permet à des utilisateurs bénéficiant d'un rôle spécifique, d'administrer le contenu des utilisateurs du site et le contenu créé par ces derniers en utilisant le **CRUD**¹⁵. Afin de définir les administrateurs, une propriété **roles** exprimée en **JSON**¹⁶ est contenue dans l'Entité **User**. Elle contiendra soit « ROLE_USER » pour un simple utilisateur, ou « ROLE_ADMIN » pour un administrateur. Les deux rôles étant cumulables

Ci-contre : la propriété roles issue de User.php.

```
/**
 * @ORM\Column(type="json")
 */
private $roles = [];
```

Symfony permet de conditionner l'accès de certaines fonctions à un rôle spécifique grâce à l'annotation **@IsGranted** suivi du rôle en question (par exemple : **@IsGranted('ROLE_ADMIN')**). Elles peuvent s'appliquer sur une ou plusieurs fonctions d'un contrôleur, ou bien sur l'intégralité du contrôleur.

¹⁵ **CRUD** (Create, Read, Update, Delete) fait référence aux quatre opérations de base effectuées dans une base de données.

¹⁶ **JSON** (JavaScript Object Notation) est un format descriptif dérivé de JavaScript qui sert à transporter et stocker des données.

Ci-contre : la méthode `admin` qui permet l'accès à l'interface d'administration issue du fichier `DashboardController.php`.

```
/**
 * @Route("/admin", name="admin")
 * @IsGranted("ROLE_ADMIN")
 */
public function index(): Response
```

Pour restreindre plus facilement l'accès à toute une partie du site. Il est également possible de renseigner un modèle d'URL complet dans l'option `access_control` du fichier de configuration `security.yaml`. Par exemple ici tout URL commençant par `/admin` requièrent un rôle administrateur (`ROLE_ADMIN`). Et tous ceux commençant par `/user` un rôle utilisateur (`ROLE_USER`).

Ci-contre : la rubrique `access_control` issue du fichier `Security.yaml`

```
access_control:
  - { path: ^/admin, roles: ROLE_ADMIN }
  - { path: ^/user, roles: ROLE_USER }
  - { path: ^/message, roles: ROLE_USER }
  - { path: ^/post, roles: ROLE_USER }
```

3.3. Captcha

Le **CAPTCHA** (de l'anglais "Completely Automated Public Turing test to tell Computers and Humans Apart") est une mesure de sécurité implémentée dans un formulaire. Cette vérification prend le plus souvent la forme d'une question auquel l'utilisateur doit répondre. Elle est ajoutée ici au formulaire d'inscription ainsi qu'au formulaire de changement de mot de passe. Le but étant d'empêcher des robots (ou logiciels automatisés) d'accéder au site. Ces derniers pouvant être utilisés dans le but pourraient compromettre la sécurité du site Web.

- Comme via une publication automatisée de formulaire, dans le but d'inscrire un grand nombre d'utilisateurs en même temps pour porter atteinte à l'intégrité d'un site web ou d'une application.
- Dans le cadre d'une **attaque par force brut**. Cette méthode vise à tester toutes les combinaisons possibles jusqu'à trouver la bonne.

La solution utilisée dans le projet est le **BotDetect CAPTCHA**, un bundle disponible à l'installation via l'invite de commande. Ce CAPTCHA utilise des images générées aléatoirement contenant une série de chiffres et de lettres, et utilisant des filtres de distorsion le rendant illisible pour une intelligence artificielle. L'utilisateur doit recopier les chiffres et les lettres dans un champ de type texte en bas de l'image. Un champ supplémentaire a été ajouté au formulaire d'inscription contenant des contraintes spécifiques à Bot Detect CAPTCHA.

À la soumission du formulaire, une vérification est établie côté serveur lorsque le champ n'est pas validé, une nouvelle image est chargée et un message d'erreur s'affiche. Deux boutons sont également disponibles sur le côté pour recharger une nouvelle image ou générer une version audio.

Ci-contre : exemple d'image générée par le Bot Detect Captcha issu du formulaire d'inscription.



L'intégration du captcha nécessite une propriété protégée **captchaCode** dans l'entité **User**, ainsi que les getters et setters correspondants (voir ci-contre). Cette propriété est non mappée donc elle n'est pas contenue en base de données.

Son intégration se déroule ensuite dans le **FormType** du formulaire d'inscription (**RegistrationFormType.php**). En ajoutant un nouveau champ de formulaire grâce à la fonction `add()`. Il appartient à la classe **CaptchaType** qui permet sa validation. Une nouvelle contrainte est ajoutée de manière à renvoyer un message d'erreur si les informations saisies ne correspondent pas à l'image.

```
public function getCaptchaCode()  
{  
    return $this->captchaCode;  
}  
  
public function setCaptchaCode($captchaCode)  
{  
    $this->captchaCode = $captchaCode;  
}  
  
public function __toString()  
{  
    return $this->userName;  
}
```

Ci-contre : le champ correspondant au captcha dans le formulaire d'inscription **RegistrationFormType.php**.

```
->add('captchaCode', CaptchaType::class, [  
    'captchaConfig' => 'ExampleCaptcha',  
    'constraints' => [  
        new ValidCaptcha([  
            'message' => 'Invalid captcha, please try again',  
        ]),  
    ],  
]);
```

3.4. Upload de fichiers

Afin de permettre d'avoir une meilleure vue d'ensemble d'un utilisateur. L'ajout de photos ainsi que de fichiers audio personnels, permet de mieux discerner la personnalité, ainsi que le niveau de pratique d'un musicien.

En effet, un utilisateur a la possibilité de compléter son profil en ajoutant des fichiers image ou un fichier audio qui seront ensuite visibles par les autres utilisateurs. Deux entités dédiées (**Image** et **Track**) liées à l'entité **User** par une relation de type **OnetoMany**. Afin d'accéder directement à la collection d'images ou de fichiers audio d'un utilisateur, deux attributs **Image** et **Track** (liés à leurs entités respectives) sont présents dans l'entité **User**.

L'ajout de fichiers est compris dans le formulaire de modification du profil d'un utilisateur (uniquement accessible par un utilisateur en session). Géré dans Symfony grâce à un `FormType` deux inputs de type `File`. Symfony permet de se prémunir de la **faille Upload**¹⁷, en renseignant des contraintes de type, ainsi que de taille de fichier. Comme illustré dans l'image ci-contre, les champs disposent de contraintes personnalisées (**'maxSize'**, **'mimeTypes'**¹⁸) en fonction du type de fichier grâce à l'attribut **'constraints'**.

