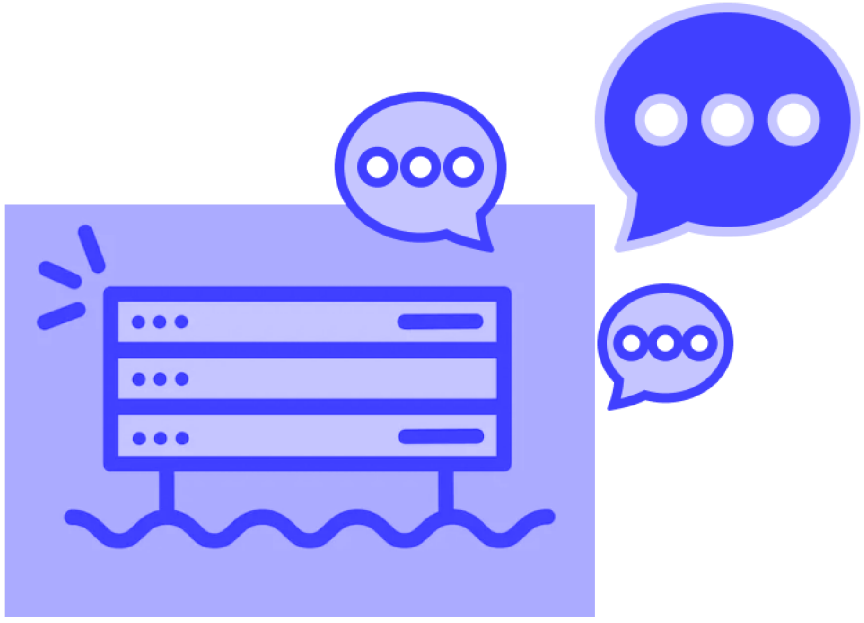
La Plateforme_

N E T W O R K



Titre de développeur web / web mobile

Dossier de projet professionnel

Robin ARBONA

Sommaire

1. Liste des compétences du référentiel couvertes par le projet

Ci-dessous, sont présentées les compétences visées au titre du projet professionnel:

- Activité type 1 « Développer la partie front-end d'une application web ou web mobile en intégrant les recommandations de sécurité »
    - Maquetter une application
    - Réaliser une interface utilisateur avec une solution de gestion de contenu ou e-commerce

- Activité type 2 « Développer la partie back-end d'une application web ou web mobile en intégrant les recommandations de sécurité »
    - - Développer les composants d'accès aux données
    - - Elaborer et mettre en œuvre des composants dans une application de gestion de contenu ou e-commerce

2. Résumé du projet

Le projet “La Plateforme_ network” est un projet de réseau social permettant aux membres de l'école la Plateforme_, les Plateformeurs, de garder contact, de partager leurs expériences et tout simplement d’échanger sur leurs vies étudiantes.

Un réseau social est un site internet qui permet aux internautes de se créer une page personnelle afin de partager et d'échanger des informations, du contenu avec leur communauté d'amis et leur réseau de connaissances.

Il doit y être possible de communiquer autant instantanément que de façon plus durable. En effet, ce réseau doit répondre à plusieurs besoins principaux:

- échanger en temps réel sur des problématiques quotidiennes,
- partager les expériences de chacun,
- partager des informations pouvant être utiles à la communauté,
- recenser les étudiants au travers d’un annuaire pour les mettre plus facilement en relation.

Dans notre cas, ce réseau social est interne à l’organisation la Plateforme_. Il ne sera accessible qu’aux Plateformeurs, au staff et éventuellement aux alumnis. Toutes ces personnes ont une adresse mail terminant par le nom de domaine laplateforme.io.

Il est évident que comme dans tout réseau social, des débordements sont à prévoir. Une administration du contenu par les membres du staff doit être possible.

Le projet la Plateforme_ network s’inscrit dans le cadre de mes études à la Coding School de la Plateforme_. J’ai travaillé en collaboration avec Monsieur Pierre MALARDIER et Monsieur Samy BENRABAH durant les mois de mai et juin 2021 sur ce projet. Le code présenté dans le chapitre 10: Réalisations correspond aux fonctionnalités dont j’avais la charge et que j’ai donc développé.

3. Cahier des charges

Le réseau social de La Plateforme_ doit permettre:

- Que chaque Plateformeur puisse créer son profil à l'aide de son adresse email @laplateforme.io.

- Les informations de base seront demandées (nom, prénom, photo de profil, date de naissance, hobbies...), et éventuellement d'autres informations telles que les expériences, entreprises, technologies maîtrisées, photo de couverture...

- Chaque profil aura son propre mur et un fil d'actualité présent sur la page d'accueil devra résumer l'actualité des autres Plateformeurs. Il devrait être possible de réagir aux posts à l'aide de commentaires et de réactions (likes, émoticônes...).

- Il sera possible de créer des conversations et de discuter avec des membres du réseau sans rechargement de la page.

- La charte graphique à utiliser est celle de la Plateforme, l'interface devra être fortement inspirée de celle de l'intranet étudiant afin d'amener de la cohérence dans les outils utilisés en interne.

4. Spécifications techniques du projet

Dans la suite du document:

- App, fait référence à l'ensemble de l'application de réseau social,
- App front à la partie front end de l'application de réseau social, visible par le client,
- App back-office, à une interface d'administration, visible uniquement de certain utilisateur,
- App back, à la partie back-end de l'application non visible par les utilisateurs,
- Chat, à l'ensemble de l'application de chat,
- Chat client, à la partie front de l'application de chat,
- Chat serveur, à la partie back de l'application de chat.

1. Liste des fonctionnalités et spécifications

Les fonctionnalités et spécifications sont regroupées selon les ensembles présentés ci-dessus. Les spécifications de sécurité sont notées en rouge italique :

- App
    - Connexion avec Google
        - *L'accès à l'application n'est possible qu'aux utilisateurs ayant un compte mail @laplateforme.io*
    - Un mur pour chaque utilisateur (il comporte les posts et données de l'utilisateurs)
        - *Le système de navigation ne devra pas permettre d'attaque de type LFI/RFI (Local/Remote File Inclusion)*
    - Un mur global (il synthétise les murs de l'ensemble des utilisateurs du réseaux)
    - Rédaction de post
        - *Le système devra prévenir les attaques de types SQLi ('injection SQL) et XSS (Cross-Site Scripting)*
    - Renseignement de données personnelles et partage avec le réseau
        - *Protection SQLi / XSS*
    - Rédaction de commentaire
        - *Protection SQLi / XSS*
    - Réagir à un post ou un commentaire (like)
    - Annuaire des Plateformeurs utilisant le réseau social
- App front
    - Affichage dynamique du contenu stocké en base de données
        - Posts / Commentaires / Likes / Utilisateurs
            - *Protection XSS*
    - Consultation des posts composants le mur sans rechargement de page
        - Scroll infini
    - Site responsive
        - Le site est accessible et utilisable pour les utilisateurs Desktop / tablette / mobile

- App back-office
    - Gestion des posts / commentaires / like (Suppression / modification)
        - *Mise en application du principe de moindre privilège*
    - Gestion des utilisateurs (bannissement)
        - *Mise en application du principe de moindre privilège*
- App back
    - Le contenu sera sauvegardé en base de données
    - API permettant de délivrer le contenu sous forme de fichier JSON
        - *Le contenu de sera délivré qu'à des personnes préalablement authentifié*
- Chat
    - Connexion au chat utilisant l'identification réalisée au travers de la connexion avec google
    - Communication avec un groupe de personne
    - Communication privée avec un utilisateur
- Chat client
    - Le contenu sera affiché dynamiquement sans rechargement de page
        - *Protection XSS*
- Chat server
    - Le contenu ne sera pas sauvegardé par le serveur
    - Liste des utilisateurs connectés à jour, sans rechargement de page

2. Choix technologiques associés

Au regard des spécifications ci-dessus, les choix technologiques suivants ont été réalisés. Ces choix sont également organisés par ensemble tel que décrit dans l'introduction du chapitre Spécifications techniques du projet.

