Université Abdelmalek Essaâdi
Faculté des Sciences et Techniques de Tanger
Département : Génie Informatique

Année universitaire 2024-2025
Filière : Licence IDAI
Prof. Badr-Eddine Boudriki Semlali

RAPPORT DE PROJET TECH HORIZONS :

DÉVELOPPEMENT WEB

Créer une application Web, sous PHP/MySQL avec framework laravel , pour gérer un magazine en ligne (Tech Horizons)

Encadré par:

PR.M'hamed AIT KBIR

PR.Yasyn EL YUSUFI

préparé par:

Ahadri Botaina
Afailal Trebak Ghizlane
Elhouaoui Hafsa
Harrak Fatima Zahrae

I. Introduction

Dans un monde en constante évolution technologique, l'accès à une information fiable et ciblée est devenu essentiel. C'est dans ce contexte que le projet **Tech Horizons** a été conçu. Ce projet consiste à développer une application web sous PHP/MySQL avec le framework **Laravel**, destinée à gérer un magazine en ligne explorant les innovations technologiques les plus marquantes, tout en mettant en lumière leurs enjeux éthiques et sociétaux.

L'objectif principal de cette application est de fournir une plateforme intuitive et sécurisée permettant aux utilisateurs de consulter des articles liés à leurs centres d'intérêt, de gérer leur historique de navigation, de proposer des articles à publier, et d'interagir avec un contenu riche et diversifié. L'application s'adresse à quatre types d'utilisateurs : les **invités**, les **abonnés**, les **responsables de thèmes** et les **éditeurs**, chacun disposant de permissions et de fonctionnalités spécifiques.

Ce projet a été l'occasion de mettre en pratique des compétences techniques avancées en développement web, notamment la gestion des utilisateurs, la sécurisation des données, la création d'une interface utilisateur personnalisée sans framework CSS, et l'implémentation d'un système de recommandation basé sur l'historique de navigation. À travers ce compte rendu, nous présenterons les objectifs, les technologies utilisées, les fonctionnalités implémentées, les défis rencontrés, ainsi que les solutions apportées pour aboutir à une application robuste et performante.

1. Objectifs du projet

- Gestion des utilisateurs : Quatre types d'utilisateurs (Invité, Abonné, Responsable de thème, Éditeur) avec des permissions spécifiques.
- Gestion des articles : Consultation, proposition, publication et notation des articles.

- Gestion des thèmes : Abonnement/désabonnement des utilisateurs à des thèmes spécifiques.
- Historique de navigation : Suivi des articles consultés par les utilisateurs.
- Recommandations : Proposer des articles en fonction des centres d'intérêt et de l'historique de l'utilisateur.
- Statistiques : Fournir des statistiques aux responsables de thèmes et aux éditeurs.

2. Technologies utilisées

- Backend : PHP, Laravel, MySQL.
- Frontend: HTML, CSS, JavaScript (sans jQuery ni framework CSS comme Bootstrap).
- Outils : Composer, npm, Vite.
- Authentification: Système d'authentification intégré de Laravel.
- Gestion des dépendances : Composer pour PHP, npm pour JavaScript.

3. Structure du projet

a. Dossiers principaux

- app/ : Contient les contrôleurs, modèles, providers et exceptions.
- database/ : Contient les migrations, seeders et factories.
- resources/ : Contient les vues (Blade), les assets (CSS, JS) et les langues.
- routes/: Contient les routes de l'application.
- public/: Contient les assets compilés (CSS, JS) et le point d'entrée de l'application

b. Fichiers clés

- app/Http/Controllers/ : Contient les contrôleurs (ex: ArticleController, SubscriptionController).
- app/Models/ : Contient les modèles (ex: User, Article, Theme).
- database/migrations/: Contient les fichiers de migration pour créer les tables de la base de données.

- resources/views/: Contient les vues Blade (ex: articles/index.blade.php, subscriptions/index.blade.php).
- routes/web.php : Contient les routes de l'application.

4. Fonctionnalités implémentées

a. Gestion des utilisateurs

- Inscription et connexion : Utilisation du système d'authentification de Laravel.
- Rôles et permissions :
  - Invité : Consulter les thèmes et les numéros publics.
  - Abonné: Gérer les abonnements,consulter l'historique, proposer des articles.
- Responsable de thème : Gérer les articles et les abonnements de son thème.
- Éditeur : Gérer les numéros, les articles, les abonnés et les responsables de thèmes.

b. Gestion des articles

- Consultation : Les utilisateurs peuvent consulter les articles liés à leurs thèmes d'abonnement.
- Proposition : Les abonnés peuvent proposer des articles pour publication.
- Notation : Les utilisateurs peuvent noter les articles de 1 à 5 étoiles.

c. Gestion des thèmes

- Abonnement/désabonnement : Les utilisateurs peuvent s'abonner ou se désabonner à des thèmes.
- Recommandations : Les articles sont recommandés en fonction des centres d'intérêt de l'utilisateur.

d. Historique de navigation

- Suivi: Les articles consultés par l'utilisateur sont enregistrés.
- Filtrage: L'utilisateur peut filtrer son historique par date ou thème.

e. Statistiques

- Tableau de bord: Les responsables de thèmes et les éditeurs peuvent consulter des statistiques (ex: nombre d'abonnés, articles les plus lus).

II. Laravel

Laravel est un framework PHP qui facilite le développement d'applications web en fournissant une structure robuste et des outils puissants.

1. Installation de Laravel

➢ Laravel est installé via Composer, le gestionnaire de dépendances PHP.

➢ Cela permet de démarrer rapidement un projet structuré avec toutes les dépendances nécessaires.

❖ Commande:

```
composer create-project laravel/laravel NomDeProjet
```

2. Installation de Vite

➢ Vite est un outil de build moderne pour les assets frontend (CSS, JavaScript).

➢ Il optimise le chargement des ressources et améliore les performances de l'application.

❖ Commande:

```
npm install
```

3. Base de Données et Modèles

- **Migrations**

➢ Les migrations sont des fichiers PHP qui définissent la structure de la base de données.

➢ Elles permettent de versionner la base de données et de la synchroniser entre les environnements de développement.

❖ Commande:

```
php artisan make:migration create_NomDuTableau_table
```

- **Modèles**

➢ Les modèles représentent les entités de l'application (ex: User, Article).

➢ Ils permettent d'interagir avec la base de données via Eloquent ORM, simplifiant les opérations CRUD.

❖ Commande:

```
php artisan make:model NomDuModele
```

- **Relations Eloquent**

➢ Les relations définissent les liens entre les modèles (ex: un utilisateur a plusieurs articles).

➢ Elles simplifient les requêtes complexes et améliorent la lisibilité du code.

4. Authentification et Autorisation

- **Authentification**

➢ Laravel fournit un système d'authentification prêt à l'emploi.

➢ Il sécurise l'accès à l'application et gère les utilisateurs connectés.

❖ Commande:

```
php artisan ui bootstrap --auth
```

- **Middlewares**

➢ Les middlewares interceptent les requêtes HTTP pour effectuer des actions spécifiques (ex: vérifier les rôles).

➢ Ils permettent de contrôler l'accès aux routes en fonction des permissions des utilisateurs.

❖ Commande:

```
php artisan make:middleware NomMiddleware
```

5. Routes

- **Définition des Routes**

➢ Les routes définissent les points d'entrée de l'application (URLs).

➢ Elles relient les requêtes HTTP aux contrôleurs et actions appropriés.

- **Route::resource**

➢ Génère automatiquement les routes CRUD pour une ressource.

➢ Simplifie la gestion des routes pour les opérations de base.

6. Contrôleurs

- **Création des Contrôleurs**

➢ Les contrôleurs gèrent la logique métier de l'application.

