

ArticleHub

Application web pour lire et créer des articles

Projet réalisé pour la présentation de l'examen du Titre Professionnel de

Développeur Web et Web Mobile

Presente par
Esteba BARE

Sommaire

Introduction

Depuis que je suis jeune, je souhaitais me bâtir un avenir dans le domaine technologique, sans vraiment savoir vers quel domaine me diriger. J'ai suivi mon parcours scolaire sans me spécialiser jusqu'à ce qu'une connaissance me parle du développement web. C'est à ce moment-là que j'ai commencé à m'intéresser à ce domaine et que j'ai décidé de me former. Après avoir obtenu mon baccalauréat général, j'ai cherché sans succès la formation qui me correspondait vraiment, jusqu'à ce que je découvre enfin La Plateforme, qui m'a permis d'apprendre la programmation web et tout ce qui l'entoure au cours d'une période de 8 mois.

Apprendre de manière autodidacte m'a permis d'acquérir de nombreuses compétences ainsi que des soft skills que j'ai développés en travaillant en groupe et en demandant de l'aide à mes collègues durant ma formation. Ce parcours a renforcé ma passion pour la programmation et a intensifié mon désir de progresser continuellement dans ce domaine.

Après ce parcours, j'ai finalement décidé de créer ArticleHub, qui, je pense, combine toutes les compétences que j'ai acquises, et me permet d'utiliser des outils que j'apprécie et que je trouve très utiles.

Liste de compétences du référentiel couvertes par le projet

1. Développer la partie front-end d'une application web ou web mobile sécurisée :

- Installer et configurer son environnement de travail en fonction du projet web ou web mobile :

Pour utiliser Laravel, j'ai eu besoin de différents logiciels sur mon ordinateur. Étant donné que j'avais déjà installé Laragon, où PHP et MySQL étaient déjà présents, j'ai simplement dû mettre à jour PHP vers la version 8.2.21 pour pouvoir créer un projet Laravel. Ensuite, j'ai installé Composer, ce qui m'a permis d'installer les dépendances de Laravel dans mon projet. Enfin, j'ai codé tout mon projet dans Visual Studio Code, ce qui m'a permis d'utiliser le terminal pour les commandes Laravel ainsi que pour utiliser Git et sauvegarder mon travail dans le dépôt GitHub de mon projet.

- Maquetter des interfaces utilisateur web ou web mobile:

Pour mieux organiser mon travail et anticiper plus clairement la structure de mon projet, je crée une maquette avec Figma. Cela facilite la compréhension de l'esthétique des pages et des liens entre les pages et les composants. De plus, grâce à la fonctionnalité de prototypage de Figma, je peux visualiser les wireframes en observant les liens entre les composants et les pages.

- Réaliser des interfaces utilisateur statiques web ou web mobile:

Dans mon application, j'ai eu besoin de créer une interface statique pour la page de contact. Cette page contient du contenu statique ainsi

qu'un formulaire qui n'interagit pas avec la base de données mais envoie simplement un email.

- Développer la partie dynamique des interfaces utilisateur web ou web mobile

Grâce à Laravel, j'ai pu utiliser le moteur de templates Blade, qui m'a permis d'interagir dynamiquement avec les données côté serveur. Dans le frontend, je vais montrer des exemples concrets de cette interaction. J'ai également utilisé JavaScript pour mettre en œuvre une barre de recherche d'articles ainsi qu'un système de likes sur la page de lecture des articles.

2. Développer la partie back-end d'une application web ou web mobile sécurisée:

- Mettre en place une base de données relationnelle:

Pour mettre en place une base de données relationnelle, j'ai suivi plusieurs étapes. Tout d'abord, j'ai créé un MCD (Modèle Conceptuel de Données) en utilisant MySQL Workbench. Cela m'a permis de visualiser clairement mes données et de créer un diagramme de classes avec leurs relations. Ensuite, grâce au système de migrations de Laravel, j'ai simplement configuré le fichier .env pour établir la connexion à ma base de données. Les tables ont été créées à l'aide de migrations que je décrirai plus en détail ultérieurement.

- Développer des composants d'accès aux données SQL et NoSQL:

Pour accéder à mes données SQL, j'ai choisi d'utiliser l'ORM de Laravel appelé Eloquent, ce qui a grandement facilité mon travail. En utilisant les migrations de Laravel, les modèles de mes tables ont été créés automatiquement avec leurs relations. J'ai simplement eu besoin de créer quelques fonctions pour obtenir des informations spécifiques, et mes composants étaient opérationnels.

- Développer des composants métier côté serveur:

Pour mon projet, j'ai eu besoin de nombreuses routes que j'ai créées avec Laravel pour afficher des vues en utilisant le MVC intégré dans Laravel. J'ai également créé des routes de suppression pour supprimer des utilisateurs ou des articles depuis le back-office destiné aux administrateurs. En plus de cela, j'ai mis en place de nombreuses routes POST pour permettre la création de comptes, la connexion au site, la création d'articles, ainsi qu'une route pour ma barre de recherche, permettant aux utilisateurs de rechercher directement à travers tous les articles du site.

Resume du projet

Pour ce projet, je souhaite m'inspirer de concepts déjà bien établis, notamment les articles journalistiques que nous avons tous eu l'occasion de lire. Cependant, je veux rendre cela plus accessible et moderne en prenant exemple sur GitHub. GitHub représente l'avancement de la technologie sur Internet et du code ouvert à tous et pour tous. Dans cette optique, je souhaite créer *ArticleHub*, une plateforme qui permettra à tout le monde de créer des articles que chacun pourra lire, liker et commenter.

ArticleHub offrira également une facilité de navigation pour trouver des articles dans les domaines qui intéressent le plus chaque utilisateur, grâce à un système de catégories. L'objectif est de faire grandir la communauté avec les utilisateurs eux-mêmes, puisque ce sont eux qui commentent, likent et donnent de la visibilité aux articles des personnes désireuses d'écrire.

En résumé, *ArticleHub* ambitionne de devenir un espace où chacun peut partager ses idées et ses connaissances à travers des articles, tout en bénéficiant de l'interaction et du soutien de la communauté.

Cahier des charges

Objectif

L'objectif principal de mon projet est de donner aux utilisateurs la possibilité de créer et de lire des articles. Je souhaite également permettre l'interaction avec ces articles en ajoutant des fonctionnalités de "like" et de commentaire. Lorsque le site contiendra un grand nombre d'articles, il sera nécessaire de les catégoriser pour faciliter la recherche. De plus, les utilisateurs pourront consulter les articles les plus anciens, les plus récents et ceux qui sont les mieux notés.

J'aurai besoin d'un back-office qui me permettra de contrôler les utilisateurs et les articles créés. Il est essentiel de créer un site sécurisé pour les utilisateurs. Pour cela, il faudra mettre en place un système d'inscription et de connexion sécurisés.

User Stories

Pour savoir mieux quel fonctionalite je besoin de savoir le cas de possible utilisation

ID	En tant que	Je veux	Afin de
1	Utilisateur non connecté	M'inscrire	Créer un compte et accéder à des fonctionnalités réservées
2	Utilisateur non connecté	Me connecter	Accéder à mon compte et interagir avec le contenu
3	Utilisateur connecté	Me déconnecter	Quitter mon compte en toute sécurité
4	Utilisateur non connecté	Consulter les articles	Lire le contenu disponible sur le site
5	Utilisateur connecté	Consulter les articles	Lire le contenu disponible et y réagir
6	Utilisateur connecté	Commenter un article	Partager mes opinions et participer aux discussions
7	Utilisateur connecté	Liker un article	Montrer mon appréciation pour le contenu
8	Utilisateur connecté	Créer un nouvel article	Partager mes connaissances ou mes opinions
9	Utilisateur connecté	Attribuer une catégorie à un article	Classer l'article pour le retrouver plus facilement
10	Utilisateur connecté	Filtrer les articles par catégorie	Trouver des articles sur des sujets spécifiques
11	Administrateur	Modérer les commentaires	Maintenir un environnement de discussion respectueux
12	Administrateur	Supprimer des articles inappropriés	Assurer la qualité et la pertinence du contenu publié
13	Administrateur	Créer et gérer des catégories	Organiser les articles de manière structurée

Fonctionnalités

En suivant les user stories je peux faire une liste de mes fonctionnalités principale:

1. **Accueil** : Présentation des articles les plus récents ou populaires.
2. **Liste des Articles** : Affichage des articles par catégories ou tags, avec options de filtre et de recherche.
3. **Détail d'un Article** : Lecture complète d'un article, avec possibilité de commenter et de liker.
4. **Profil Utilisateur** : Gestion des informations personnelles, des articles publiés, des commentaires et des likes.
5. **Authentification** : Système de connexion et d'inscription.
6. **Administration** : Tableau de bord pour l'administration du site, gestion des articles et utilisateurs