- App
    - Front (Langages utilisés: HTML / CSS / Javascript)
        - Google OAuth (Authentification avec google)
        - Twig (Moteur de template pour php)
        - Bulma (Framework CSS)
    - Back (Langages utilisés: PhP / SQL)
        - Google APi client (Authentification avec google, vérification authenticité du token)
        - Slim (Micro framework PhP facilitant la mise en place d'API et application web: rooting, gestion des requêtes / réponses ..)
        - PhP DI (Conteneur pour injection de dépendance)
        - MySQL (Base de données de l'application)
- Chat (Langage utilisés: Javascript / HTML / CSS)
    - Websocket (communication client <--> serveur)
    - Client (javascript)
        - Socket.io (Facilite la mise en oeuvre d'une connection websocket, permet une communication en temps réels, bi directionnelle et basée sur l'occurrence d'événement)
    - Serveur (javascript node)

- Socket.io
- google auth library (Authentification avec google, vérification authenticité du token)

3. Architecture

L'App utilise le langage PHP avec un paradigme de programmation orienté objet.

L'App est organisée selon une architecture de type DDD (Domain-Driven-Design). Comme en MVC (Modèle View Controller), la logique applicative est séparée de la gestion des données et des controllers.

Les routes invoquent des SAC (Single Action Controller).

Ces contrôleurs font appelle à des services organisés eux-mêmes par domaine.

Les dépôts sont également regroupés en fonction de leurs domaines.

Voici l'architecture de dossier de l'App:

```
App
├── config ← Fichiers de configuration (Info. de co. à la bdd, Google clientId, routes ...)
├── public ← Contient l'index (point d'entré de l'App) et assets accessibles à l'utilisateur
│   ├── css
│   ├── image
│   └── js
├── src
│   ├── Action ← SACs (gère la requête / réponse HTTP et appel les services adéquates)
│   ├── Domain
│   │   ├── Comment
│   │   │   ├── Repository ← Repository associé / Requête SQL
│   │   │   └── Service ← Business logique organisée par service
│   │   .
│   │   .
│   │   └── User
│   │       ├── Repository
│   │       └── Service
│   ├── Exception ← Classe pour les modèles d'exception spécifiques
│   └── Page ← Single Action Controller responsable du chargement des pages
├── templates ← Fichiers twig (template / page / composant)
└── vendor ← Dossiers contenant les bibliothèques tierces (Twig, Slim ..)
```

Cette approche a pour objectif de simplifier la lisibilité du code en organisant l'application selon la logique métier et non la structure de donnée.

L’objectif est également d’avoir des classes spécialisées, de plus petites tailles et donc plus maintenables et testables.

Cette approche permet d’avoir du code lisible, qui exprime littéralement les concepts, domaines et processus de l'activité.

5. Réalisations

1. Maquettage

Une fois les spécifications techniques écrites, les fonctionnalités à implémenter sont listées (Trello) et le travail de maquettage peut commencer.

Wireframe

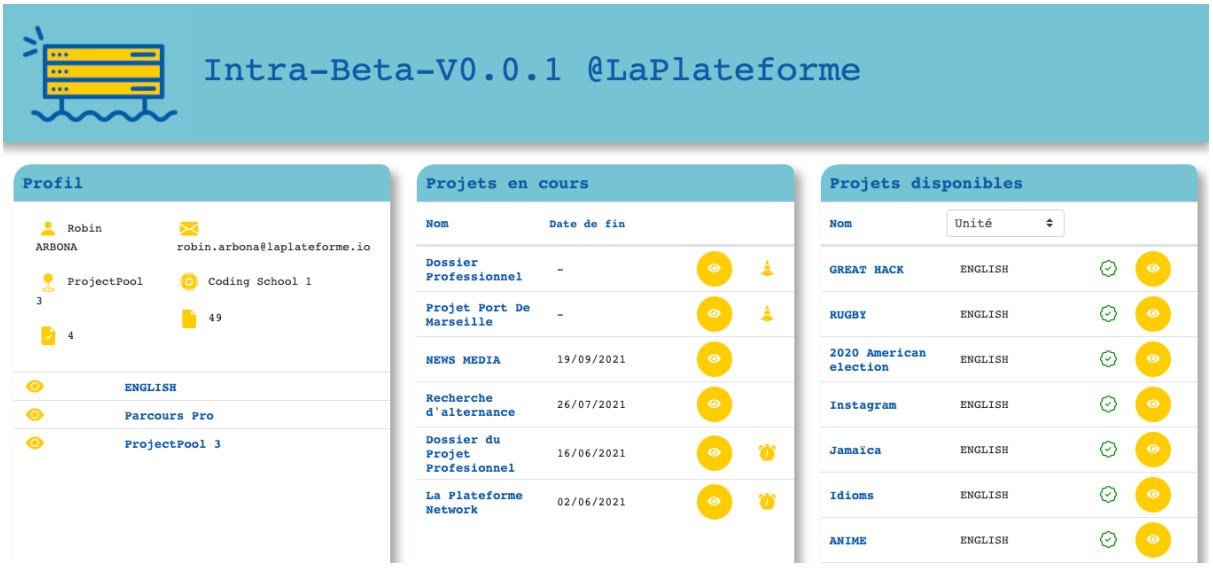
Avant de passer au maquettage, un wireframe des pages principales a été réalisé.

Les annexes Annexe : Wireframe page Connexion et Annexe : Wireframe page Wall] présentent les wireframes de la page de connexion, et de la page wall.
La page Wall correspond à la page d'accueil directement visible après que l'utilisateur se soit connecté.

Les wireframes ont été réalisés grâce à Figma. Ces wireframes sont une ébauche rapide de l'application. Ils permettent de faire des essais sur la disposition du contenu sans se préoccuper de la partie graphique.

Maquette

Afin de respecter une certaine cohérence graphique dans les outils utilisés par la Plateforme_, la Plateforme_ network utilisera la charte graphique de la Plateforme_ et s'inspirera également fortement de l'intranet accessible aux étudiants visible sur la capture d'écran ci-après.



[Capture d'écran de l'intranet étudiant]

Les annexes Annexe : Maquette page Connexion et Annexe : Maquette page Wall présentent le travail de maquettage réalisé sur les pages de connexion et wall.

Il n'a pas été fait ici de prototype qui permet de matérialiser les interactions entre les différentes pages.

2. Interface utilisateur intégrant une solution de gestion de contenu

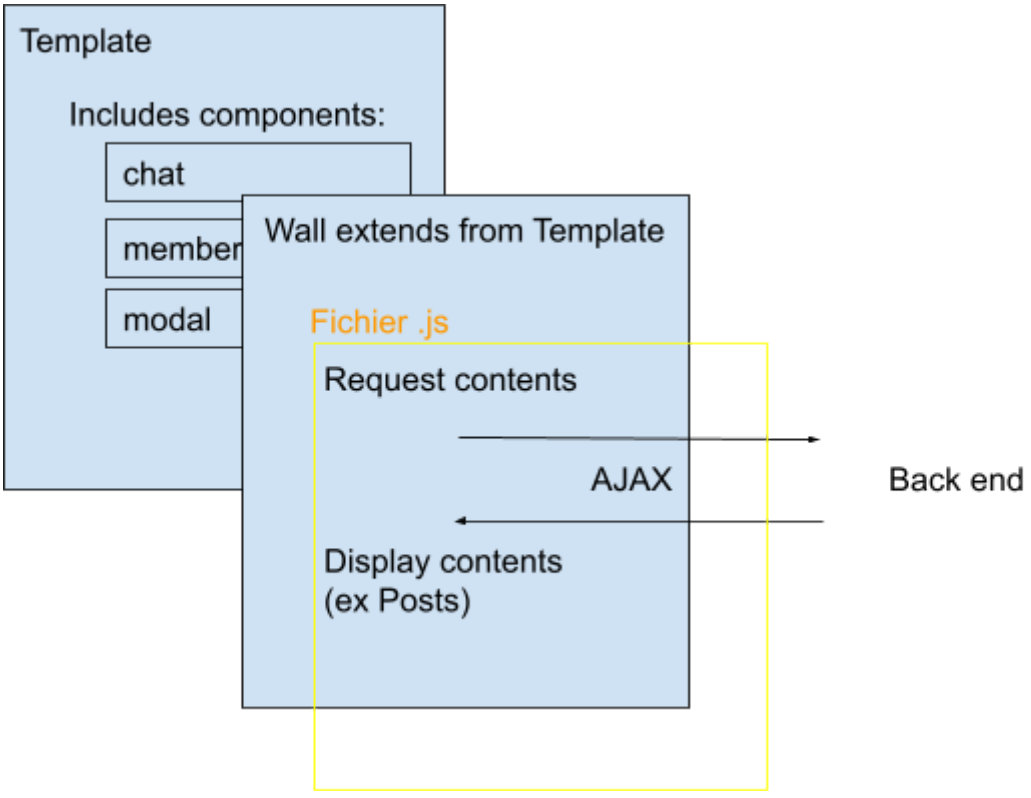
Utilisation de TWIG

L'interface utilisateur est composée de vues écrites en TWIG (moteur de template pour PHP).

