PVT

Un projet de Florian Duchesne
Développeur web et web mobile
Elan Formation - Colmar 2021

Table des matières

Remerciements

Merci à Hannah Lafargue pour son soutien, et d'avoir supporté d'être à mes côtés durant cette période particulière de notre vie commune.
Merci à Corinne Ringeisen, puisque c'est elle qui m'a aiguillé sur cette formation.
Merci à Gilles Muess de m'avoir pris en formation, et pour nous avoir accompagnés alors que nous étions encore des bébés du code.
Merci à Stéphane Smail. Il mérite de nombreux cookies faits par Jean-Philippe. Son engagement total à nos côtés m'a motivé à donner à mon tour le meilleur de moi-même dans cette formation.
Merci à Mickaël Murmann, Virgile Gibello, Cindy Cahen, Paul Van Kalck et Céline Hugonnot pour leurs interventions ponctuelles et bienvenues.
Enfin, merci à tous mes camarades : Jean-Philippe, Béa, Alexis, Martin, Melvin, Thomas, Terence, Kiliann et Victor. Depuis janvier, à travers nos écrans d'ordinateurs, on a été dans le même bateau, je dirais même dans la même galère, et d'une certaine façon ça nous a unis.

I/ Introduction

A) A propos de moi

Je m'appelle Florian Duchesne, j'ai trente et un ans et je m'inscris dans un projet de reconversion professionnelle. J'ai fait mes études à l'école des arts-décoratifs de Strasbourg (aujourd'hui la HEAR) à l'atelier illustration. J'ai fait plusieurs jobs alimentaires (animation périscolaire, courses à vélo) tout en poursuivant ma pratique artistique en parallèle. J'ai eu la chance de faire un PVT (programme vacances-travail) au Japon, une expérience très enrichissante.
C'est le désir de faire une reconversion qui m'a amené au développement web.
J'ai commencé à me familiariser avec l'algorithmie et le front-end grâce aux sites web France-IOI et OpenClassroom, avant d'être accepté en formation par ELAN à Colmar.
J'ai fait mon stage à l'agence Tiz à Strasbourg.

B) A propos d'ELAN

ELAN Formation, c'est plus de 25 ans d'expérience dans les domaines de la Bureautique, la PAO, le Multimédia, d'Internet, des Techniques de secrétariat.

ELAN Formation, c'est avant tout l'individualisation des parcours de formation.

ELAN est un organisme de formation local.

La méthode pédagogique d'ELAN

- Ecouter et comprendre la demande précise de l'entreprise ou du stagiaire
- Adapter une formation qui prenne en compte la singularité de l'apprenant
- Guider le stagiaire en permanence grâce à un formateur
- Anticiper ses attentes
- Suivre l'évolution des acquis tout en avançant à son rythme
- Valoriser la formation et certifier les compétences acquises
- Valider la pertinence de notre formation

(source : http://elan-formation.info/qui-sommes-nous)

C) A propos du projet

C'est le fait d'avoir accompli un PVT (programme vacances-travail) au Japon qui m'a donné l'idée de mon projet.
Le PVT est un programme qui permet aux ressortissants de pays partenaires en dessous d'un certain âge de bénéficier d'un visa autorisant à résider un an dans le pays et à y travailler.
Il existe déjà des sites de ressources à propos du PVT (en particulier https://pvtistes.net/), mais mon idée était de faire, sur le thème du PVT, un site associant blogging et réseau social.
Les usagers ont ainsi l'opportunité de faire découvrir leur voyage et de découvrir ceux des autres. C'est aussi l'occasion de rencontrer virtuellement d'autres voyageurs, d'échanger ensemble, éventuellement de se donner des conseils ou des recommandations. Un service de messagerie fait également partie du projet, pour une meilleure interaction encore entre utilisateurs.
Très concrètement, l'idée est pour un usager non-inscrit de pouvoir explorer les articles catalogués sur le site, rangés par membres, pays et thèmes.
Mais une personne inscrite aura en plus un profil avec ses informations publiques, un « fil », qui contiendra les publications les plus récentes des personnes qu'elle a décidé de suivre, la possibilité d'elle-même publier des articles, de liker et commenter des articles, ainsi que la possibilité d'envoyer et recevoir des messages privés sur la plateforme.

D) Compétences couvertes par le projet

- **Maquetter une application**

  J'ai d'abord fait un wireframe de mon projet à l'aide d'indesign, pour m'aider à concevoir l'ergonomie, la navigation et l'arborescence du site. Puis j'ai maquetté quelques pages du site pour définir sa charte graphique.

- **Réaliser une interface utilisateur web statique et adaptable**

  L'interface est en effet statique pour un visiteur non-connecté. Et les pages sont en responsive (elles s'adaptent à l'écran de l'usager).

- **Développer une interface utilisateur web dynamique**

  L'interface est dynamique dès le moment où un utilisateur se connecte. A partir de là, le contenu est adapté à l'utilisateur.

- **Créer une base de données**

  J'ai en effet créé une base de données pour mon projet, sur phpmyadmin, avec l'aide de Looping pour définir mon MCD (modèle conceptuel de données).

- **Développer les composants d'accès aux données**

  L'accès aux données se fait via Doctrine, l'ORM (définir) de Symfony.

- **Développer la partie back-end d'une application web ou web mobile**

  J'ai développé la partie back-end de mon application à l'aide de Symfony, dont le patron de conception est le MVP, proche du MVC.

II/ Organisation du travail

A) Méthodes

a) La méthode Agile

La méthode Agile est un ensemble de **méthodes et pratiques basées sur les valeurs et les principes du Manifeste Agile**, un texte datant de 2001 rédigé par des experts du développement d'applications informatiques. Elles permettent notamment de toujours **s'adapter au client** (appelé le **« product owner »**), et de le garder au cœur de la tenue du projet. La **méthode Scrum** est une méthode agile très connue, dont l'une des composantes est le travail par **itérations**, autrement appelés **sprints**. Il s'agit d'un intervalle de temps fixé, durant lequel sont donnés des objectifs à atteindre d'ici la fin de l'itération. Les sprints sont destinés à se succéder jusqu'à la fin du travail.

b) Le MVP

Une notion qui va avec, toujours issue de la méthode Scrum, est la notion de MVP : « **minimum viable product** », soit **produit minimum valable** en français. Il s'agit un produit fonctionnel mais minimal, destiné à être enrichi lors d'autres itérations.

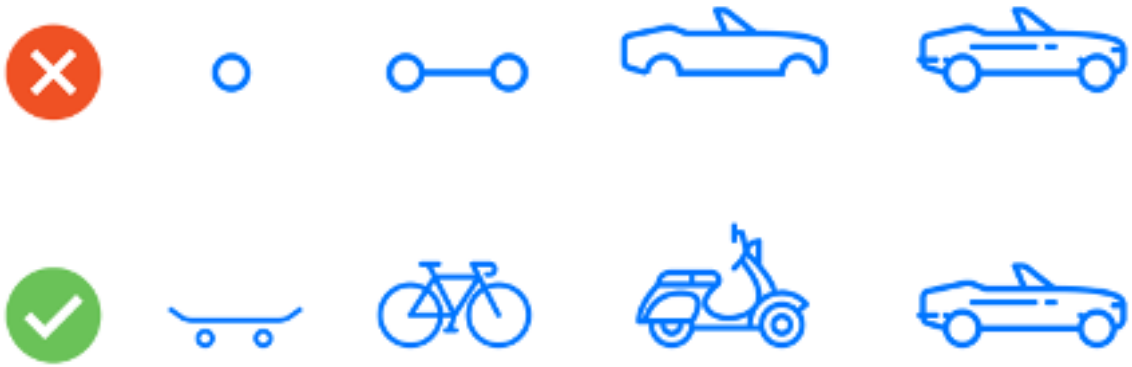


Illustration du MVP

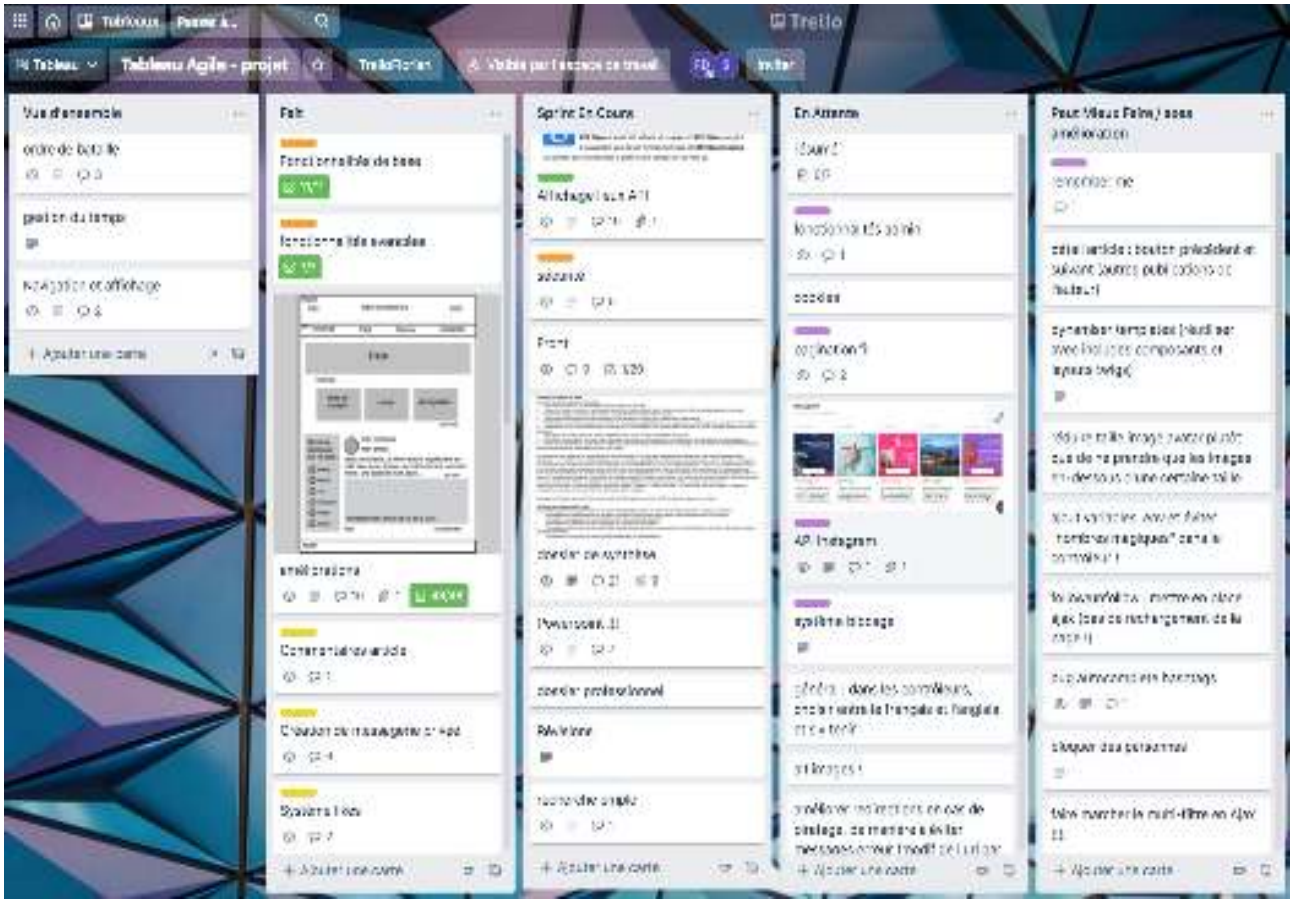
L'idée derrière le MVP est de pouvoir rapidement fournir à des utilisateurs ou à un client un produit destiné à être testé pour obtenir les retours des usagers (participant à l'amélioration du produit par la suite) voire valider l'intérêt du produit.
Il rentre dans la logique de mener un projet adaptable.

c) la méthode MoSCoW

Pour décider de l'ordre dans lequel faire ces autres fonctionnalités, j'ai essayé de me référer à un autre outil organisationnel, la méthode MoSCoW, un outil de priorisation des tâches dont l'acronyme correspond aux expressions suivantes :
M – must have : ce qui doit être fait
S – should have : ce qui devrait être fait si c'est possible
C – could have : ce qui pourrait être fait si c'est possible et que cela n'affecte pas les tâches précédentes
W – Won't have : ce qui ne sera pas fait cette fois-ci mais pourra l'être s'il y a une autre itération sur le projet.

d) Trello

Je me suis inspiré de ces méthodes dans l'organisation de mon travail, comme en témoigne le trello de mon projet.



En plus de chercher à opérer par sprints successifs, je me suis efforcé de travailler dans un ordre qui m'a permis de construire rapidement un MVP avant de consolider le projet avec des fonctionnalités moins prioritaires, effectuées dans un ordre donné en appliquant la méthode MoSCoW.

B) Le cahier des charges

Voici les fonctionnalités du projet que j'ai réalisées

Les fonctionnalités accomplies prioritairement, faisant partie du MVP :

- inscription et connexion d'un utilisateur
- création et édition de son profil
- envoi d'email en cas de mot de passe oublié
- rédaction d'articles catégorisés par thème et pays, avec upload de fichiers
- au clic sur la galerie d'un post, déclenchement d'un lightbox en carousel
- suppression et édition de ses articles
- filtrage des articles catalogués sur le site en fonction de critères
- système de follow
- barre de recherche
- prévention des failles de sécurité
- pour un admin, accès à un panneau dédié
- pour un admin, modification et ajout de thèmes et pays, suppression d'articles, de commentaires et de membres

Celles accomplies au cours des sprints suivants :

- système de hashtags
- système de commentaires (liés aux articles)
- système de likes pour les articles et les commentaires
- messagerie instantanée
- Utilisation de cartes grâce à l'API de Google Maps

C) Technologies

a) Langages informatiques

Au cours du projet, j'ai utilisé les langages informatiques suivants :

HTML5

HTML signifie « HyperText Markup Language », c'est-à-dire « langage de balises pour l'hypertexte ». C'est donc un langage de balisage conçu pour représenter les pages web.

CSS3

Les CSS, pour « cascading style sheets », c'est-à-dire « feuilles de style en cascade », forment le langage informatique employé pour présenter une page web. C'est un langage de feuille de style, qui permet d'appliquer des styles sur différents éléments d'un document HTML.

Js

JS est l'abréviation de javascript : c'est un langage de programmation de scripts, pouvant être employé en tant que langage procédural ou bien orienté objet.

Php

PHP, pour « Hypertext Preprocessor », est un langage de programmation libre, et plus précisément un langage de script utilisé le plus souvent côté serveur. PHP est très utile à la création de pages web dynamiques.

Sql

Le SQL, pour « structured query langage », soit langage de requêtes structurées, est un langage informatique servant à exploiter des bases de données relationnelles. Elle nous permet ainsi de rechercher, supprimer, ajouter et modifier des données dans les bases de données relationnelles.