On accepte ici uniquement des fichiers de type mp3 ou MPEG d'une taille maximale de 256Mo pour l'audio et jpeg, png, ou webp d'une taille maximale de 8Mo pour les images.

```
->add('images', FileType::class, [
    'multiple' => true,
    'mapped' => false,
    'required' => false,
    'constraints' => [
        new Images([
            'maxSize' => '8M',
            'mimeTypes' => [
                'image/jpeg',
                'image/png',
                'image/webp'
            ]
        ])
    ],
    'attr' => [
        'accept'=>'.jpg, .jpeg, .png, .gif'
    ]
])
->add('tracks', FileType::class, [
    'multiple' => true,
    'mapped' => false,
    'required' => false,
    'constraints' => [
        new Tracks([
            'maxSize' => '256M',
            'mimeTypes' => [
                'application/octet-stream',
                'audio/mpeg',
                'audio/mp3'
            ]
        ])
    ]
])
```

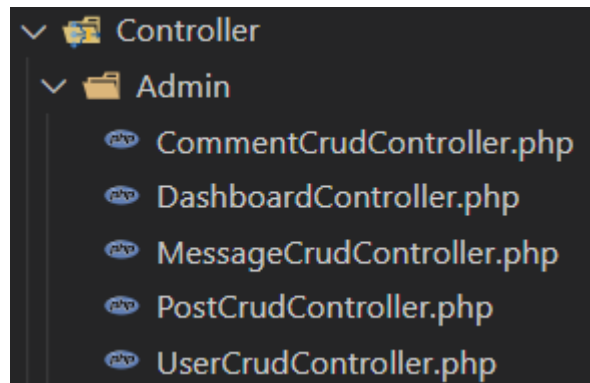
¹⁷ **Faillle Upload** : faille de sécurité permettant à un utilisateur malveillant de charger un ou des fichiers avec une extension non autorisée.

¹⁸ **Type MIME** (Multipurpose Internet Mail Extensions) est un standard permettant d'indiquer la nature et le format d'un document.

3.5. Interface d'administration

L'interface d'administration permet à un utilisateur possédant le rôle administrateur d'effectuer un travail de modération. Il peut ainsi supprimer tout contenu qui nuirait à la sécurité et la crédibilité de l'application. L'interface est gérée elle, grâce à l'extension **EasyAdmin** de Symfony. Son intérêt réside en la création automatique d'un **Dashboard** ou tableau de bord et son contrôleur associé (**DashboardController.php**) ; dont l'accès est conditionné à la possession du rôle administrateur (ROLE_ADMIN). Ainsi seul un administrateur a accès au lien vers le Dashboard depuis le menu principal.

Ci-contre : contenu du sous-dossier Admin présent dans le dossier Controller.



EasyAdmin, permet également d'afficher les entités à administrer et de créer des fonctions CRUD. L'ensemble des fonction CRUD que le Dashboard Controller sont contenus dans un sous-dossier « Admin » à l'intérieur du dossier « Controller » de Symfony. On administre ici quatre entités **User**, **Post**, **Comment** et **Message** dont les listes détaillées sont disponibles via un menu.

Ci-dessous : la fonction `configureMenuItems()` contenant l'ensemble des fonction CRUD pour chaque entité administrée.

```
public function configureMenuItems(): iterable
{
    yield MenuItem::linktoDashboard('Dashboard', 'fa fa-home');
    yield MenuItem::linkToCrud('Users', 'fas fa-user', User::class);
    yield MenuItem::linkToCrud('Posts', 'fas fa-list', Post::class);
    yield MenuItem::linkToCrud('Comments', 'fas fa-comment-dots', Comment::class);
    yield MenuItem::linkToCrud('Messages', 'fas fa-envelope', Message::class);
}
```

Afin d'optimiser la modération du contenu et de garantir sa sécurité, j'aimerais à terme proposer à tout utilisateur de l'application de signaler à un administrateur un post, un commentaire, un message, ou un autre utilisateur. Cette fonction se caractérise d'abord au sein des entités concernées par une propriété **isReported** de type booléen. Ensuite une fonction reporting présente là aussi dans chacune des entités, définie la valeur de cette propriété à true puis envoie une notification visible depuis le Dashboard d'un administrateur.

4. FONCTIONNALITÉS MAJEURES

4.1. Recherche multi-filtres

La recherche multi-filtres permet à un utilisateur de rechercher un utilisateur tiers en fonction de son ou ses instrument(s) pratiqué(s), ou de ses styles de musiques affectionnés. Ces filtres reposent sur les entités **Instrument** et **Style** qui ont une relation de type **Many To Many** avec l'entité **User**. Le formulaire du filtre intègre deux champs de types **EntityType** se référant aux entités **Style** et **Instrument**. Elles sont présentées sous forme de liste déroulante avec les noms de chaque instrument ou style sélectionnable par l'utilisateur. La requête du formulaire vers la base de données est assurée par Doctrine ORM via le fichier **User Repository**. Il permet de définir une fonction établissant les requêtes DQL qui permettent de retourner un tableau contenant des objets de type **User**.

La requête est créée à l'aide de la méthode **createQueryBuilder()**. Un **QueryBuilder** est un objet qui permet à Doctrine de construire des requêtes DQL en plusieurs étapes. Il fournit un ensemble de méthodes et de classes qui autorisent la création de requêtes par programmation. Les méthodes **select()**, **join()** et **andWhere()** s'apparentent à leur équivalents en SQL (SELECT, JOIN et WHERE). L'appel de la fonction **setParameter()** permet de définir le type de la valeur utilisée et de préparer la requête afin de se prémunir de l'**injection SQL**.

Ci-contre : extrait des requêtes DQL permettant une recherche d'utilisateur par style par instrument ou par son nom ; issues du fichier **UserRepository.php**.

```
$query = $this
->createQueryBuilder('u')
->select('i', 's', 'u')
->join('u.styles', 's')
->join('u.instruments', 'i');

if (empty($search->styles)) {
    $query = $query
        ->andWhere('s.id IN (:styles)')
        ->setParameter('styles', $search->styles);
}

if (empty($search->instruments)) {
    $query = $query
        ->andWhere('i.id IN (:instruments)')
        ->setParameter('instruments', $search->instruments);
}

if (empty($search->q)) {
    $query = $query
        ->andWhere('u.userName Like :q')
        ->setParameter('q', "%{$search->q}%");
}
```