➢ Ils séparent la logique de l'interface utilisateur, améliorant la maintenabilité.

❖ Commande:

```
php artisan make:controller NomController
```

- **Méthodes des Contrôleurs**

➢ Les méthodes des contrôleurs gèrent les actions spécifiques (ex: afficher, créer, mettre à jour).

➢ Elles centralisent la logique pour chaque action, rendant le code plus organisé.

❖ Exemple:

```
public function index()
```

```
public function create()
```

```
public function show()
```

7. Vues (Blade)

- **Création des Vues**

➢ Les vues sont des templates HTML générés dynamiquement avec Blade.

➢ Elles permettent de séparer la présentation de la logique métier.

❖ Exemple :

```
resources/views/articles/index.blade.php
```

III. Tech-Horizons

A. structure de projet

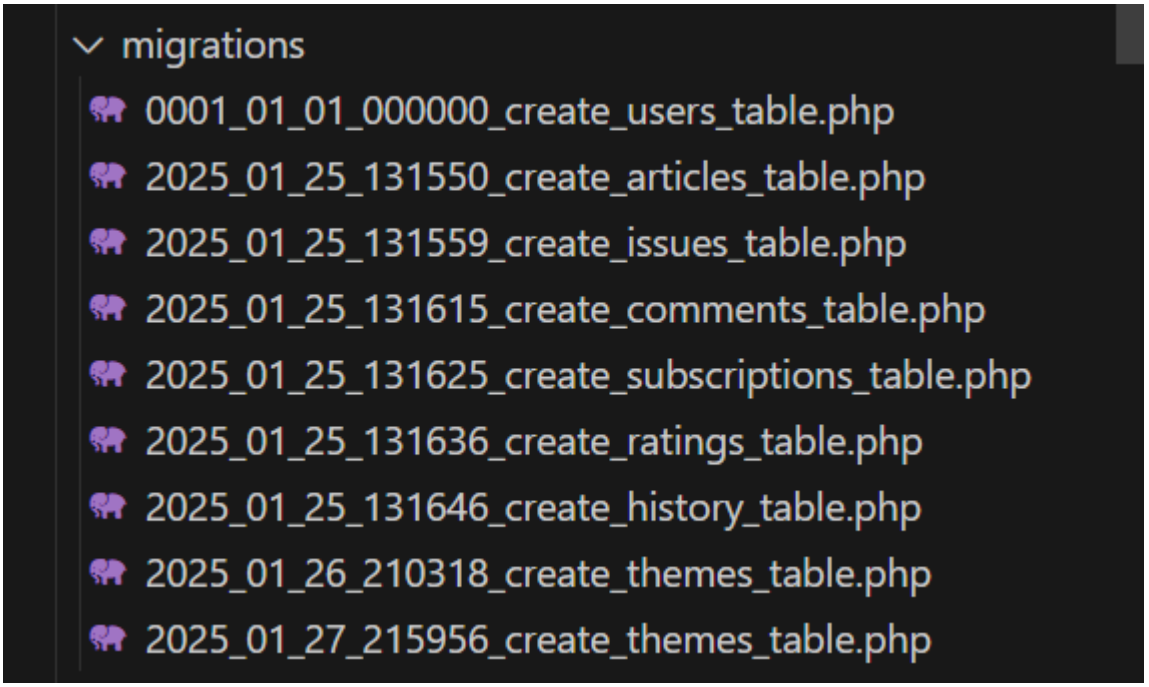
1. Installation de Laravel

```
composer create-project laravel/laravel TechHorizons
```

2. Base de Données et Modèles

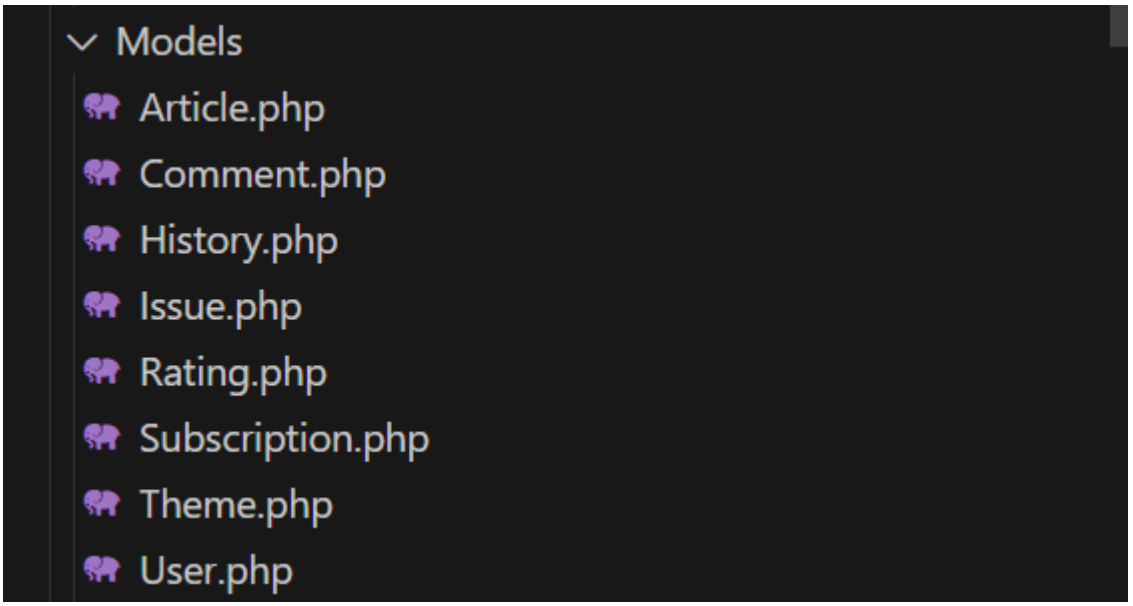
❖ **Création des Migrations**

- ➢ `users` : Gestion des utilisateurs.
- ➢ `articles` : Stockage des articles.
- ➢ `themes` : Gestion des thèmes.
- ➢ `issues` : Gestion des numéros du magazine.
- ➢ `comments` : Gestion des commentaires sur les articles.
- ➢ `ratings` : Gestion des notations des articles.
- ➢ `subscriptions` : Gestion des abonnements aux thèmes.
- ➢ `history` : Suivi de l'historique de navigation des utilisateurs.



❖ **Création des Modèles**

➢ `User` : Gestion des utilisateurs.
➢ `Article` : Gestion des articles.
➢ `Theme` : Gestion des thèmes.
➢ `Issue` : Gestion des numéros.
➢ `Comment` : Gestion des commentaires.
➢ `Rating` : Gestion des notations.
➢ `Subscription` : Gestion des abonnements.
➢ `History` : Gestion de l'historique de navigation.



❖ Relations entre les Modèles

➢ Un utilisateur peut avoir plusieurs articles.

```
// Relation avec les articles
    public function articles()
  {
        return $this->hasMany(Article::class);
    }
```

➢ Un article appartient à un thème.

```
// Relation avec le thème (un article appartient à un thème)
    public function theme(){
        return $this->belongsTo(Theme::class);
}J
```

➢ Un utilisateur peut avoir plusieurs abonnements.

```
// Relation avec les abonnements
    public function subscriptions()
    {
        return $this->belongsToMany(Theme::class, 'subscriptions',
'user_id', 'theme_id');
    }
```

➢ Un article peut avoir plusieurs commentaires et notations.

```
// Relation avec les commentaires (un article peut avoir plusieurs
commentaires)
    public function comments()
    {
        return $this->hasMany(Comment::class);
    }
    // Relation avec les notes (un article peut avoir plusieurs
notes)
```

```php
    public function ratings()

    {

        return $this->hasMany(Rating::class);

    }
```

3. Contrôleurs (Controllers)

❖ **Auth**

➔ Contrôleurs pour la gestion de l'authentification.