Sass

Sass est un langage de script préprocesseur qui est compilé ou interprété en css.
Il permet l'utilisation de variables, de mixins et d'héritages des sélecteurs.

b) Applications, frameworks, librairies

xampp	Xampp, comme Laragon, est un outil permettant la configuration d'un serveur de test local avant la mise en œuvre d'un site Web. C'est concrètement un ensemble de logiciels permettant la mise en place d'un serveur web local, avec comme logiciels Apache (un serveur http), MariaDB, PHP (entre autres).
MariaDB	MariaDB est un **système de gestion de base de données**, dérivé de MySQL.
Symfony	Symfony est un **framework**, créé par SensioLabs, basé sur un ensemble de composants PHP et s'inspirant du modèle d'architecture logicielle MVC (pour Modèle-Vue-Contrôleur)
Doctrine	Doctrine est **l'ORM** par défaut de Symfony. L'ORM est l'acronyme de « **object-relational mapping** », soit mapping objet-relationnel en français. C'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.
Composer	Composer est un logiciel **gestionnaire de dépendances** libre écrit en PHP.
Bootstrap	Bootstrap est une collection d'outils utiles à la création du design d'applications web.
JQuery	jQuery est une bibliothèque JavaScript libre et multiplateforme créée pour faciliter l'écriture de scripts côté client dans le code HTML des pages web. Jquery m'a notamment permis de faciliter l'installation de fonctionnalités employant ajax. Ajax est une méthode qui permet d'effectuer des requêtes au serveur web et, en conséquence, de modifier partiellement la page web affichée sur le poste client sans avoir à afficher une nouvelle page complète.
Vscode	Visual Studio Code est un éditeur de code développé par Microsoft.
Photoshop	Photoshop est un logiciel de retouche, de traitement et de dessin assisté par ordinateur, édité par Adobe.
Indesign	Adobe Indesign est un logiciel de PAO produit par Adobe.
Maps JavaScript API	API de Google permettant de cartographier des lieux
FontAwesome	FontAwesome est une police d'écriture d'icônes se basant sur CSS.
Google Fonts	Google Fonts est un service d'hébergement gratuit de polices d'écritures pour le Web.
Trello	Trello est un outil de gestion de projet en ligne.
Github Desktop	Github Desktop est une application permettant d'interagir avec Github à travers une interface graphique. Github est un service web d'hébergement et de gestion de développement, utilisant le logiciel de gestion de versions Git.

D) Le RGPD.

a) Présentation du RGPD

Le RGPD est un acronyme pour « **règlement général sur la protection des données** ».
C'est un règlement de l'Union Européenne adopté en 2016 et entré en vigueur en 2018, qui constitue un texte de référence en matière de protection des données à caractère personnel. Il **renforce et unifie la protection des données pour les individus au sein de l'Union européenne**.
Une donnée personnelle est une information concernant directement ou indirectement une personne physique.
Tous les organismes présents dans l'UE doivent respecter le RGPD, ainsi que les entreprises traitant les données de clients européens.
Il est interdit de collecter certains types de données personnelles sensibles, sauf cas particuliers et justifiés : l'origine raciale, l'orientation sexuelle, la religion, l'opinion politique, syndicale, l'état de santé, les données génétiques, biométriques, les condamnations pénales et infractions.
Les personnes ont le droit de contrôler l'usage de leurs données, la collecte de données par l'organisme doit être pertinente et transparente, et nécessite le consentement explicite et éclairé des usagers.

b) Gestion du RGPD

Concernant mon projet, voilà comment j'ai géré la question du RGPD :

- **Minimisation des données** : je ne collecte que les données personnelles nécessaires à la meilleure expérience du site possible : ainsi, je propose à l'utilisateur, s'il le souhaite, de mentionner dans son profil les voyages qu'il a réalisés ou qu'il souhaite réaliser, ce qui est pertinent dans le cadre de mon projet. Mais **je ne demande aucune information personnelle dont je n'ai pas besoin**.

- **Je ne collecte aucune donnée sensible** (font partie des données jugées sensibles celles liés à la santé, aux opinions politiques, à la sexualité…)

- **Droit à l'accès des données** : L'utilisateur a la possibilité de **consulter** et de **modifier** toutes ses informations personnelles sur le site.

- Lorsque l'utilisateur ferme son compte, toutes les données qui lui sont affiliées sont effacées : son **droit à l'oubli** est respecté, les données ont une **temporalité.**

- **Consentement à l'utilisation des données** : l'utilisateur est prévenu lors de sa première visite sur le site de la politique des cookies à travers une fenêtre modale que la personne pourra accepter ou personnaliser, puis lors de son inscription de la gestion de ses données personnelles et de l'application du RGPD à travers un règlement à valider une fois lues les conditions d'utilisation du site. Son **consentement libre et éclairé** est respecté, ainsi que son **droit à la transparence** (rappel de ses droits d'utilisateur).

III/ Développement

A) Modélisation des données

a) Définition

La modélisation des données est l'analyse et la conception de l'information contenue dans un système afin de représenter la structure de ces informations et en structurer le stockage et les traitements informatiques.
C'est donc un **processus de description de la structure, des associations, des relations et des contraintes relatives aux données disponibles**. Elle sert à établir des normes et à coder des règles de gestion des data.

b) Méthode Merise

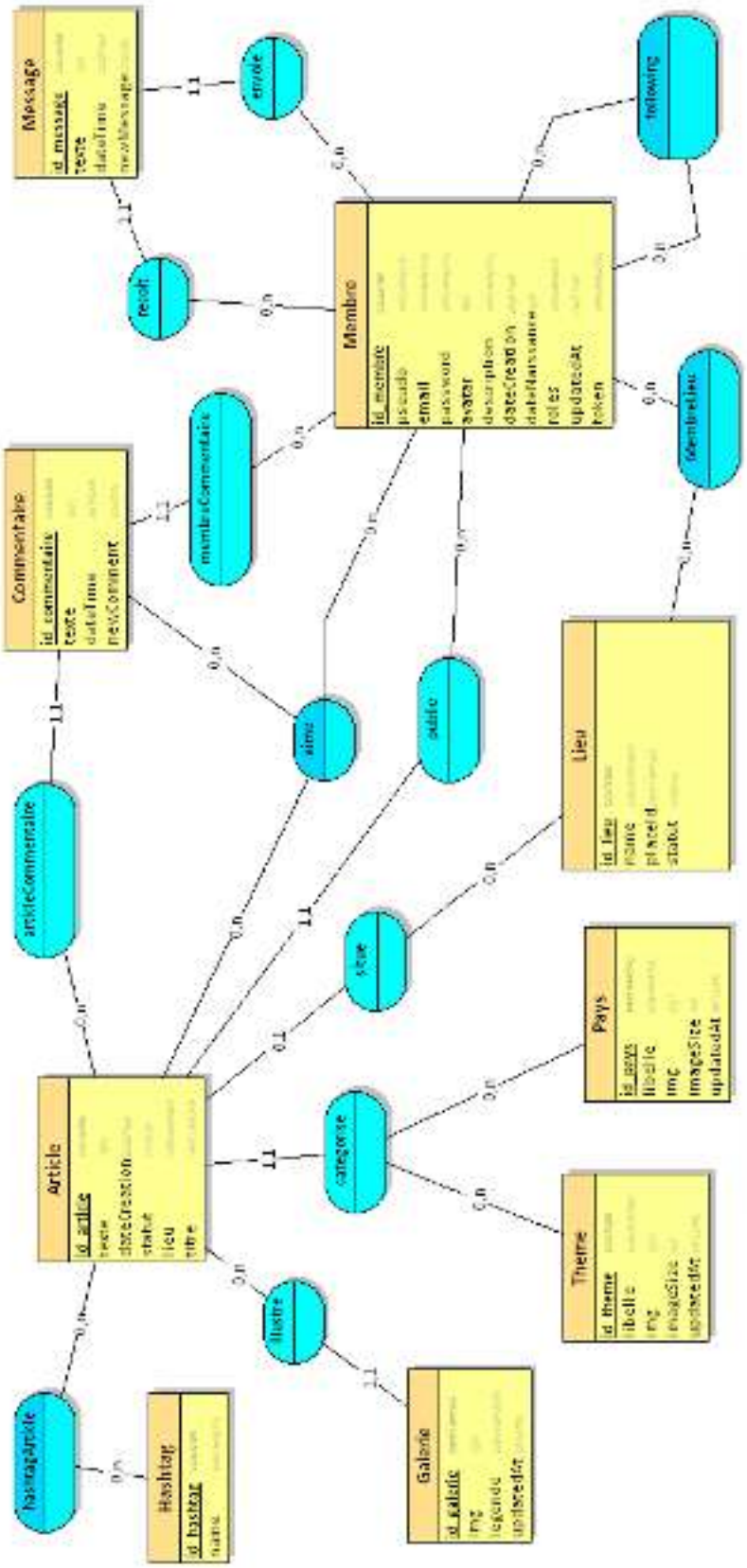
La méthode Merise est une **méthode d'analyse, de conception et de gestion de projet** informatique créée dans les années 70, que j'ai employée à l'aide de Looping pour structurer mon projet.
Les MCD et MLD sont des modèles de données issus de cette méthode.

c) MCD

MCD est l'acronyme de « **modèle conceptuel de données** ». Le MCD est la couche la plus abstraite de la représentation des données, au sens où les données sont représentées de la façon la plus réduite et essentielle possible, et permet de comprendre un système de données.
Il est constitué **d'entités** (constituées d'une propriété identifiante et de propriétés quelconque), de **relations** et de **cardinalités** (ce sont des caractérisations d'une relation constituées d'un couple minimal et maximal : 0,1 / 1,1 / 0,n / 1,n).
Un MCD permet de représenter graphiquement comment les différents éléments d'un site seront liés entre eux.
On remarquera dans mon MCD une relation ternaire entre Article, Commentaire et Membre (« aime »), et une autre entre Article, Pays et Theme (« categorise »), une relation ternaire étant une relation reliant trois entités au lieu de deux (auquel cas on parle de relation binaire).
On remarquera aussi la présence d'une association réflexive (qui relie des occurrences d'une même entité) avec following, qui relie Membre à lui-même.

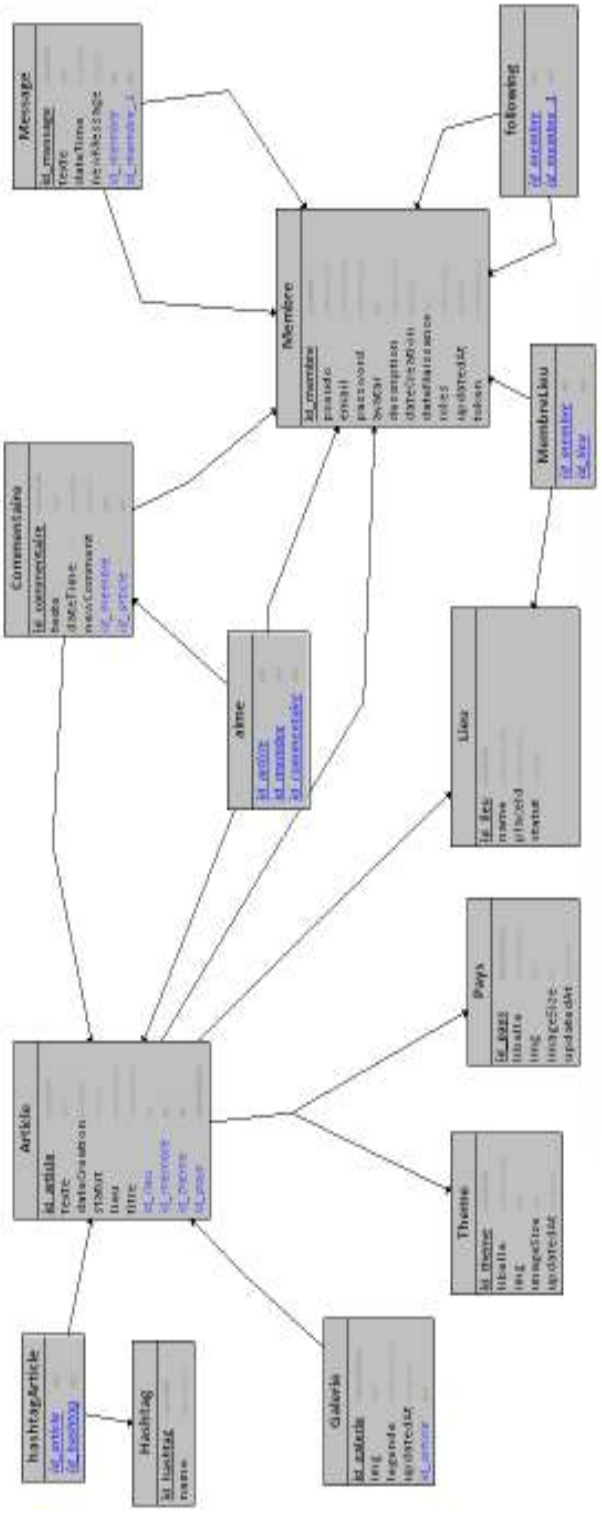
Concrètement et pour donner un exemple, l'entité Membre est constitué d'une propriété identifiante (le champ « id ») et de propriétés quelconques, et peut envoyer entre zéro et un nombre indéfini de messages à d'autres membres, comme le montre la cardinalité entre membre et message. Par ailleurs, un message est forcément envoyé par un seul et unique membre, et envoyé à un seul et unique membre.

Voici le MCD de mon projet :



d) MLD

Il existe aussi le MLD, qui est le **modèle logique des données**. C'est une représentation logique issue du MCD : l'entité devient une table, une propriété identifiante devient une clé primaire, et des cardinalités du MCD découlent les clés étrangères du MLD. Voici le MLD de mon projet :



Une **clé primaire** correspond à l'identifiant unique, et **permet d'identifier une ligne de la table de façon unique**.
Une **clé étrangère** est une **contrainte qui garantit l'intégrité référentielle entre deux tables**.
Les clés étrangères identifient la clé primaire d'une autre table référencée, et sont créées en fonction des relations du MCD.

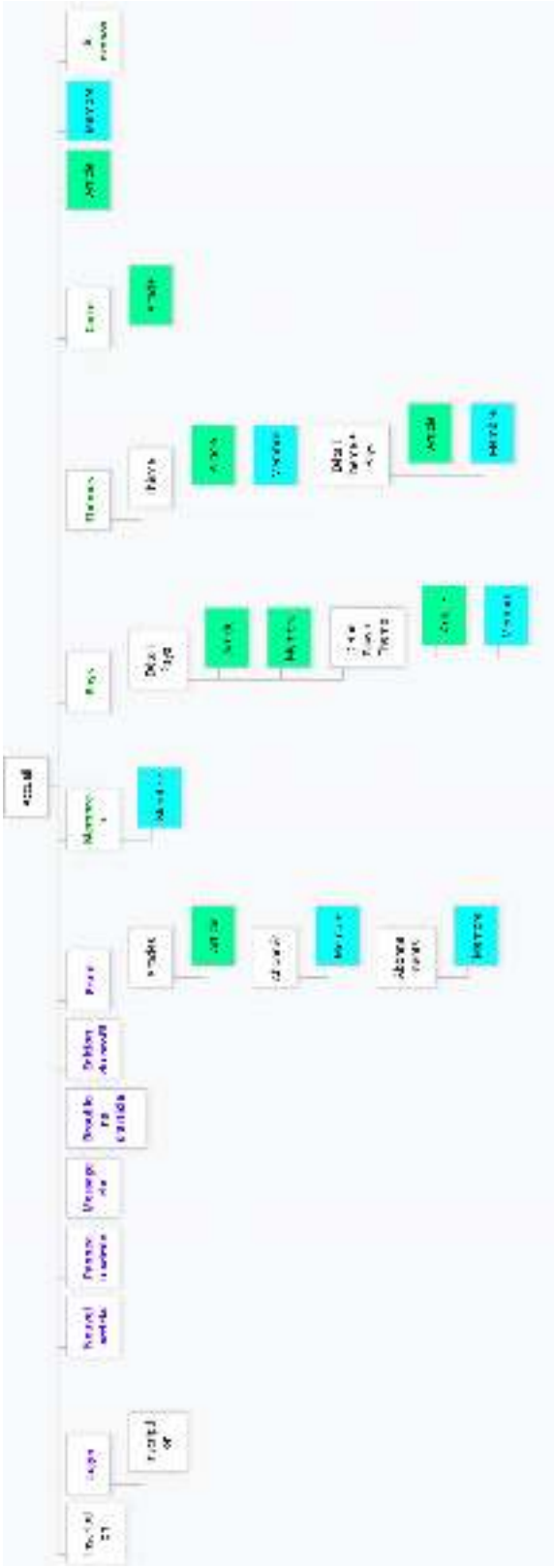
A nouveau, prenant un exemple à partir du MCD de mon projet : un article est associé à un auteur, avec un couple de cardinalités 1,1 / 0,n : un article ne peut être attaché qu'à un seul et unique auteur, tandis qu'un auteur peut publier zéro articles ou un nombre indéfini d'articles.
En conséquence, dans le MLD, la table Membre n'a pas de clés étrangères, mais la table Article a la clé étrangère #id_membre, qui permet d'identifier son auteur.
Comment faire quand deux tables sont associées en ayant de chaque côté un couple de cardinalité 0,n ?
Pour le savoir, étudions la relation entre la table Article et la table Hashtag : un article peut avoir entre zéro et un nombre indéfini de hashtags, tandis qu'un hashtag peut être attaché à zéro ou un nombre indéfini d'articles.
De ce type de cardinalités découle une **table associative** dans le MLD. Il s'agit d'une **table dont la clé primaire est composée de ses clés étrangères associées**, et dont **les clés étrangères renvoient chacune à un côté de la relation**.
Imaginons par exemple un article intitulé « quelques jours à Osaka » ayant pour identifiant la valeur 31, qui correspond à la clé primaire, et un hashtag intitulé « château », ayant l'identifiant 17, lui aussi correspondant à la clé primaire. L'auteur va donner ce hashtag à son article, car il parle de sa visite du château d'Osaka. Que se passe-t-il alors en base de données ? Une ligne se crée dans la table « hashtag_article », avec dans la colonne « hashtag_id » le chiffre 17, et dans la colonne « article_id » le chiffre 31. Le partage de la même ligne représente l'association, et permettra d'identifier que ce hashtag se trouve associé à cet article, et que cet article est associé à ce hashtag.