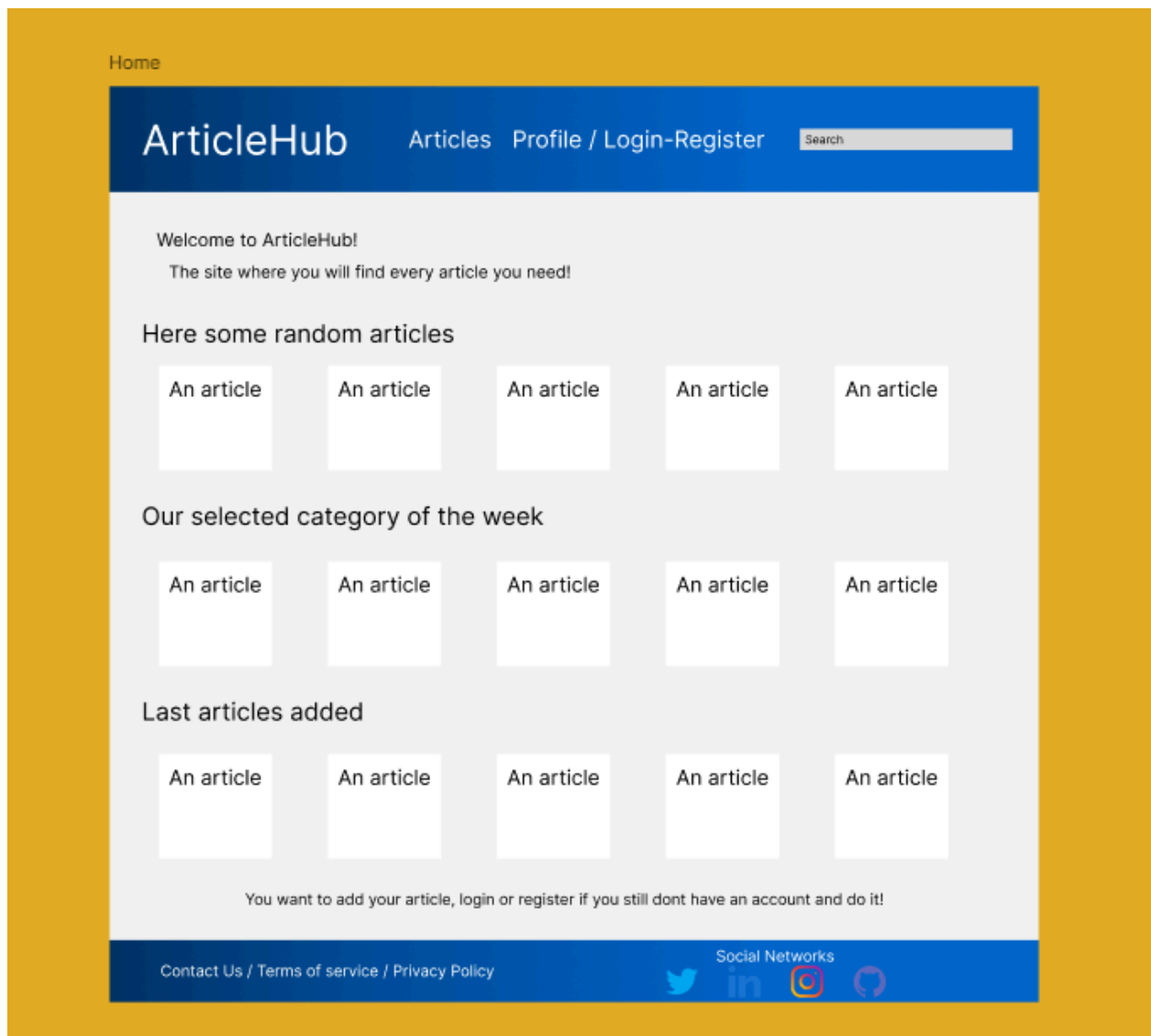
Grace a ca on se les rôles et permission:

1. **Utilisateur Non Authentifié** :
 - Consulter les articles.
 - Utiliser la recherche et les filtres.
2. **Utilisateur Authentifié** :
 - Publier des articles.
 - Commenter et liker les articles.
 - Modifier son profil et gérer ses publications.
3. **Administrateur** :
 - Gérer l'ensemble des utilisateurs, articles et catégories.
 - Superviser et modérer les interactions (commentaires, likes).

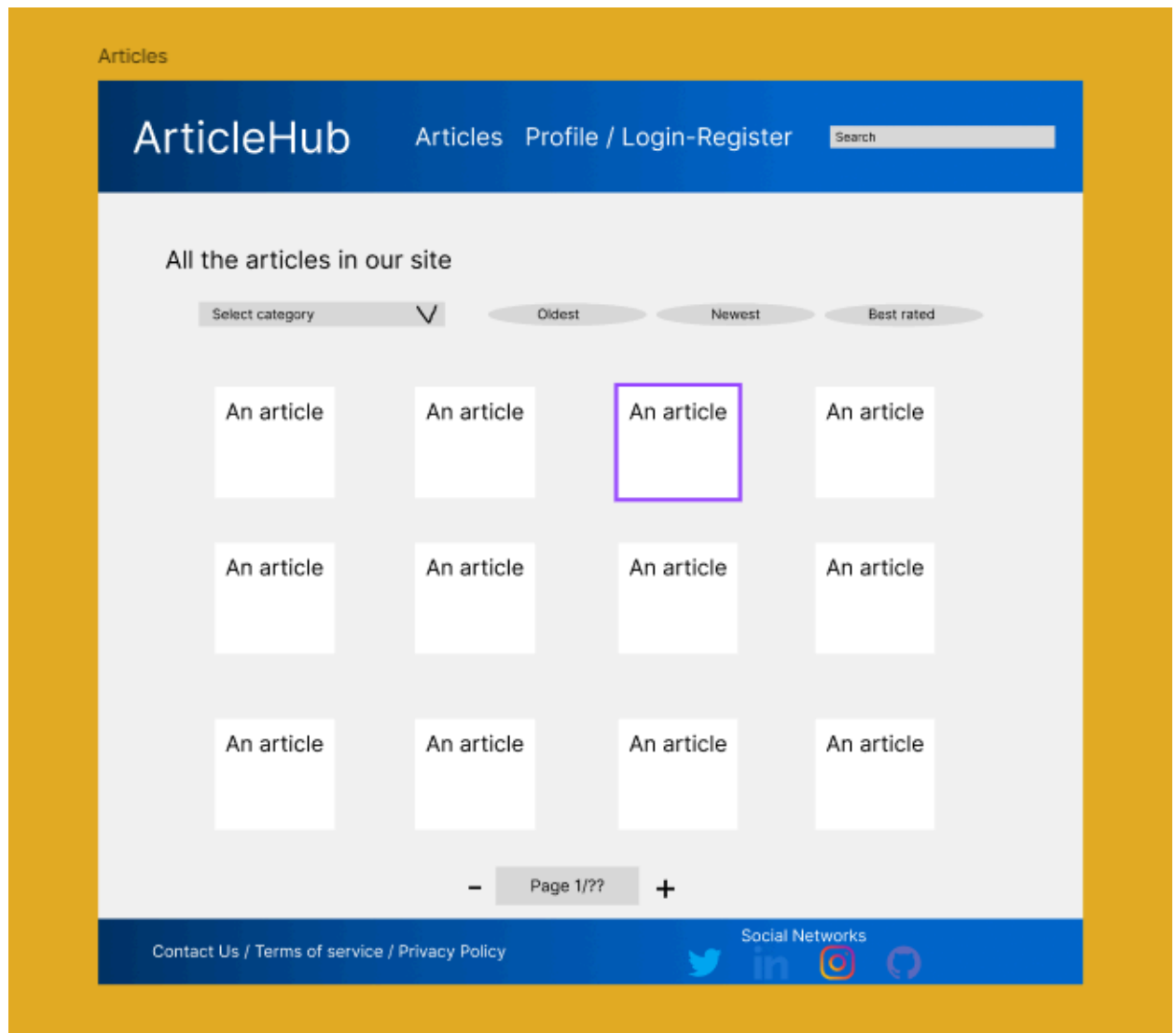
Maquette et Wireframes

Pour ma maquette je utilise figma qui ma permis de pense a l'avance à ma charte graphique et à comment j'ai voule structure mes page et donne l'esthétique nécessite