➢ ConfirmPasswordController.php : Gère la confirmation du mot de passe.

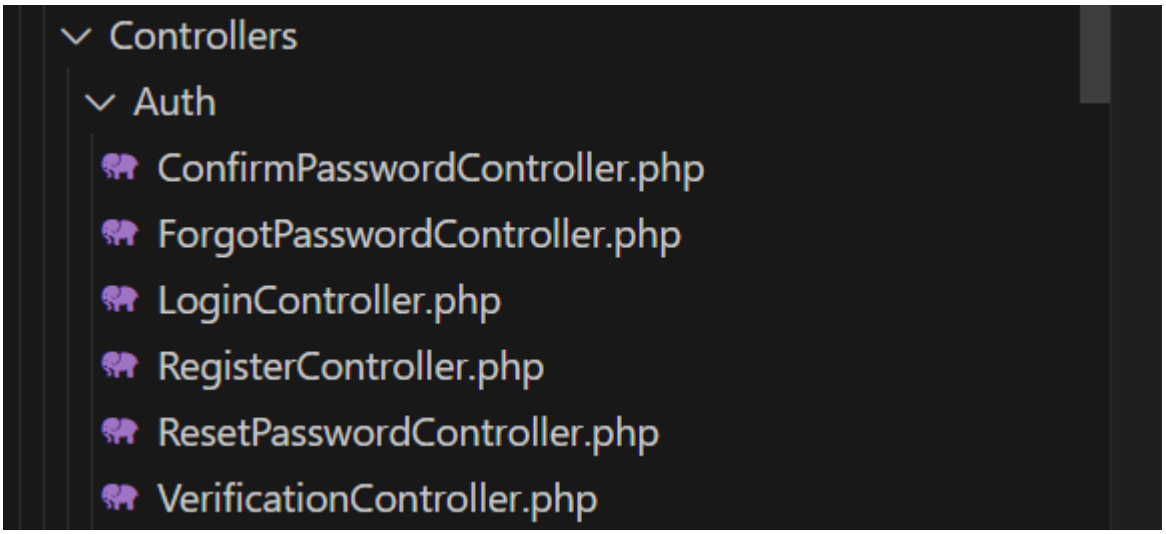
➢ ForgotPasswordController.php : Gère la réinitialisation du mot de passe.

➢ LoginController.php : Gère la connexion des utilisateurs.

➢ RegisterController.php : Gère l'inscription des utilisateurs.

➢ ResetPasswordController.php : Gère la réinitialisation du mot de passe.

➢ VerificationController.php: Gère la vérification des emails.



❖ **Controllers**

➢ ArticleController.php : Gère les articles.

➢ ArticleProposalController.php: Gère les propositions d'articles.

➢ AuthController.php: Gère l'authentification.

➢ CommentController.php : Gère les commentaires.

➢ Controller.php : Contrôleur de base.

➢ HistoryController.php : Gère l'historique des actions.

➢ HomeController.php : Gère la page d'accueil.

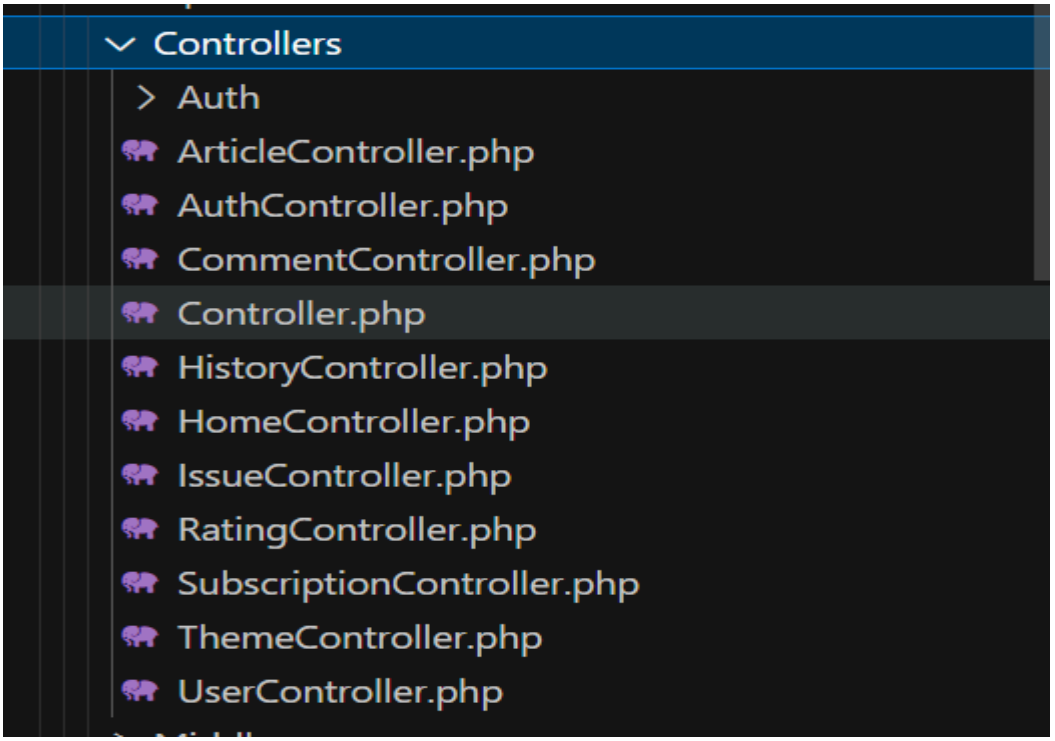
➢ IssueController.php : Gère les problèmes ou les numéros.

➢ RatingController.php : Gère les évaluations.

➢ SubscriptionController.php : Gère les abonnements.

➢ ThemeController.php : Gère les thèmes.

➢ UserController.php : Gère les utilisateurs.



4. Middlewares

➢ CanAccessArticle.php : Vérifie l'accès aux articles.

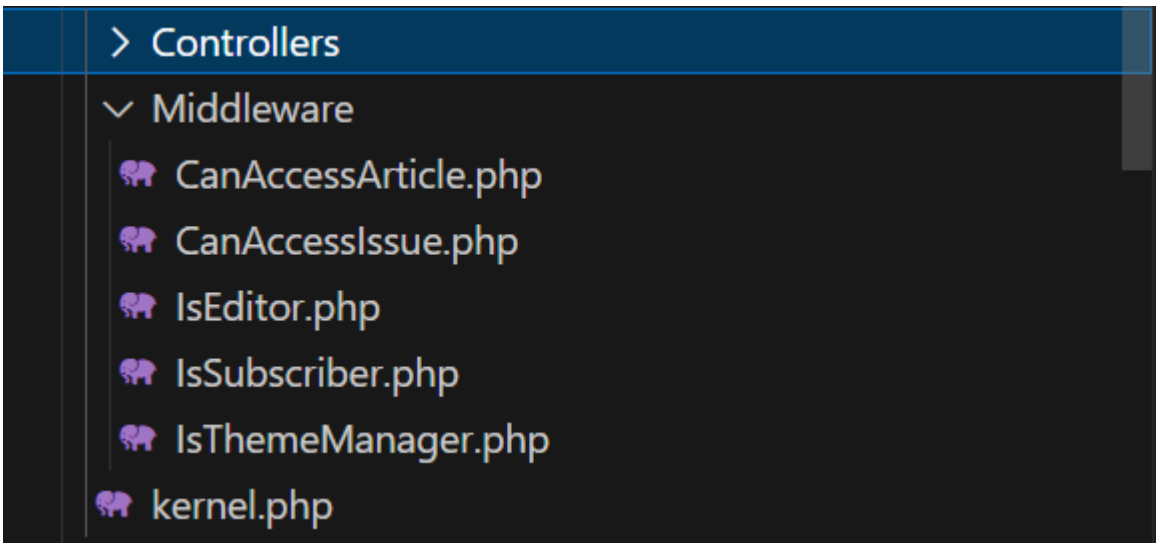
➢ CanAccessIssue.php : Middleware pour vérifier l'accès aux problèmes ou numéros.