L'ORM de Symfony, Doctrine, nous permet de manipuler la base de données tout en nous présentant les données sous formes de classes et d'objets : c'est grâce à sa **couche d'abstraction**.
Ainsi, lorsqu'on crée une nouvelle entité sur le projet, Doctrine crée une table en base de donnée, et transforme les propriétés relationnelles en clés.
Des méthodes de la classe permettent par ailleurs d'accéder aux objets auxquels la propriété se rapporte, sans qu'aucune table associative ne soit donc figurée dans le code.
Les annotations d'ORM au-dessus des propriétés permettent cependant à Doctrine de reconnaître les associations, comme nous y reviendrons plus tard dans la partie consacrée à la fonction « following ».

Une fois que les données ont été modélisées, j'ai pu commencer à imaginer la structure du site web. J'ai pour cela constitué une arborescence.

B) Arborescence

L’arborescence d’un site est la charpente sur laquelle on vient ajouter le contenu, c’est-à-dire les différents éléments que les visiteurs auront à consulter. Voici celle que j'ai conçue :



Cette arborescence permet de réfléchir à **l'ergonomie** du site, c'est-à-dire à la manière de l'organiser pour qu'il corresponde le plus possible aux règles en vigueur de **l'UX** et **UI** design (soit respectivement **l'étude des attentes et besoins de l'utilisateur**, et **l'amélioration de l'interaction d'un utilisateur avec le produit**).
Un élément particulièrement important à prendre en compte à ce titre est l'accès le plus rapide possible aux informations voulues (la **règle des 3 clics**). Pour cela, j'ai constitué deux outils de navigation :

- la barre de navigation du header, commun à chaque page, qui renvoie aux différentes pages essentielles du site : la liste des membres, la liste des pays et des thèmes, l'à propos et la carte répertoriant géo-localement les articles.

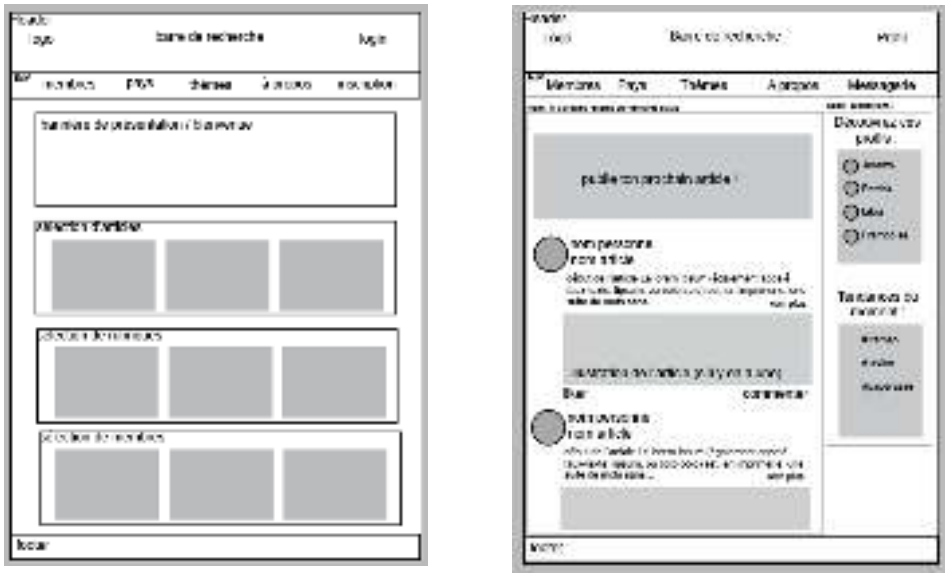
- L'outil de navigation qui s'ouvre au clic sur l'icône en haut à droite, qui donne accès à tout ce qui à trait à l'utilisation du site par un de ses membres : le login bien sûr, la publication d'un nouvel article, le panneau administateur pour l'administrateur, la messagerie, les brouillons d'articles, l'édition du profil, et le profil du membre connecté.

Ainsi, beaucoup de pages sont déjà accessibles depuis n'importe où sur le site via ces deux outils de navigations. Les autres (le profil d'un autre membre, la page dédiée à un article précis…) sont disponibles en très peu de clics.
Une fois cette étape passée, j'ai pu m'attaquer à la maquette du site.

C) Maquette

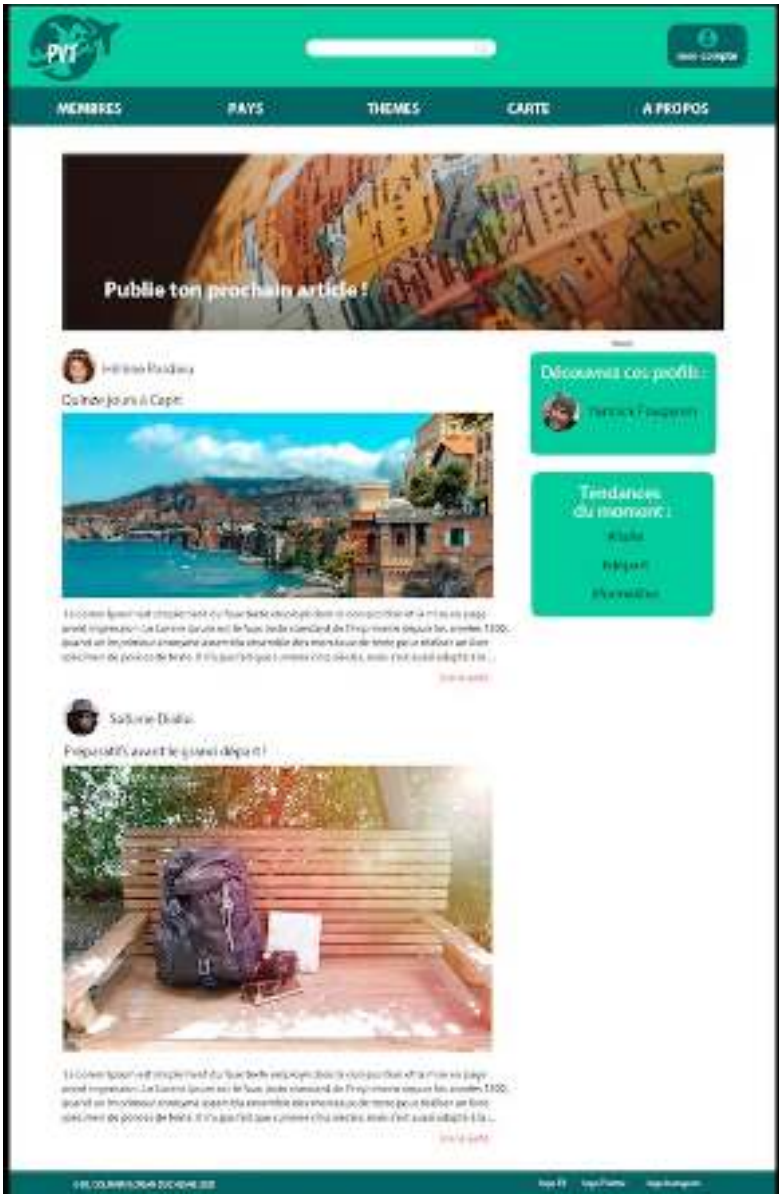
a) Le Wireframe

Le Wireframe est une étape de maquette fonctionnelle, qui permet de définir les zones et composants d'une page.
J'ai d'abord fabriqué les wireframes de mes pages pour m'imaginer leur rendu et leur contenu précis.



Wireframe de la page d'accueil en tant que visiteur, et en tant qu'utilisateur connecté.

J'ai ensuite fabriqué une maquette plus précise de façon à concevoir la charte graphique de mon site : les couleurs utilisées, le style des formes…



maquette de la page d'accueil d'un utilisateur connecté

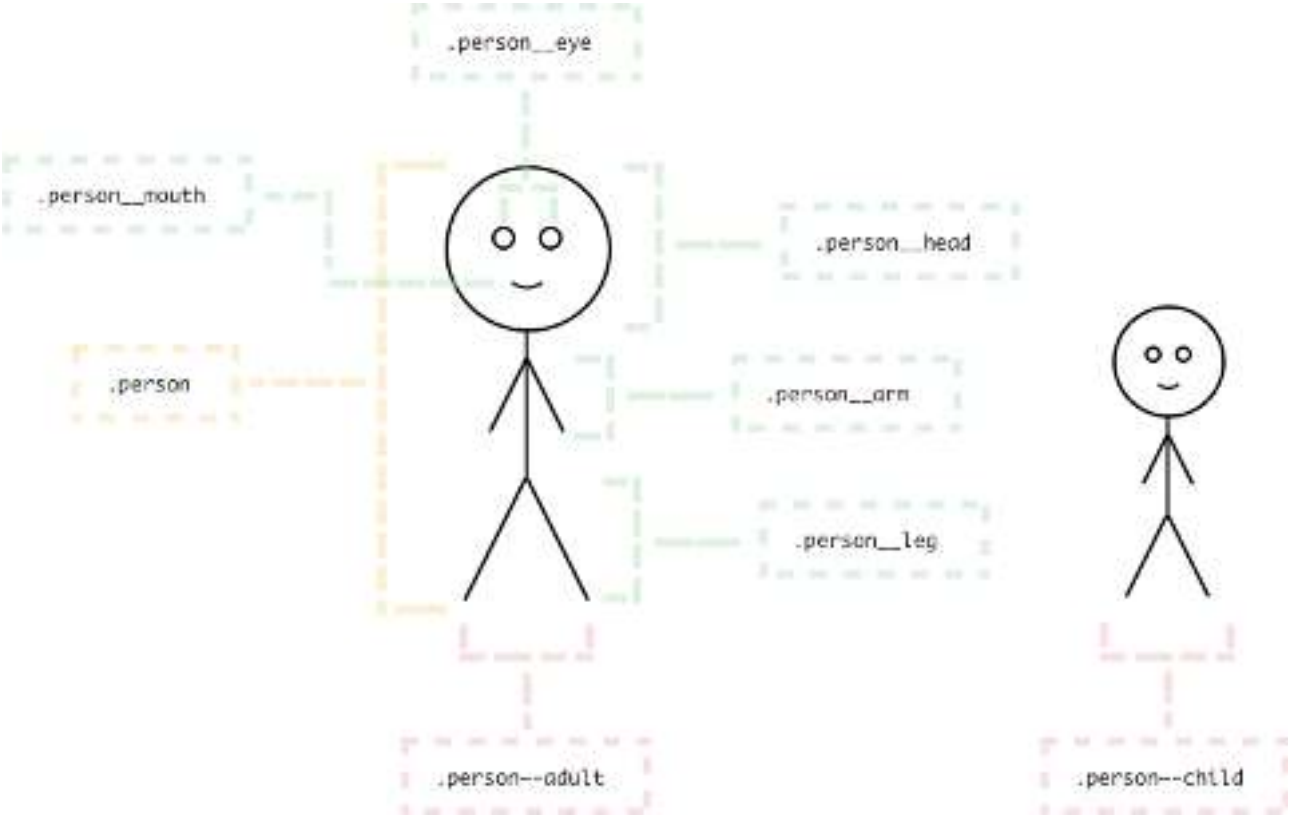
E) Partie Front-End

Pour travailler la partie front du site, j'ai cherché à employer des méthodes reconnues de production de code CSS, et employé des technologies qui m'ont paru pertinentes pour accomplir ce travail : Sass, Twig et Bootstrap.

a) méthodes

- **méthode BEM**

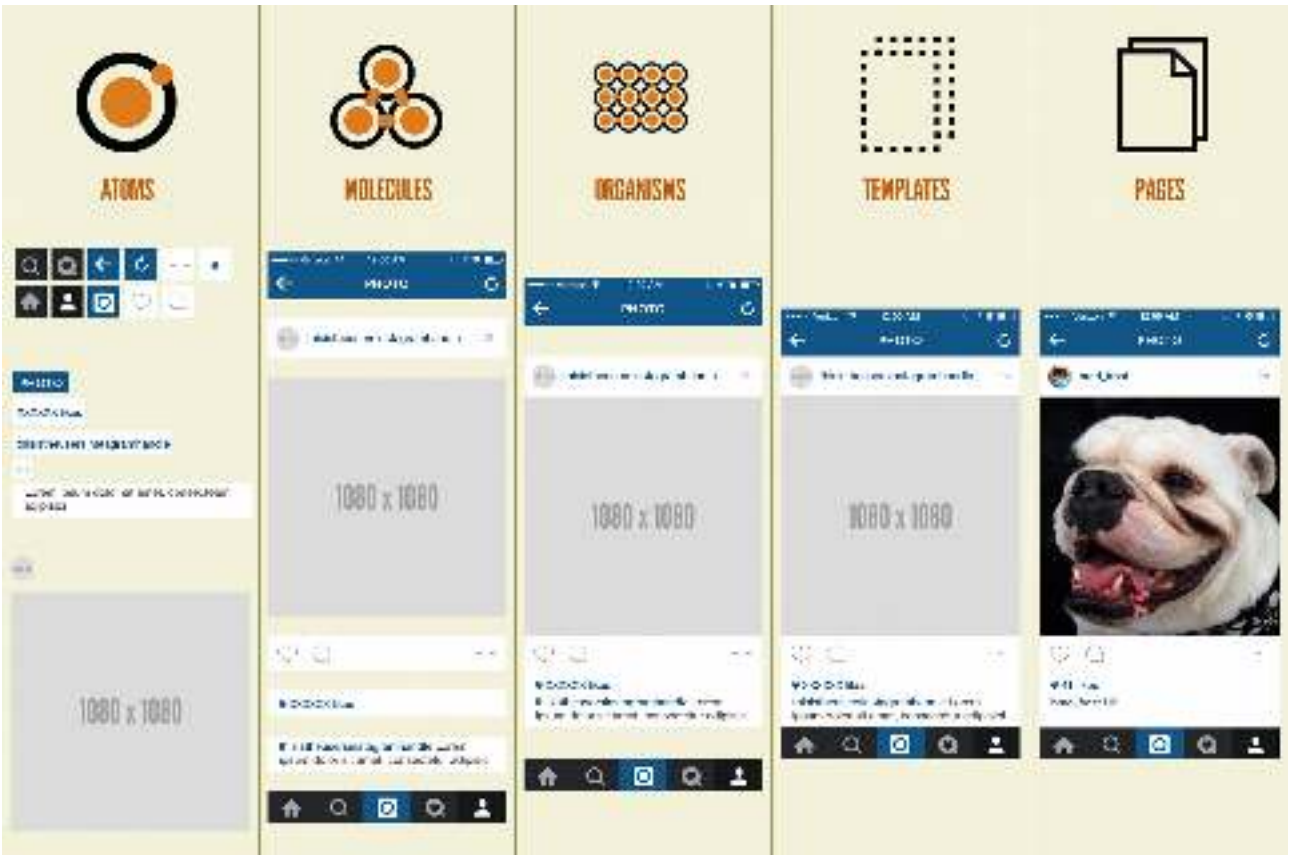
La méthode BEM (pour « block, element, modifier ») est une technique d'écriture du CSS inventée en 2010, permettant d'optimiser et structurer l'emploi du CSS.