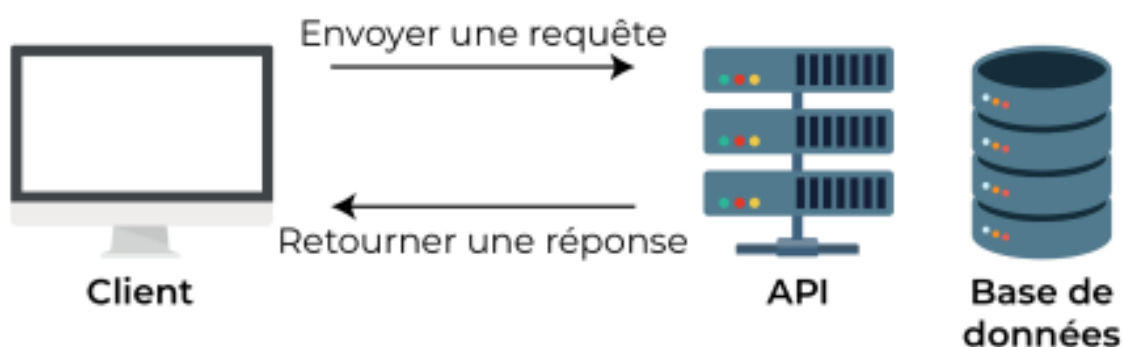

Afin de proposer une meilleure expérience utilisateur et d'éviter le rechargement systématique de la page. La soumission du formulaire passe par une requête asynchrone en **AJAX**¹⁹. L'**AJAX** (Asynchronous Javascript and XML) permet d'effectuer une requête en arrière-plan pendant le chargement de la page web. Cette méthode permet ainsi d'afficher du contenu sans attendre le chargement de la page en entier.

Un système d'historique permet de créer un URL personnalisé en fonction des filtres sélectionnés. De cette manière, si l'utilisateur désire partager la page affichant le résultat de ses recherches le lien renverra bien vers le bon contenu.

4.2. Gestion d'A.P.I. et géolocalisation

Pour ce projet, il est nécessaire de faire appel à une **API** pour répondre à un besoin essentiel de l'application : le fait de pouvoir rechercher des musiciens proches de soi. Une **API** (de l'anglais Application Programming Interface) est un ensemble de protocoles qui facilite la création et l'intégration de logiciels d'applications. Pour consommer l'API, c'est à dire utiliser ses données dans le cadre d'une application on utilise une requête sécurisée nécessitant un code (ou clé d'API) l'API permet l'accès à une base de données. Sa réponse ensuite doit être traitée afin d'en exploiter les données dans l'application. L'API est parfois considérée comme un contrat entre un fournisseur et un utilisateur d'informations.

C'est donc une A.P.I de géolocalisation qui est ici utilisée afin dans un premier temps de localiser l'utilisateur en session. Ce qui, par la suite, permet à celui-ci de rechercher d'autres utilisateurs (ayant eux aussi renseigné leur localisation) dans un périmètre donné. Le site fait appel à l'API **REST**²⁰ **Mapquest Geocode API**.



Ci-dessus : schéma représentant le principe de fonctionnement d'une API.

¹⁹ **AJAX** (Asynchronous Javascript and XML) permet d'effectuer une requête en Javascript qui permet au navigateur de ne pas attendre la réponse du serveur web. Cette méthode permet ainsi d'afficher certains contenus sans attendre le chargement de la page en entier.

²⁰ **REST** (Representational State Transfer) désigne un ensemble de contraintes appliquées à l'architecture d'une API.

Cette A.P.I. permet, à partir de l'emplacement de l'utilisateur (que ce dernier a renseigné au préalable dans son profil), de retrouver la latitude et la longitude associée. Lorsqu'un utilisateur édite son profil, il a la possibilité de renseigner ou d'actualiser sa localisation grâce au bouton « **Edit Location** ». Ce dernier renvoie vers une vue contenant un formulaire avec une barre de recherche ainsi qu'un bouton de soumission. Lors de la saisie un système d'auto-complétions affiche des suggestions extraites de L'A.P.I. Cette fonction est assurée par le kit de développement **PlaceSearch.js** créé par **Mapquest** et disponible grâce à un **C.D.N.** En bas de la vue se situe une carte qui permet de situer l'emplacement entré par l'utilisateur.

À la soumission du formulaire, une requête de type POST contenant en paramètre les données de localisation (\$location) saisies est soumise à l'A.P. I grâce au **HttpClient** de Symfony. Ce dernier permet l'accès à des méthodes pour effectuer la requête et gérer la réponse. La clé d'API est comprise dans l'URL de la requête.

```
//si ils ne sont pas nuls
if ($location != null) {

    $this->client = $client;

    //on effectue la requête vers l'API Mapquest avec CallApiService et on envoie $location en paramètres
    $response = $this->client->request('POST', 'http://www.mapquestapi.com/geocoding/v1/address?key=wxK9UXivqVivA6iIhZK
        'json' => [
            "location" => $location,
            "options" => [
                "thumbMaps" => false
            ]
        ]
    );

    //on transforme la réponse en tableau PHP
    $content = $response->toArray();
    //on récupère la latitude dans la réponse de l'API et on la stock en base de données
    $latitude = $content['results'][0]['locations'][0]['latLng']['lat'];
    $userConnected->setLatitude($latitude);
    //pareil pour la longitude
    $longitude = $content['results'][0]['locations'][0]['latLng']['lng'];
    $userConnected->setLongitude($longitude);
}
```

Ci-dessus : extrait de la fonction exécutant la requête vers l'A.P. I Mapquest Geocode issu du fichier **UserController.php**.

En retour, on obtient un tableau JSON contenant la latitude et la longitude correspondantes. Ces données sont ensuite extraites du tableau en identifiant leur emplacement et stockées en base de données dans la table **User**.

Ci-contre : extrait du tableau JSON renvoyé par l'API.

```
"latLng": {  
  "lat": 38.892062,  
  "lng": -77.019912  
},  
"displayLatLng": {  
  "lat": 38.892062,  
  "lng": -77.019912  
}
```

Pour pouvoir localiser l'utilisateur, et rechercher des utilisateurs dans un périmètre donné. Il est possible, à partir des latitudes et des longitudes stockées, de faire une requête SQL (à l'aide de **Doctrine ORM**). Afin de :

- Calculer une distance entre l'utilisateur connecté et un utilisateur tiers.
- Comparer cette distance à une valeur renseignée dans un filtre de recherche.
- Afficher les utilisateurs à une distance inférieure ou égale à la distance renseignée dans le filtre.

Ci-dessus : extrait de la requête SQL permettant une recherche d'utilisateur par distance à partir de la longitude et la latitude de deux utilisateurs issus du fichier UserRepository.php.



```
if (!empty($search->distance) && !empty($user->getLatitude()) && !empty($user->getLongitude())) {  
    $query = $query  
        ->select('u')  
        ->andWhere('(6353 * 2 * ASIN(SQRT( POWER(SIN((u.latitude - :latitude) * pi()/180 / 2), 2) +COS(u.latitude *  
pi()/180) * COS(:latitude * pi()/180) * POWER(SIN((u.longitude - :longitude) * pi()/180 / 2), 2) ))) <= :distance')  
        ->setParameter('longitude', $user->getLongitude())  
        ->setParameter('latitude', $user->getLatitude())  
        ->setParameter('distance', $search->distance);  
}  
  
return $query->getQuery()->getResult();
```

4.3. Systèmes de following

Le fait de suivre un utilisateur représente une fonctionnalité majeure pour un réseau social. Le système de following est ici directement lié aux fonctions de publication ainsi qu'à la messagerie. Un utilisateur pour pouvoir accéder aux posts d'un autre utilisateur doit suivre ce dernier. Pour contacter un utilisateur via la messagerie, les deux utilisateurs doivent se suivre réciproquement.