Les vues principales héritent d'un template contenant les éléments présents sur l'ensemble des pages (par exemple head / header / footer / meta / script ...).

Les vues intègrent ensuite les éléments qui leurs sont spécifiques. Le contenu est quant à lui chargé dynamiquement à l'aide de requêtes AJAX.

Des composants peuvent également être chargés séparément dans des modals comme par exemple le formulaire de création de poste ou intégrés directement dans un template chat, annulaires utilisateurs (members)



[Graphique organisation fichiers TWIG]

Le graphique précédent présente l'organisation des fichiers TWIG (template / page / component). Le rôle du fichier javascript est détaillé dans le paragraphe suivant.

Exemple de l'affichage des posts

Afin d'illustrer le fonctionnement de l'interface utilisateur, nous allons prendre en file rouge l'affichage de posts. La partie back end sera détaillée dans le chapitre Back end.

Supposons que l'utilisateur demande la page wall, le contrôleur associé à la route /wall va charger le fichier twig et l'intégrer à la réponse HTTP envoyé à l'utilisateur.

L'extrait de code ci-après correspond à un extrait du fichier "page.wall.twig":

```
{% extends 'template.twig'%}

{% block title %}
   {{ parent() }}
   Wall
{% endblock %}

{% block content %}
<div class="post">
</div>
<button id="loadContent" class="button is-primary is-light">Load more</button>
{% endblock %}

{% block script %}
   {{ parent() }}
   <script src="{{pathPublic}}/js/wall.js" async defer></script>
{% endblock %}
```

[Extrait de code: page.wall.php]

Ce fichier hérite du fichier template.twig, cela signifie qu'il reprendra en totalité le code écrit dans ce fichier, sauf si le code d'un "block" est réécrit:

- Cas du block title: le block title présent dans le fichier template.twig est conservé et complété par le code suivant (ici le text "Wall")
- Cas du block content: le block content présent dans le fichier template.twig n'est pas conservé au profit de celui définit dans le présent fichier (page.wall.php)

Une fois le fichier servi à l'utilisateur, le navigateur va télécharger et exécuter le fichier /js/wall.js.

La variable pathPublic est définie coté PhP, elle contient le chemin d'accès absolu au dossier /public du serveur, cela est nécessaire lorsque le dossier /public n'est pas la racine du serveur web.

Le fichier wall.js contient notamment une classe nommée PageLoader. Celle-ci permet de charger du contenu (scroll infini), de le mettre en forme et de l'afficher sans rechargement de la page.

Pour cela, elle contient ou fait appel à des méthodes/attributs permettant:

- de charger du contenu en faisant un appel AJAX à la route spécifiée lors de l'instanciation de l'objet à partie de la classe
- de mettre en place un observateur d'intersection sur un bouton: quand celui-ci apparaît dans la zone d'affichage, du contenu supplémentaire est chargé
- la mise en forme du contenu et son ajout à la page
- la gestion de l'index du contenu (contenu actuellement affiché par rapport au contenu disponible)

L'extrait de code suivant correspond à la méthode loadContent, appelée lorsque le bouton observé apparaît dans la zone d'affichage:

```
async loadContent(){
      if( this.currentPage <= this.totalPage ){
          let results = await this.fetchJson( this.getUrl() );
          this.totalPage = results.totalPage;
          this.displayResults( results.posts );
          this.currentPage = this.currentPage + 1;
          this.setTargetMsg('Loaded');
      } else {
          this.setTargetMsg('Nothing more to load');
          this.observer.disconnect();
      }
  }
```

[Extrait de code wall.js, Class PageLoaded]

A chaque appel de la fonction, il est vérifié si du contenu supplémentaire reste à charger. currentPage et totalPage correspondent respectivement à l'index de la prochaine page de résultat à charger et au nombre total de page de résultat. Ces variables ont une valeur initiale de 1.

Lors du premier appel, le premier bloc de code est exécuté (la condition est remplie). Une requête GET est réalisée au travers de l'API fetch de javascript et le résultat est retourné sous forme de JSON par la fonction fetchJSON.

L'annexe Annexe : JSON posts utilisateur identifié présente un exemple de fichier retourné par l'API lorsque l'utilisateur est authentifié. L'annexe Annexe : JSON posts utilisateur non

identifié présente un exemple de fichier retourné par l'API lorsque l'utilisateur n'est pas authentifié.

En plus de contenir les résultats à afficher, la réponse contient le nombre total de pages de résultat disponible, l'attribut correspondant est mis à jour.

Les résultats (posts) sont mis en forme et ajoutés au DOM. L'attribut correspondant à la page de résultat actuelle est incrémenté, le texte du bouton observé est mis à jour.

Cette même logique est répétée à chaque appel jusqu'à qu'il n'y ai plus de résultat à charger. Dans ce cas, le texte du bouton observé est modifié en conséquence et l'observateur est déconnecté. Il n'y aura plus d'appel ultérieur à la méthode loadContent

L'image suivante présente le résultat final du post mis en forme.

[Capture d'écran d'un post présent sur la page /wall]

En plus du post, l'interface affiche les données de l'utilisateur ayant rédigé le post, les éventuelles commentaires et réactions au post / commentaire(s).

Exemple d'un ajout de commentaire

Il est possible de réagir au post, prenons l'exemple d'un ajout de commentaire:

Lors de la mise en forme du post, un lien Add Comment a été créé, voici l'extrait de code correspondant:

```
                <a href="#" class="card-footer-item new-comment"
onClick="reply(${post.post_pk_id})">Add comment</a>
```

[Extrait de code wall.js, fonction formatPost]

Lorsqu'un utilisateur clique sur le lien, la callback reply est appelée avec pour argument l'id du post sur lequel porte le commentaire.

```
const reply = async (id) => {
   let content = await loadContent(pathMain + '/comment/form');
   displayModal("New comment",content,id);
}
```

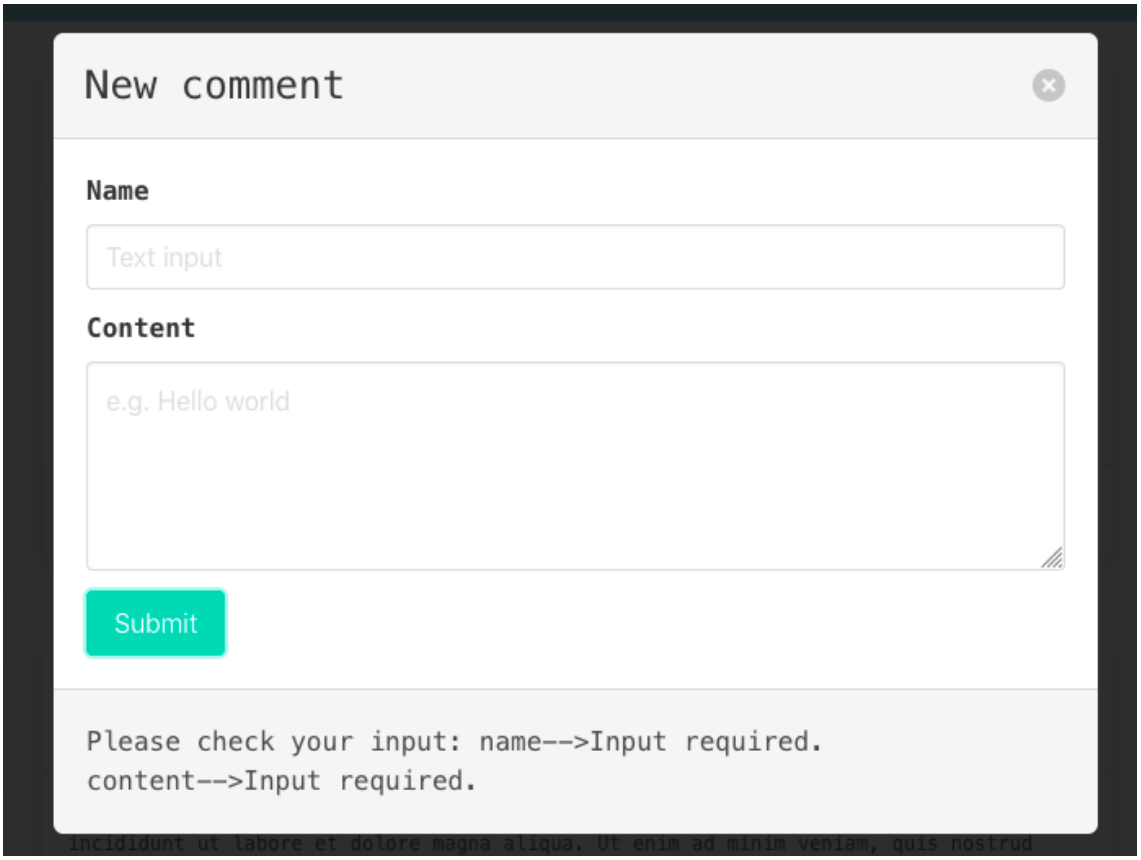
[Extrait de code wall.js, fonction reply]