- Page d'accueil



- Page des articles



- Page des détails d'un article



- Page d'enregistrement et de connexion

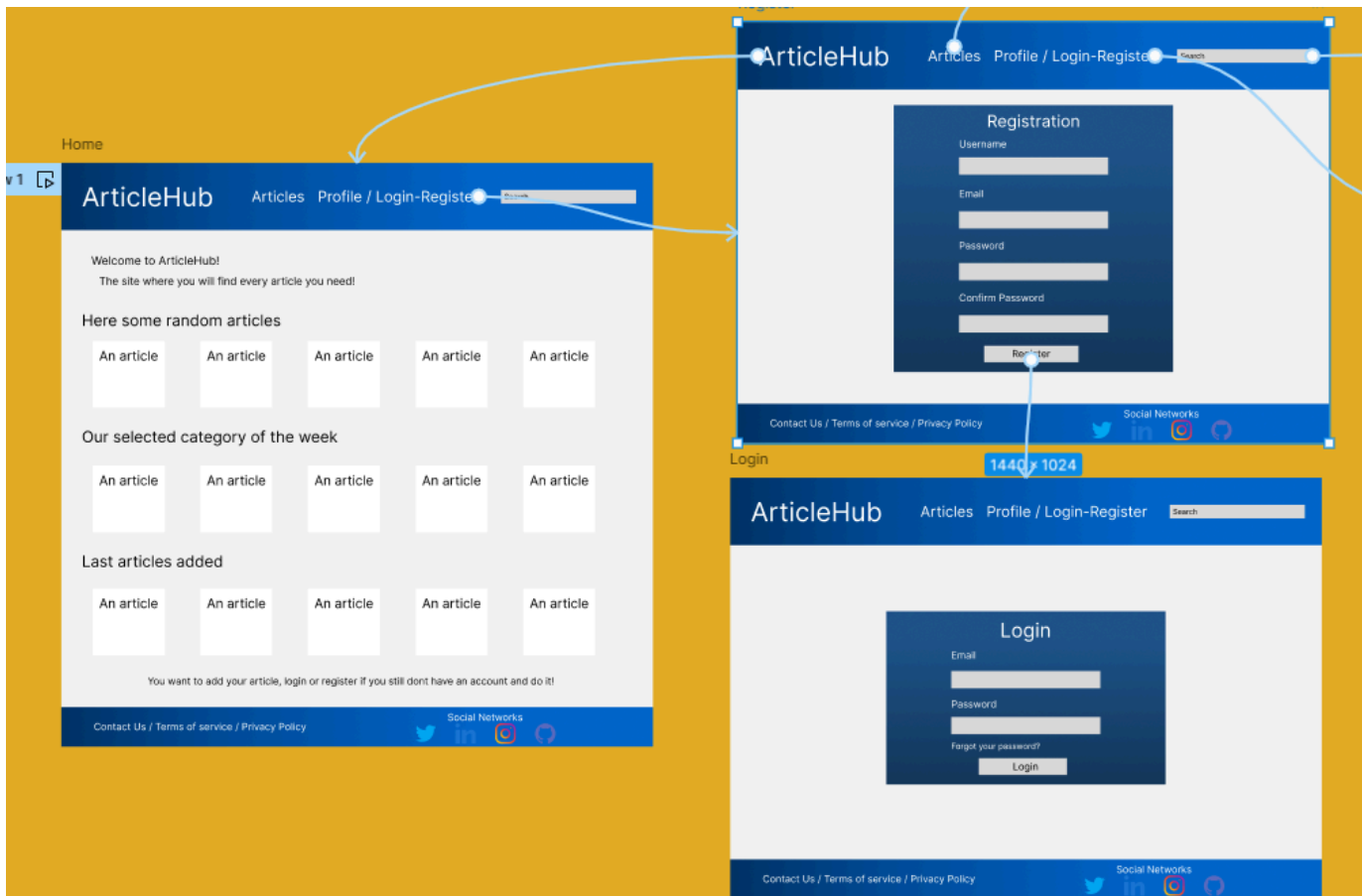


- Page profilé et page ajout de article

- Voici les page du back-office

Grâce à cette page appelée frames je peux utiliser une fonctionnalité de figma appelez prototype qui me permet de voir le wireframes et comment me composent et page vont interagir.

- Lien entre accueil et page register



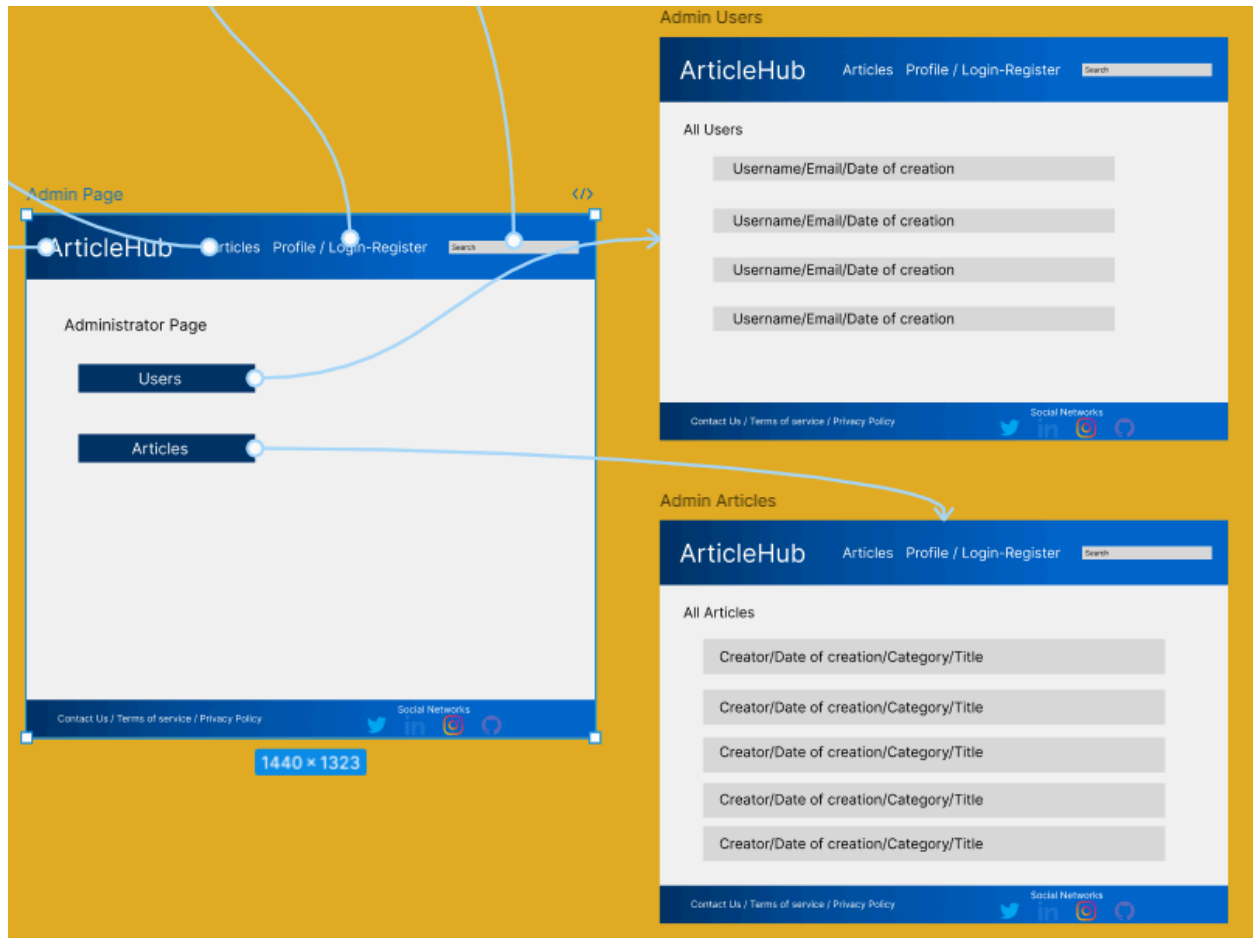
Ici on peut voir que le lien dans le header me permet d' aller dans la page d' enregistrement pour apre une fois enregistré être renvoyé dans la page de connexion

- Lien entre la page des articles et la page de de détails d'un article



On peut voir qu'il y a beaucoup de liens entre ces deux pages, ce qui est normal puisque chaque article renvoie à sa page de détails. Cependant, il y a aussi un lien entre la barre de recherche et la page de détails. En effet, lorsqu'un article proposé est cliqué, il renvoie également à la page de détails.

- Lien entre la page principale d'administration et le pages de gestion des données



Ici on peut voir le page seulement accessible aux compte avec le rôle admin et que de la page principale on a deux lien qui renvoie au pages de gestion.

Stack et technologies utilisées

Front-End

Pour le front-end, j'ai décidé d'utiliser HTML pour la structure, CSS pour le style des pages, et JavaScript pour l'interactivité du site, utilisé dans quelques fonctionnalités.

Back-End

Pour le back-end, c'est plus intéressant. J'utilise le framework Laravel. Ayant travaillé avec PHP pendant toute la formation, j'ai décidé de tenter de réaliser un projet avec Laravel, car je pense que c'est une bonne façon d'apprendre le framework et de créer un bon côté serveur. Grâce à la documentation et à mes connaissances en routage avec alterrouter et à la méthode de travail MVC, j'ai réussi à créer un bon back-end.

Base de Données

Pour la base de données, j'utilise le système de gestion MySQL avec phpMyAdmin pour visualiser graphiquement la base de données. Pour créer la base de données, j'utilise l'ORM Eloquent de Laravel.

En plus de tout ce que j'ai mentionné, j'utilise Blade, qui est largement utilisé dans Laravel pour le templating et qui apporte beaucoup de dynamisme à mon site.

Accessibilite

Navigateur disponible

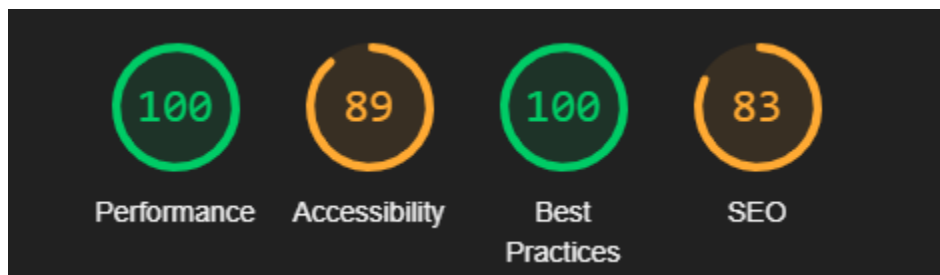
Les navigateurs compatibles avec mon application sur desktop sont Chrome, Edge, et Opera. Sur téléphone, les navigateurs compatibles sont Chrome et Safari.

Type d'appareil

Grâce à l'utilisation des DevTools de Chrome et de mon téléphone, j'ai pu vérifier si mon application était responsive pour téléphone et tablette en utilisant bien sûr la technologie des media queries présente dans CSS.