Dans la base de données la fonction de following est donc directement liée à l'entité **User**. Deux nouvelles propriétés **following** et **followers** sont ajoutées dans l'entité. Les deux propriétés renvoient une collection d'utilisateurs. Following permettant d'accéder aux utilisateurs suivis et followers aux utilisateurs suiveurs. Ces deux attributs représentent une relation reliant l'entité **User** à elle-même de type **Many-To-Many** qui aura pour effet de créer une nouvelle table associative nommée **following** en base de données. Cette dernière comprendra les colonnes **user_id** et **following_user_id**.

Ci-contre : extrait de la table associative following.

 user_id	 following_user_id
3	11

Du point de vue de Doctrine ORM, **following** établie une relation dite **bidirectionnelle** de type **Many-to-Many** notée **@ORM/ManyToMany**. Une relation bidirectionnelle possède un côté propriétaire (**followers**) et un côté inverse (**followings**). Ce qui permet à Doctrine ORM de seulement prendre en compte le côté propriétaire en cas de changements (**voir traduction d'une documentation anglophone p.30**).

Le mapping se déroule en deux étapes, on définit à quelle colonne et à quelle table le côté propriétaire fait référence, grâce à l'annotation **@ORM/JoinTable** puis **@ORM/JoinColumn**. Ensuite, on définit à quelle colonne le côté inverse fait référence à l'aide de **inverseJoinColumn** puis à nouveau **@ORM/JoinColumn**.

```

/**
 * @ORM\ManyToOne(targetEntity=User::class, mappedBy="following")
 */
private $followers;

/**
 * @ORM\ManyToOne(targetEntity=User::class, inversedBy="followers")
 * @ORM\JoinTable(name="following", joinColumns={
 *     @ORM\JoinColumn(name="user_id", referencedColumnName="id")
 * },
 *     inverseJoinColumns={
 *         @ORM\JoinColumn(name="following_user_id", referencedColumnName="id")
 *     }
 * )
 */
private $following;

```

Ci-dessus : les propriétés `followers` et `following` présentent dans l'entité `User` issues du fichier `User.php`.

TRADUCTION D'UNE DOCUMENTATION ANGLOPHONE

Texte original :

Le texte ci-dessous est tiré de la documentation officielle de Doctrine ORM et porte sur la création de relation entre les entités, qui est une des fonctions principales de l'ORM dans Symfony.

<https://www.doctrine-project.org/projects/doctrine-orm/en/latest/reference/unitofwork-associations.html>

Association Updates: Owning Side and Inverse Side

When mapping bidirectional associations it is important to understand the concept of the owning and inverse sides. The following general rules apply:

- Relationships may be bidirectional or unidirectional.
- A bidirectional relationship has both an owning side and an inverse side
- A unidirectional relationship only has an owning side.
- Doctrine will **only** check the owning side of an association for changes.

Bidirectional Associations

The following rules apply to **bidirectional** associations:

- The inverse side has to have the mappedBy attribute of the OneToOne, OneToMany, or ManyToMany mapping declaration. The mappedBy attribute contains the name of the association-field on the owning side.
- The owning side has to have the inversedBy attribute of the OneToOne, ManyToOne, or ManyToMany mapping declaration. The inversedBy attribute contains the name of the association-field on the inverse-side.
- ManyToOne is always the owning side of a bidirectional association.
- OneToMany is always the inverse side of a bidirectional association.
- The owning side of a OneToOne association is the entity with the table containing the foreign key.
- You can pick the owning side of a many-to-many association yourself.

Traduction française :

MISE A JOUR D'ASSOCIATIONS : Côté propriétaire et côté inverse

Lors du mapping d'une association bidirectionnelle, il est important de comprendre le concept de côté inverse et de côté propriétaire.

Les règles suivantes s'appliquent :

- Les relations peuvent être bidirectionnelles ou unidirectionnelles.
- Une association bidirectionnelle possède un côté inverse et un côté propriétaire.
- Une relation unidirectionnelle possède seulement un côté propriétaire.
- Doctrine va **seulement** contrôler le côté propriétaire d'une association en cas de changements.

ASSOCIATIONS BIDIRECTIONNELLES

Les règles suivantes s'appliquent aux associations **bidirectionnelles** :

- Le côté inverse doit avoir l'attribut « mappedBy » de la déclaration de mapping OneToOne, OneToMany ou ManyToMany. L'attribut mappedBy contient le nom du champ d'association du côté propriétaire.
- Le côté propriétaire doit avoir l'attribut inversedBy de la déclaration de mapping OneToOne, OneToMany ou ManyToMany. L'attribut inversedBy contient le nom du champ d'association du côté inverse.
- ManyToOne est toujours le côté propriétaire d'une association bidirectionnelle.
- OneToMany est toujours le côté inverse d'une association bidirectionnelle.
- Le côté propriétaire d'une association OneToOne est l'entité avec la table contenant la clé étrangère.
- Vous pouvez choisir vous-même le propriétaire d'une association multiple.

4.4. Systèmes de publication et commentaires

Le système de publication permet aux utilisateurs de publier un post pouvant contenir du texte, un fichier audio, ou une image. Les posts sont représentés par l'entité **Post** et contient les propriétés suivantes :

- **id**
- **releaseDate** (Date de création)
- **creator** (nom de l'utilisateur créateur)
- **content** (contenu texte non nullable)
- **images**
- **tracks** (contenu audio)
- **videos**
- **likedBy** (id des utilisateurs)

Afin d'ajouter plus d'interactivité au sein du système de post une fonction de like/unlike y est associée. L'entité **Post** contiendra un attribut **likedBy** établissant une relation de **type Many To Many**, permettant d'accéder à tous les utilisateurs qui ont « liké » un post et renvoyant une collection d'objets de type **User**. De la même manière, dans l'entité **User** l'attribut **postLiked** renvoie les posts « likés » par un utilisateur sous la forme d'une collection d'objets de type **Post**. La création de ses nouveaux attributs engendre une table associative **post_likes** en base de données qui comporte les colonnes **post_id** et **user_id** et associe un post avec un utilisateur qui a « liké » ce dernier.

```
/**
 * @ORM\ManyToOne(targetEntity=User::class, inversedBy="postLiked")
 * @ORM\JoinTable(name="post_likes",
 *     joinColumns={@ORM\JoinColumn(name="post_id", referencedColumnName="id")},
 *     inverseJoinColumns={@ORM\JoinColumn(name="user_id", referencedColumnName="id")}
 * )
 */
private $likedBy;
```

Ci-dessus : la propriété **likedBy** présente dans l'entité **User** issue du fichier **User.php**.