➢ IsEditor.php : Vérifie si l'utilisateur est un éditeur.

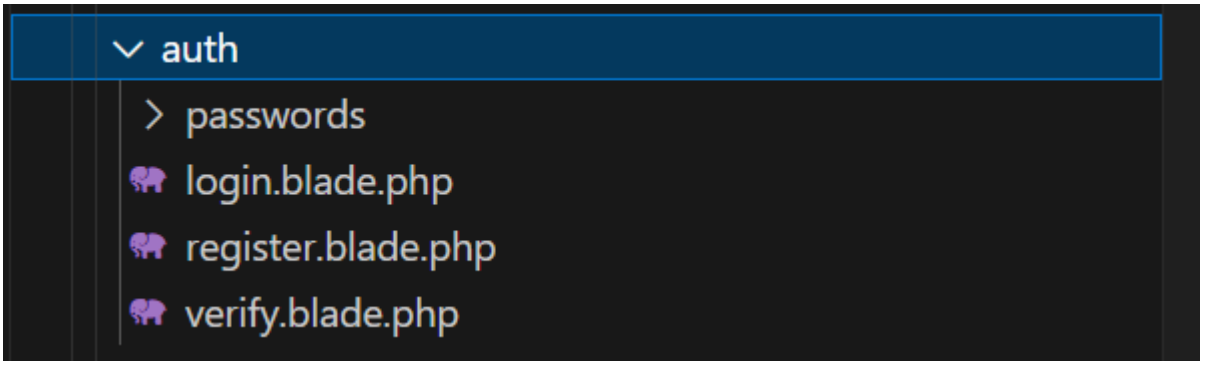
➢ IsSubscriber.php : Vérifie si l'utilisateur est un abonné.

➢ IsThemeManager.php : Vérifie si l'utilisateur est un gestionnaire de thèmes.

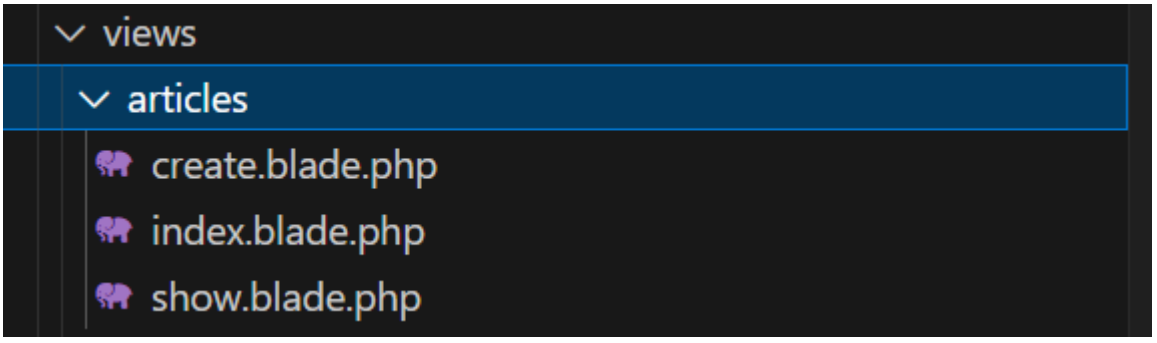
➢ kernel.php : Définit les middlewares globaux.



5. Vues (Views)

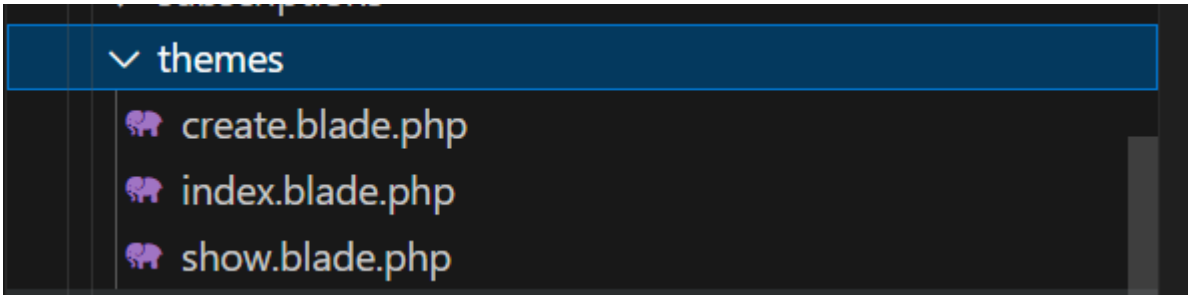


- articles : Vues pour la gestion des articles (create, index, show).
  - ➢ create.blade.php : Vue pour créer un article.
  - ➢ index.blade.php : Vue pour lister les articles.
  - ➢ show.blade.php: Vue pour afficher un article.



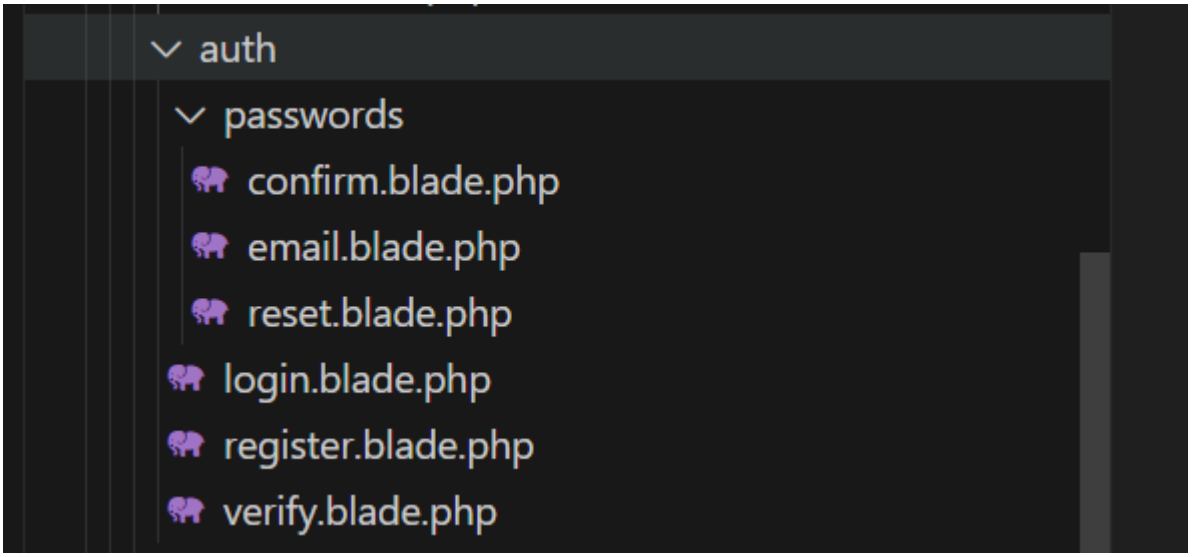
- themes: Vues pour la gestion des thèmes (create, index, show).

➢ create.blade.php : Vue pour créer un thème.
➢ index.blade.php: Vue pour lister les thèmes.
➢ show.blade.php : Vue pour afficher un thème.



❖ auth : Vues pour l'authentification.
➢ passwords : Vues pour la gestion des mots de passe.
- confirm.blade.php : Vue pour confirmer le mot de passe.
- email.blade.php : Vue pour l'email de réinitialisation.
- reset.blade.php : Vue pour réinitialiser le mot de passe.