Le formulaire de commentaire est chargé et affiché dans une modal. Lors de la soumission du formulaire présent dans la modal, la méthode postContent est appelée et les données renseignées sont transmises à l'API.

```
const postContent = async (formElement,url) => {
   let form = new FormData(formElement);
   let json = await fetch(url, {
     method: "POST",
     body: form
   }).then(response => response.json())
   return json
}
```

[Extrait de code main.js, fonction postContent]

En cas de succès de la soumission d'information, la modal se ferme. En cas d'erreurs, les erreurs sont affichées dans la modal. Cf image ci-après.



[Capture d'écran modal : Création d'un commentaire]

3. Base de données

Pour réaliser la base de données, il a été utile de procéder par étape.

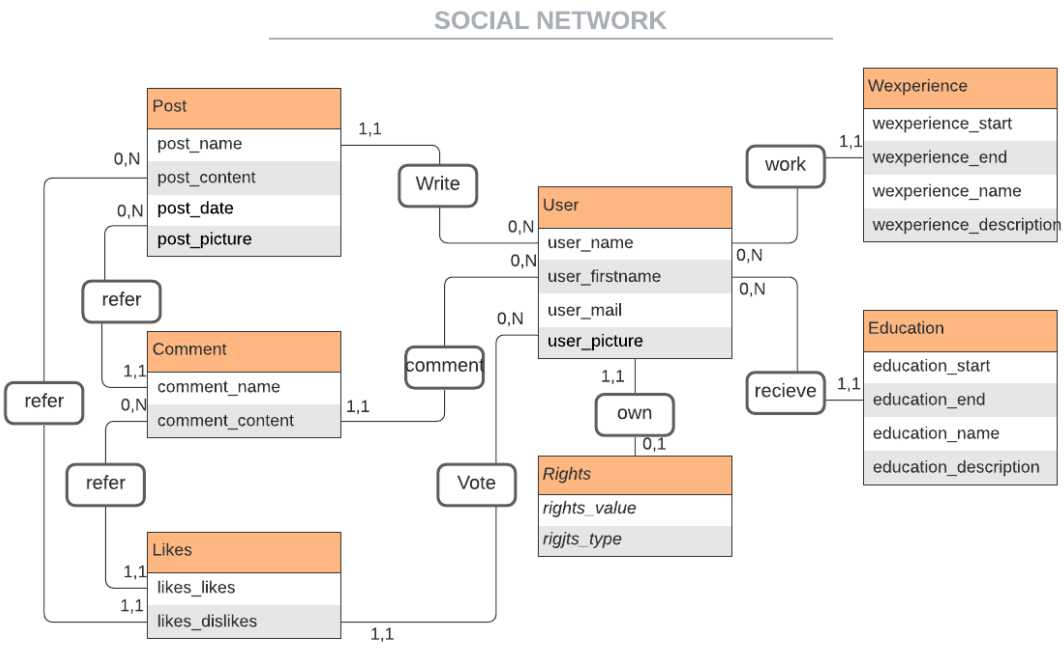
Création du dictionnaire de données

Réalisation du dictionnaire des données, il présente pour chaque donnée son code mnémonique, sa désignation son type et sa taille.

Création du MCD

Réalisation du MCD (Modèle Conceptuel de données).

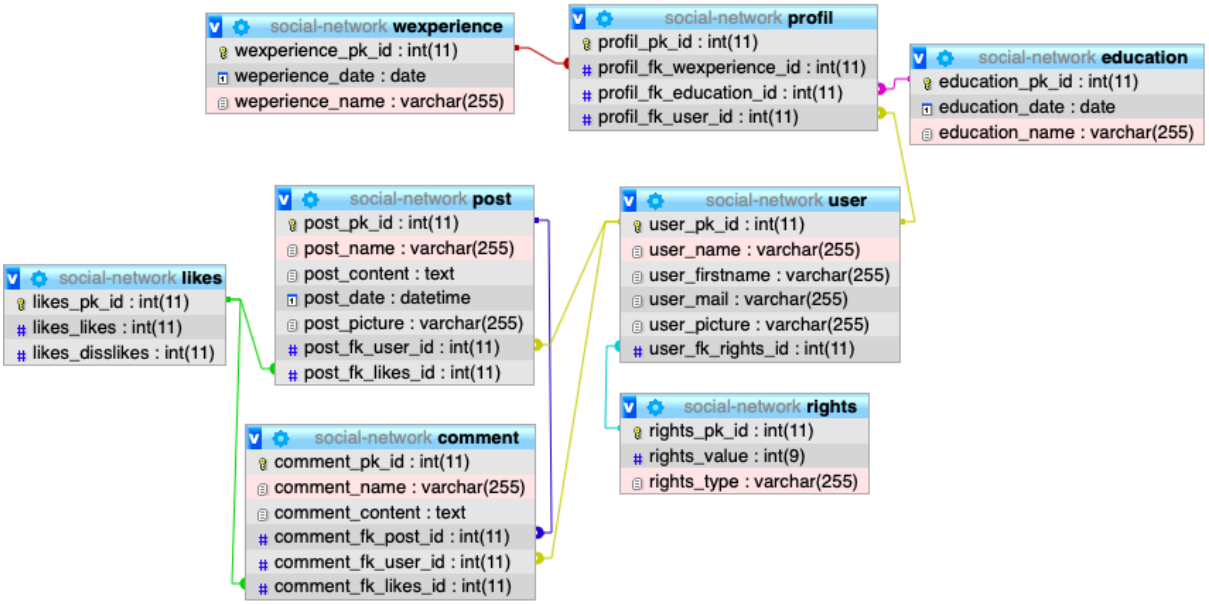
Ce modèle est la représentation abstraite des données. Il est indépendant de toute technologie. Les données y sont représentées sous forme d'entités et d'associations entre entités. Le diagramme suivant correspond au MCD du projet.



[Diagramme: Modèle Conceptuel de Données]

Implémentation dans MySQL

Une fois le MCD créé il est plus facile de réaliser l'implémentation sous MySQL. Le diagramme suivant est une représentation de l'implémentation réalisée.



[Capture d'écran interface MySQL : Organisation des tables]

Celui-ci est relativement similaire en termes d'organisation au MCD.

On voit apparaître en plus les clefs primaires des tables, ainsi que les clefs étrangères qui permettent de faire le lien entre les tables.

On voit également l'ajout de la table d'association profil entre user et wexperience / education.

Une table similaire est également nécessaire pour l'association likes / post - comment, afin de déterminer l'origine d'un like. Celle- ci n'a pas été implémentée pour le moment.