Du point de vue du contrôleur, la fonction **like** (permettant de créer un like) retourne une réponse JSON contenant le nombre de likes du post. Tandis que la fonction **unlike** se contente de supprimer le like de l'utilisateur en base de données.

Les posts peuvent également être commenté. Les commentaires sur les posts sont régis par l'entité **Comment** qui contient les propriétés suivantes :

- **id**
- **releaseDate** (Date de création)
- **author** (l'auteur du post, non nullable)
- **content** (contenu texte, non nullable)

- **post** (le post auquel le commentaire est rattaché, non nullable)

```
/**
 * @ORM\ManyToOne(targetEntity=User::class, inversedBy="comments")
 * @ORM\JoinColumn(nullable=false)
 */
private $author;

/**
 * @ORM\ManyToOne(targetEntity=Post::class, inversedBy="comments")
 * @ORM\JoinColumn(nullable=false)
 */
private $post;
```

Ci-dessus : extrait de l'entité **Comment** présente dans le fichier **Comment.php**.

Un commentaire est lié à son auteur ainsi qu'au post qui le contient. Son contenu est uniquement textuel et peut être visible par toutes les personnes ayant accès au post lié. L'auteur du commentaire aussi le supprimer ainsi que l'auteur du post auquel le commentaire est rattaché.

4.5. Messagerie

La messagerie représente une fonctionnalité majeure dans l'application. Elle permet aux utilisateurs ayant accepté une invitation au préalable de se contacter directement. Son fonctionnement est régi par une entité principale **Message** possédant les attributs suivants :

- **id**
- **releaseDate** (date de création)
- **sender** (id de l'expéditeur)
- **recipient** (id du destinataire)
- **content** (contenu text)
- **isRead** (Booléen)

L'entité **Message** possède un attribut « **isRead** » permettant de définir si un message a été lu, sa valeur de type booléen est par défaut égale à zéro. L'expéditeur (« **Sender** ») ainsi que le destinataire (« **Recipient** ») du message sont aussi renseignés non nullable tout comme la date de création au format **DateTime**. Le contenu du message est représenté par l'attribut « **Content** » au format **Text**.


```

/**
 * @ORM\ManyToOne(targetEntity=User::class, inversedBy="messages")
 * @ORM\JoinColumn(nullable=false)
 */
private $sender;

/**
 * @ORM\ManyToOne(targetEntity=User::class, inversedBy="messages")
 * @ORM\JoinColumn(nullable=false)
 */
private $recipient;

```

Ci-dessus : les propriétés **sender** et **recipient** présentent dans l'entité **Message** issues du fichier **Message.php**.

Du côté de l'entité **User**, les attributs **sent** et **received** directement liés à **sender** et **recipient** de l'entité **Message** permettent de récupérer les messages reçus et envoyés pour l'utilisateur en session en utilisant « **app.user.received** » ou « **app.user.sent** ».

```

/**
 * @ORM\OneToMany(targetEntity=Message::class, mappedBy="sender", orphanRemoval=true)
 */
private $sent;

/**
 * @ORM\OneToMany(targetEntity=Message::class, mappedBy="recipient", orphanRemoval=true)
 */
private $received;

```

Ci-dessus : les propriétés **sent** et **received** présentent dans l'entité **User** issu du fichier **User.php**.

Concernant l'interface, un utilisateur accédant à sa messagerie personnelle peut consulter le nombre de messages reçus et les consulter. De même pour les messages envoyés. Un autre lien lui permet également d'accéder au formulaire, pour créer ou d'envoyer un nouveau message.

AXES D'AMÉLIORATION

L'application a aujourd'hui atteint son objectif de Minimum Viable Product. Cependant différents aspects peuvent encore être améliorés.

- Concernant la partie front-end l'aspect responsive de l'interface d'administration pourrait être amélioré.
- Dans le but d'améliorer l'expérience utilisateur des fonctions en AJAX pourraient être rajoutées (sur le système de likes par exemple).
- Un système de notification concernant la messagerie les likes et les suivis de l'utilisateur en session pourrait être ajouté.
- Un système de signalement permettrait de signaler un post, un commentaire, un message ou un utilisateur directement à un administrateur.
- L'intégration d'une API d'une application de musique en ligne comme Spotify ou Deezer permettrait aux utilisateurs de partager des playlists. Et renforcer l'interactivité de l'application.

CONCLUSION

Ce projet a bénéficié d'un rapport avec une problématique que j'ai pu rencontrer personnellement et dans un domaine qui me concerne. Il m'a permis d'effectuer un bond en avant dans mes connaissances des différentes technologies utilisées et notamment le framework Symfony. D'autre part, cela fut l'occasion de mettre en pratique les connaissances acquises durant la formation sur un projet de plus grande ampleur.