➢ login.blade.php: Vue pour la connexion.
➢ register.blade.php: Vue pour l'inscription.
➢ verify.blade.php : Vue pour la vérification de l'email.



- history: Vues pour l'historique.
  ➢ index.blade.php : Vue pour lister l'historique.

- comments : Vues pour les commentaires.
- issues : Vues pour les problèmes ou numéros.
- ratings : Vues pour les évaluations.
- subscriptions : Vues pour les abonnements.
- user : Vues pour les utilisateurs.

➢ home.blade.php : Vue pour la page d'accueil de l'utilisateur.

6. Routes

➢ console.php : Définit les commandes Artisan personnalisées.

➢ web.php: Définit les routes web de l'application.

B. Explication du code

1. controller

❖ auth

- **ConfirmPasswordController**

```
use App\Http\Controllers\Controller;
use Illuminate\Foundation\Auth\ConfirmsPasswords;

class ConfirmPasswordController extends Controller
```

➢ Ce contrôleur est responsable de la confirmation du mot de passe de l'utilisateur.Il est utilisé pour s'assurer que l'utilisateur qui tente d'accéder à une partie sensible de l'application est bien celui qu'il prétend être.

```
use ConfirmsPasswords;
```

➢ Utilise le trait ConfirmsPasswords pour gérer la logique de confirmation du mot de passe.

```
protected $redirectTo = '/home';
```

➢ Redirige l'utilisateur vers /home après une confirmation réussie.

```
 $this->middleware('auth');
```

➢ Applique le middleware auth pour s'assurer que seuls les utilisateurs authentifiés peuvent accéder à cette fonctionnalité.

- **ForgotPasswordController**

```
use App\Http\Controllers\Controller;
use Illuminate\Foundation\Auth\SendsPasswordResetEmails;

class ForgotPasswordController extends Controller
```

➢ Ce contrôleur gère les demandes de réinitialisation de mot de passe. Il envoie un email à l'utilisateur avec un lien pour réinitialiser son mot de passe.

```
use SendsPasswordResetEmails;
```

➢ Utilise le trait SendsPasswordResetEmails pour envoyer des emails de réinitialisation de mot de passe.
➢ Ne nécessite pas d'authentification, car il est accessible aux utilisateurs qui ont oublié leur mot de passe.

- **LoginController**

```
use App\Http\Controllers\Controller;
use Illuminate\Foundation\Auth\AuthenticatesUsers;

class LoginController extends Controller
{
```

➢ Ce contrôleur gère l'authentification des utilisateurs, permettant aux utilisateurs de se connecter à l'application.

```
use AuthenticatesUsers;
```

➢ Utilise le trait AuthenticatesUsers pour gérer le processus de connexion.
➢ Redirige l'utilisateur vers /home après une connexion réussie.

```
$this->middleware('guest')->except('logout');
        $this->middleware('auth')->only('logout');
```

- ➢ Applique le middleware guest pour s'assurer que seuls les utilisateurs non authentifiés peuvent accéder à la page de connexion.
- ➢ Applique le middleware auth uniquement pour la déconnexion, garantissant que seuls les utilisateurs authentifiés peuvent se déconnecter.

- **RegisterController**

```
use App\Http\Controllers\Controller;
use App\Models\User;
use Illuminate\Foundation\Auth\RegistersUsers;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Facades\Validator;

class RegisterController extends Controller
```

- ➢ Ce contrôleur gère l'inscription de nouveaux utilisateurs dans l'application.

```
use RegistersUsers;
```

- ➢ Utilise le trait RegistersUsers pour gérer le processus d'inscription.

- ➢ Redirige l'utilisateur vers /home après une inscription réussie.

```
$this->middleware('guest');
```

- ➢ Applique le middleware guest pour s'assurer que seuls les utilisateurs non authentifiés peuvent s'inscrire.

```
protected function validator(array $data)
```

```
protected function create(array $data)
```

- ➢ Valide les données d'inscription (nom, email, mot de passe) avant de créer un nouvel utilisateur.
- ➢ Hash le mot de passe de l'utilisateur avant de le stocker dans la base de données.

- **ResetPasswordController**

```
use App\Http\Controllers\Controller;
use Illuminate\Foundation\Auth\ResetsPasswords;

class ResetPasswordController extends Controller
{
```

➢ Ce contrôleur gère la réinitialisation du mot de passe après que l'utilisateur a reçu un lien de réinitialisation par email.

```
use ResetsPasswords;
```

➢ Utilise le trait ResetsPasswords pour gérer la logique de réinitialisation du mot de passe.
➢ Redirige l'utilisateur vers /home après une réinitialisation réussie.

- **VerificationController**

```
use App\Http\Controllers\Controller;
use Illuminate\Foundation\Auth\VerifiesEmails;

class VerificationController extends Controller
```

➢ Ce contrôleur gère la vérification de l'adresse email des utilisateurs après leur inscription.

```
use VerifiesEmails;
```

➢ Utilise le trait VerifiesEmails pour gérer la vérification des emails.
➢ Redirige l'utilisateur vers /home après une vérification réussie.

```
$this->middleware('auth');
        $this->middleware('signed')->only('verify');
        $this->middleware('throttle:6,1')->only('verify', 'resend');
```

➢ Applique le middleware auth pour s'assurer que seuls les utilisateurs authentifiés peuvent accéder à cette fonctionnalité.
➢ Applique le middleware signed pour vérifier que le lien de vérification est valide.
➢ Applique le middleware throttle pour limiter le nombre de tentatives de vérification et de renvoi d'email.

❖ controller

- **ArticleController**

```
use Illuminate\Http\Request;
use App\Models\Article;
```

```
use App\Models\Theme;
use Illuminate\Support\Facades\Auth;
use App\Models\History;

class ArticleController extends Controller
{
```

➢ Ce contrôleur gère les articles, leur création, affichage, et gestion.

- **Méthodes:**

➢ **index()** : Affiche la page d'accueil avec les articles publiés et les thèmes disponibles.

➢ **create(Theme $theme)** : Affiche le formulaire de création d'un article pour un thème spécifique.

➢ **store(Request $request, Theme $theme)** : Enregistre un nouvel article dans la base de données.

➢ **status($id)**: Affiche la page de gestion du statut d'un article.

➢ **aboutUs()** : Affiche la page "À propos de nous".

➢ **themes()** : Affiche la liste des thèmes disponibles.

➢ **subscribe()** : Affiche la page d'inscription.

➢ **show($id)** : Affiche un article spécifique et enregistre sa consultation dans l'historique de l'utilisateur connecté.

➢ **edit($id)** :Affiche le formulaire de modification d'un article.

➢ **update(Request $request, $id)**: Met à jour un article existant.

➢ **destroy($id)** : Supprime un article spécifique.

➢ **updateStatus(Request $request, Article $article)**: Met à jour le statut d'un article.

- **AuthController**

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;

class AuthController extends Controller
```

➢ Ce contrôleur gère l'authentification des utilisateurs.

- **Méthodes:**

➢ **showLoginPage()**: Affiche la page de connexion.

- **CommentController**

```
use Illuminate\Http\Request;

use App\Models\Comment;
class CommentController extends Controller
{
```

➢ Ce contrôleur gère les commentaires sur les articles.

- **Méthodes :**

➢ **store(Request $request, $articleId)** : Ajoute un commentaire à un article spécifique.

- **HistoryController**

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;
use App\Models\History;

class HistoryController extends Controller
```

➢ Ce contrôleur gère l'historique des articles consultés par l'utilisateur.

- **Méthodes :**

➢ **showHistory(Request $request)** : Affiche l'historique des articles consultés par l'utilisateur, avec des filtres par date et par thème.