Lorsqu'on applique la méthode BEM, une classe CSS est découpée entre la première partie, le bloc, et la deuxième partie, l'élément. On peut ajouter aussi un « modifier » à un bloc.
La méthode BEM permet de réduire les risques de conflits de nommage, puisqu'un élément se trouve rattaché à son bloc parent dans la dénomination de sa classe CSS.
Par ailleurs, BEM est modulaire et réutilisable : on réutilise en effet facilement les différentes classes produites, en utilisant si nécessaire des « modifier ».

- **Design modulaire**

J'ai aussi essayé de garder à l'esprit l'idée de design modulaire (« atomic design » en anglais), une approche du front consistant à designer des éléments destinés à pouvoir être combinés pour composer une page. L'idée est de **réutiliser le code de façon la plus dynamique possible**. On va utiliser les mixins de Sass dans ce sens, mais aussi créer des vues utilisées par d'autres vues à chaque fois que le même type d'élément est reproduit.



b) Technologies

Pour faire la partie front du site web, j'ai employé le langage de script préprocesseur Sass, essayé d'optimiser mon usage du moteur de template Twig, et utilisé aussi la librairie Bootstrap.

- **Sass**

Sass est un **préprocesseur** au sens où il génère dynamiquement des fichiers CSS.
Il nécessite donc une étape de compilation en code CSS, j'ai pour cela utilisé une extension VSCode : Live Sass Compiler.
Sass permet d'optimiser considérablement le code grâce aux **variables** et aux **mixins**. J'ai ainsi consacré un fichier .sass à la déclaration de variables (par exemple pour l'usage des couleurs que j'ai décidé d'employer dans le projet) que j'utilise dans le reste de mes

fichiers .sass. Si je change d'avis sur une couleur employée, je n'ai qu'à modifier la variable et tous les usages de cette couleur se trouveront modifiés.
Les mixins sont également extrêmement pratiques.
Ce sont en quelque-sorte des fonctions CSS. Un mixin peut recevoir un paramètre, et décrit un code CSS à exécuter lorsqu'il est appelé.
J'ai utilisé les mixins notamment pour les avatars des utilisateurs, ou pour l'aperçu visuel (« thumbnail ») d'un article ou d'une catégorie.
Concernant l'avatar, je rentre en paramètre la taille voulue de l'avatar.

```
@mixin avatar($taille)
  width: $taille
  border-radius: 100%
  overflow: hidden
  height: $taille
  margin: 0
  display: flex
  justify-content: center
  .user__avatar
    width: auto
    height: 100%
    overflow: initial
```

Lorsque j'ai besoin de définir un avatar en CSS dans mon code, je n'ai plus qu'à appeler ce mixin.

```
.post
  border-bottom: #f7a2a273 dashed 2px
  margin-top: 3em
  padding-bottom: 3em
  .post__entete
    @include blockFlex(space-between)
    width: 80%
  .post__user
    text-decoration: none
    @include blockFlex(start)
    margin-bottom: 1em
    .user__avatar
      @include avatar(100px)
```

J'ai rangé mes fichiers Sass par composants, layouts et pages, de façon à définir le code CSS de mes pages le plus possible par composants, de manière à les réutiliser facilement.

La classe .aside par exemple (voir page suivante) est appelée pour différentes pages, et correspond aux suggestions de membres que l'utilisateur est susceptible de vouloir follower. Elle est définie dans un fichier à part rangé parmi les composants. Si je modifie cette classe, elle sera donc modifiée sur toutes les pages où elle est appelée.

public > css > components > _aside.sass

```
.aside
  top: 1em
  position: sticky
  background-color: $info
  margin: 2em
  padding: 2em
  border-radius: 15px
  .aside__h6
    color: $white
  .aside__user
    @include blockFlex(start)
    .aside__user__avatar
      @include avatar(50px)
    .aside__user__name
      //variable
      color: $white
      font-size: medium
      margin-left: 0.5em
  .aside__hashtags
    a
      text-decoration: none
      color: $white
      font-style: italic
.aside--bottom
  top: 230px
```

J'ai opté pour la même approche pour le gestionnaire de template Twig : j'ai rangé mon dossier « template », composé de fichiers twig, également par composants, layouts et pages.
Voici par exemple la page Twig faisant référence à un index d'articles (voir page suivante).

templates > pages > article > index.html.twig

```
20  {% include 'components/lightbox.twig' %}
21
22  <div class="wrapper-whole">
23    {% if brouillon %}
24      <h1 class="text-center m-5">Brouillons de
25        {{user.pseudo|capitalize}}</h1>
26    {% else %}
27      {% if user is defined %}
28        <h1 class="text-center m-5">Articles de
29          {{user.pseudo|capitalize}}</h1>
30      {% else %}
31        {% if tag is defined %}
32          <h1 class="text-center m-5">Articles avec le mot-clé
33            {{tag.name|capitalize}}</h1>
34        {% else %}
35          <h1 class="text-center m-5">Articles situés à
36            {{lieu|capitalize}}</h1>
37        {% endif %}
38      {% endif %}
39    {% endif %}
40
41    <div class="wrapper-main wrapper-main--indexArticles">
42      <div class="wrapper-posts">
43        {% for publication in publications %}
44          {% if publication.statut == 1 %}
45            {% include 'components/publication.twig' %}
46            <p class="post__statut">Publié</p>
47          {% endif %}
48
49          {% if publication.statut == 0 %}
50            {% if user is defined %}
51              {% if user == app.user %}
52                {% include 'components/publication.twig' %}
53
54                <p class="post__statut">Brouillon</p>
55              {% endif %}
56            {% endif %}
57          {% endif %}
58        {% endfor %}
59      </div>
60    </div>
61  </div>
62  {% endblock %}
```

On peut voir qu' il appelle le composant « lightbox.twig », dont le code permet de déclencher l'aperçu de photos en lightbox au clic.
Ensuite, il s'adapte aux données transmises par le contrôleur. En effet, cette vue Twig est retournée pour différents usages : elle est employée pour renvoyer vers les brouillons d'articles d'un utilisateur, pour lister les articles d'un membre précis, pour lister les articles avec un certain hashtag, ou bien pour lister les articles situés dans un certain lieu. Pour cela, je profite des conditions que Twig nous laisse faire, en optant pour différents titres en

fonction de la condition respectée.
Ensuite, j'appelle le composant « publication.twig » pour chaque publication transmise par le contrôleur grâce à la boucle for proposée par Twig, en ne donnant à voir les brouillons que si un membre est connecté, et si ce membre est celui dont le profil est consulté.

Twig est très avantageux dans le sens où il se prête à un **usage très dynamique**, grâce aux **conditions**, aux **boucles** for, aux **includes** mais aussi à toutes ses autres fonctions et **filtres**.
Je vais l'appuyer en montrant un extrait de la feuille publication.twig, ce qui me permet d'aller au bout de la démonstration.

```
41      {% if (publication.hashtags|length > 0) %}
42        <p>Mots-clés :
43          {% for hashtag in publication.hashtags %}
44            <strong>
45              <a href="{{ path('indexTag', {id: hashtag.id}) }}">{{hashtag.name}} </a>
46            </strong>
47          {% endfor %}
48        </p>
49      {% endif %}
50      {% if app.user %}
51        {% set liked = false %}
52        {% for like in publication.likes %}
53          {% if (app.user == like.user) %}
54            {% set liked = true %}
55          {% endif %}
56        {% endfor %}
57        <div>
58          <a href="{{ path('likeArticle', {idArticle: publication.id}) }}" class="like">
59            {% if (liked == true) %}
60              <i class="fa fa-heart"></i>
61            {% else %}
62              <i class="far fa-heart"></i>
63            {% endif %}
64          </a>
65          <span class="nbLikes">
66            {{publication.likes|length }}</span>
67        </div>
68        {% include 'components/commentaires.twig'%}
69
70      {% endif %}
71
72  </div>
```

Dans cet extrait, je teste le nombre de hashtags associés à la publication grâce au filtre length. Si il y a au moins un hashtag, j'ouvre un paragraphe qui cite les hashtags en question dans une boucle for.
Ensuite, si un utilisateur est connecté, je vérifie s'il a déjà liké la publication. Si un like de la publication est le sien, j'utilise la fonction set qui me permet de créer une variable. Cette variable me permettra de vérifier en dehors de la boucle si la publication a été likée ou non par l'utilisateur, et en fonction, j'affiche un icône qui indique que la publication est déjà likée (un cœur colorié en bleu), ou pas likée pour le moment (un cœur dont le contour est bleu mais l'intérieur est vide).

Ensuite, toujours si un utilisateur est bien connecté, j'inclus encore un composant : celui qui permet l'affichage des commentaires de la publication.

c) Le responsive

Pour rendre mon site responsive, j'ai utilisé d'une part bootstrap, et d'autre part les media queries.

- **Bootstrap**

Bootstrap a un fonctionnement très intéressant par rapport au responsive : son système de colonnes.
La largeur de la **page** est **découpée en douze colonnes**, il suffit d'appeler la classe CSS « **col** » sur une rangée de balises au même niveau, la classe « **row** » sur leur balise parente, et d' ajouter le nombre de colonnes qu'on souhaite voir occupées par chaque balise, par exemple « col-6 » pour indiquer qu'une balise devra occuper la moitié de la largeur de la page.
Au-delà de ça, il est possible d'ajouter une autre valeur entre « col » et le nombre de colonnes à occuper : « **sm** » pour **small**, « **md** » pour **medium**, « **lg** » pour **large** et « **xl** » pour **extra large** : chacune de ces classes sera effective pour les tailles d'au-dessus si d'autres classes avec une autre taille n'est pas rajoutée.

```
15  {% block body %}
16    <h1 class="titre">Pays</h1>
17    <div class="wrapperFlex row">
18      {% for pays in pays %}
19        <a href="{{ path('pays_show', {id: pays.id}) }}" class="a col-lg-3 col-sm-5 col-11 m-2">
20          <figure class="pays picture">
21            <img src="{{ asset('img/pays/' ~ pays.img) }}" alt="#" class="thumbnail">
22            <figcaption class="title">
23              {{ pays.libelle }}
24            </figcaption>
25          </figure>
26        </a>
27      {% endfor %}
28    </div>
29  {% endblock %}
```

Par exemple, sur la page donnant l'index des pays listés sur mon site, chaque vignette présentant le lien vers un pays a pour classe « col-lg-3 col-sm-5 col-11 » : cela signifie que les vignettes prendront presque toute la largeur de la page tant qu'elle fera moins de 576 pixels de large (valeur par défaut de small), puis cinq colonnes sur 12 jusqu'à 992 pixels (valeur par défaut de large), puis trois colonnes sur 12 au-delà.

J'ai par ailleurs également réalisé des media queries.

- Les **media queries**

Les media queries (« requêtes média » en français) servent à donner des **instructions** CSS différentes **en fonction du type d'un appareil et de ses caractéristiques**. Une requête va donc se composer d'un type de média optionnel et d'une ou plusieurs expressions de caractéristiques.

Dans le cas du responsive, ce qui nous intéressera, c'est les appareils dotés d'un écran, et la largeur de la zone d'affichage.

Dans les requêtes visibles ci-dessous, dédiées à des parties du header de mon site, je demande donc d'exécuter les instructions données si le type de média est un écran et si sa largeur est en-dessous de 576 pixels (breakpoint minimal suggéré par Bootstrap sous la dénomination « extra small », pour les petits téléphones orientés en portrait).

```
13  #myLinks
14    @media screen and (max-width: 576px)
15      display: none
16    @media screen and (min-width: 576px)
17      display: flex
18
19  #searchBig
20    display: flex
21    width: 40%
22    input
23      background-color: $clearBlue
24    @media screen and (max-width: 576px)
25      display: none
26
27  #searchLittle
28    @media screen and (min-width: 576px)
29      display: none
30
31  #menuBurger
32    @media screen and (min-width: 576px)
33      display: none
```

d) Le référencement

Pour améliorer le **référencement naturel** du site, voici les points sur lesquels j'ai été vigilant :

- utiliser les **bonnes balises** html **pour le bon contenu**

  Utiliser des balises <article> pour un article par exemple. Plus on respecte l'usage des balises, plus le référencement sera amélioré.

- Le système des **hashtags**

  Les hashtags permettent d'user de mots-clés inscrits dans des balises (on en revient au point précédent), ce qui encore une fois apporte des points sur le référencement.

- Les **maillage interne** et les **liens externes** au site améliorent également le référencement.

- La page d'à propos, qui emploie des mots-clés importants pour la description du contenu du site dans des balises , et renvoie vers des sites de référence par rapport à la question du PVT.

Un bon référencement naturel contribue à une meilleure optimisation pour les moteurs de recherche (SEO).

D) Partie Back-End

a) Symfony

Pour développer ce projet, j'ai décidé d'utiliser le framework Symfony.
En effet :

- ce framework fournit un excellent **cadre de travail** et facilite de très nombreux aspects. Il nous fournit une structure établie dans laquelle travailler, qu'il nous suffit de respecter et d'alimenter.
- Il a l'avantage de fournir **Doctrine**, un ORM extrêmement utile qui facilite beaucoup l'exploitation de la base de données.
- La **gestion de la sécurité** par Symfony est aussi très pratique. Il gère notamment les failles XSS, CSRF et SQL, que nous étudierons plus tard.
- **Symfony est très bien documenté** sur internet, et peut afficher avec précision les erreurs commises en cas de bug.
- Enfin, l'utilisation de Symfony m'a aussi permis d'installer plus facilement des **bundles**, qui m'ont permis d'accélérer certains aspects du développement (l'upload d'images et l'éditeur de texte, particulièrement).