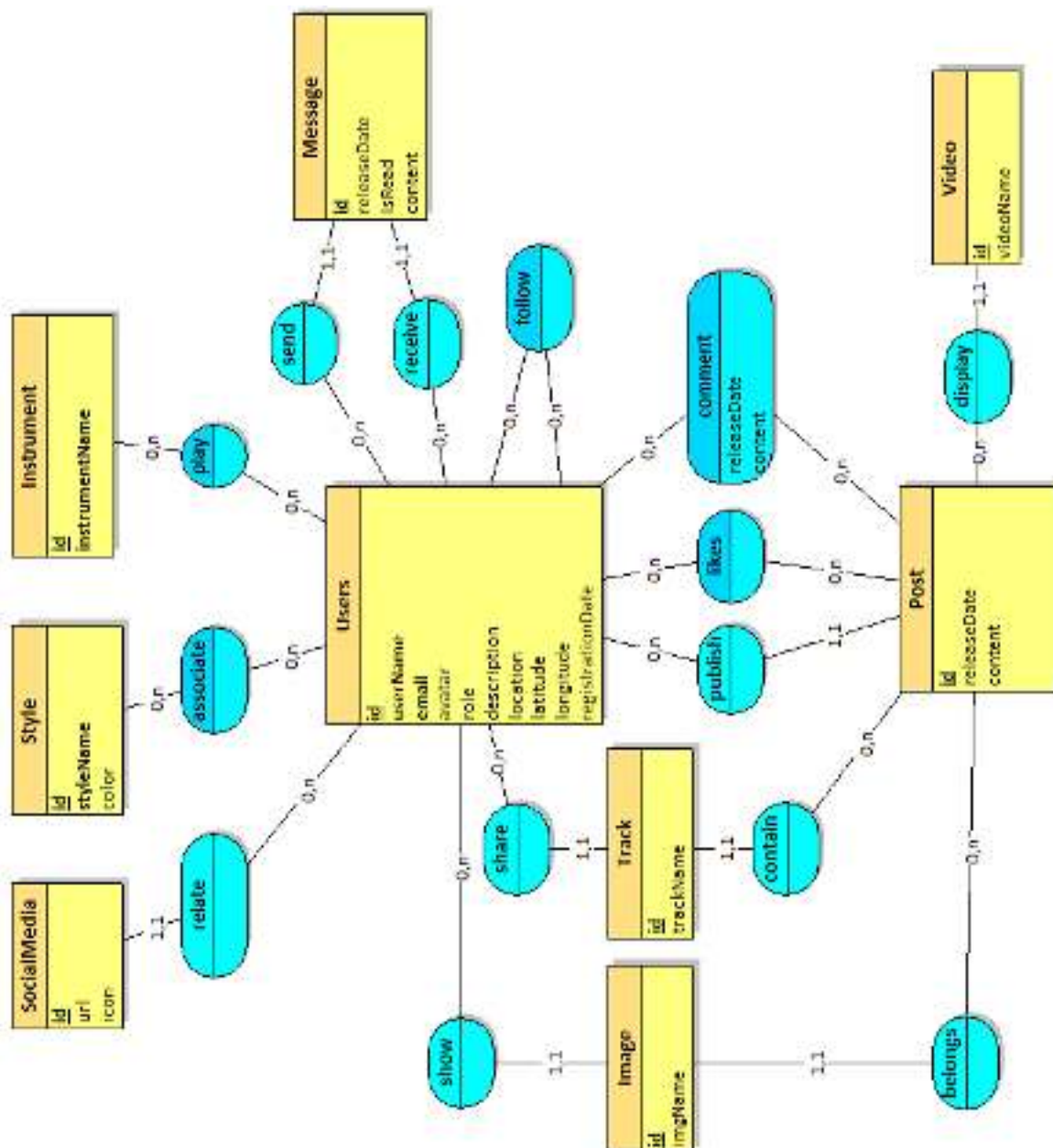
Et si l'application bénéficiera encore grandement de certaines améliorations. Ce projet et plus généralement la formation, restent pour moi une expérience très enrichissante. Je reste donc satisfait de mon parcours et espère pouvoir mettre en pratique l'expérience accumulée lors dans la suite de ma vie professionnelle.

BIBLIOGRAPHIE / SOURCES

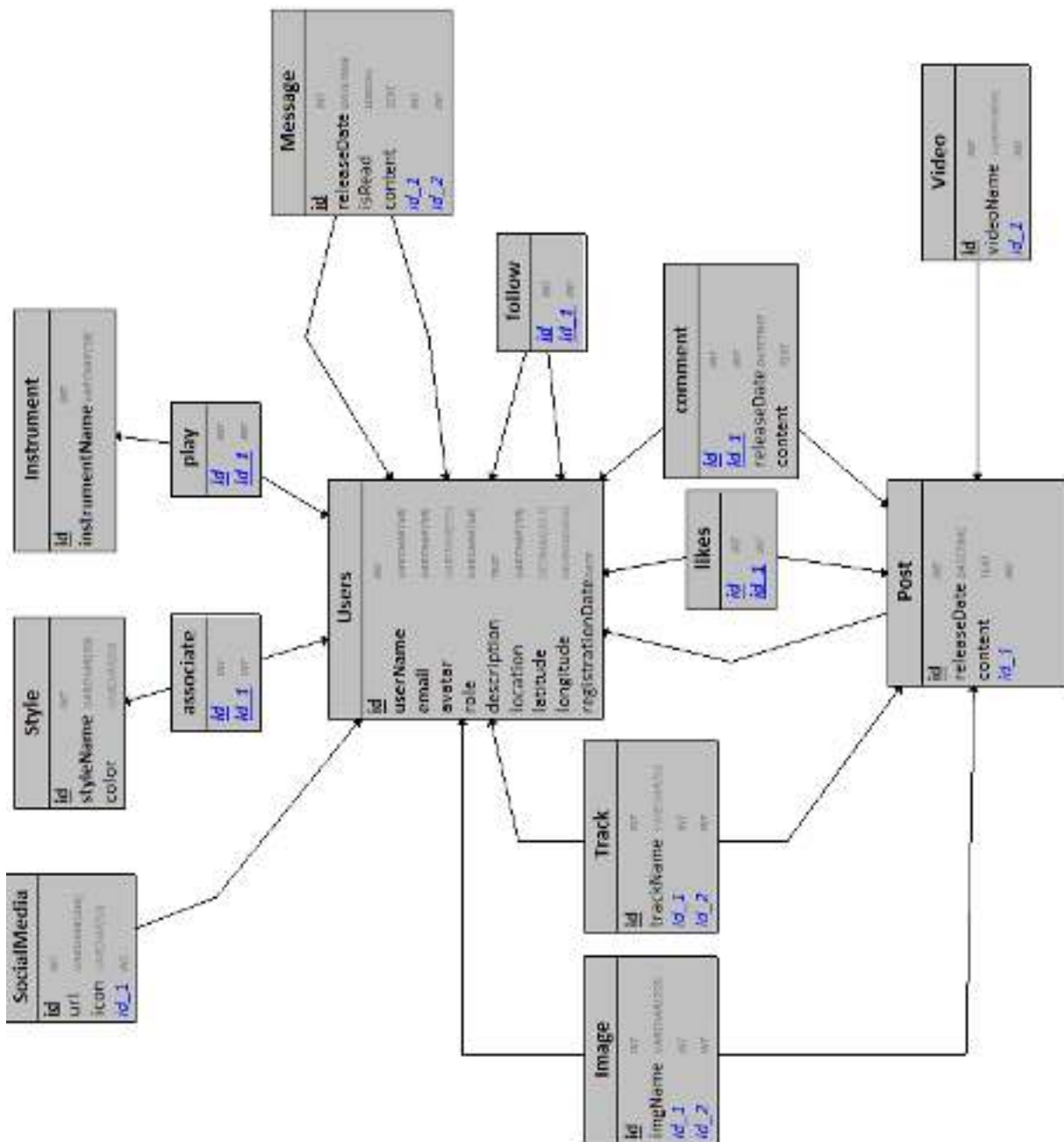
- Définition du RGPD par la CNIL / *RGPD de Quoi parle-t-on ?* :
<https://www.cnil.fr/fr/rgpd-de-quoi-parle-t-on>
- Documentation officielle du framework Symfony :
<https://www.symfony.com/>
- Documentation officielle du moteur de template Twig :
<https://twig.symfony.com/>
- Définition de la faille CSRF par l'OWASP : <https://owasp.org/www-community/attacks/csrf>
- Définition du CAPTCHA par Google :
<https://support.google.com/a/answer/1217728?hl=fr>
- Documentation officielle du Bot Detect CAPTCHA :
<https://captcha.com/>
- Documentation officielle de l'ORM Doctrine :
<https://www.doctrine-project.org>
- Définition du type MIME par MDN :
https://developer.mozilla.org/fr/docs/Web/HTTP/Basics_of_HTTP/MIME_types
- Définition de l'API et de l'API REST par RedHat.com :
<https://www.redhat.com/fr/topics/api/what-is-a-rest-api>
- Définition du JSON par W3Schools.com : https://www.w3schools.com/js/js_json_intro.asp
- Documentation officielle de l'extension EasyAdmin de Symfony :
<https://symfony.com/bundles/EasyAdminBundle/current/index.html>
- Documentation officielle de l'API Mapquest :
<https://developer.mapquest.com/documentation/>
- Récapitulatif des failles de sécurité par MDN:
https://developer.mozilla.org/fr/docs/Learn/Server-side/First_steps/Website_security