- **HomeController**

```
use Illuminate\Http\Request;

class HomeController extends Controller

```

➢ Ce contrôleur gère la page d'accueil après connexion.

- **Méthodes:**

➢ **index()** : Affiche la page d'accueil pour les utilisateurs authentifiés.

- **IssueController**

```
use App\Models\Issue;
use Illuminate\Http\Request;

```

```
class IssueController extends Controller
```

➢ Ce contrôleur gère les numéros (issues) des articles.

- **Méthodes :**

➢ **index()** : Affiche la liste des numéros.

➢ **create()** : Affiche le formulaire de création d'un numéro.

➢ **store(Request $request)**: Enregistre un nouveau numéro.

➢ **show($id)**: Affiche les détails d'un numéro spécifique.

➢ **publish($id)** : Publie un numéro.

➢ **toggle($id)** : Active ou désactive un numéro.

- **RatingController**

```
use App\Models\Rating;
use App\Models\Article;
use Illuminate\Http\Request;
```

➢ Ce contrôleur gère les notations des articles.

- **Méthodes:**

➢ **store(Request $request, $articleId)** : Attribue une note à un article.

- **SubscriptionController**

```
use App\Models\Subscription;
use App\Models\Theme;
use Illuminate\Http\Request;

class SubscriptionController extends Controller
```

➢ Ce contrôleur gère les abonnements aux thèmes.

- **Méthodes :**

➢ **subscribe($themeId)**: Abonne l'utilisateur à un thème.

➢ **unsubscribe($themeId)** : Désabonne l'utilisateur d'un thème.

➢ **index()** : Affiche les thèmes auxquels l'utilisateur est abonné.

- **ThemeController**

```
use App\Models\Theme;
use Illuminate\Http\Request;

class ThemeController extends Controller
```

```
{
```

➢ Ce contrôleur gère les thèmes des articles.

- **Méthodes :**

➢ **index()** : Affiche la liste des thèmes.

➢ **create()**: Affiche le formulaire de création d'un thème.

➢ **store(Request $request)** : Enregistre un nouveau thème.

➢ **show(Theme $theme)**: Affiche les détails d'un thème spécifique.

➢ **stats($id)** : Affiche les statistiques d'un thème (articles et abonnés).

➢ **statistics(Theme $theme)**: Affiche des statistiques détaillées pour un thème (articles, abonnés, commentaires).

• UserController

```
use App\Models\User;
use Illuminate\Http\Request;

class UserController extends Controller
```

➢ Ce contrôleur gère les utilisateurs.

- **Méthodes:**

➢ **index()** : Affiche la liste des utilisateurs.

➢ **create()** : Affiche le formulaire de création d'un utilisateur.

➢ **store(Request $request)** : Enregistre un nouvel utilisateur.

➢ **show($id)** : Affiche les détails d'un utilisateur spécifique.

➢ **edit($id)** : Affiche le formulaire d'édition d'un utilisateur.

➢ **update(Request $request, $id)** : Met à jour les informations d'un utilisateur.

➢ **destroy($id)** : Supprime un utilisateur.

❖ EditorController

```
se Illuminate\Http\Request;
use App\Models\Article;
use App\Models\Subscription;
use App\Models\Theme;

class EditorController extends Controller
```

➢ Il est responsable de la gestion du tableau de bord de l'éditeur, en fournissant des statistiques et des données pertinentes pour l'interface d'administration. Voici une analyse détaillée du code :
➢ La classe `EditorController` étend la classe de base `Controller` de Laravel, ce qui permet d'utiliser les fonctionnalités de base d'un contrôleur.

- **Méthode `dashboard`**

➢ Cette méthode est responsable de la récupération des données statistiques pour le tableau de bord de l'éditeur et de leur transmission à la vue.

- **Statistiques des Articles**

➢ **`$totalArticles`** : Compte le nombre total d'articles dans la base de données.
➢ **`$articlesPending`** : Compte le nombre d'articles ayant le statut "pending" (en attente).
➢ **`$articlesAccepted`** : Compte le nombre d'articles ayant le statut "accepted" (acceptés).
➢ **`$articlesRejected`** : Compte le nombre d'articles ayant le statut "rejected" (rejetés).
➔ Ces statistiques permettent à l'éditeur de comprendre l'état actuel des articles soumis.

- **Statistiques des Abonnés**

➢ **`$totalSubscribers`** : Compte le nombre total d'abonnés dans la base de données.

➢ **Statistiques des Thèmes**

➢ **`$totalThemes`** : Compte le nombre total de thèmes dans la base de données.
➢ **`$themesWithMostArticles`** : Récupère les 5 thèmes les plus actifs en fonction du nombre d'articles associés à chaque thème.
➢ Utilise `withCount('articles')` pour compter les articles associés à chaque thème.
➢ Trie les résultats par ordre décroissant (`orderBy('articles_count', 'desc')`).

➢ **`$themesWithMostSubscribers`** : Récupère les 5 thèmes les plus populaires en fonction du nombre d'abonnés associés à chaque thème.
➢ Utilise `withCount('subscribers')` pour compter les abonnés associés à chaque thème.
➢ Trie les résultats par ordre décroissant (`orderBy('subscribers_count','desc')`).

- **Transmission des Données à la Vue**

➢ Les données récupérées sont transmises à la vue `editor.dashboard` via la fonction `compact()`
➢ **Les variables transmises sont** :
  - `totalArticles`, `articlesPending`, `articlesAccepted`, `articlesRejected`
  - `totalSubscribers`
  - `totalThemes`, `themesWithMostArticles`, `themesWithMostSubscribers`

2. Models

❖ Article

```
use Illuminate\Database\Eloquent\Model;

class Article extends Model
```

➢ Ce modèle représente un article dans le système. Un article peut être en attente, accepté ou rejeté.

- **Attributs :**

➢ **title** : Titre de l'article.
➢ **content** : Contenu de l'article.
➢ **theme_id** : ID du thème associé à l'article.
➢ **user_id** : ID de l'utilisateur qui a créé l'article.
➢ **status** : Statut de l'article (en attente, accepté, rejeté).

- **Relations:**

➢ **user()** : Relation avec l'utilisateur (un article appartient à un utilisateur).
➢ **theme()** : Relation avec le thème (un article appartient à un thème).
➢ **history()** : Relation avec l'historique (un article peut avoir plusieurs entrées d'historique).
➢ **comments()** : Relation avec les commentaires (un article peut avoir plusieurs commentaires).

➢ **ratings()** : Relation avec les notes (un article peut avoir plusieurs notes).

- **Accesseurs :**

➢ **getIsAcceptedAttribute()** : Vérifie si l'article est accepté.

- **Règles de validation :**

➢ title, content, theme_id, user_id, et status sont requis et doivent respecter certaines contraintes.

❖ Comment

```
use Illuminate\Database\Eloquent\Model;

class Comment extends Model
```

➢ modèle représente un commentaire sur un article.

- **Attributs :**

➢ **message** : Contenu du commentaire.

➢ **article_id** : ID de l'article associé.

➢ **user_id** : ID de l'utilisateur qui a posté le commentaire.

➢ Relations :

➢ **user()** : Relation avec l'utilisateur (un commentaire appartient à un utilisateur).

➢ **article()** : Relation avec l'article (un commentaire appartient à un article).

- **Règles de validation :**

➢ message, article_id, et user_id sont requis et doivent respecter certaines contraintes.

❖ History

```
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Factories\HasFactory;

class History extends Model
```

➢ Ce modèle représente l'historique de navigation des utilisateurs.

- **Attributs:**

➢ **user_id** : ID de l'utilisateur.

➢ **article_id** : ID de l'article consulté.