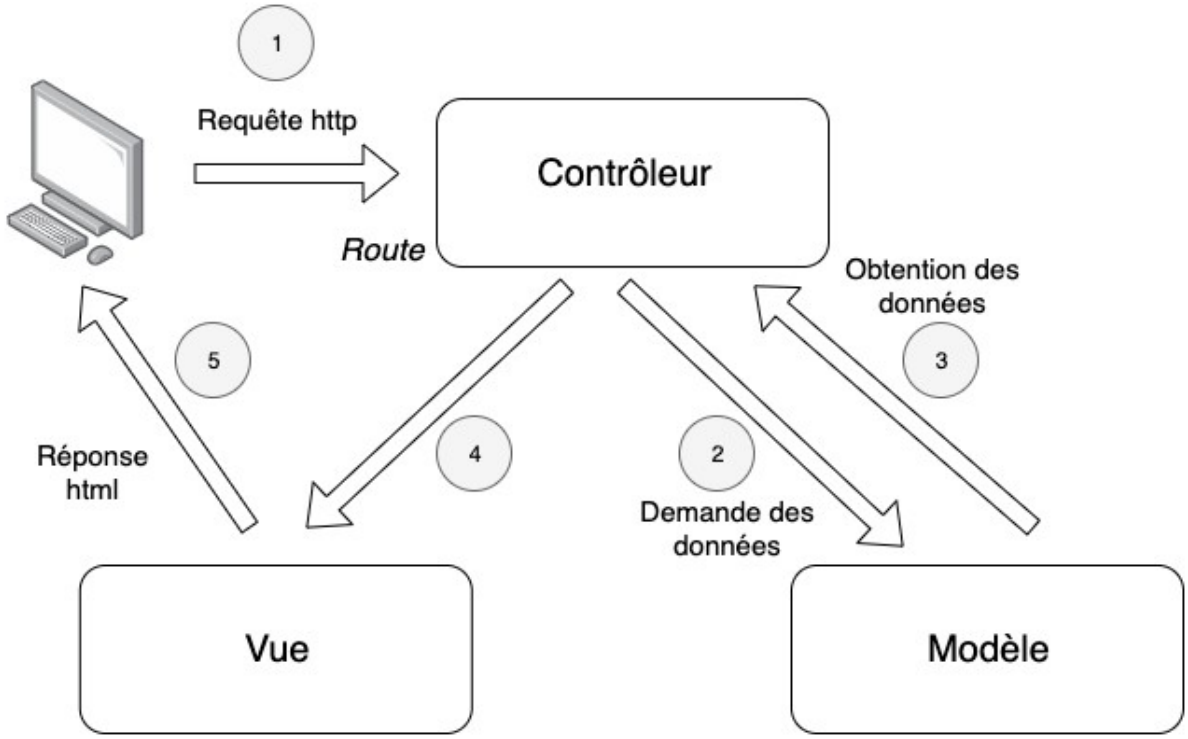
b) Design Pattern

Symfony suit une certaine architecture, qui a son propre design pattern.
Le Design Pattern, autrement appelé « **patron de conception** », est un modèle de conception, reconnu comme bonne pratique lors de la conception d'une application informatique. Il permet :

- un développement plus rapide en suivant des modèles ayant fait leurs preuves,
- une meilleure organisation d'un ensemble de codes qui constituent une application,
- la reprise ou la relecture par d'autres développeurs.

c) Le MVC

J'ai utilisé le design pattern fourni par la structure de Symfony, le MVC.
Le MVC est l'acronyme de « **Modèle-Vue-Contrôleur** ». C'est donc un patron de conception qui se divise en trois couches distinctes.
La couche **Modèle** est celle qui **interagit avec la base de données** et la manipule.
La couche **Contrôleur reçoit les requêtes et les traite** : elle fait le lien entre la couche Modèle et la couche Vue.
La couche **Vue renvoie les données à l'utilisateur à travers une interface graphique**.



Dans le cas précis de Symfony, lorsqu'un client demande une **URL**, celle-ci est d'abord **traitée par le contrôleur frontal de Symfony** (il s'agit de la page index.php dans le dossier « public ») avant d'être envoyée au **noyau** (il s'agit de la page kernel.php dans le dossier « src »).
Quand le **noyau reçoit la demande** d'une URL, il **appelle le service de routing**, celui-ci va alors lui indiquer le contrôleur à appeler pour l'URL que le client demande. Le noyau appelle alors le **contrôleur** qui va si il en a besoin, appeler le **modèle** et générer la **vue**.
Quand le contrôleur a fini de générer la vue, il retourne le résultat au noyau, celui-ci va alors transmettre la réponse au client.

Concrètement, à quoi ressemble le MVC dans le cadre de mon projet ? Voici un exemple.
Lorsqu'on est sur la page d'accueil du site, voici la fonction qui a été appelée à travers la requête http dans la couche contrôleur :

```php
    /**
     * @Route("/", name="home")
     */
    public function index(User $user = null)
    {
        // Symfony nous permet d'identifier avec "getUser" si le visiteur est un utilisateur identifié ou non.
        $user = $this->getUser();

        // On rassemble les données nécessaires à la vue grâce aux repositories des entités User, Pays et Theme.
        // C'est Doctrine, l'ORM de Symfony qui nous permet d'accéder aux repositories des différentes classes.
        // On accède aux méthodes de chaque repository, ce qui nous permet d'utiliser la méthode findAll.
        $users = $this->getDoctrine()->getRepository(User::class)
            ->findAll();
        $countries = $this->getDoctrine()->getRepository(Pays::class)
            ->findAll();
        $themes = $this->getDoctrine()->getRepository(Theme::class)
            ->findAll();
        // Si la variable $user est vide, on redirige vers la route nommée "homeVisitor".
        if (empty($user)){
            return $this->redirectToRoute('homeVisitor');
        }

        // On renvoie la vue twig avec les données instanciées dans la fonction.
        return $this->render('pages/home/indexBEM.html.twig', [
            'users' => $users,
            'countries' => $countries,
            'themes' => $themes
        ]);
    }

        /**
     * @Route("/homeVisitor", name="homeVisitor")
     */
    public function indexVisitor()
    {
        $users = $this->getDoctrine()->getRepository(User::class)
            ->findAll();
        $countries = $this->getDoctrine()->getRepository(Pays::class)
            ->findAll();
        $themes = $this->getDoctrine()->getRepository(Theme::class)
            ->findAll();
        $articles = $this->getDoctrine()->getRepository(Articles::class)
            ->findAll();
            return $this->render('pages/home/indexVisitor.html.twig', [
                'users' => $users,
                'countries' => $countries,
                'themes' => $themes,
                'articles' => $articles
            ]);
    }
```

Le contrôleur appelle donc la couche Modèle grâce à Doctrine, avant de retourner une vue ou de renvoyer vers la fonction indexVisitor si aucun utilisateur n'est reconnu.

Les entités et les repositories font partie de la couche Modèle, ils permettent de représenter le contenu de la base de données et d'interagir avec elle.

```php
<?php

namespace App\Entity;

use App\Repository\ArticleRepository;
use Doctrine\Common\Collections\ArrayCollection;
use Doctrine\Common\Collections\Collection;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass=ArticleRepository::class)
 */
class Article
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="text")
     */
    private $texte;

    /**
     * @ORM\Column(type="datetime", nullable=false, options={"default": "CURRENT_TIMESTAMP"})
     */
    private $dateCreation;

    /**
     * @ORM\Column(type="string", length=255, nullable=true)
     */
    private $lieu;

    /**
     * @ORM\Column(type="boolean")
     */
    private $statut;

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

    /**
     * @ORM\OneToMany(targetEntity=Galerie::class, mappedBy="article", orphanRemoval=true)
     */
    private $galeries;
```

Une **entité** (ici Article) représente une **classe orienté objet**: elle a des propriétés (ici l'id, le texte, la date de création, le lieu…) et des méthodes (en particulier les getters et setters). Elle représente cependant une table en base de données. C'est grâce à Doctrine, l'ORM de Symfony.

```php
<?php

namespace App\Repository;

use App\Entity\Article;
use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
use Doctrine\Persistence\ManagerRegistry;

/**
 * @method Article|null find($id, $lockMode = null, $lockVersion = null)
 * @method Article|null findOneBy(array $criteria, array $orderBy = null)
 * @method Article[]    findAll()
 * @method Article[]    findBy(array $criteria, array $orderBy = null, $limit = null, $offset = null)
 */
class ArticleRepository extends ServiceEntityRepository
{
    public function __construct(ManagerRegistry $registry)
    {
        parent::__construct($registry, Article::class);
    }

    /**
     * @return Article[] Returns an array of Article objects
     */
    public function findByFollow($following, $user)
    {
        return $this->createQueryBuilder('a')
            ->where('a.statut = 1')
            ->andWhere('a.auteurArticle IN (:following) OR a.auteurArticle IN (:user)')
            ->setParameter('following', $following)
            ->setParameter('user', $user)
            ->orderBy('a.id', 'DESC')
            ->getQuery()
            ->getResult();
    }

    /**
     * @return Article[] Returns an array of Article objects
     */
    public function findByAuteurArticleAndStatut($user)
    {
        return $this->createQueryBuilder('a')
            // ->where('a.statut = 1')
            ->where('a.auteurArticle = (:val)')
            ->setParameter('val', $user)
            ->orderBy('a.id', 'DESC')
            ->getQuery()
            ->getResult();
    }
```

Les repositories nous permettent d'aller chercher les données notamment grâce au Query Builder de Doctrine, qui fournit une manière orienté objet d'écrire des requêtes.

Concernant l'entité Article, son repository contient deux fonctions. La première est celle qui permet au contrôleur d'afficher sur le « fil » d'un utilisateur les articles partagés par lui-même et par les personnes qu'il suit, et dont le statut est publié (un article peut être enregistré comme brouillon).

L'autre est la fonction qui permet de trouver tous les articles d'un utilisateur.

Le contrôleur qui a demandé les données aux repositories retourne ensuite une vue.

Voici la vue de la page d'accueil réservée aux visiteurs non-inscrits :

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

{% block title %}Hello PaysController!{% endblock %}

{% block stylesheets %}

    {{ parent() }}

    {# <link
    href="{{ asset('css/home/visitor/style.css') }}" rel="stylesheet"/>  #}
    {#{{ encore_entry_link_tags('app') }}#}
{% endblock %}

{% block body %}


<div class="wrapper-whole">
    <figure class="publishStuff">
        <img class="publishStuff__img" src="{{ asset('img/inscription4.jpg') }}"/>
        <h1 class="publishStuff__h1 publishStuff__h1--register">Inscris-toi !</h1>
    </figure>

    {# <div class="wrapper-main"> #}
    <h1 class="titre">Pays</h1>

    <div class="wrapperFlex">
    {# Ici, on fait une boucle, mais on ne retient que les trois premiers résultats de la boucle grâce au filtre "slice" #}
     {% for pays in countries|slice(0,3) %}
    <a href="{{ path('pays_show', {id: pays.id}) }}">
    <figure class="pays picture">
        <img src="{{ asset('img/pays/' ~ pays.img) }}" alt="#" class="thumbnail">
        <figcaption class="title"> {{ pays.libelle }} </figcaption>
    </figure></a>
     {% endfor %}
</div>
  <button class="link">Voir plus</button>

    <h1 class="titre">Thèmes</h1>
    <div class="wrapperFlex">

  {# Ici, on fait une boucle, mais on ne retient que les trois premiers résultats de la boucle grâce au filtre "slice" #}
     {% for theme in themes|slice(0,3) %}
    <a href="{{ path('theme_show', {id: theme.id}) }}">
        <figure class="theme picture">
            <img src="{{ asset('img/themes/' ~ theme.img) }}" alt="#" class="thumbnail">
            <figcaption class="title"> {{ theme.libelle }} </figcaption>
        </figure>
    </a>
     {% endfor %}

    </div>
    <button class="link">Voir plus</button>

    <h1 class="titre">Articles populaires</h1>
    <div class="wrapperFlex">

   {# Ici, on fait une boucle, mais on ne retient que les trois premiers résultats de la boucle grâce au filtre "slice" #}
     {% for article in articles|slice(0,3) %}
     {# On inclut la feuille twig dédiée aux vignettes d'articles #}
     {% include 'components/thumbnailArticle.twig' %}

     {% endfor %}

    </div>
    <button class="link">Voir plus</button>

{% endblock %}
```

La vue appelle elle-même d'autres vues grâce à «include » : cela peut permettre d'avoir une vue par composant et de créer ainsi des vues plus dynamiques, qu'on pourra réutiliser.

Ici, on appelle la vue qui nous donne à voir les vignettes d'articles.

```
    </div>
    <button class="link">Voir plus</button>

    <h1 class="titre">Articles populaires</h1>
    <div class="wrapperFlex">

  {# Ici, on fait une boucle, mais on ne retient que les trois premiers résultats de la boucle grâce au filtre "slice" #}
    {% for article in articles|slice(0,3) %}
    {# On inclut la feuille twig dédiée aux vignettes d'articles #}
    {% include 'components/thumbnailArticle.twig' %}

    {% endfor %}

    </div>
    <button class="link">Voir plus</button>

  {% endblock %}
```

Comme on a pu le deviner en regardant le MCD, l'entité article est notamment composée de la propriété galeries, qui est elle-même une collection d'instances d'une autre entité : galerie.

L'entité galerie est principalement définie par son chemin qui renvoie vers une image. Un article dispose donc d'une galerie d'images.

J'ai souhaité pouvoir représenter un article par une image de sa galerie, ou par un fond de couleur s'il n'en a pas.

Dans la vue ci-dessous, j'ai donc fait une condition pour le cas où un article est dépourvu d'images, et pour le cas où il contient des images.

S'il contient des images, je fais une boucle pour accéder à celles-ci, puis je fais une nouvelle condition qui me permet d'utiliser uniquement la première image de l'article.