Test du site

Pour vérifier le référencement et l'accessibilité pour tous les utilisateurs, j'ai utilisé les fonctionnalités Lighthouse de Dev Tools de Google et j'ai obtenu des notes positives.



Architecture du projet

Laravel est un framework MVC (Model-View-Controller) et comme son nom l'indique, il utilise trois concepts :

Modèle (Model) : Le modèle contient les données et interagit avec elles. Il se charge des requêtes CRUD (Create, Read, Update, and Delete).

Vue (View) : La vue permet de montrer l'interface graphique. Dans mon cas, elle consiste en des templates créés en HTML avec du CSS pour le style et Blade pour la gestion des templates.

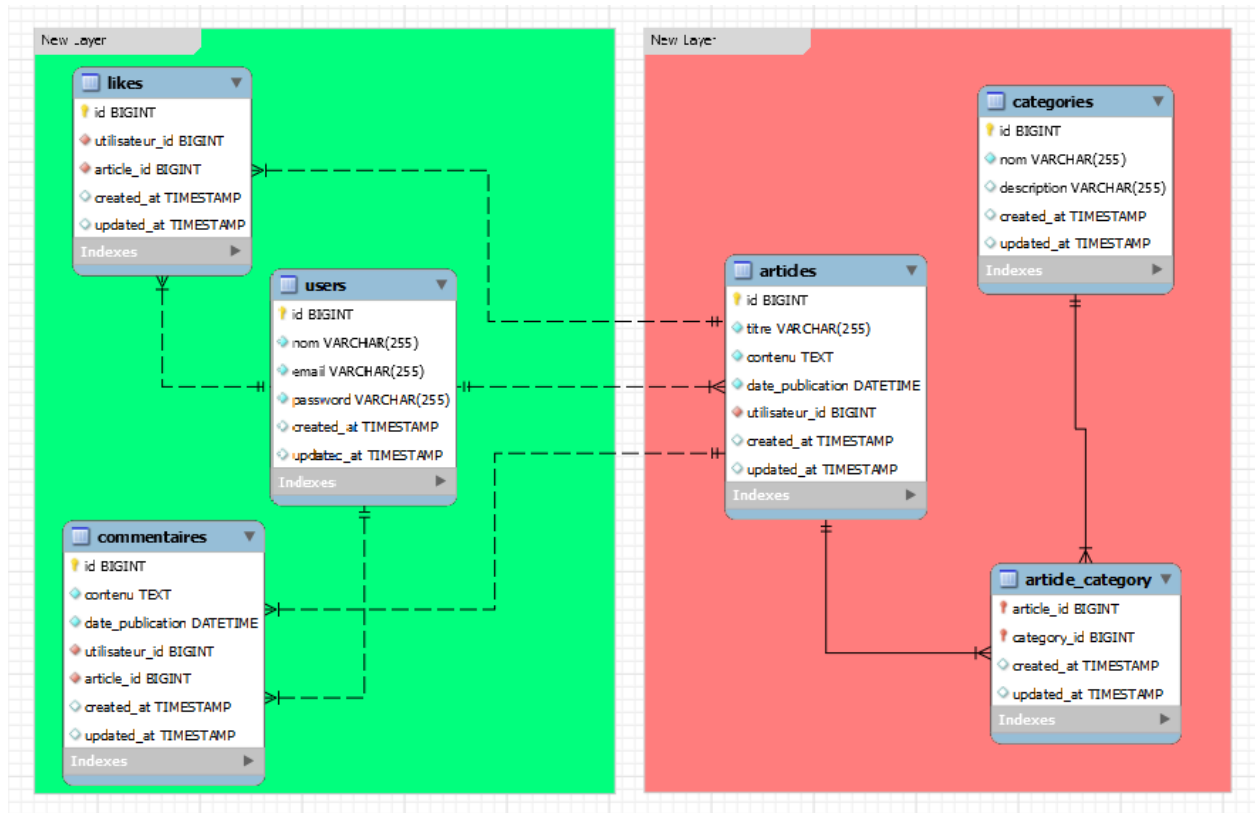
Contrôleur (Controller) : Le contrôleur contient la logique des actions de l'application et se charge de la coordination entre la vue et le modèle.

Par exemple, dans mon projet, lorsqu'un utilisateur fait une requête HTTP, mon côté serveur lit la requête et la recherche dans mes routes qui se trouvent dans le fichier **web.php**. Une fois que la route est trouvée, il voit la fonction qui se trouve dans le contrôleur et peut soit afficher une vue avec des informations de la base de données que le contrôleur a récupérées dans un modèle, soit, par exemple, effacer un utilisateur dans le cas de l'appel d'une route dans le back-office pour effacer un utilisateur.

Base de données

MCD:

Utilisation de mySQL Workbench pour faire le schéma



Après avoir créé le MCD, j'ai pu observer les relations entre les entités. On peut voir une relation Many-to-Many entre les articles et les catégories. Dans ce cas, j'ai créé une table de pivot appelée **article_category**, puisque un article peut avoir plusieurs catégories et une catégorie peut avoir plusieurs articles.

J'ai également identifié des relations plus simples, comme le fait qu'un utilisateur peut mettre plusieurs likes et plusieurs commentaires, mais qu'un commentaire ne peut pas provenir de

plusieurs utilisateurs (One-to-Many). De même, un utilisateur peut créer plusieurs articles.

Créer la base de données:

Pour créer la base de données, j'utilise les migrations de Laravel. Avant tout, je configure mon fichier `.env` (environnement).

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=project-articlehub
DB_USERNAME=root
DB_PASSWORD=
```

Une fois cela fait, je peux lancer la commande `php artisan migrate` pour établir la connexion une première fois et permettre à Laravel de créer les tables prédéfinies nécessaires au bon fonctionnement de l'application. Ensuite, je crée une migration pour chaque table spécifique que je souhaite ajouter à ma base de données.

```
class CreateUsersTable extends Migration
{
    0 references | 0 overrides
    public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->id();
            $table->string('nom');
            $table->string('email')->unique();
            $table->string('password');
            $table->enum('role', ['user', 'admin'])->default('user');
            $table->timestamps();
        });
    }

    0 references | 0 overrides
    public function down()
    {
        Schema::dropIfExists('users');
    }
}
```

```

class CreateArticlesTable extends Migration
{
    0 references | 0 overrides
    public function up()
    {
        Schema::create('articles', function (Blueprint $table) {
            $table->id();
            $table->string('titre');
            $table->text('contenu');
            $table->text('small_description')->nullable();
            $table->dateTime('date_publication')->default(DB::raw('CURRENT_TIMESTAMP'));
            $table->foreignId('utilisateur_id')->constrained('users')->onDelete('cascade');
            $table->timestamps();
        });
    }

    0 references | 0 overrides
    public function down()
    {
        Schema::dropIfExists('articles');
    }
}

```

```

class CreateLikesTable extends Migration
{
    0 references | 0 overrides
    public function up()
    {
        Schema::create('likes', function (Blueprint $table) {
            $table->id();
            $table->foreignId('utilisateur_id')->constrained('users')->onDelete('cascade');
            $table->foreignId('article_id')->constrained('articles')->onDelete('cascade');
            $table->timestamps();
        });
    }

    0 references | 0 overrides
    public function down()
    {
        Schema::dropIfExists('likes');
    }
}

```

```

0 references | 0 implementations
class CreateCommentsTable extends Migration
{
    0 references | 0 overrides
    public function up()
    {
        Schema::create('commentaires', function (Blueprint $table) {
            $table->id();
            $table->text('contenu');
            $table->dateTime('date_publication');
            $table->foreignId('utilisateur_id')->constrained('users')->onDelete('cascade');
            $table->foreignId('article_id')->constrained('articles')->onDelete('cascade');
            $table->timestamps();
        });
    }

    0 references | 0 overrides
    public function down()
    {
        Schema::dropIfExists('commentaires');
    }
}

```

```

0 references | 0 implementations
class CreateCategoriesTable extends Migration
{
    0 references | 0 overrides
    public function up()
    {
        Schema::create('categories', function (Blueprint $table) {
            $table->id();
            $table->string('nom');
            $table->string('description')->nullable();
            $table->timestamps();
        });
    }

    0 references | 0 overrides
    public function down()
    {
        Schema::dropIfExists('categories');
    }
}

```



```

class CreateArticleCategoryTable extends Migration
{
    0 references | 0 overrides
    public function up()
    {
        Schema::create('article_category', function (Blueprint $table) {
            $table->foreignId('article_id')->constrained('articles')->onDelete('cascade');
            $table->foreignId('category_id')->constrained('categories')->onDelete('cascade');
            $table->primary(['article_id', 'category_id']);
            $table->timestamps();
        });
    }

    0 references | 0 overrides
    public function down()
    {
        Schema::dropIfExists('article_category');
    }
}

```

On peut voir l'utilisation de la classe Schema::create de laravel dont la fonction up() de la classe create(nom de ma table) qui étend Migration de laravel et que ce lancera quand je utilisera 'php artisan migrate' dans le terminal de Visual Studio Code, dans chaque fonction up() de cette migration, on voit les caractéristiques de la table que l'on a pu voir dans le MCD .

Avec Laravel, j'ai eu la possibilité de créer des seeders qui me permettent de remplir la base de données une fois celle-ci créée avec les migrations. Ces seeders remplissent la table de données fictives pour tester le bon fonctionnement de l'application. J'ai découvert cette utilité lors de la création de mon projet.