- **Relations :**

➢ **article()** : Relation avec l'article (une entrée d'historique appartient à un article).
➢ **user()** : Relation avec l'utilisateur (une entrée d'historique appartient à un utilisateur).

- **Règles de validation :**

➢ user_id et article_id sont requis et doivent respecter certaines contraintes.

❖ Issue

```
use Illuminate\Database\Eloquent\Model;

class Issue extends Model
```

➢ Ce modèle représente un numéro (ou édition) qui peut contenir plusieurs articles.

- **Attributs :**

➢ **title** : Titre du numéro.
➢ **publication_date** : Date de publication.
➢ **status** : Statut du numéro (brouillon, publié, archivé).

- **Relations :**

➢ **articles()** : Relation avec les articles (un numéro peut contenir plusieurs articles).

- **Accesseurs :**

➢ **getIsPublishedAttribute()** : Vérifie si le numéro est publié.

- **Règles de validation :**

➢ title, publication_date, et status sont requis et doivent respecter certaines contraintes.

❖ Rating

```
use Illuminate\Database\Eloquent\Model;

class Rating extends Model
```

➢ Ce modèle représente une note attribuée à un article.

- **Attributs :**

➢ **rating**: Note attribuée (entre 1 et 5).
➢ **article_id** : ID de l'article noté.
➢ **user_id** : ID de l'utilisateur qui a noté l'article.

- **Relations :**

➢ **article()** : Relation avec l'article (une note appartient à un article).

➢ **user()** : Relation avec l'utilisateur (une note appartient à un utilisateur).

- **Accesseurs :**

➢ **getRatingStarsAttribute()** : Retourne la note sous forme d'étoiles.

- **Règles de validation :**

➢ rating, article_id, et user_id sont requis et doivent respecter certaines contraintes.

- **Événements :**

➢ Des actions peuvent être effectuées après la création, la mise à jour ou la suppression d'une note.

❖ **Subscription**

```
use Illuminate\Database\Eloquent\Model;

class Subscription extends Model
```

➢ Ce modèle représente un abonnement d'un utilisateur à un thème.

- **Attributs :**

➢ **user_id**: ID de l'utilisateur.

➢ **theme_id** : ID du thème.

- **Relations :**

➢ **user()** : Relation avec l'utilisateur (un abonnement appartient à un utilisateur).

➢ **theme()** : Relation avec le thème (un abonnement appartient à un thème).

- **Accesseurs :**

➢ **getCreatedAtFormattedAttribute()**: Retourne la date de création formatée.

- **Règles de validation:**

➢ **user_id** et **theme_id** sont requis et doivent respecter certaines contraintes.

- **Événements:**

➢ Des actions peuvent être effectuées après la création, la mise à jour ou la suppression d'un abonnement.

- **Méthodes :**

➢ **listSubscribers(Theme $theme)** : Liste les abonnés d'un thème.

❖ **Theme**

```
use Illuminate\Database\Eloquent\Model;

class Theme extends Model
```

```
{
```

➢ Ce modèle représente un thème auquel les articles peuvent être associés.

- **Attributs :**

➢ **name**: Nom du thème.
➢ **description** : Description du thème.
➢ **image** : Image associée au thème.
➢ **created_by** : ID de l'utilisateur qui a créé le thème.

- **Relations :**

➢ **articles()** : Relation avec les articles (un thème peut avoir plusieurs articles).
➢ **subscribers()** : Relation avec les abonnés (un thème peut avoir plusieurs abonnés).
➢ **responsible()** : Relation avec le responsable du thème (un thème appartient à un utilisateur responsable).

- **Accesseurs :**

➢ **getTotalArticlesAttribute()** : Retourne le nombre total d'articles associés au thème.

- **Règles de validation :**

➢ name et description sont requis et doivent respecter certaines contraintes.

❖ **User**

```
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
```

➢ Ce modèle représente un utilisateur du système.

- **Attributs :**

➢ **name** : Nom de l'utilisateur.
➢ **email** : Email de l'utilisateur.
➢ **password** : Mot de passe de l'utilisateur.
➢ **role** : Rôle de l'utilisateur (invité, abonné, gestionnaire de thème, éditeur).

- **Relations :**

➢ **articles()** : Relation avec les articles (un utilisateur peut avoir plusieurs articles).
➢ **comments()** : Relation avec les commentaires (un utilisateur peut avoir plusieurs commentaires).
➢ **subscriptions()** : Relation avec les abonnements (un utilisateur peut s'abonner à plusieurs thèmes).

➢ **history()** : Relation avec l'historique de navigation (un utilisateur peut avoir plusieurs entrées d'historique).
➢ **ratings()** : Relation avec les notes (un utilisateur peut avoir plusieurs notes).

- **Accesseurs:**

➢ **getTotalArticlesAttribute()** : Retourne le nombre total d'articles écrits par l'utilisateur.

- **Mutateurs** :

➢ **setPasswordAttribute($value)** : Hache le mot de passe avant de le stocker.

- **Méthodes :**

➢ **isGuest()**, **isSubscriber()**, **isThemeManager()**, **isEditor()** : Vérifient le rôle de l'utilisateur.

- **Règles de validation :**

➢ name, email, password, et role sont requis et doivent respecter certaines contraintes.

3. Migrations

❖ Table users

```
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
```

➢ Cette table stocke les informations des utilisateurs de l'application.

- **Colonnes :**

➢ **id**: Clé primaire auto-incrémentée.
➢ **name** : Nom de l'utilisateur.
➢ **email** : Adresse email de l'utilisateur (unique).
➢ **email_verified_at** : Timestamp de la vérification de l'email (nullable).
➢ **password** : Mot de passe de l'utilisateur.
➢ **remember_token** : Token pour la fonction "Se souvenir de moi".
➢ **created_at** et **updated_at** : Timestamps pour la création et la mise à jour.

❖ password_reset_tokens

```
Schema::create('password_reset_tokens', function (Blueprint $table)
```

➢ Cette table stocke les tokens pour la réinitialisation des mots de passe.

- **Colonnes :**

➢ **email** : Adresse email de l'utilisateur (clé primaire).
➢ **token** : Token de réinitialisation.
➢ **created_at** : Timestamp de création du token (nullable).

❖ Table sessions

```
Schema::create('sessions', function (Blueprint $table)
```

➢ Cette table stocke les sessions des utilisateurs.

- **Colonnes :**

➢ **id** : Identifiant de la session (clé primaire).
➢ **user_id** : Clé étrangère référençant l'utilisateur (nullable).
➢ **ip_address** : Adresse IP de l'utilisateur (nullable).
➢ **user_agent**: Agent utilisateur (nullable).
➢ **payload** : Données de la session.
➢ **last_activity** : Timestamp de la dernière activité.

❖ Table articles

```
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;rz
class CreateArticlesTable extends Migration
 public function up(): void
    {
        Schema::create('users', function (Blueprint $table)
```

➢ Cette table stocke les articles publiés.

- **Colonnes :**

➢ **id** : Clé primaire auto-incrémentée.
➢ **title**: Titre de l'article.
➢ **content** : Contenu de l'article.
➢ **image** : Image associée à l'article (nullable).
➢ **theme_id** : Clé étrangère référençant le thème de l'article.
➢ **created_at** et **updated_at** : Timestamps pour la création et la mise à jour.

❖ Table issues

```
use Illuminate\Database\Migrations\Migration;
```

```php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;
return new class extends Migration
{
    public function up()
    {
        Schema::create('issues', function (Blueprint $table)
```

➢ Cette table stocke les numéros de publication.