4. Back end

Pour présenter la partie Back-end, nous allons conserver l'exemple pris pour le front avec l'affichage des posts. La logique concernant l'ajout d'un commentaire étant sensiblement identique, elle ne sera pas détaillée ici. L'ajout d'un commentaire fait en revanche l'objet d'un zoom sur l'aspect sécurité. Il fait l'objet du chapitre Vulnérabilités de sécurité.

Exemple des posts

Route

Pour afficher les posts, la partie front fait un appel à la route /post.

Les routes sont définies dans le fichier routes.php (dossier config). L'extrait de code suivant présente la route en question.

```
    $app->get('/post[/{page}[/{user_id}]]', \App\Action\GetPost::class);
```

[Extrait de code config/route.php, fonction anonyme]

Dans cet extrait, la route contient des parties optionnelles: l'argument page et l'argument user_id. S' ils sont spécifiés, ils seront transmis au SAC GeTPost associé à la route.

Single Action Controller

Étant donné qu'aucune méthode n'est spécifiée, Slim fait appel au SAC comme à une fonction, dans ce cadre, c'est la méthode magique __invoke du contrôleur qui est appelé.

Lors de l'instanciation du contrôleur, le constructeur est tout d'abord appelé.

```
final class GetPost
{
```

```
    private $postFetcher;
    private $postsPerPage;
    private $likesFetcher;
    private $commentFetcher;

    public function __construct(ContainerInterface $container, PostFetcher
$postFetcher, CommentFetcher $commentFetcher, LikesFetcher $likesFetcher)
    {
        $this->postFetcher = $postFetcher;
        $this->commentFetcher = $commentFetcher;
        $this->likesFetcher = $likesFetcher;
        $this->postsPerPage = $container->get('settings')['postsPerPage'];
    }
```

[Extrait de code src/Action/GetPost.php, class GetPost]

PHP-DI permet ici l'injection de dépendances et la récupération de paramètres définis dans les settings au travers d'un container.

La fonction magique __invoke est ensuite appelée. Dans l'architecture choisie, le SAC est uniquement chargé de faire appel aux services nécessaires (logique business) et de gérer la requête et la réponse HTTP.

Slim envoie systématiquement aux méthodes appelées par les routes un objet correspondant à la requête HTTP reçue, un objet correspondant à la réponse HTTP à retourner, et les éventuels arguments passés à la route.

```
    public function __invoke(ServerRequestInterface $request,  ResponseInterface
$response, array $args): ResponseInterface
    {
        //Check if user is authentified
        if (isset($_SESSION["user"])) {
            // Invoke the Domain with inputs and retain the result
            // get posts
            $result = $this->postFetcher->fetch($args, $this->postsPerPage);

            // Get matching comments
            $result = $this->commentFetcher->fetch($result);

            // Get matching likes
            $result = $this->likesFetcher->fetch($result);


            if ($result["success"]) {
                $status = 200;
            } else {
```

```
                $status = 500;
            }
        } else {
            $result["success"] = false;
            $result["message"] = 'Permission denied';
            $status = 403;
        }

        // Build the HTTP response
        $response->getBody()->write((string)json_encode($result));

        return $response
            ->withHeader('Content-Type', 'application/json')
            ->withStatus($status);
    }
```

[Extrait de code src/Action/GetPost.php, class GetPost]

Dans l'extrait de code précédent, il est tout d'abord vérifié que l'utilisateur demandant les posts est bien identifié. Si oui, le script récupère les posts, commentaires, likes correspondant à la route, le détail du code pour la récupération des posts est à suivre.

Si le script a effectivement retourné des résultats, le statut de la réponse HTTP est fixé à 200, si une erreur est survenue ou si l'utilisateur n'est pas identifié les statuts seront respectivement fixé à 500 et 403.

Le tableau contenant les résultats (qui peuvent être nul) et les messages d'erreurs est ensuite encodé en JSON, et écrit dans le corps de la réponse HTTP.

La réponse est ensuite retournée, le statut de la réponse ainsi que le header spécifiant que la réponse est au format JSON sont ajoutés.

L'annexe Annexe : JSON posts utilisateur identifié présente un exemple de fichier retourné par l'API lorsque l'utilisateur est authentifié. L'annexe Annexe : JSON posts utilisateur non identifié présente un exemple de fichier retourné par l'API lorsque l'utilisateur n'est pas authentifié.

Service PostFetcher

Le service post fetcher a uniquement pour objectif de récupérer les posts en base de données par le biais d'une classe repository associée (la dépendance a été injectée dans le constructeur de la classe PostFetcher).

La fonction fetch présentée ci après, gère tout d'abord la pagination de résultat.
Elle récupère les informations des posts en base de données (en totalité ou filtré par user_id) au travers du repository. Elle ajoute dans le tableau associatif retourné des

informations supplémentaires relatives à la pagination de résultat (page actuelle de résultat, nombre total de page de résultat ..)

```php
    /**
     * Get post.
     *
     * @param array $args array could be empty / contain a page number for the result / and and user_id
     *
     * @return array $results array of post
     */
    public function fetch(array $args = [], int $postsPerPage = 10): array
    {
        // Handle pagination
        $page =  isset($args["page"]) ? (int) $args["page"] : 1;

        $offset = $page == 1 ? 0 : ($page - 1) * $postsPerPage;

        // Fetch Post by user_id or all post
        if (isset($args["user_id"])) {
            $posts = $this->repository->getPostsBydId($args["user_id"], $offset, $postsPerPage);
        } else {
            $posts = $this->repository->getPosts($offset, $postsPerPage);
        }

        $results["success"] =  empty($posts) ? false : true;

        if ($results["success"]) {
            $results["resultsNb"] = $this->repository->countPosts(isset($args["user_id"]) ? $args["user_id"] : 0);
            $results["currentPage"] = $page;
            $results["totalPage"] = ceil($results["resultsNb"] / $postsPerPage);
        }

        $results["posts"] = $posts;

        return $results;
    }
```

[Extrait de code src/Domain/Post/Service/PostFetcher.php, fonction fetch]

Afin de voir l'accès à la base de donnée, nous prenons l’exemple ici de la fonction getPosts

Repository PostRepository

Le lien de connexion à la base de données est injecté par PHP-DI dans le contructeur de la classe PostRepositoy. C'est également PHP-DI qui, si cela n'a pas été déjà fait, instancie l'objet PDO avec les informations de connexion contenues dans le fichier settings.

La fonction getPosts présentés ci-après correspond à la définition d'une requête SQL. On remarque ici l'utilisation d'un INNER JOIN afin de récupérer les posts mais aussi les informations relatives à leur auteur. Les posts sont ordonnés par dates décroissantes, dans une limite de 10 posts (pagination spécifiée dans le fichier settings) avec un offset correspondant à la page de résultat demandée.

Une requête est réalisée à la base de donnée au travers de l'objet PDO et les résultats sont retournés. L'usage de la méthode prépare n'est ici pas nécessaire étant donné que les informations utilisées pour bâtir la requête ne proviennent pas de l'utilisateur. Le chapitre Vulnérabilités de sécurité présente le cas de l'utilisation de la méthode prepare.

Par ailleurs, le typage des variables $offset et $limit est précisée au niveau de la définition des arguments et correspond à un entier. Une requête par injection SQL n'est donc pas possible.

```php
/**
 * Get posts
 *
 * @param int $offset - Result offset
 *
 * @param int $limit - Nb max of result
 *
 * @return array $posts - set of result
 */
public function getPosts(int $offset = 0, int $limit = 100): array
{
    $sql = "SELECT *
            FROM `post`
            INNER JOIN `user` ON `post`.`post_fk_user_id` = `user`.`user_pk_id`
            ORDER BY `post`.`post_date` DESC
            LIMIT $limit OFFSET $offset;";
    return $this->connection->query($sql)->fetchAll();
}
```

[Extrait de code src/Domain/Post/Repository/PostRepository.php, fonction getPost]

5. Système de chat

Le système de chat est basé sur le protocole Websocket.

L’utilisation des websockets avec la librairie socket.io permet d’établir une connexion en temps réel, à double sens et basée sur l'occurrence d'événement.

Le système de chat permet :

- la communication public au travers d’une room générale,
- la communication privé au travers d’une room partagée entre deux utilisateurs,
- d’obtenir une liste des utilisateurs connectés.