```

0 references | 0 implementations
class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     */
    0 references | 0 overrides
    public function run(): void
    {
        $this->call([
            UserSeeder::class,
            CategorySeeder::class,
            ArticleSeeder::class,
        ]);
    }
}

```

Ici, on voit que DatabaseSeeder est le seeder principal créé par Laravel, suivant la convention de sa documentation. Il appelle différents seeders, par exemple ArticleSeeder, en dernier. Ce dernier est le plus complexe car il a besoin des catégories et des IDs des utilisateurs pour fonctionner correctement.

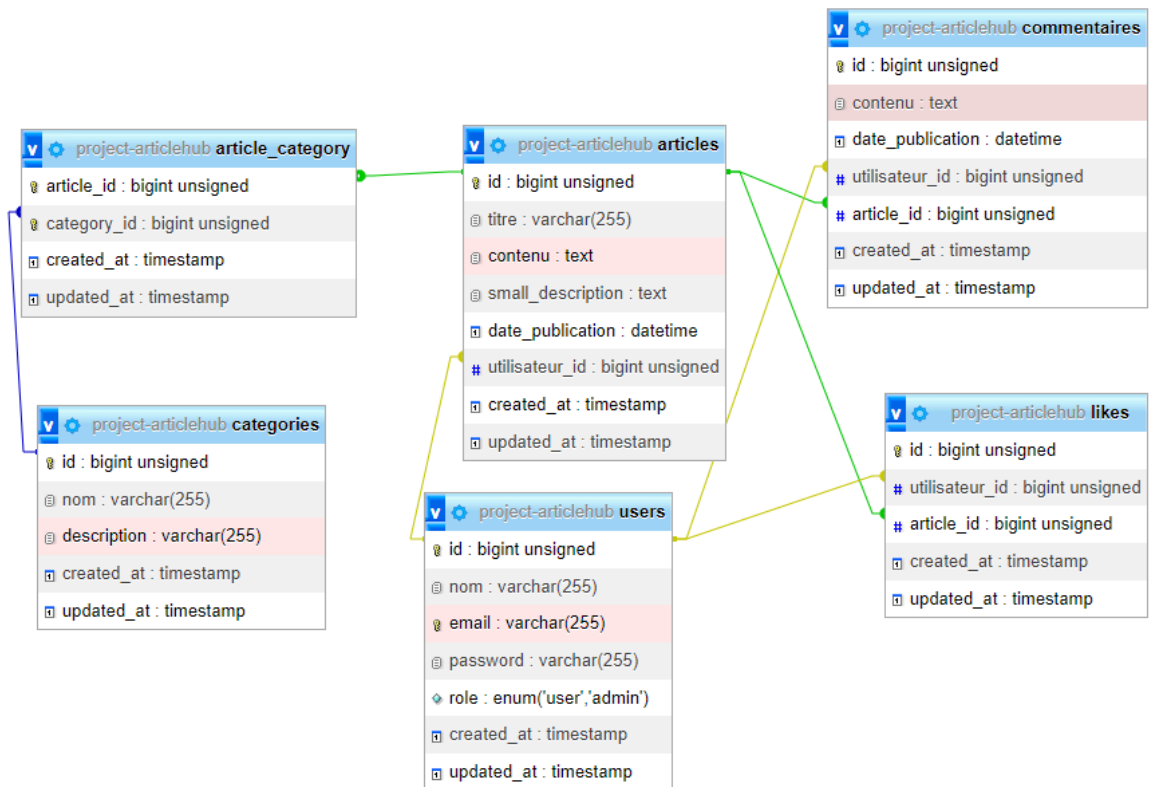
```

0 references | 0 implementations
class ArticleSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    0 references | 0 overrides
    public function run(): void
    {
        Article::factory()->count(20)->create();
    }
}

```

Ici, on voit que le seeder d'articles, lorsqu'on l'appelle, utilise Article::factory pour créer des articles fictifs. Dans ce cas, j'ai la possibilité de spécifier une quantité, et je choisis d'en créer 20.

Avec la facilité de l'ORM Eloquent, j'ai pu créer une base de données complète avec aisance, tout en me permettant d'interagir très rapidement avec les modèles.



Voici ma base de données finale telle qu'elle apparaît dans phpMyAdmin, prête à être utilisée.

Routes

Pour illustrer et voir l'utilité générale de toutes mes route je crée un tableau

Endpoint	Méthode	Fonctionnalité	Contrôleur
/	GET	Afficher la page d'accueil	HomeController
/profile	GET	Afficher le profil de l'utilisateur	ProfileController
/profile/liked	GET	Afficher les articles aimés par l'utilisateur	ProfileController
/profile/articles	GET	Afficher les articles de l'utilisateur	ProfileController
/profile/articles/{id}	DELETE	Supprimer un article de l'utilisateur	ProfileController
/profile/password/change	POST	Changer le mot de passe de l'utilisateur	ProfileController
/admin	GET	Afficher la vue d'administration	AdminController
/admin/users	GET	Afficher la liste des utilisateurs	AdminController
/admin/articles	GET	Afficher la liste des articles	AdminController
/admin/articles/{id}	DELETE	Supprimer un article	AdminController
/admin/users/{id}	DELETE	Supprimer un utilisateur	AdminController
/auth	Resource	Gérer l'authentification (CRUD)	RegistrationController

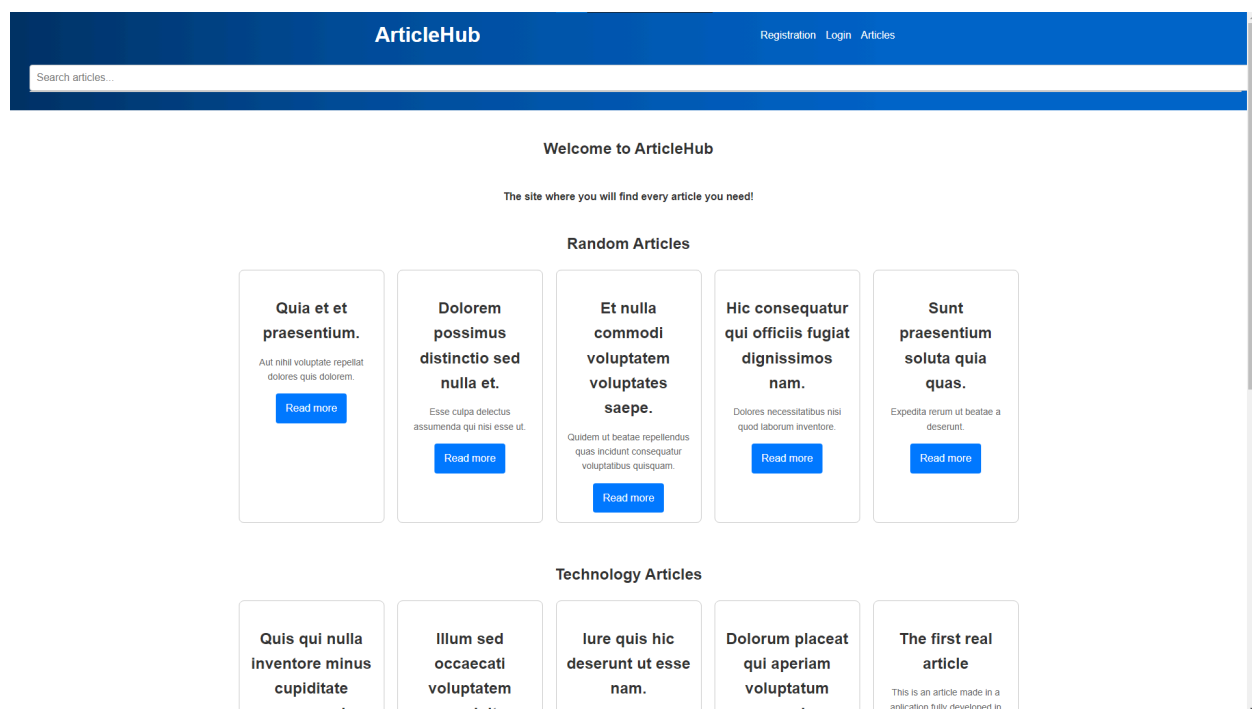
/login	GET	Afficher le formulaire de connexion	AuthController
/login	POST	Gérer la connexion utilisateur	AuthController
/logout	POST	Déconnexion de l'utilisateur	AuthController
/articles	GET	Afficher tous les articles	ArticlesController
/articles/category/{category}	GET	Afficher les articles d'une catégorie spécifique	ArticleController
/articles/create	GET	Afficher le formulaire de création d'article	ArticlesController
/articles/create	POST	Créer un nouvel article	ArticlesController
/article/{id}	GET	Afficher un article spécifique	ArticleController
/article/{id}/like	POST	Gérer la fonction "J'aime" d'un article	ArticleController
/article/{id}/comments	POST	Créer un commentaire sur un article	CommentController
/search	GET	Effectuer une recherche d'articles	ArticleController

Dans le tableau, nous pouvons voir quatre éléments : l'endpoint à la fin de l'URL, la méthode HTTP utilisée, l'utilité principale de chaque route, et le contrôleur associé à chaque route.

Key Features et Fonctionnalités

Dans mon projet je développe beaucoup de fonctionnalités et je vais expliquer les plus importantes les Key Feature plus quelques fonctionnalités de mon application. En commençant par la lecture et création de articles

Lecture d'articles



Lorsque l'on arrive sur le site, on voit des articles aléatoires ainsi que des articles provenant d'une catégorie choisie au hasard qui correspond à ce code:

Ici la route

```
Route::get('/', [HomeController::class, 'viewHome'])->name('home');
```

Et ici le controller qui se charge d'envoyer les articles

```
class HomeController extends Controller
{
    1 reference | 0 overrides
    public function viewHome() {
        $randomArticles = Article::inRandomOrder()->take(5)->get();

        $category = Category::inRandomOrder()->first();

        if ($category) {
            $categoryArticles = $category->articles()->take(5)->get();
        } else {
            $categoryArticles = collect();
        }

        $newestArticles = Article::orderBy('date_publication', 'desc')->take(5)->get();

        return view('home', compact('randomArticles', 'category', 'categoryArticles', 'newestArticles'));
    }
}
```

On peut voir plusieurs étapes : tout d'abord, je sélectionne 5 articles aléatoires grâce à mon modèle Article et à l'utilisation de l'ORM Eloquent. Je fais de même pour choisir une catégorie aléatoire dans ma base de données. Si une catégorie est trouvée, je sélectionne alors 5 articles appartenant à cette catégorie. Une fois cela terminé, je renvoie la vue avec les articles et les

données correspondantes au template.