```
<a href="{{ path('article_show', {id: article.id}) }}">
<figure class="article__picture">
{# Si la propriété galeries de article n'est pas vide #}
  {% if article.galeries is not empty %}
  {# Pour chaque galerie dans galeries #}
  {% for galerie in article.galeries %}
  {# Si l'index de la boucle est le premier #}
  {% if loop.first %}
    <img src="{{ asset('img/galerie/' ~ galerie.img) }}" alt="#" class="article__thumbnail">
  {% endif %}
  {% endfor %}
  {# Si la propriété galeries de article EST vide #}
  {% else %}
  <div class="article__thumbnail article__thumbnail--empty"><div>
  {% endif %}
  <figcaption class="article__title"> {{ article.titre }} </figcaption>
</figure>
</a>
```

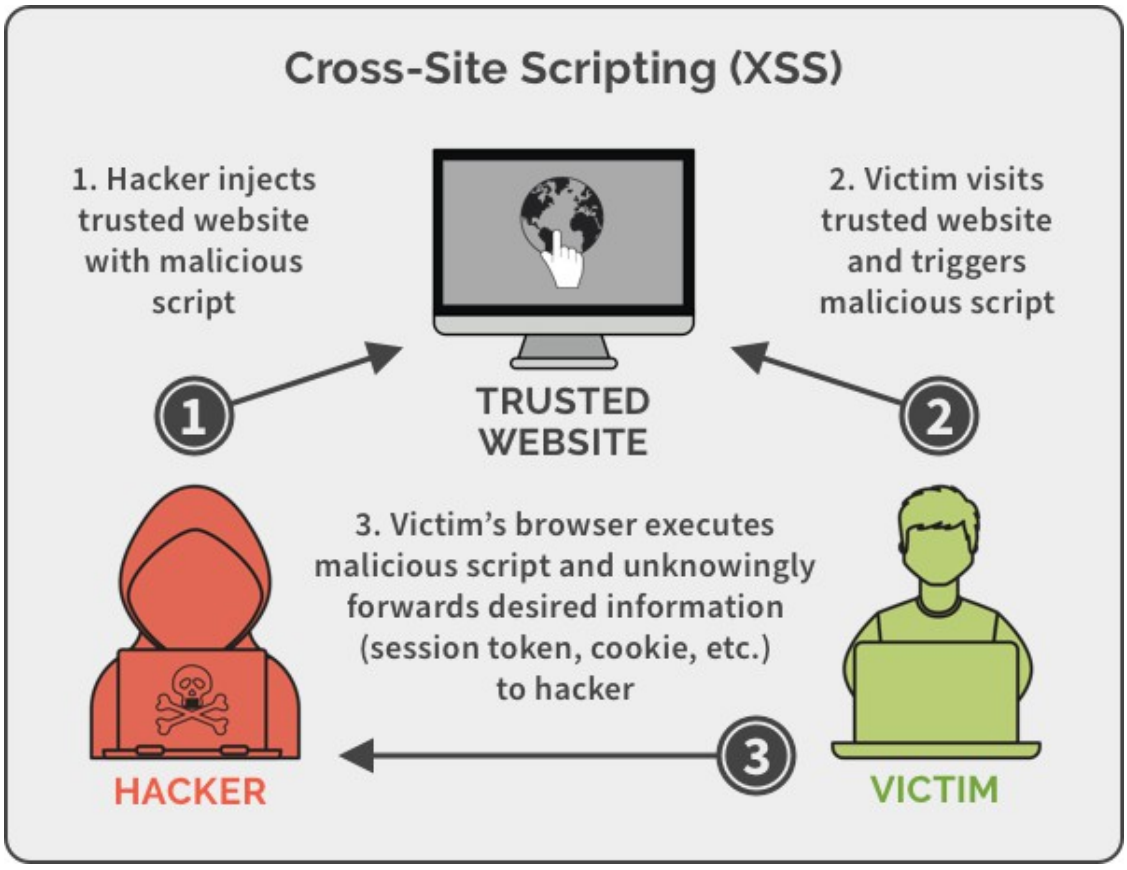
Voici le résultat affiché :

d) Sécurité

Je propose d'aborder les principales failles de sécurité, leur fonctionnement, comment on s'en prémunit, et comment ces failles sont gérées par Symfony.

- **La faille XSS**

La faille XSS consiste à l'injection de code malveillant à faire interpréter par le navigateur, par exemple via javascript. Il peut notamment avoir pour conséquence le vol de cookies.



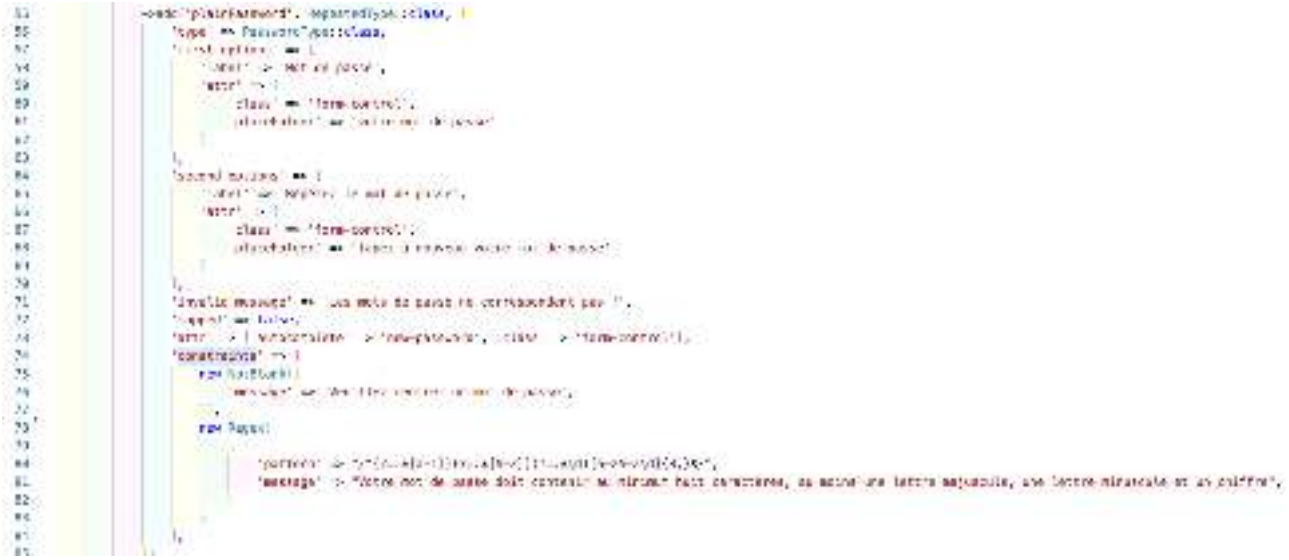
On s'en prémunit en filtrant les entrées et en échappant les sorties.

En PHP, on peut ainsi utiliser les fonctions **htmlspecialchars** ou **htmlentities** pour filtrer les données reçues, ou encore **filter_input**, qui permet de personnaliser le filtrage en fonction des paramètres donnés à la fonction.

Nous n'avons cependant pas besoin de le faire en Symfony à partir du moment où nous utilisons des formTypes. En effet, les formulaires construits avec Symfony filtrent les données de façon native, en fonction du type de données attendues et des contraintes apportées au formulaire.

Par exemple, à l'enregistrement d'un utilisateur, je vais indiquer la classe « RepeatedType » au champ « plainPassword », de type « PasswordType » : Symfony attend donc deux champs identiques de type mot de passe, et vérifie à la validation du formulaire que

les champs correspondent à ses attentes. J'ai par ailleurs ajouté une contrainte de type Regex au champ, ne sera donc validé qu'un mot de passe qui respecte l'expression régulière indiquée.



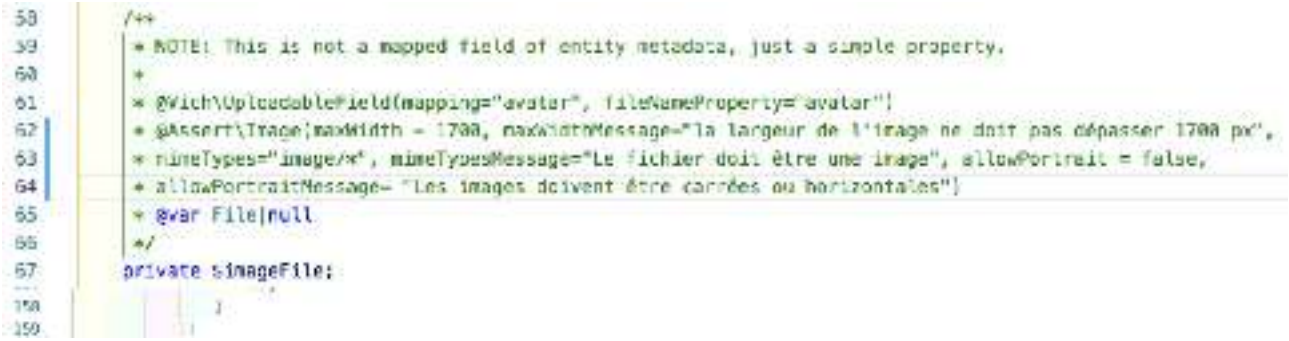
Pour prendre un autre exemple, le formulaire réclame aussi une date de naissance, et indique donc « DateType::class » au champ « date_naissance ». Une entrée qui ne correspondra pas à un format de date ne sera donc pas accepté. J'ai par ailleurs rajouté la contrainte « Range », qui permet d'indiquer une date minimale et une date maximale. Au-delà de ces limites, la date de naissance entrée ne sera pas validée.

Il est également possible de fixer des contraintes au niveau de l'entité au lieu du formulaire. Je l'ai fait par exemple pour le champ « imageFile » (qui en l'occurrence ne correspond pas à quelque-chose destiné à rentrer en base de données, mais qui sera uploadé sur le serveur).

Le même composant Symfony que pour les contraintes de formulaire est appelé :

« Symfony\Component\Validator\Constraints », dénommé ici « Assert ».

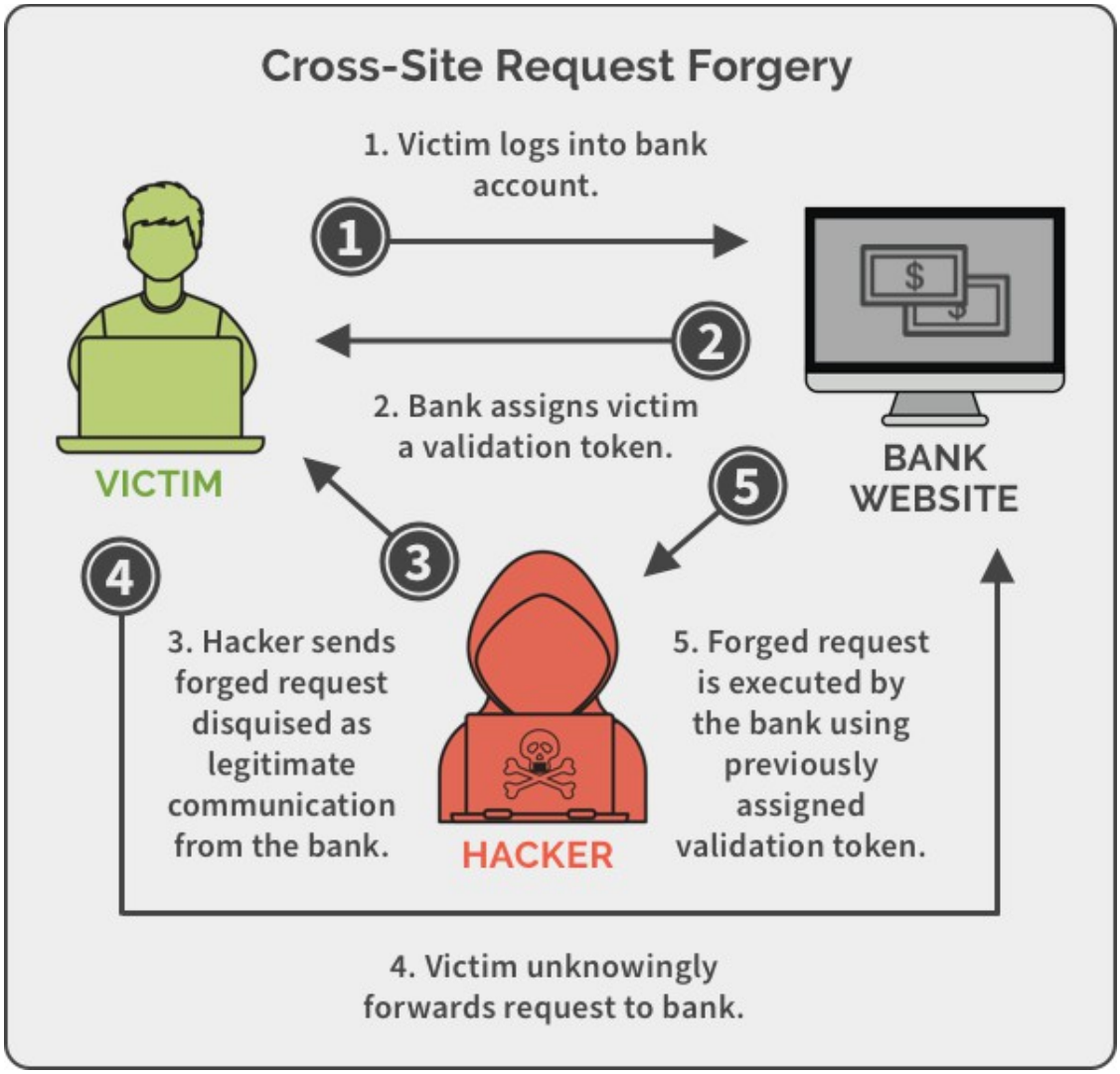
De la même manière qu'on pourrait le faire dans un formulaire, j'indique le type de fichier attendu (un fichier de type image), les dimensions minimales et maximales attendues, le type de dimensions, et les messages indiqués en cas de non-validation.



En ce qui concerne l'échappement des sorties, le gestionnaire de template Twig échappe par défaut les caractères : ils sont transformés en entités html, non-interprétables en tant que code. J'ai d'ailleurs dû utiliser un filtre spécial pour les articles publiés par un utilisa-

teur : en effet, l'éditeur de texte utilisé intègre des balises paragraphe, strong ou autres en fonction de l'usage de l'utilisateur. Par défaut, les balises sont donc affichées sur la page au lieu de prendre effet. Il faut dans ce contexte utiliser le filtre « raw » de Twig, qui n'échappe pas les caractères (ce qui réclame d'être d'autant plus vigilant sur la validation d'un article).

- **La faille CSRF**



La faille CSRF consiste à transmettre à un utilisateur authentifié une requête HTTP falsifiée qui pointe sur une action interne au site, afin qu'il l'exécute sans en avoir conscience et en utilisant ses propres droits. L'utilisateur devient donc complice d'une attaque sans même s'en rendre compte. L'attaque étant actionnée par l'utilisateur, un grand nombre de systèmes d'authentification sont contournés.

On s'en prémunit en utilisant un système de « **jetons** », soit de « **tokens** » en anglais.

C'est-à-dire qu'on transmet un jeton haché à l'utilisateur à l'ouverture d'une session, et qu'on attribue le même jeton haché aux requêtes délicates du site. Lorsque l'utilisateur

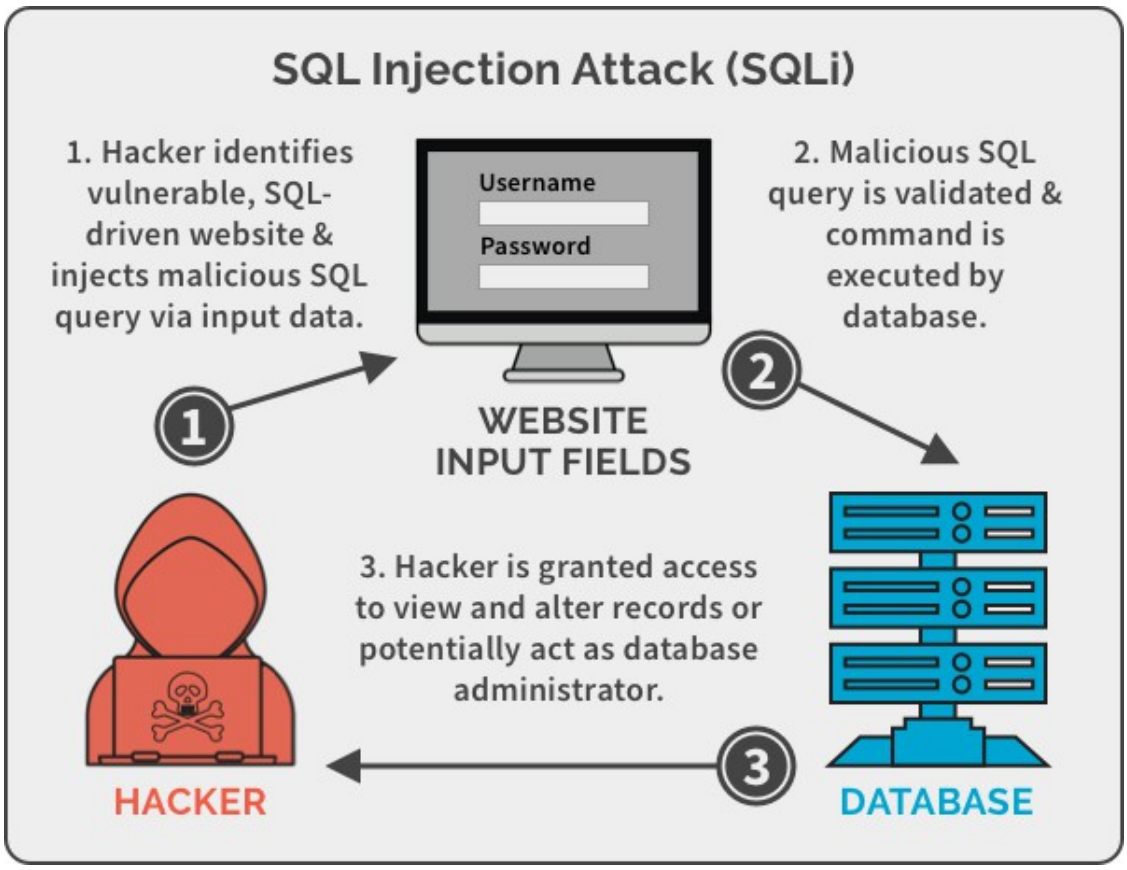
accomplit une action, on vérifie si la valeur du jeton haché de la requête correspond à celle de l'utilisateur (par exemple avec la fonction **hash_equals**). Si c'est le cas, c'est bien que la requête n'a pas été falsifiée.

Symfony gère la faille CSRF de façon native : encore une fois, ses formulaires sont très utiles, et disposent par défaut d'un champ hidden correspondant à des jetons d'authentification qui seront vérifiés à la soumission du formulaire.

Exemple sur le formulaire d'un commentaire d'article

- **L'injection SQL**

L'injection SQL consiste à, alors que le pirate remplit un formulaire destiné à produire une requête SQL, détourner la requête en écrivant quelque-chose dans le formulaire qui la modifiera.



Pour s'en prémunir, il faut ne faire appel qu'à des requêtes préparées : une compilation de la requête est réalisée avant d'y insérer les paramètres et de l'exécuter, ce qui empêche

un éventuel code inséré dans les paramètres d'être interprété.

Symfony le fait de manière automatique grâce à Doctrine. Comme nous l'avons vu précédemment, dans les repositories, Doctrine nous permet d'instancier des objets QueryBuilder, qui construisent des requêtes DQL en plusieurs étapes. La requête est compilée, paramétrée puis exécutée.