ANNEXES

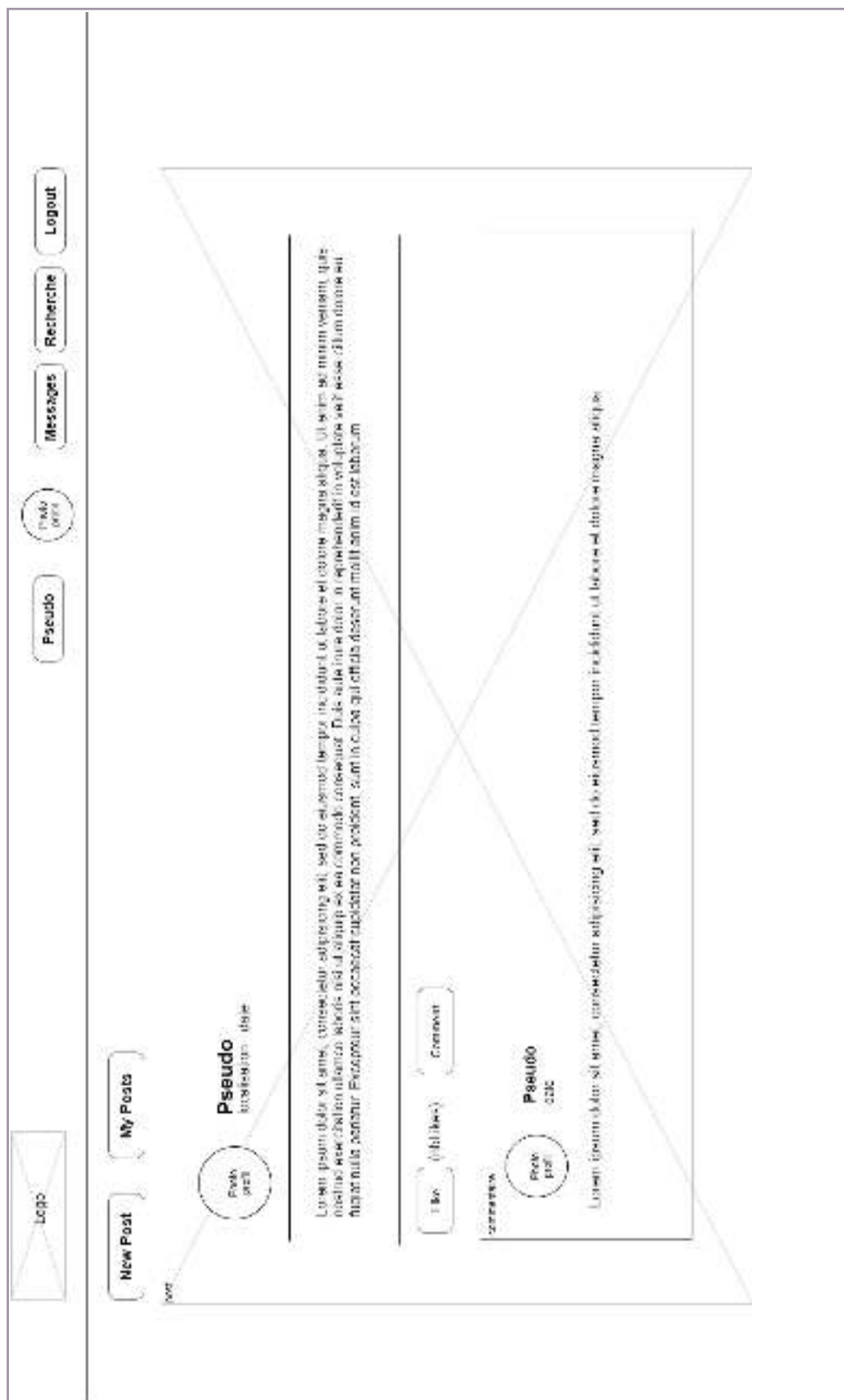
Ci-dessous : Modèles conceptuel des données (MCD).