```
@extends('layouts.main')

@section('title', 'Home')

@section('content')
<h2 class="ah-welcome-heading">Welcome to ArticleHub</h2>
<h4 class="ah-subheading">The site where you will find every article you need!</h4>

<h2 class="ah-section-heading">Random Articles</h2>
<div class="ah-article-grid">
    @foreach ($randomArticles as $article)
        <div class="ah-article-item">
            @include('articles.article_card', ['article' => $article])
        </div>
    @endforeach
</div>

<h2 class="ah-section-heading">{{ $category->nom }} Articles</h2>
<div class="ah-article-grid">
    @foreach ($categoryArticles as $article)
        <div class="ah-article-item">
            @include('articles.article_card', ['article' => $article])
        </div>
    @endforeach
</div>

<h2 class="ah-section-heading">Newest Articles</h2>
<div class="ah-article-grid">
    @foreach ($newestArticles as $article)
        <div class="ah-article-item">
            @include('articles.article_card', ['article' => $article])
        </div>
    @endforeach
</div>

<p class="ah-info-text">You want to add your article, login or register if you still don't have an account and do it!</p>
@endsection
```

Dans mon template, on peut voir que j'ai la capacité, grâce à Blade intégré dans Laravel, de faire une boucle foreach sur les articles que j'ai renvoyés depuis mon contrôleur.

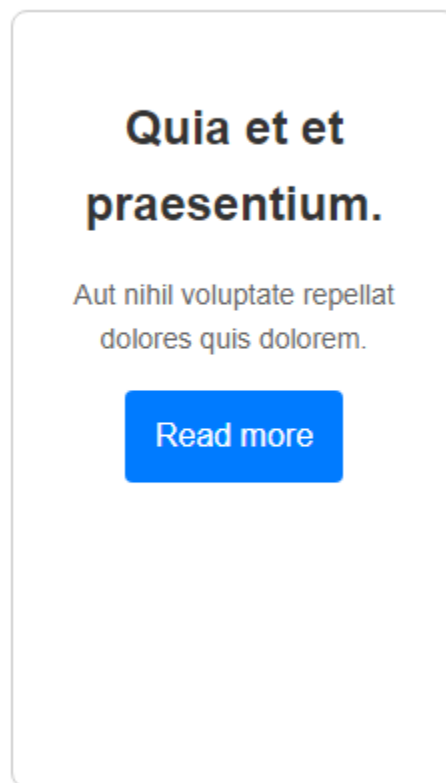
```
return view('home', compact('randomArticles', 'category', 'categoryArticles', 'newestArticles'));
```

On constate que compact se charge de passer les variables contenant les articles à la vue, ce qui permet ultérieurement de créer des cartes pour ces produits. Les cartes sont conçues dans un fichier article_card.blade.php.

```
<div class="ah-article-card">
    <h3 class="ah-article-title">{{ $article->titre }}</h3>
    <p class="ah-article-description">{{ $article->small_description }}</p>
    <a href="{{ route('article.show', ['id' => $article->id]) }}" class="ah-read-more-link">Read more</a>
</div>
```


Cela me permettra de créer plusieurs cartes directement dans ma vue grâce à Blade, qui offre un dynamisme permettant d'utiliser les données envoyées par le back-end.

Voici le résultat d'une carte :



Dans cette carte, on voit le titre, une brève description de l'article (dans ce cas, du Lorem Ipsum), et un bouton 'Lire la suite' qui renvoie à la page de lecture.

```
<a href="{{ route('article.show', ['id' => $article->id]) }}" class="ah-read-more-link">Read more</a>
```

```
Route::get('article/{id}', [ArticleController::class, 'showArticle'])->name('article.show');
```

Dans la route, je passe en argument l'ID de l'article, ce qui est nécessaire pour accéder à la page de lecture. Cela permet déjà de lire un article sur mon site, démontrant ainsi son accessibilité.

Sinon, il est possible d'accéder à la page de tous les articles depuis le header.

ArticleHub

[Registration](#) [Login](#) [Articles](#)

```
@endif  
<a href="/articles">Articles</a>  
</nav>
```

```
Route::get('articles' , [ArticlesController::class, 'showAllArticlesView'])->name('articles.index');
```

Here are all the articles

Sort by Category: Sort by:

test
test

The first real article
This is an article made in a application
fully developed in laravel

**Quis qui nulla inventore
minus cupiditate
assumenda dolorem.**
Quod officia laboriosam voluptatibus aut
ea.

**Et nulla commodi
voluptatem voluptates
saepe.**
Quidem ut beatae repellendus quas
incidunt consequatur voluptatibus
quisquam.

**Quis optio commodi vero
earum.**
Ipsa iusto ut dolores voluptatum magni
qui.

**Illum sed occaecati
voluptatem suscipit.**
Est officis commodi eaque officis omnis
ex perspiciatis.

Illo sit suscipit fuga vel qui.
Ea vero aliquid voluptas quaerat.

**Asperiores tenetur
quibusdam sit
exercitationem.**
Et ut aperiam doloremque dolorem.

**Nostrum vero omnis non
temporibus eum.**
Nihil rem nobis quo suscipit nisi eligendi.

**Illo omnis excepturi ut
numquam dolores.**
Tenetur dolorem sit aut dignissimos ut
alias et.

**Ea non ut sint ut atque
neque placeat molestias.**
Quasi labore incidunt et impedit
voluptatibus.

**Omnis corrupti eos maxime
facilis.**
Molestias tempore quos aut rerum
perspiciatis nam molestiae.

Ici, on voit tous les articles avec, par défaut, un filtre pour toutes les catégories et les plus récents. De cette manière, le contrôleur que nous avons vu dans la route rend les articles.

Page 34

```
$query = Article::query();
```

Ici, le contrôleur prépare la requête pour récupérer les articles.

```
$sort = $request->query('sort');
switch ($sort) {
    case 'oldest':
        $query->orderBy('date_publication', 'asc');
        break;
    case 'most-liked':
        $query->withCount('likes')->orderByDesc('likes_count');
        break;
    default:
        $query->orderBy('date_publication', 'desc');
        break;
}
```

Ici, on voit que le contrôleur vérifie si un filtre de tri (sort) est présent dans la requête. Sinon, par défaut, les articles sont triés par date de publication de manière descendante.

```
$articles = $query->paginate(12);
```

À la fin, le contrôleur finalise la requête au modèle en créant une pagination de 12 articles, puis renvoie ces articles à la vue.

```
return view('articles.index', [
    'articles' => $articles,
    'categories' => $categories,
    'selectedCategory' => $request->category ?? null,
    'sort' => $sort,
]);
```

```

@if ($articles->isEmpty())
    <p>No articles found.</p>
@else
    <div class="article-grid">
        @foreach ($articles as $article)
            <div class="article-item">
                <a href="{{ route('article.show', ['id' => $article->id]) }}">
                    <div class="article-content">
                        <h3>{{ $article->titre }}</h3>
                        <p>{{ $article->small_description }}</p>
                    </div>
                </a>
            </div>
        @endforeach
    </div>
    <div class="pagination">
        {{ $articles->links() }}
    </div>
@endif
@endsection

```

Dans le template, j'utilise une boucle foreach pour afficher chaque article, avec un lien permettant d'accéder à la page de lecture de chaque article.

```
Route::get('article/{id}', [ArticleController::class, 'showArticle'])->name('article.show');
```

Voici la route qui gère le lien vers la page de lecture des articles et prend en argument l'ID de l'article, nécessaire pour accéder à cette page. Voici comment le contrôleur ArticleController affiche la vue avec l'article.

```

1 reference | 0 overrides
public function showArticle($id){
    $article = Article::find($id);

    return view('articles.show', ['article' => $article]);
}

```

La fonction est simple : elle nécessite un ID et renvoie l'article correspondant à la vue.