Afin de présenter le système de chat, nous prendrons en exemple la fonctionnalité correspondant à la liste des utilisateurs connectés.

Côté client, le chat est construit autour de la classe Chat définie dans le fichier chat.js.

L’extrait de code ci-après présente l’instanciation de la classe.

```
let chatInit = {
   url: "social.network:3001",
   userListEl: document.querySelector('.chat-user-list'),
   inputEl: document.querySelector('#chat-form'),
   displayEl: document.querySelector('.chat-message')
}

var chat = new Chat(chatInit)
```

[Extrait de code chat.js]

Le constructeur prend en argument un objet contenant l’url du serveur websocket ainsi que des éléments du DOM utiles au chat (liste utilisateur / input message / zone d’affichage des message)

La méthode connect est appelée par le constructeur. L’extrait de code suivante présente le code de la méthode:

```
connect(url){
       this.socket = io(url);
       this.socket.emit('identification', getCookie('id_token'));
   }
```

[Extrait de code chat.js, classe Chat, méthode connect]

Cette méthode établit la connexion au serveur websocket et sauvegarde en tant qu’attribut l’objet correspondant au socket retourné.

Afin d'identifier chaque utilisateur côté serveur, le client émet l'événement 'identification' avec pour paramètre l' id_token retourné par la fonction google authentification. Ce token avait été sauvegardé au préalable dans un cookie lors de la connexion de l'utilisateur.

Passons côté serveur, celui-ci autorise les connexions d'origine étrangère au serveur grâce à la clef CORS (Cross-origin resource sharing) de l'objet passé en paramètre lors de la création du serveur.

Cette configuration est nécessaire dans la mesure ou le serveur HTTP servant les pages du site n'est pas le serveur de chat. En revanche l'origine pourrait être plus spécifique que "*". Pour des tests lors du développement il est plus commode de garder "*", ce parametre devra être changé lors de la mise en production.

```
const httpServer = require("http").createServer();
const hostname = 'social.network';
const port = 3001;
const io = require("socket.io")(httpServer, {
 cors: {
     origin: "*",
     methods: ["GET", "POST"]
   }
});
```

[Extrait de code server/index.js]

Le serveur doit ensuite être lancé en écoutant un port spécifique ici 3001 sur un nom d'hôte donné.

```
httpServer.listen(port, hostname, () => {
 console.log(`Server running at http://${hostname}:${port}/`);
});
```

[Extrait de code server/index.js]

La configuration du serveur websocket est basée sur un système d'événement déclenchant des callbacks.

L'extrait de code suivant présente le code exécuté lors de l'occurrence de l'événement 'identification' émis par le client après s'être connecté au serveur websocket.

```
socket.on('identification', (id_token)=>{
     googleAuth.verify(id_token)
       .then((payload)=>{
         newUser = {
           id: socket.id,
           name: payload.name
         };
```

```
        if(isNewUser(newUser,userList) == false){
          userList.push(newUser)
          io.emit('chat message', newUser.name + ' has joined the room');
        }
        io.emit('user list', extractUsersName(userList));
      })
      .catch(console.error('Authentification failed'));
  })
```

[Extrait de code server/index.js]

Le token transmis par le client est vérifié, si celui-ci est authentifié les informations utilisateurs sont récupérées. Un objet NewUser contenant le socket id du client et son nom est ajouté à une liste d'utilisateurs (lors de la déconnexion de l'utilisateur, celui-ci est supprimé).

Le serveur émet ensuite deux événement reçu par tous les utilisateurs:

- un premier de type 'chat message' correspondant à un message informant les utilisateurs qu'un nouvel utilisateur les a rejoint,
- un second de type 'user list' contenant la liste des utilisateurs connectés.

Si l'authentification google a échoué, un message d'erreur est affiché dans la console du serveur, l'utilisateur n'aura pas accès aux fonctionnalités du chat. La vérification étant faite grâce à une clef CLIENT_ID délivrée par google et propre à l'application, une personne extérieure à l'organisation ne pourra pas se connecter.

Retournons côté client pour voir la logique de code déclenchée par l'événement 'user list' émis par le serveur:

```
this.socket.on('user list', (list) => {
          this.setState({
              userList:list
          } );
          this.updateUserList(this.getState());
      })
```

[Extrait de code chat.js, classe Chat, méthode initialiseSocket]

Un état local est sauvegardé au niveau de l'attribut de la classe et la méthode updateUserList est appelée.

```
updateUserList(state){
      if(state.userList.length <= 0){
          return;
      }
      let list = state.userList.map(user=>formatUser(user)).join('')
```

```
        this.userListEl.innerHTML = list;
        document.querySelectorAll('.user-item').forEach(initiateUserItem.bind(this))
    }
```

[Extrait de code chat.js, classe Chat, méthode updateUserList]

Si la liste contient des utilisateurs, celle-ci est parcourue grâce à la méthode map et chaque élément est mis en forme grâce à la fonction formatUser qui génère du code html.

Le code HTML est ajouté à l'élément du DOM correspondant à la liste des utilisateurs.

Enfin, pour chaque élément de la liste, la callback initiateUserItem est appelée afin de mettre à jour les événements déclenchés lors d’un clic sur un élément de la liste. Ceux-ci sont chargés de la logique de code à exécuter pour déclencher un chat privé avec un utilisateur.

6. Vulnérabilités de sécurité

Une attention particulière doit être apportée à la sécurité de l'application pendant toute la durée de la conception de l'application mais également lors de sa mise en exploitation.

Le site du centre gouvernemental de veille d'alerte et de réponse aux attaques informatiques permet d'être informé des alertes de sécurité en cours https://www.cert.ssi.gouv.fr/.

Il propose aussi au travers de guide tel que "RECOMMANDATIONS POUR LA MISE EN ŒUVRE D'UN SITE WEB : MAÎTRISER LES STANDARDS DE SÉCURITÉ CÔTÉ NAVIGATEUR", les recommandations de sécurité à intégrer lors de la conception d'un site web.

Parmi les attaques les plus communes, il y a l'injection SQL et les attaques XSS.

L'extrait de code suivant correspond au code exécuté pour ajouter un commentaire dans la base de données.

```
    /**
     * Create comment
     *
     * @param int $id
     *
     * @param array $data
     *
     * @return int the new id
     */
    public function newComment(array $comment): int
    {
        $row = [
            'comment_name' => htmlspecialchars($comment['name']) ,
            'comment_content' => htmlspecialchars($comment['content']),
            'comment_fk_user_id' =>
htmlspecialchars($comment['comment_fk_user_id']),
            'comment_fk_post_id' => htmlspecialchars($comment['comment_fk_post_id'])
        ];
        $sql = "INSERT INTO comment SET
                comment_name=:comment_name,
                comment_content=:comment_content,
                comment_fk_user_id=:comment_fk_user_id,
                comment_fk_post_id=:comment_fk_post_id;";

        $this->connection->prepare($sql)->execute($row);

        return (int)$this->connection->lastInsertId();
```

```
    }
```

[Extrait de code src/Domain/Comment/Repository/CommentRepository.php, fonction newComment]

Un utilisateur malveillant ne peut ici pas réaliser d'injection SQL, cette attaque correspond à intégrer dans notre cas des query SQL dans les champs du formulaire de commentaire afin que ces requêtes soient exécutés lors de l'insertion du commentaire en base de données. L'utilisation de la méthode prepare rend ici cette attaque impossible. En effet la query est déclarée séparément des données à intégrer dans la table de la base de données.

Il n'est également pas possible de réaliser une attaque de type XSS qui correspond à injecter du code HTML (exemple intégration du balise <script> permettant de lier un fichier js) qui serait envoyé aux autres utilisateurs par le serveur lorsque ceux-ci consulteront la page où le commentaire a été posté. Cette attaque est rendue impossible par l'utilisation de la fonction système htmlspecialchar() qui échappe les caractères spéciaux html sur toutes les données en provenance de l'utilisateur.

7. Anglais

1. Situation de travail ayant nécessité une recherche sur un site anglophone

Nous avons décidé d'utiliser la bibliothèque PHP DI afin de disposer d'un outil pour créer un conteneur chargé de l'injection de dépendances.