```
62      /**
63       * @return Article[] Returns an array of Article objects
64       */
65      public function findByLikes($user)
66      {
67          return $this->createQueryBuilder('a')
68              ->where(':user = a.auteurArticle')
69              ->innerJoin('a.likes', 'l')
70              ->setParameter('user', $user)
71              ->groupBy('l.post')
72              ->orderBy('COUNT(l.post)', 'DESC')
73              ->setMaxResults(3)
74              ->getQuery()
75              ->getResult();
76      }
```

IV/ Fonctionnalité phare : système de follow.

Dans mon application, je souhaitais qu'un utilisateur puisse suivre un autre utilisateur, permettant l'affichage des articles des personnes suivies sur son « fil ».

Cela signifie qu'un utilisateur doit avoir une collection de « followers » (les personnes qui le suivent) et une collection de « following » (les personnes que lui suit). Ces collections se reflètent comme propriétés dans l'entité User. Elles sont constituées d'objets qui sont eux-mêmes des instances de l'entité User. On a donc besoin d'une relation many-to-many auto-référencée, ce que Symfony ne peut pas fabriquer automatiquement via l'utilisation du terminal.

```
110     /**
111      * @ORM\ManyToMany(targetEntity=User::class, mappedBy="following")
112      */
113     private $followers;
114
115     /**
116      * @ORM\ManyToMany(targetEntity=User::class, inversedBy="followers")
117      * @ORM\JoinTable(name="following",
118      *      joinColumns={
119      *          @ORM\JoinColumn(name="user_id", referencedColumnName="id")
120      *      },
121      *      inverseJoinColumns={
122      *          @ORM\JoinColumn(name="following_user_id", referencedColumnName="id")
123      *      }
124      * )
125      */
126     private $following;
127
128     public function __construct()
129     {
130         $this->publications = new ArrayCollection();
131         $this->followers = new ArrayCollection();
132         $this->following = new ArrayCollection();
133     }
```

Nous avons donc besoin d'ajouter deux nouvelles instances d'ArrayCollection dans la fonction __construct de l'objet User, pour pouvoir ajouter, en plus des publications (les articles d'un « user »), les personnes suivies ou suiveuses.

Mais surtout, il faut personnaliser les annotations. Celle pour « followers » est une annotation ManyToMany encore classique, mais l'annotation de « following » sort du cadre habituel de manière à pouvoir faire fonctionner la chose.

L'annotation « ordonne » de créer une nouvelle table en base de données appelée « following », avec une colonne « user_id » et une colonne « following_user_id », se référant à la table « user ».

Avant de continuer, un petit rappel sur les relations entre entités dans Doctrine.

Une **association bidirectionnelle** entre entités se définit par une **association où chaque entité a une propriété se référant à l'autre**. Il y a pour cela un **« owning » side** (soit un côté « propriétaire ») et un **« inverse » side** (le côté inverse).

Le « owning » side est celui qui permet à Doctrine de détecter les changements, et le « inverse » side permet d'identifier l'objet concerné de l'autre côté. Cela remplace le système de clés étrangères qu'on trouve en base de données.

Dans les annotations, la propriété ayant l'attribut « **mappedBy** » est le **côté inverse**, tandis que la propriété ayant l'attribut « **InversedBy** » est le **côté propriétaire**. Les deux contiennent le nom de la propriété associée de l'autre côté.

On ne peut pas spontanément avoir une entité ayant une relation bidirectionnelle avec elle-même, il faut en quelque-chose tricher.

L'annotation de $following crée une table intermédiaire entre la table User et elle-même, permettant cette relation bidirectionnelle. La propriété « followers », qui a l'atttribut « mappedBy=''following'' », est donc le côté inverse de la relation, et la propriété « following » le côté « propriétaire » avec l'attribut « inversedBy=''followers'' » .

Lorsqu'on rédige l'annotation « inverseJoinColumns » dans la propriété following, la colonne se réfère donc au côté inverse de la relation.

La propriété « following » est donc le côté propriétaire et indique les personnes suivies par l'utilisateur concerné. En base de donnée, il correspond à la colonne « user_id » de la table « following ».

La propriété « followers » est le côté inverse et indique les utilisateurs qui suivent l'utilisateur concerné. La colonne « following_user_id » indique donc la personne suivie, et la colonne « user_id » indique la personne « suiveuse ».

Quant à l'attribut « referencedColumnName=''id'' », il permet d'indiquer à Doctrine que la colonne se réfère à celle de l'id dans la table User.

C'est cette complexité qui permet à un utilisateur d'en suivre un autre sans que ce soit mutuel. Un système d'amitié comme sur facebook aurait été plus simple, car il n'aurait pas nécessité de distinction entre suiveur et suivi.

Ici, si deux personnes se suivent mutuellement, ce seront donc deux lignes différentes qui seront inscrites en base de données, où les rôles seront inversés à chaque ligne.

Quant aux méthodes reliées à ces propriétés, nous avons getFollowers, addFollower et removeFollower, ainsi que getFollowing, addFollowing et removeFollowing.

```php
/**
 * @return Collection|self[]
 */
public function getFollowers(): Collection
{
    return $this->followers;
}

public function addFollower(self $follower): self
{
    if (!$this->followers->contains($follower)) {
        $this->followers[] = $follower;
    }

    return $this;
}

public function removeFollower(self $follower): self
{
    $this->followers->removeElement($follower);

    return $this;
}
```

```php
/**
 * @return Collection|self[]
 */
public function getFollowing(): Collection
{
    return $this->following;
}

public function addFollowing(self $following): self
{
    if (!$this->following->contains($following)) {
        $this->following[] = $following;
        $following->addFollower($this);
    }

    return $this;
}

public function removeFollowing(self $following): self
{
    if ($this->following->removeElement($following)) {
        $following->removeFollower($this);
    }

    return $this;
}
```

Les getters sont tout ce qu'il y a de plus classique.

La méthode removeFollower dicte que l'on enlève l'objet paramétré de la propriété followers de l'objet courant.

La méthode « removeFollowing » dicte que si l'on enlève l'objet paramétré de la propriété following de l'objet User courant, alors on appelle la méthode removeFollower, qui elle-même enlève l'objet courant de la propriété following de l'objet paramétré.

La méthode removeFollowing permet donc d'utiliser simultanément removeFollowing et removeFollower.

La méthode addFollower dicte que si l'objet paramétré n'est pas déjà contenu dans la propriété followers — qui est un tableau — de l'objet courant, alors il y est ajouté.

La méthode addFollowing dicte que si l'objet paramétré n'est pas contenu dans la propriété following, qui est un tableau, alors il y est ajouté, mais en plus, il appelle la méthode addFollower en paramétrant l'objet courant, sur l'objet paramétré.

Il suffit donc d'appeler la méthode addFollowing pour que le addFollower s'effectue également.

```
67      {% if user.followers.contains(app.user) %}
68        <a class="header__follow" href="{{ path('unfollowing', {id: user.id}) }}">Ne plus suivre ce membre</a>
69      {% else %}
70        <a class="header__follow" href="{{ path('following', {id: user.id}) }}">suivre ce membre</a>
71      {% endif %}
```

Sur la vue Twig du profil d'un utilisateur, une condition vérifie si la propriété followers de l'utilisateur consulté contient l'utilisateur connecté. Si c'est le cas, un bouton « ne plus suivre ce membre » est affiché, sinon un bouton « suivre ce membre ». Le premier appelle une fonction qui utilise la méthode removeFollowing en paramétrant l'utilisateur dont le profil était consulté, et en l'appliquant sur l'utilisateur connecté.

```
81      /**
82       * @Route("/unfollowing/{id}", name="unfollowing")
83       */
84      public function unfollow(User $user, EntityManagerInterface $manager)
85      {
86          $countries =  $this->getDoctrine()->getRepository(Pays::class)
87              ->findAll();
88          $themes =  $this->getDoctrine()->getRepository(Theme::class)
89              ->findAll();
90
91          $userFollowing = $this->getUser();
92
93          $userFollowing->removeFollowing($user);
94
95          $manager->flush();
96
97          return $this->render('pages/user/show.html.twig', [
98              'user' => $user,
99              'countries' => $countries,
100             'themes' => $themes
101         ]);
102     }
```

Le deuxième appelle la fonction addFollowing sur l'utilisateur connecté, en paramétrant l'utilisateur dont le profil était consulté.

```
58      /**
59       * @Route("/following/{id}", name="following")
60       */
61      public function follow(User $user, EntityManagerInterface $manager)
62      {
63          $countries =  $this->getDoctrine()->getRepository(Pays::class)
64              ->findAll();
65          $themes =  $this->getDoctrine()->getRepository(Theme::class)
66              ->findAll();
67
68          $userFollowing = $this->getUser();
69
70          $userFollowing->addFollowing($user);
71
72          $manager->flush();
73
74          return $this->render('pages/user/show.html.twig', [
75              'user' => $user,
76              'countries' => $countries,
77              'themes' => $themes
78          ]);
79      }
```

Mettons qu'un utilisateur appelé Jean-Michel ne suive pas encore un autre utilisateur, appelé Jérôme. Jean-Michel consulte le profil de Jérôme. Comme Jean-Michel n'est pas contenu dans le tableau de followers de Jérôme, un bouton « suivre ce membre » est affiché. Jérôme décide de cliquer dessus. La vue twig appelle alors la fonction « follow », avec en paramètre l'id de Jérôme.

La fonction « follow » identifie l'instance objet liée à l'id de Jérôme grâce au paramètre en tant que $user, et identifie Jean-Michel grâce à « $this->getUser() », qui permet d'identifier l'utilisateur connecté, et l'enregistre dans la variable $userFollowing. La fonction appelle ensuite la méthode addFollowing sur $userFollowing en paramétrant $user.

La fonction est donc appliquée sur l'objet User lié à Jean-Michel, en paramétrant l'objet User lié à Jérôme.

La méthode addFollowing permet ensuite de vérifier que Jérôme n'est pas déjà contenu dans le tableau « following » de Jean-Michel, c'est-à-dire que Jérôme ne suit pas déjà Jean-Michel.
Si la condition est respectée, Jérôme est ajouté au tableau en question, puis la méthode addFollower est appelée en paramétrant l'objet courant (c'est-à-dire Jean-Michel) sur l'objet paramétré dans la méthode addFollowing, c'est-à-dire Jérôme. Jean-Michel se trouve donc inscrit dans le tableau de followers de Jérôme.

Pour que la page d'accueil d'un utilisateur soit nourrie des articles de lui-même et des personnes qu'il suit, la fonction qui retourne la page d'accueil charge dans les données envoyées à la vue les articles trouvés par la méthode findByFollow du repository lié à l'entité Article.

```
32      // On rassemble les données nécessaires à la vue grâce aux repositories des entités User, Pays et Theme.
33      // C'est Doctrine, l'ORM de Symfony qui nous permet d'accéder aux repositories des différentes classes.
34      // On accède aux méthodes de chaque repository, ce qui nous permet d'utiliser la méthode findAll.
35      $users = $this->getDoctrine()->getRepository(User::class)
36          ->findAll();
37      $countries = $this->getDoctrine()->getRepository(Pays::class)
38          ->findAll();
39      $themes = $this->getDoctrine()->getRepository(Theme::class)
40          ->findAll();
41      $following = $user->getFollowing();
42
43      $articles = $this->getDoctrine()->getRepository(Article::class)->findByFollow($following, $user);
44
45
46      // On renvoie la vue twig avec les données instanciées dans la fonction.
47      return $this->render('pages/home/indexRFM.html.twig', [
48          'users' => $users,
49          'countries' => $countries,
50          'themes' => $themes,
51          'publications' => $articles
52      ]);
53  }
```

```
87      /**
88       * @return Article[] Returns an array of Article objects
89       */
90      public function findByFollow($following, $user)
91      {
92          return $this->createQueryBuilder('a')
93              ->where('a.statut = 1')
94              ->andWhere('a.auteurArticle IN (:following) OR a.auteurArticle IN (:user)')
95              ->setParameter('following', $following)
96              ->setParameter('user', $user)
97              ->orderBy('a.id', 'DESC')
98              ->getQuery()
99              ->getResult();
100     }
```

On paramètre la variable $following, qui contient le tableau $following de l'utilisateur connecté, c'est-à-dire les personnes que suit l'utilisateur, et on paramètre aussi l'utilisateur lui-même.

La méthode findByFollow retourne un tableau d'objets de la classe Article. Il cherche les articles dont le statut est 1 (c'est-à-dire les articles publiés, et non les brouillons), où l'auteur de l'article fait partie des personnes que suit l'utilisateur, ou bien, où l'auteur de l'article est l'utilisateur connecté.

V/ Consommation d'API

Pour mon projet, j'ai souhaité qu'une personne qui s'inscrit puisse indiquer et enregistrer sur une carte les voyages qu'elle a fait et les voyages qu'elle désire faire, et que ceux-ci apparaissent sur son profil.

Pour parvenir à ce résultat, j'ai dû consommer une API.

Une API est une interface de programmation permettant à deux applications logicielles de communiquer entre elles et d'échanger des données.

J'ai personnellement utilisé la « Maps JavaScript API » de Google pour cette fonctionnalité, et sa libraire « Places », qui m'a paru très bien convenir à ce que je voulais faire.

En effet, cette librairie permet de rechercher des lieux sur une carte, et offre une fonctionnalité d'autocomplétion des lieux disponibles.

Après avoir obtenu une clé API, j'ai chargé le script permettant de l'utiliser, en suivant la documentation.

Capture d'écran de la documentation fournie

L'URL appelé par le script a deux paramètres requis : la clé d'API, et le callback, où je spécifie le nom d'une fonction globale à appeler une fois que Maps Javascript API a fini de charger.
Je spécifie aussi en paramètre optionnel la librairie que je veux charger de l'API (Places).

Cela donne ça dans l'entête de ma page :

Je place ensuite les éléments dans le code dans lesquels je vais situer la carte et le formulaire de recherche avec autocomplete.

```
40    <div>
41      <h5>quels sont vos projets de voyage ?</h5>
42      <input id="autocomplete" placeholder="Indiquer un pays" type="text"/>
43    </div>
44    <div id="map" class="mapHidden"></div>
```

Dans la fonction appelée par le callback du script, on initialise un nouveau service Place Autocomplete et on l'attache à l'élément html #autocomplete.

On spécifie ensuite le type de lieux attendus par l'autocomplete et le type de champs qu'on souhaite obtenir attachés au lieu retourné.