```

@extends('layouts.main')

@section('title', $article->titre)

@section('content')
<div class="article-container">
    <div class="article-header">
        <h1>{{ $article->titre }}</h1>
        <div class="article-meta">
            <p>Published on: {{ $article->date_publication }}</p>
            <p>Author: {{ $article->user->nom }}</p>
        </div>
    </div>

    <p>{{ $article->small_description }}</p>
    <p>{{ $article->contenu }}</p>

    <!-- Display categories -->
    <ul>
        @foreach ($article->categories as $category)
            <li>{{ $category->nom }}</li>
        @endforeach
    </ul>

    <div class="like-section">...
    </div>

    <hr>

    <div class="comments-section">...
    </div>
</div>
@endsection

```

Voici le template de la page de lecture. Comme toujours, Blade se charge d'afficher les informations de l'article dynamiquement grâce à la variable \$article, qui est passée en tant qu'objet puisque c'est un modèle de la base de données. Voici le rendu

The first real article

Published on: 2024-07-18

10:34:56

Author: esteban

This is an article made in a application fully developed in laravel

The description is way to long for a description

Technology

Science

Business



0 likes

Comments

No comments yet

Add a comment

Please [login](#) to add a comment.

On peut voir le titre, la description, la date de création, l'utilisateur, et les catégories attribuées à l'article.

Creation d'articles

Maintenant, nous allons voir la création d'un article. Pour cela, il sera nécessaire de se connecter ou de s'enregistrer si ce n'est pas déjà fait.

```

@if (session('userId'))
    <a href="/profile">Profile</a>
@else
    <a href="/auth/create">Registration</a>
    <a href="/login">Login</a>
@endif

```

Ici, on voit que dans le header, il y a une condition qui vérifie si l'utilisateur est connecté, en utilisant une variable de session que je crée lorsque l'utilisateur se connecte. Voici les deux formulaires:

```

<form action="{{ route('auth.store') }}" method="POST" class="form-create">
    @csrf
    <div class="form-group-create">
        <label for="username" class="form-label-create">Username:</label>
        <input type="text" id="username" name="username" class="form-control-create" required>
    </div>
    <div class="form-group-create">
        <label for="email" class="form-label-create">Email:</label>
        <input type="email" id="email" name="email" class="form-control-create" required>
    </div>
    <div class="form-group-create">
        <label for="password" class="form-label-create">Password:</label>
        <input type="password" id="password" name="password" class="form-control-create" required>
    </div>
    <div class="form-group-create">
        <label for="password_confirmation" class="form-label-create">Confirm Password:</label>
        <input type="password" id="password_confirmation" name="password_confirmation" class="form-control-create" required>
    </div>
    <button type="submit" class="btn btn-primary btn-register-create">Register</button>
</form>

```

Ici, le formulaire d'enregistrement utilise la fonction @csrf de Blade, qui ajoute un token CSRF au formulaire pour se protéger contre ce type d'attaque, comme recommandé dans la documentation de Laravel. Une fois le formulaire rempli, il utilise la route auth.store, qui sert à créer un utilisateur.

```

public function store(Request $request)
{
    $request->validate([
        'nom' => 'required|string|max:255',
        'email' => 'required|string|email|max:255|unique:users',
        'password' => 'required|string|min:8|confirmed',
    ]);

    $user = User::create([
        'nom' => $request->input('nom'),
        'email' => $request->input('email'),
        'password' => bcrypt($request->input('password')),
    ]);

    return redirect()->route('auth.login')->with('success', 'Registration successful!');
}

```

Comme on peut le voir dans le code, la fonction **store** de la route **auth.store** valide les données avant de les insérer dans un modèle, ce qui les enregistrera dans la base de données. Le mot de passe est crypté à l'aide de la fonction **bcrypt** de Laravel, qui se charge de crypter les mots de passe et de vérifier les mots de passe durant la connexion en les comparant avec ceux cryptés dans la base de données. Une fois l'utilisateur créé, il est redirigé vers la route de la vue du formulaire de connexion.

```
@section('content')
<div class="login-form">
    <h1 class="form-heading-login">Login</h1>
    @if (session('success')) ...
    @endif
    @if ($errors->any()) ...
    @endif
    <form action="{{ route('login') }}" method="POST" class="form-login">
        @csrf
        <div class="form-group-login">
            <label for="email" class="form-label-login">Email:</label>
            <input type="email" id="email" name="email" class="form-control-login" value="{{ old('email') }}" required>
        </div>
        <div class="form-group-login">
            <label for="password" class="form-label-login">Password:</label>
            <input type="password" id="password" name="password" class="form-control-login" required>
        </div>
        <button type="submit" class="btn btn-primary btn-login">Login</button>
    </form>
</div>
@endsection
```

De même, le formulaire utilise la route **login** pour vérifier les données saisies.

```
public function login(Request $request) {
    $credentials = $request->only('email','password');

    if(Auth::attempt($credentials)) {
        $request->session()->put('userId', Auth::user()->id);
        $request->session()->put('userEmail', Auth::user()->email);

        return redirect()->intended('/');
    }

    return back()->withInput()->withErrors(['email'=> 'These credentials do not match our records.']);
}
```

Ici on prend le email et mot de passe et on le passe dans la variable \$credentials puis dans la condition

Ici, on prend l'email et le mot de passe, puis on les passe dans la variable **\$credentials**. Ensuite, dans la condition **if (Auth::attempt(\$credentials))**, la fonction retourne **true** si l'email et le mot de passe correspondent à un utilisateur dans la base de données. Si les informations sont correctes, je crée deux variables de session, **userId** et **userEmail**, que j'utilise pour déterminer si l'utilisateur est connecté.

```
@if (session('userId'))
    <a href="/profile">Profile</a>
@else
    <a href="/auth/create">Registration</a>
    <a href="/login">Login</a>
@endif
```

Comme je l'ai déjà montré auparavant, ici on peut voir que l'utilisateur voit désormais 'Profile' au lieu de 'Registration' et 'Login'.

Hello esteban

Logout

Back-office

Change Password

Current Password:

New Password:

Confirm New Password:

Change Password

Make an article

View Liked Articles

View Created Articles

Voici ce que l'utilisateur va voir dans profile mais ce que nous intéresse ce le bouton **“make an article”**

```
class= profile-links >  
<li><a href="{{ route('articleCreate') }}">Make an article</a></li>  
<li><a href="{{ route('profile.liked') }}">View Liked Articles</a></li>
```

Qui renvoie vers la route de la vue **“articleCreate”**

```

<form action="{{ route('createArticle') }}" method="post">
    @csrf
    <div>
        <label for="titre">Title:</label>
        <input type="text" name="titre" id="titre" value="{{ old('titre') }}" required>
    </div>
    <div>
        <label for="small_description">Small description:</label>
        <textarea name="small_description" id="small_description">{{ old('small_description') }}</textarea>
    </div>
    <div>
        <label for="contenu">Content:</label>
        <textarea name="contenu" id="contenu">{{ old('contenu') }}</textarea>
    </div>
    <div>
        <label>Categories:</label>
        @foreach ($categories as $category)
            <div>
                <input type="checkbox" name="categories[]" id="category{{ $category->id }}" value="{{ $category->id }}">
                <label for="category{{ $category->id }}">{{ $category->nom }}</label>
            </div>
        @endforeach
    </div>
    <input type="hidden" name="utilisateur_id" id="utilisateur_id" value="{{ session('userId') }}">
    <button type="submit">Create</button>
</form>
@endsection

```

Voici le formulaire qui permet de créer un article et qui, entre autres, affiche de manière dynamique les catégories présentes dans la base de données. Une fois que l'utilisateur a créé son article et cliqué sur le bouton **'Create'**, le formulaire utilise la route **('createArticle')**.

```
Route::post('articles/create', [ArticlesController::class, 'createArticle'])->name('createArticle');
```

Dans cette route on peut voir que je utilise la methode **createArticle** de **ArticlesController** la voici:

```

public function createArticle(Request $request) {
    $request->validate([
        'titre' => 'required|unique:articles,titre',
        'contenu' => 'required',
        'small_description' => 'nullable|string|max:255',
        'categories' => 'required|array',
        'categories,*' => 'exists:categories,id',
        'utilisateur_id' => 'required|exists:users,id',
    ]);

    $article = Article::create($request->only(['titre', 'contenu', 'small_description', 'utilisateur_id', 'date_publication']));
    $article->categories()->attach($request->categories);

    return redirect()->route('articles.index')->with('success', 'Article created successfully.');
```

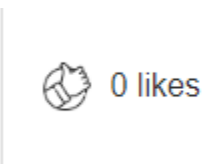
Ici, on peut voir qu'avec `$request->validate`, je vérifie les données

passées dans le formulaire en utilisant la configuration nécessaire pour éviter les injections SQL malveillantes. Ensuite, avec `Article::create`, je crée un article grâce à l'ORM Eloquent. Avant de rediriger l'utilisateur, je crée les liens pour les catégories que l'utilisateur souhaite attribuer à son article en utilisant une méthode de mon modèle `Article`. Enfin, je renvoie l'utilisateur à la page des articles pour que l'article soit prêt à être lu.

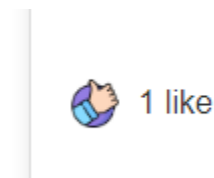
Fonctionnalité pour 'liker' un article

Une des manières dont j'ai permis aux utilisateurs d'interagir avec les articles est de pouvoir 'liker' ou cliquer sur 'J'aime' un article. Pour cela, j'ai utilisé JavaScript, le dynamisme de Blade, et une route de mon application.

Voici comment il apparaît dans la vue :



Voici comment cela se présente lorsque l'on clique sur l'image de "j'aime":



Pour cela je suivi cette logique:

```
<div class="like-section">
  id }}">
  <p>{{ $article->likes->count() }} {{ Str::plural('like', $article->likes->count()) }}</p>
</div>
```

Voici comment cela fonctionne : lorsque l'image se charge pour la première fois, une fonction appelée `userHasLiked()` dans le

modèle des articles est appelée. Cette fonction vérifie, à l'aide de l'ID de l'utilisateur (qui est stocké dans une variable de session), si l'utilisateur a déjà aimé cet article. Si c'est le cas, l'image sera affichée en couleur ; sinon, elle sera affichée en version non colorisée.