L'injection de dépendances est un design pattern qui permet d'implémenter le principe d'inversion de contrôle. Il permet d'injecter dynamiquement les dépendances entre les objets en s'appuyant sur leurs descriptions ou le typage des arguments d'une fonction …
De cette façon, les dépendances entre les objets ne sont plus exprimées de manière statique.

En regroupant ces dépendances dans un conteneur, il devient facile de changer une dépendance pour une autre. C'est ce que permet de faire PHP-DI.

La documentation de PHP-DI est uniquement disponible en anglais.

2. Extrait du site anglophone et traduction

Site anglophone : https://php-di.org/doc/autowiring.html

Extrait du site

Autowiring

Autowiring is an exotic word that represents something very simple: the ability of the container to automatically create and inject dependencies.

In order to achieve that, PHP-DI uses PHP's reflection to detect what parameters a constructor needs.

Autowiring does not affect performances when compiling the container.

Let's take this example:

```
class UserRepository
{
    // ...
```

```
}

class UserRegistrationService
{
    public function __construct(UserRepository $repository)
    {
        // ...
    }
}
```

When PHP-DI needs to create the `UserRegistrationService`, it detects that the constructor takes a `UserRepository` object (using the type hinting).

Without any configuration, PHP-DI will create a `UserRepository` instance (if it wasn't already created) and pass it as a constructor parameter. The equivalent raw PHP code would be:

```
$repository = new UserRepository();
$service = new UserRegistrationService($repository);
```

As you can imagine, it's very simple, doesn't require any configuration, and it just works!

Traduction du site

Autowiring

Autowiring (câblage automatique) est un mot complexe qui représente un concept simple: la capacité du conteneur à automatiquement créer et injecter des dépendances.

Pour y arriver, PHP-DI utilise les classes de reflection de PHP pour détecter quels paramètres sont requis par le constructeur.

L'autoriwing n'affecte pas les performances lorsque le container est compilé.

Prenons l'exemple suivant:

```
class UserRepository
{
    // ...
}

class UserRegistrationService
{
    public function __construct(UserRepository $repository)
    {
```

```
        // ...
    }
}
```

Quand PHP-DI doit créer le `UserRegistrationService`, il détecte que le constructeur prend un objet `UserRepository` en argument (en utilisant l'indice de typage).

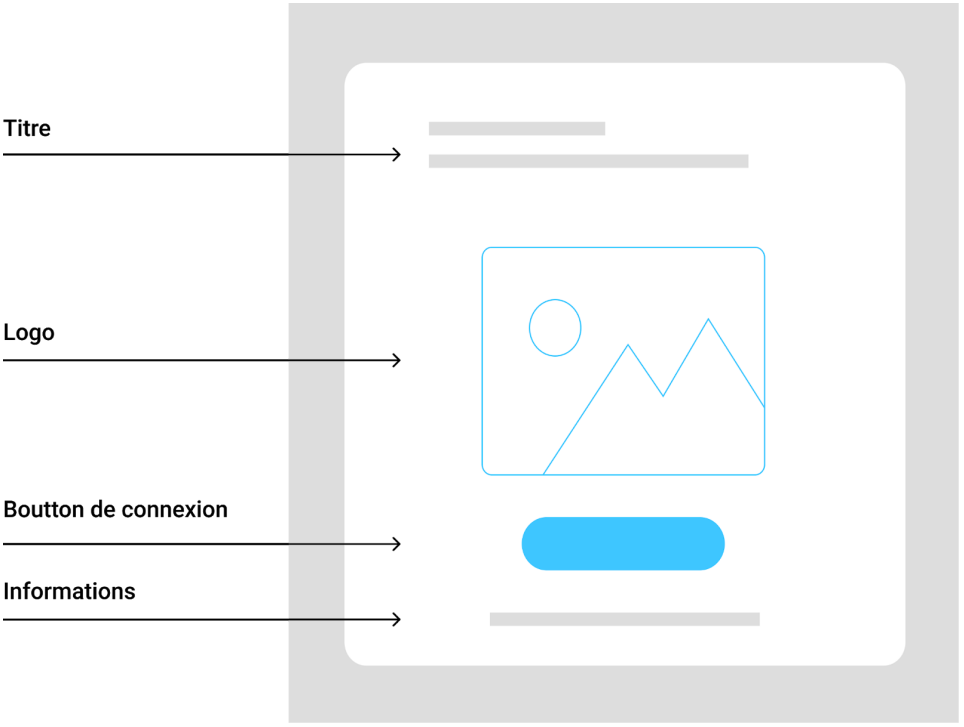
Sans aucune configuration, PHP-DI créera une instance de `UserRepository` (si cela n'a pas déjà été fait) et le passera en paramètre au constructeur. Le code PHP brut équivalent serait:

```php
$repository = new UserRepository();
$service = new UserRegistrationService($repository);
```

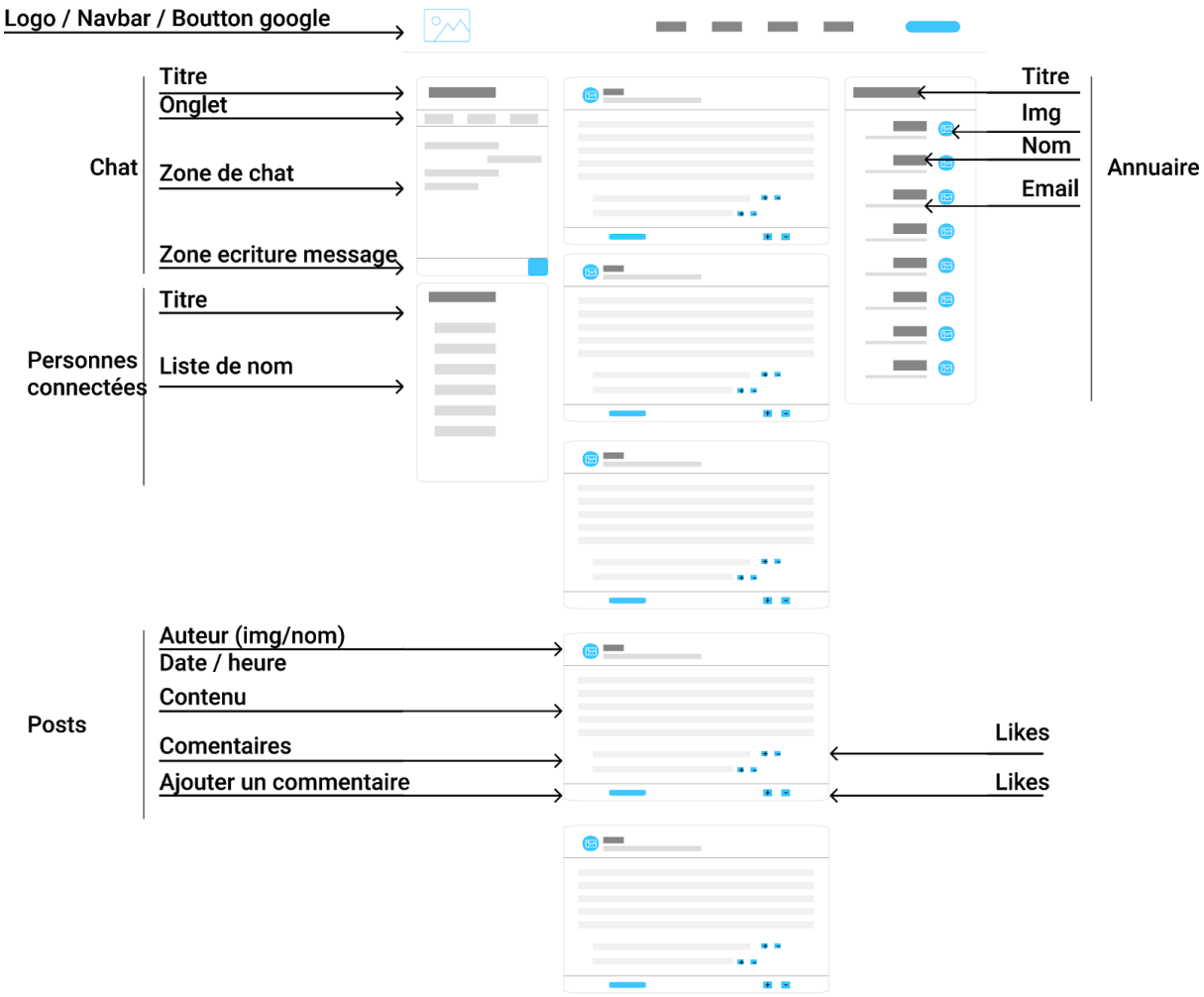
Comme vous pouvez le voir, c'est très simple, ça ne requiert aucune configuration, et ça marche !