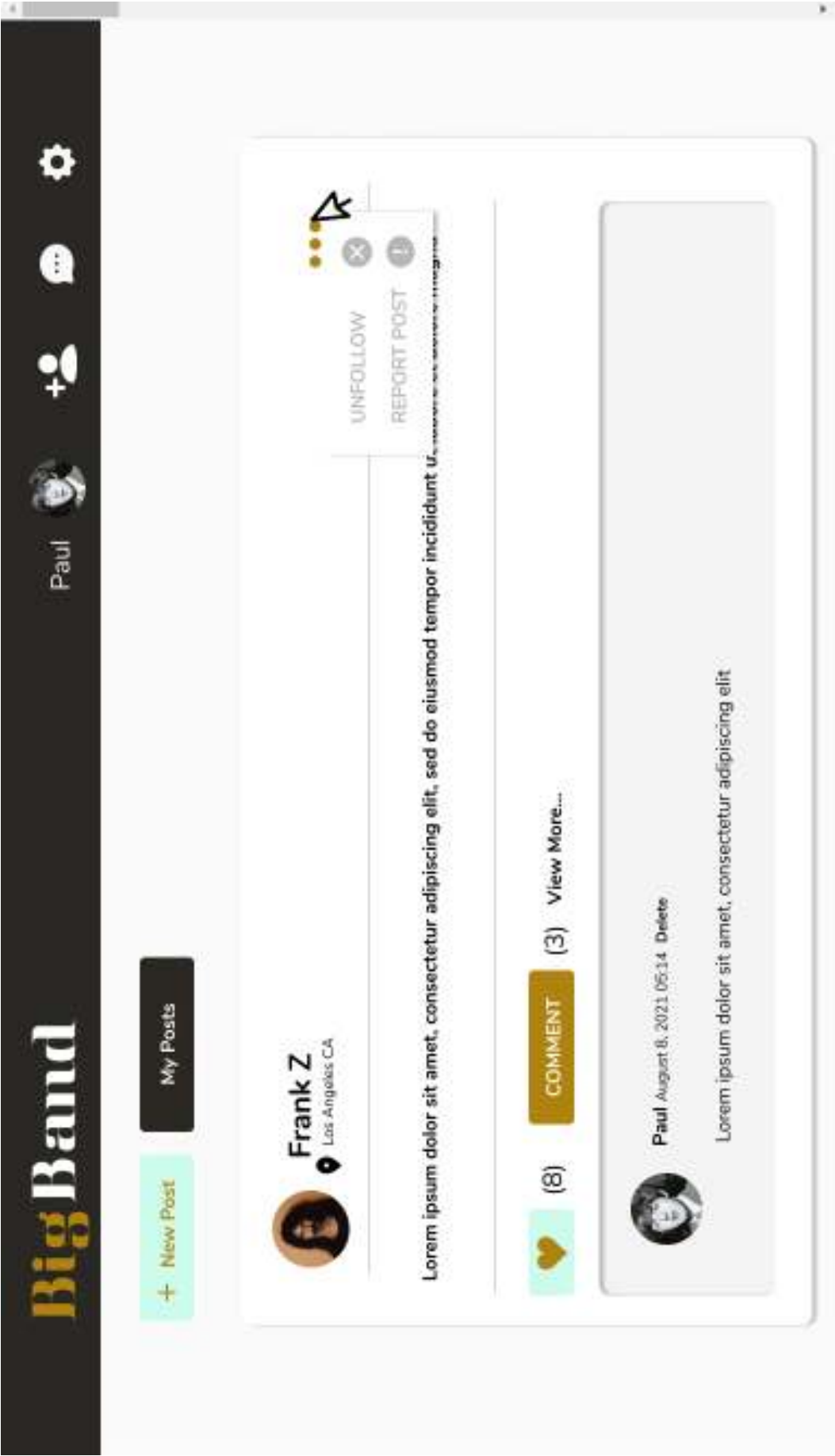


Ci-dessous : Modèles logique des données (MLD).

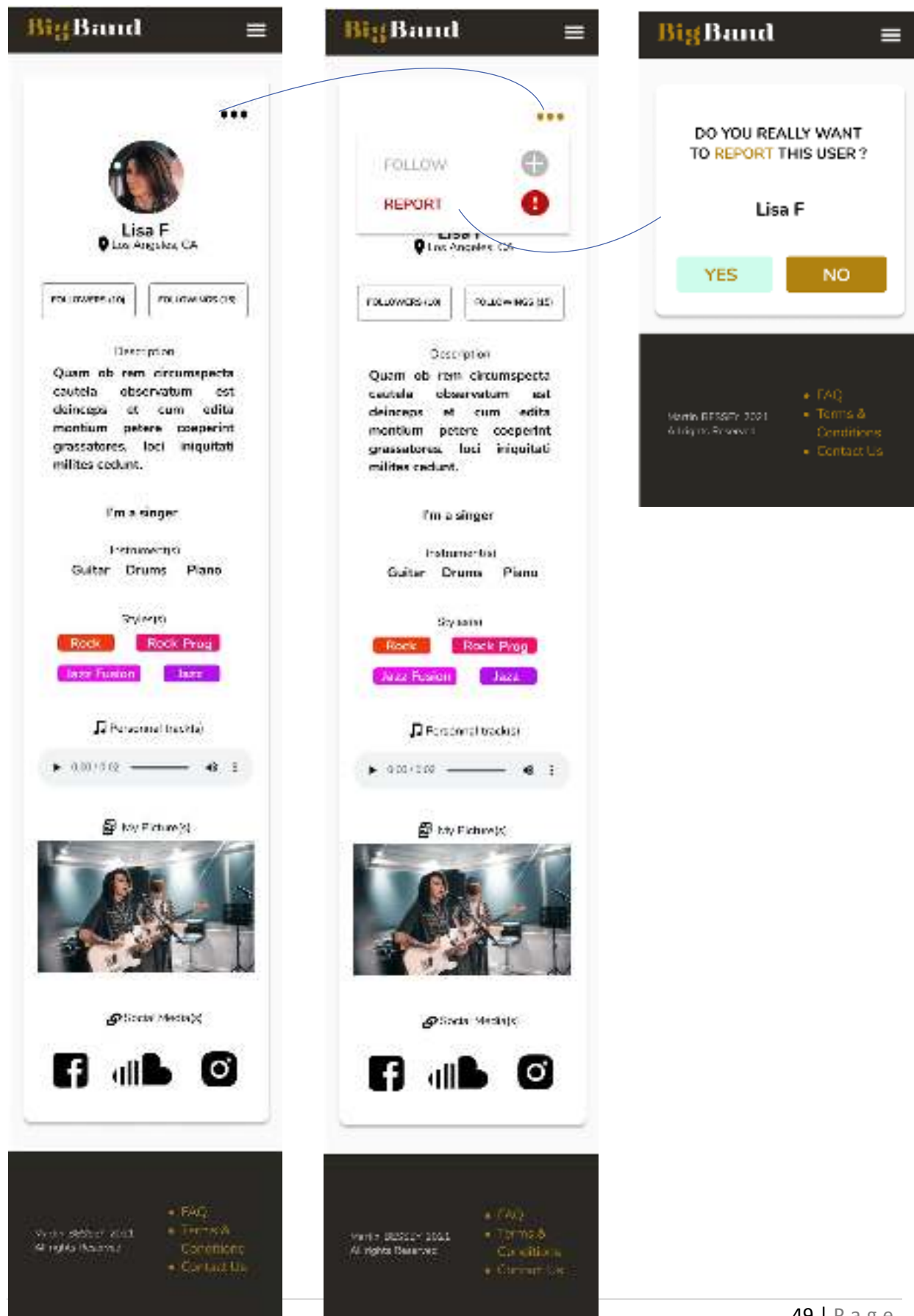


Ci-dessous et page suivante : maquettage de la page contenant la liste des post des utilisateurs suivit par l'utilisateur en session





. Ci-dessous: maquetage d'une version mobile prenant en compte l'UI/UX. Ici pour signaler un utilisateur.



Ci-dessous : maquettage de la page permettant la recherche d'autres utilisateurs.