```
public function likes()
{
    return $this->hasMany(Like::class, 'article_id');
}

0 references | 0 overrides
public function userHasLiked($userId)
{
    return $this->likes()->where('utilisateur_id', $userId)->exists();
}
```

Voici les deux méthodes que j'utilise pour vérifier si l'utilisateur a aimé cet article. Tout d'abord, avec la méthode likes(), je récupère tous les likes associés à l'article. Ensuite, la méthode userHasLiked() vérifie, parmi les likes récupérés, si l'un d'eux correspond à l'ID de l'utilisateur. Si c'est le cas, la méthode exists() renvoie True; sinon, elle renvoie False.

Maintenant, je vais vous montrer le code JavaScript qui permet à l'utilisateur de liker et de dislike un article de manière dynamique.

```
const csrfToken = document.querySelector('meta[name="csrf-token"]');
if (!csrfToken) {
    console.error('CSRF token not found');
    return;
}

const token = csrfToken.getAttribute('content');
```

Tout d'abord, pour permettre au JavaScript d'interagir avec les données de mon back-end, je lui donne accès au token CSRF de

l'utilisateur, qui se trouve dans la balise `<head>` du layout de mon application.

```
<meta name="csrf-token" content="{{ csrf_token() }}">
```

Une fois l'accès accordé

```
document.querySelectorAll('.like-button').forEach(button => {
  button.addEventListener('click', function() {
    const articleId = this.dataset.articleId;
    if (!articleId) {
      console.error('Article ID not found');
      return;
    }

    fetch(`/article/${articleId}/like`, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
        'X-CSRF-TOKEN': token
      },
      body: JSON.stringify({})
    })
    .then(response => response.json())
    .then(data => {
      if (data.status === 'liked') {
        // console.log('liked');
        this.src = window.likeFilledUrl;
      } else {
        // console.log('unliked');
        this.src = window.likeUrl;
      }
    })
    .catch(error => console.error('Error:', error));
  });
});
```

Il faudra effectuer une requête fetch vers la route `/article/{id}/like` en incluant le token pour que le back-end accepte la requête. Le serveur répondra par `liked` ou `unliked`. Si la réponse est `liked`, cela signifie que l'article a été aimé, et nous changerons l'image pour la version colorisée. En revanche, si la réponse est `unliked`, cela

signifie que l'article a été disliké, et nous reviendrons à la version non colorisée de l'image.

Voici la route `/article/{id}/like`

```
public function like($id)
{
    $article = Article::findOrFail($id);
    $userId = session('userId');

    if ($userId) {
        if ($article->likes()->where('utilisateur_id', $userId)->exists()) {

            $article->likes()->where('utilisateur_id', $userId)->delete();
            return response()->json(['status' => 'unliked']);
        } else {
            $article->likes()->create(['utilisateur_id' => $userId]);
            return response()->json(['status' => 'liked']);
        }
    } else {
        return response()->json(['status' => 'unliked']);
    }
}
```

Voici comment fonctionne la méthode associée à cette route : elle prend en argument l'ID de l'article, puis le recherche dans la base de données. Ensuite, elle vérifie l'ID de l'utilisateur. Si l'utilisateur n'est pas connecté, la méthode retourne `unliked`, ce qui fait que l'image reste en version non colorisée, car un utilisateur non connecté ne peut pas utiliser cette fonctionnalité.

En revanche, si l'utilisateur est connecté, la méthode utilise les fonctions du modèle `Article` pour vérifier si l'utilisateur a déjà aimé cet article. Si c'est le cas, cela signifie qu'il a disliké l'article, et la méthode retourne `unliked`. Dans le cas contraire, la méthode retourne `liked`, indiquant que l'utilisateur vient de liker l'article.

Fonctionnalité barre de recherche

Pour cette fonctionnalité, j'ai également utilisé JavaScript en écoutant chaque touche du clavier dans le champ de saisie pour effectuer la recherche.

```
<div class="search-container">
  <input type="text" id="search" placeholder="Search articles..." onkeyup="searchArticles()">
  <ul id="search-results"></ul>
</div>
```

Voici l'élément `<input>` associé à la fonction `searchArticles()`, suivi de la balise `` qui affichera une liste de tous les produits trouvés.

```
async function searchArticles() {
  const query = document.getElementById('search').value;
  const searchResults = document.getElementById('search-results');
  searchResults.innerHTML = '';

  if (query.length < 2) {
    return;
  }

  const response = await fetch(`/search?query=${query}`);
  const articles = await response.json();

  articles.forEach(article => {
    const li = document.createElement('li');
    li.innerHTML = `\${article.titre}</a>`;
    searchResults.appendChild\(li\);
  }\);
}
```

Voici comment cela fonctionne : la fonction commence à rechercher des produits uniquement lorsque l'entrée contient au moins 2 caractères. Si l'entrée est plus courte, tous les articles de la base de données sont affichés. Lorsque l'entrée atteint 2 caractères, un `fetch` est effectué vers la route `/search`, qui est responsable de renvoyer les produits correspondants. Une fois la réponse du serveur reçue, la fonction crée des éléments `` pour

la liste , chacun contenant un lien permettant de consulter l'article.

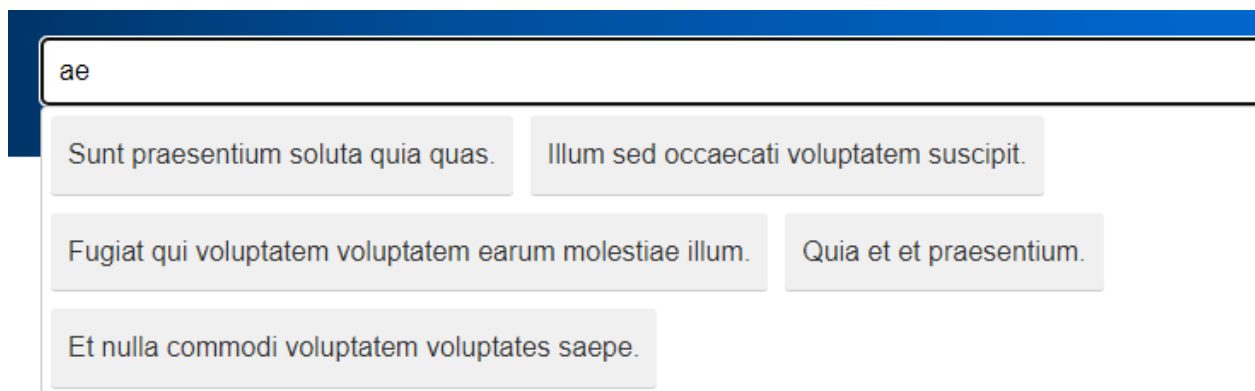
```
public function search(Request $request)
{
    $query = $request->input('query');

    if ($query) {
        $articles = Article::where('titre', 'LIKE', "%{$query}%")->get();
        return response()->json($articles);
    }

    return response()->json([]);
}
```

Voici la méthode search, qui permet de rechercher des articles en fonction du texte saisi dans l'input. Elle effectue une recherche de titres similaires parmi tous les articles de la base de données, puis renvoie les résultats au format JSON au JavaScript. Celui-ci utilise ces données pour construire les éléments avec les liens correspondants.

Voici le resultat:



The screenshot shows a web interface with a search bar at the top containing the text 'ae'. Below the search bar, there is a list of search results displayed as light gray boxes. The results are Latin phrases: 'Sunt praesentium soluta quia quas.', 'Illum sed occaecati voluptatem suscipit.', 'Fugiat qui voluptatem voluptatem earum molestiae illum.', 'Quia et et praesentium.', and 'Et nulla commodi voluptatem voluptates saepe.'

On observe qu'en recherchant 'ae', la méthode trouve des articles dont le titre contient 'ae' et crée des liens permettant de les lire.

Conclusion

Ce projet m'a permis d'apprendre énormément sur la gestion et l'organisation d'un projet, depuis sa conception initiale jusqu'à sa réalisation finale. J'ai appris à élaborer une maquette et à conceptualiser la base de données à l'avance, en comprenant mieux les relations entre les tables et leurs attributs. J'ai eu l'opportunité de réaliser mon premier projet complet en utilisant le framework Laravel et son ORM Eloquent. Grâce à la documentation fournie, j'ai acquis de nombreuses compétences et une compréhension approfondie de l'architecture MVC. J'ai également appris à créer des routes et, avec Blade, à assurer l'interaction entre le front-end et le back-end tout en garantissant la sécurité du site, notamment en cryptant les données des utilisateurs et en les protégeant efficacement.

L'utilisation de ce framework pour un projet individuel a éveillé en moi le désir de pousser mes compétences plus loin et de collaborer sur des projets d'envergure en groupe. J'aspire à continuer à me former dans ce métier qui me passionne profondément.