Annexes

1. Annexe : Wireframe page Connexion

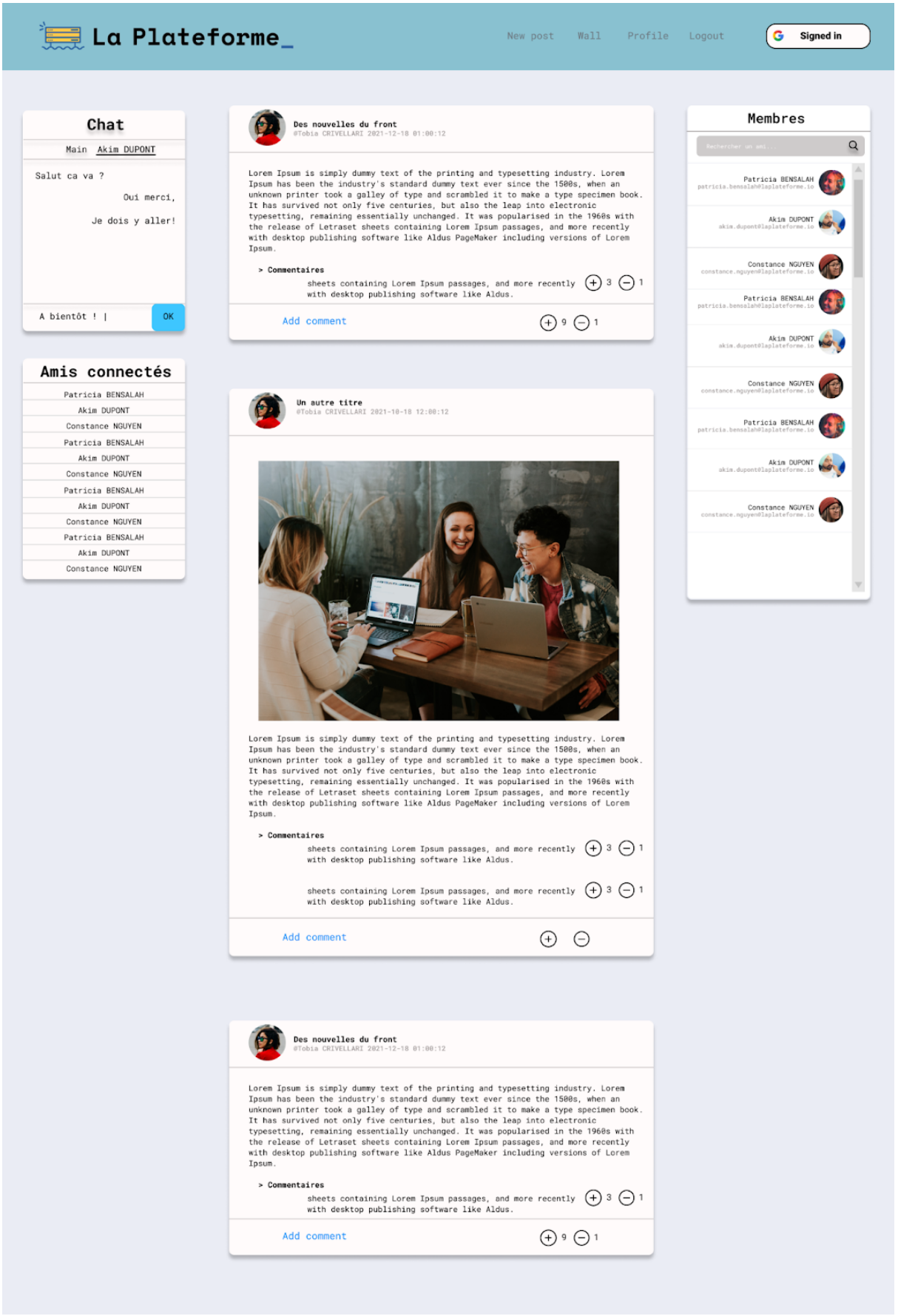


2. Annexe : Wireframe page Wall



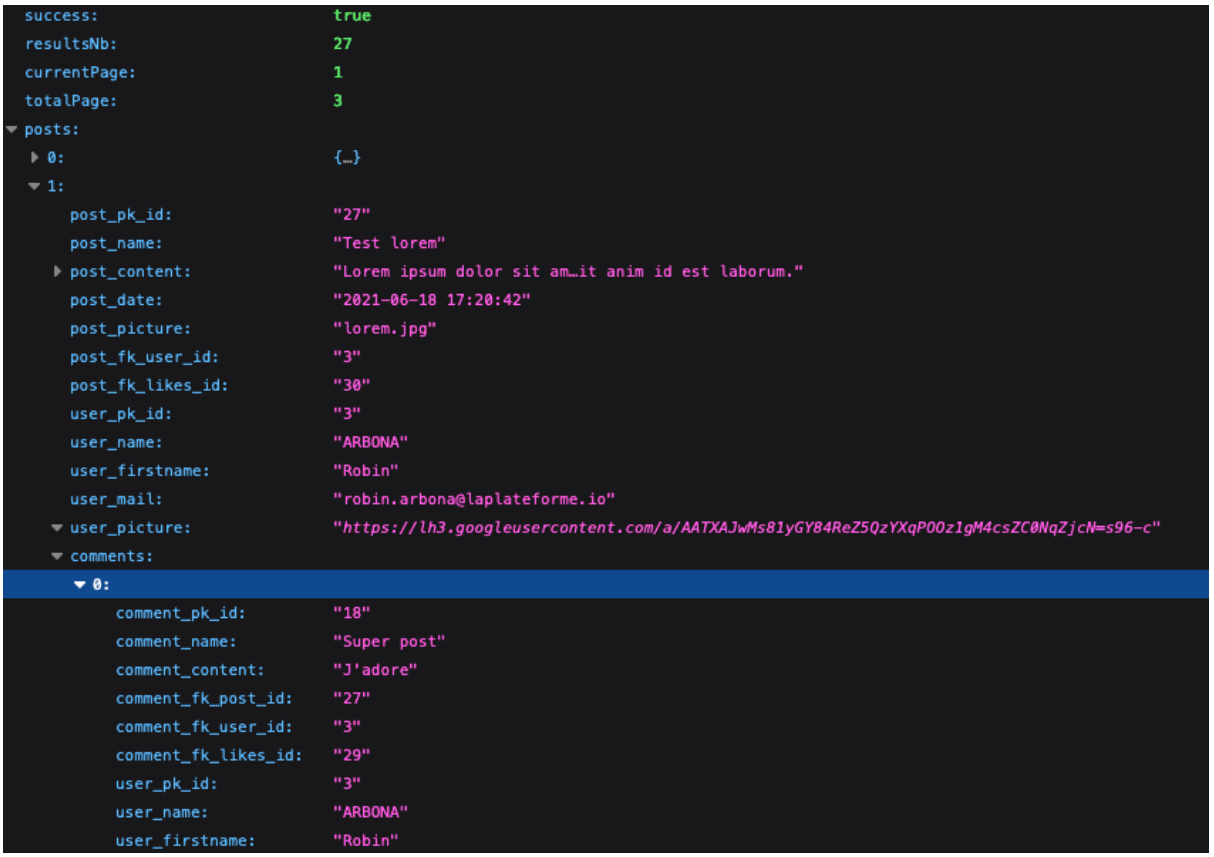
3. Annexe : Maquette page Connexion

4. Annexe : Maquette page Wall



5. Annexe : JSON posts utilisateur identifié

Structure fichier JSON: réponse retournée pour la route http://social.network:8888/post/1 lorsque l'utilisateur est identifié



6. Annexe : JSON posts utilisateur non identifié

Structure fichier JSON: réponse retournée pour la route http://social.network:8888/post/1 lorsque l'utilisateur n'est pas identifié