```
let autocomplete;
function initAutocomplete() {
  autocomplete = new google.maps.places.Autocomplete(
    document.getElementById("autocomplete"),
    { types: ["(regions)"], fields: ["place_id", "geometry", "name"] }
  );
```

Par la suite, on ajoute un addListener à l'autocomplete qui va appeler un événement dénommé « place_changed » qui provient de l'API, et paramétrer la fonction à déclencher quand une prédiction de l'autocomplete est cliquée.

```
    autocomplete.addListener("place_changed", onPlaceChanged);
```

Dans la fonction en question, on appelle la méthode getPlace() du service autocomplete, ce qui retourne les détails du lieu.

```
function onPlaceChanged() {
  document.getElementById("map").classList.remove("mapHidden");
  place = autocomplete.getPlace();
```

Par la suite, à chaque fois que l'utilisateur valide le lieu sélectionné, je l'ajoute à une collection du formulaire en donnant les valeurs de son nom et de son place_id aux champs correspondants.

```
document.querySelectorAll(".add_item_link").forEach((btn) =>
  btn.addEventListener("click", function (e) {
    addFormToCollection(e);
    document.getElementById(
      "registration_form_projetsVoyages_" + index + "_name"
    ).value = place.name;
    document.getElementById(
      "registration_form_projetsVoyages_" + index + "_placeId"
    ).value = place.place_id;
    let node = document.createElement("li");
    let textnode = document.createTextNode(place.name);
    node.appendChild(textnode);
    document
      .getElementById("registration_form_projetsVoyages_" + index)
      .appendChild(node);
  })
);
```

Le place_id est un code attaché à un lieu, qui permet de retrouver ses coordonnées ou son adresse.

Toute cette opération est accomplie pour les voyages faits par un utilisateur, et répétée pour les voyages à venir par l'utilisateur.

A la validation du formulaire, dans le contrôleur, j'enregistre les lieux rentrés dans le formulaire, en donnant un booléen différent aux voyages selon qu'ils sont passés ou à venir.

```
39          if ($form->isSubmitted() && $form->isValid()) {
40              foreach ($form->get('projetsVoyages') as $item) {
41                  $place = new Place;
42                  $place->setName($item->get('name')->getViewData());
43                  $place->setPlaceId($item->get('placeId')->getViewData());
44                  $place->setStatut("0");
45                  $place->addUser($user);
46              }
47              foreach ($form->get('voyagesAccomplis') as $item) {
48                  $place = new Place;
49                  $place->setName($item->get('name')->getViewData());
50                  $place->setPlaceId($item->get('placeId')->getViewData());
51                  $place->setStatut("1");
52                  $place->addUser($user);
53              }
```

Lors de la consultation du profil d'un utilisateur, un autre script, presque identique mais avec un autre callback, appelle cette fois la fonction « initMap », qui va charger une carte dans l'élément #mapProfile .

```
10  function initMap() {
11    // The map, centered at Uluru
12    const mapProfile = new google.maps.Map(
13      document.getElementById("mapProfile"),
14      {
15        zoom: 2,
16        center: { lat: 23.8862915, lng: 0 },
17      }
18    );
```

Dans une autre fonction, j'identifie et récupère l'id du user grâce à l'URL de la page et j'appelle la fonction « searchPlaceId » du contrôleur en ajax.

```
let markersPlaces = [];
let markersNames = [];
let markersStatuts = [];

$(document).ready(function () {
    let adress = document.URL;
    let userId = adress.slice(27, adress.length);
    let url = "/searchPlaceId/" + userId;
    $.ajax({
        url: url,
        type: "POST",
    }).done(function (response) {
        markersPlaces.push(response.placesId);
        markersNames.push(response.placesNames);
        markersStatuts.push(response.placesStatut);
        var i = 0;
```

Dans la fonction du contrôleur, grâce au repository de l'entité Place - et en sachant qu'il y a une relation Many-to-Many entre l'entité Place et l'entité User - je cherche les objets Place dont le user paramétré faire partie de la propriété « users » de l'objet.

Je renvoie dans une réponse Json les PlaceId, noms et statuts des lieux récupérés, que je récupère dans ma fonction ajax

.

```
/**
 * @return Place[] Returns an array of Place objects
 */
public function findByUser($user)
{
    return $this->createQueryBuilder('p')
        ->where(':user MEMBER OF p.users')
        ->setParameter('user', $user)
        ->orderBy('p.id', 'ASC')
        ->getQuery()
        ->getResult();
}
```

J'appelle ensuite un service de l'API me permettant d'identifier les détails d'un lieu donné.

Pour chaque lieu transmis par la réponse ajax, j'indique le placeId, puis je définis une icône différente en fonction du statut récupéré (voyage accompli ou à venir). Enfin, je définis un marqueur auquel j'attribue l'icône, la carte et la location, retrouvée grâce au placeId et au résultat de la méthode getDetails.

```
46      var service = new google.maps.places.PlacesService(mapProfile);
47      markersPlaces[0].forEach((element) =>
48        service.getDetails(
49          {
50            placeId: element,
51          },
52          function (result, status) {
53            if (markersStatuts[0][i] == true) {
54              var icon = "/img/redPin.png";
55            } else {
56              var icon = "/img/bluePin.png";
57            }
58            i++;
59            var marker = new google.maps.Marker({
60              icon: icon,
61              map: mapProfile,
62              place: {
63                placeId: element,
64                location: result.geometry.location,
65              },
66            });
67          }
68        )
69      );
```

VI/ Situation de travail ayant nécessité une recherche sur un site anglophone

De nombreuses fois, je suis tombé sur un obstacle au cours de mon projet, à chaque fois j'ai fini par trouver une solution à mon problème, notamment grâce au site Stackoverflow.

Concernant le cas de figure que je vais vous présenter, plusieurs problèmes à la fois se sont posés, dont j'ai trouvé la résolution sur différents sites anglophones.

J'avais l'habitude que mes formulaires se trouvent dans une page Twig précise, qu'ils soient uniques, que leur vue soit créée dans le contrôleur, et qu'ils soient soumis et validés dans la même fonction que celle d'où elle avait été envoyée.

Mais un système de commentaires posait plusieurs défis : leurs formulaires doivent apparaître plusieurs fois dans une même page, existent sur de nombreuses pages du site, retournées par différentes fonctions du contrôleur.

Je me suis donc posé ces questions :

- Comment traiter le formulaire depuis une seule fonction du contrôleur, alors que les pages sont retournées depuis des fonctions différentes (page d'accueil, détail d'une catégorie et/ou d'un pays, détail d'un article…) ?
- Comment faire pour renvoyer sur la page d'où l'on est venu après avoir soumis son commentaire ?
- Comment afficher plusieurs fois le même formulaire dans la vue Twig ?

Concernant la première question, c'est la documentation de Symfony qui m'a fourni la réponse, et m'a permis de me rendre compte que je pouvais outrepasser l'action du formulaire (par défaut la fonction du contrôleur par laquelle le formulaire a été rendu) en précisant dans le template l'action préférée.


Finally, you can override the action and method in the template by passing them to the `form()` or the `form_start()` helper functions:

```
{# templates/task/new.html.twig #}
{{ form_start(form, {'action': path('target_route'), 'method': 'GET'}) }}
```


templates > components > commentaires.twig

```
{% if commentaire is defined %}
  {% set form = commentaire.createView %}
  <span class="btnPublierComm btn btn-info">Commenter</span>
  {% if publication.commentaires|length > 0 %}
    <span class="btnCommentaires btn btn-info">Voir les commentaires ({{publication.commentaires|length}})</span>
  {% else %}
    <span class="btnCommentaires"></span>
  {% endif %}
  <div class="hidden">
    {{ form(form, {'action': path('addComment', {id: publication.id})}) }}
  </div>
{% endif %}
```

Concernant la seconde question, après renseignement sur un site anglophone inconnu (un site bizarrement dédié au langage python, alors que la question concernait symfony), je me suis rendu compte que la requête détenait l'information de sa propre provenance, et comment l'identifier.

Answer #1:

This is an alternative version of Naitsirch and Santi their code. I realized a trait would be perfect for this functionality. Also optimized the code somewhat. I preferred to give back all the parameters including the slugs, because you might need those when generating the route.

This code runs on PHP 5.4.0 and up. You can use the trait for multiple controllers of course. If you put the trait in a seperate file make sure you name it the same as the trait, following PSR-0.

```php
<?php
trait Referer {
    private function getRefererParams() {
        $request = $this->getRequest();
        $referer = $request->headers->get('referer');
        $baseUrl = $request->getBaseUrl();
        $lastPath = substr($referer, strpos($referer, $baseUrl) + strlen($baseUrl
        return $this->get('router')->getMatcher()->match($lastPath);
    }
}
```

```php
    /**
     * @Route("/article/addComment/{id}/", name="addComment")
     */
    public function addComment(Article $article, Request $request)
    {
        // dd($request);
        $user = $this->getUser();

        $form = $this->createForm(CommentaireType::class);
        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid()) {

            $commentaire = $form->getData();

            $commentaire->setAuteur($user);
            $commentaire->setArticle($article);
            $commentaire->setDateTime(new DateTime);
            $commentaire->setNewComment(1);


            $entityManager = $this->getDoctrine()->getManager();
            $entityManager->persist($commentaire);
            $entityManager->flush();

            $referer = $request->headers->get('referer');

            return new RedirectResponse($referer);
        }
    }
```

Concernant la troisième question, je dois d'abord expliquer qu'initialement, quand j'ai simplement cherché à créer la vue du formulaire dans une fonction et à l'afficher dans une vue Twig, mon formulaire ne s'affichait qu'une seule fois, malgré le fait qu'il se trouve à l'intérieur d'une boucle.

Après une recherche Google, je me suis aperçu que c'était un problème courant dans mon cas de figure. C'est stackoverflow qui m'a fourni une solution.

15

I came across this and another question about the similar issue. You can find my first answer for a solution here.

To wrap it up, I did not call the `createView()` function on the form in the controller, as usually done when passing the form to the view, but in the twig view itself.

E.g. in your controller action you do return the form object itself:

```
return $this->render('AppBundle:Cart:list.html.twig', ['formObject' => $form];
```

and in your view you would set the form in each loop:

```
{% for cart in carts %}
    {# Some template stuff #}
    {% set form = formObject.createView %}
    {{ form_start(form) }}
        <div class="form-input">
        <label for="country" class="middle-color">Country <span class="active-color
        {{ form_widget(form.country) }}
    {{ form_end(form) }}
{% endfor %}
```

Je me suis rendu compte que, dans une fonction du contrôleur où je créais un formulaire de commentaire, je ne devais pas retourner dans la vue Twig le createView() du formulaire, mais le définir dans la vue twig elle-même à l'intérieur de la boucle.

templates > components > commentaires.twig

```
1  {% if commentaire is defined %}
2    {% set form = commentaire.createView %}
3  <span class="btnPublierComm btn btn-info">Commenter</span>
4    {% if publication.commentaires|length > 0 %}
5      <span class="btnCommentaires btn btn-info">Voir les commentaires ({{publication.commentaires|length}})</span>
6    {% else %}
7      <span class="btnCommentaires"></span>
8    {% endif %}
9    <div class="hidden">
10     {{ form(form, {'action': path('addComment', {id: publication.id})}) }}
11   </div>
12   {% endif %}
```

Après la résolution de ces trois problèmes, la mise en place de mon système de commentaires s'est faite sans autres accrocs particuliers.

VII/ Traduction

Pour cette traduction, je propose de traduire un extrait d'un article de la documentation officielle de Doctrine m'ayant permis de mieux comprendre comment mettre en place mon système de follow.

Version d'origine en anglais :

Association Mapping
This chapter explains mapping associations between objects.

Instead of working with foreign keys in your code, you will always work with references to objects instead and Doctrine will convert those references to foreign keys internally.

- A reference to a single object is represented by a foreign key.
- A collection of objects is represented by many foreign keys pointing to the object holding the collection

This chapter is split into three different sections.

- A list of all the possible association mapping use-cases is given.
- Association Mapping are explained that simplify the use-case examples.
- Association Mapping are introduced that contain entities in associations.

One tip for working with relations is to read the relation from left to right, where the left word refers to the current Entity. For example:

- OneToMany - One instance of the current Entity has Many instances (references) to the referred Entity.
- ManyToOne - Many instances of the current Entity refer to One instance of the referred Entity.
- OneToOne - One instance of the current Entity refers to One instance of the referred Entity.

See below for all the possible relations.

An association is considered to be unidirectional if only one side of the association has a property referring to the other side.

To gain a full understanding of associations you should also read about owning and inverse sides of associations

Version traduite par mes soins :

Mappage d'associations

Ce chapitre explique le mappage d'associations entre des objets.

Au lieu de travailler avec des clés étrangères dans son code, on va travailler plutôt avec des références à des objets, et Doctrine va convertir en interne ces références en clés étrangères.

- Une référence à un objet unique est représenté par une clé étrangère
- Une collection d'objets est représentée par plusieurs clés étrangères pointant vers l'objet détenant la collection.

Ce chapitre est découpé en trois différentes sections.

- Une liste de toutes les possibles cas d'utilisation de mappages d'associations est donnée.
- Le mappage d'associations est expliqué en simplifiant les exemples de cas d'utilisation.
- Les mappages d'associations qui contiennent des entités dans les associations sont introduits.

Une recommandation pour travailler avec les relations est de lire les relations de gauche à droite, le mot de gauche se référant à l'entité courante. Par exemple :

- UnePourPlusieurs – Une instance de l'Entité courante a plusieurs instances (références) à l'Entité référencée.
- PlusieursPourUne – Plusieurs instances de l'Entité courante se réfèrent à une instance de l'Entité référencée.
- UnePourUne – Une instance de l'Entité courante réfère à une instance de l'Entité référencée.

Regardez ci-dessous pour toutes les relations possibles.

Une association est considérée comme unidirectionnelle si un seul côté de l'association a une propriété se référant à l'autre côté.

Pour comprendre entièrement les associations vous devriez lire aussi ce chapitre sur les côtés propriétaire et inverse des associations.

VIII/ Axes d'améliorations

A ce stade du projet, bien que je sois plutôt fier du travail accompli dans le temps imparti et avec le bagage technique que j'ai pour l'instant, il y a tout de même encore beaucoup d'éléments auxquels j'ai renoncé pour le moment à la fois par manque de temps et parce qu'elles n'ont pas été priorisées :

- un système de signalement pour faciliter le travail de modération par l'administrateur.
- un système de blocage, de manière à ce qu'une personne puisse en empêcher une autre d'entrer en contact avec elle ou de voir ce qu'elle partage sur le site.
- Un système de pagination
- Utiliser l'API d'instagram pour partager les derniers posts instagram d'un membre du site
- une gestion des cookies
- pouvoir se connecter via Facebook ou Google
- avoir une option « remember me »
- donner accès à une traduction automatique du contenu
- compresser les images fournies au site plutôt que de refuser les images au-delà d'un certain nombre de pixels en largeur.
- Systématiser l'usage d'ajax sur mon site (de manière à avoir une messagerie instantanée par exemple)

IX/ Bibliographie

Voici les sites qui m'ont été les plus utiles durant le développement de ce projet :

- https://stackoverflow.com/
- https://symfony.com/doc/current/index.html
- https://twig.symfony.com/doc/3.x/
- https://www.doctrine-project.org/projects/doctrine-orm/en/2.9/index.html
- https://developers.google.com/maps/documentation/javascript/overview?hl=fr
- https://github.com/dustin10/VichUploaderBundle/blob/master/docs/index.md
- https://symfony.com/bundles/FOSCKEditorBundle/current/index.html