- **Colonnes:**

➢ **id** : Clé primaire auto-incrémentée.
➢ **title** : Titre du numéro.
➢ **publication_date** : Date de publication.
➢ **status** : Statut du numéro (draft, published, archived).
➢ **is_active** : Indicateur d'activité (par défaut true).
➢ **created_at** et **updated_at** : Timestamps pour la création et la mise à jour.

❖ **Table comments**

```php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;
return new class extends Migration
{
    public function up()
    {
        Schema::create('comments', function (Blueprint $table)
```

➢ Cette table stocke les commentaires des utilisateurs sur les articles.

- **Colonnes:**

➢ **id** : Clé primaire auto-incrémentée.
➢ **message** : Contenu du commentaire.
➢ **article_id** : Clé étrangère référençant l'article commenté.
➢ **user_id** : Clé étrangère référençant l'utilisateur qui a posté le commentaire.
➢ **created_at** et **updated_at** : Timestamps pour la création et la mise à jour.

❖ **Table subscriptions**

```php
use Illuminate\Database\Migrations\Migration;
```

```php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;
return new class extends Migration
{
    public function up()
    {
        Schema::create('subscriptions', function (Blueprint $table)
```

➢ Cette table stocke les abonnements des utilisateurs à des thèmes.

- **Colonnes :**

➢ **id** : Clé primaire auto-incrémentée.
➢ **user_id** : Clé étrangère référençant l'utilisateur.
➢ **theme_id** : Clé étrangère référençant le thème.
➢ **created_at** et updated_at : Timestamps pour la création et la mise à jour.

❖ Table ratings

```php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;
return new class extends Migration
{
    public function up()
    {
        Schema::create('ratings', function (Blueprint $table)
```

➢ Cette table stocke les notes attribuées par les utilisateurs aux articles.

- **Colonnes :**

➢ **id** : Clé primaire auto-incrémentée.
➢ **rating** : Note attribuée (de 1 à 5).
➢ **article_id** : Clé étrangère référençant l'article noté.
➢ **user_id** : Clé étrangère référençant l'utilisateur qui a noté.
➢ **created_at** et **updated_at** : Timestamps pour la création et la mise à jour.

❖ Table history

```php
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;
return new class extends Migration
```

```
{
    public function up()
    {
        Schema::create('history', function (Blueprint $table)
```

➢ Cette table stocke l'historique des articles consultés par les utilisateurs.

- **Colonnes :**

➢ **id** : Clé primaire auto-incrémentée.
➢ **user_id** : Clé étrangère référençant l'utilisateur.
➢ **article_id** : Clé étrangère référençant l'article consulté.
➢ **created_at** et **updated_at** : Timestamps pour la création et la mise à jour.

❖ Table themes

```
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;
return new class extends Migration
{
    public function up()
    {
        Schema::create('themes', function (Blueprint $table)
```

➢ Cette table stocke les thèmes disponibles pour les articles.

- **Colonnes :**

➢ **id** : Clé primaire auto-incrémentée.
➢ **name**: Nom du thème (unique).
➢ **description** : Description du thème (nullable).
➢ **image** : Image associée au thème (nullable).
➢ **created_by** : Clé étrangère référençant l'utilisateur qui a créé le thème.
➢ **created_at** et updated_at : Timestamps pour la création et la mise à jour.

4. Middelware

❖ CanAccessArticle

```
use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class CanAccessArticle
```

- Ce middleware est conçu pour vérifier si l'utilisateur actuellement authentifié a le droit d'accéder à un article spécifique.

```
return $next($request);
```

- passer la requête au prochain middleware ou au contrôleur en appelant `return $next($request);`.

❖ CanAccessIssue

```
use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class CanAccessIssue
```

- Ce middleware est conçu pour vérifier si l'utilisateur actuellement authentifié a le droit d'accéder à un numéro spécifique (issue) d'une publication.
- Comme CanAccessArticle,passer la requête au prochain middleware ou au contrôleur.

❖ IsEditor

```
use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class IsEditor
```

- Ce middleware est conçu pour vérifier si l'utilisateur actuellement authentifié a le rôle d'éditeur.
- passer la requête au prochain middleware ou au contrôleur.

❖ IsSubscriber

```
use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class IsSubscriber
```

- Ce middleware est conçu pour vérifier si l'utilisateur actuellement authentifié est un abonné.
- passer la requête au prochain middleware ou au contrôleur.

❖ IsThemeManager

```
use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class IsThemeManager
```

➢ Ce middleware est conçu pour vérifier si l'utilisateur actuellement authentifié a le rôle de gestionnaire de thèmes.
➢ passer la requête au prochain middleware ou au contrôleur.

➔ **kernel.php**

❖ Middleware Global ($middleware)

➢ Ces middlewares sont exécutés pour chaque requête entrante dans l'application. Ils sont appliqués globalement, quel que soit le type de requête ou la route.

- **Middlewares inclus** :

➢ **TrustProxies::class** : Gère les proxies pour obtenir les informations correctes sur le client (comme l'adresse IP).
➢ **HandleCors::class**: Gère les en-têtes CORS (Cross-Origin Resource Sharing) pour les requêtes cross-origin.
➢ **PreventRequestsDuringMaintenance::class** : Bloque les requêtes lorsque l'application est en mode maintenance.
➢ **ValidatePostSize::class** : Valide la taille des données POST pour éviter les surcharges.
➢ **TrimStrings::class** : Supprime les espaces blancs inutiles des chaînes de caractères dans les données de requête.
➢ **ConvertEmptyStringsToNull::class**: Convertit les chaînes vides en `null` dans les données de requête.

❖ Groupes de Middleware (`$middlewareGroups`)

➢ Les middlewares peuvent être regroupés pour être appliqués à des ensembles de routes spécifiques. Laravel propose deux groupes par défaut : `web` et `api`.

- **Groupe `web`** :

➢ Appliqué aux routes qui gèrent les requêtes web (comme les formulaires, les sessions, etc.).

- **Middlewares inclus :**

➢ **EncryptCookies::class** : Chiffre les cookies pour la sécurité.

➢ **AddQueuedCookiesToResponse::class** : Ajoute les cookies en file d'attente à la réponse.
➢ **StartSession::class** : Démarre une session pour l'utilisateur.
➢ **ShareErrorsFromSession::class** : Partage les erreurs de validation avec les vues.
➢ **VerifyCsrfToken::class** : Vérifie le jeton CSRF pour les requêtes POST.
➢ **SubstituteBindings::class**: Gère le binding des modèles dans les routes.
• **Groupe `api`** :
➢ Appliqué aux routes qui gèrent les requêtes API.
• **Middlewares inclus :**
➢ **throttle:api** : Limite le taux de requêtes pour éviter les abus.
➢ **SubstituteBindings::class**: Gère le binding des modèles dans les routes API.

❖ **Middleware de Route (`$routeMiddleware`)**
➢ Ces middlewares peuvent être appliqués individuellement à des routes spécifiques ou à des groupes de routes. Ils sont souvent utilisés pour des fonctionnalités spécifiques comme l'authentification, l'autorisation, etc.
• **Middlewares inclus :**
☐ Middlewares par défaut de Laravel :
➢ **auth** : Vérifie si l'utilisateur est authentifié.
➢ **auth.basic** : Authentification basique HTTP.
➢ **bindings** : Gère le binding des modèles dans les routes.
➢ **cache.headers** : Définit les en-têtes de cache pour les réponses.
➢ **can** : Vérifie les autorisations de l'utilisateur.
➢ **guest** : Redirige les utilisateurs authentifiés.
➢ **signed** : Valide les URLs signées.
➢ **throttle** : Limite le taux de requêtes.
➢ **verified** : Vérifie que l'email de l'utilisateur est confirmé.
☐ Middlewares personnalisés :
➢ **is.subscriber** : Vérifie si l'utilisateur est un abonné.
➢ **is.theme.manager** : Vérifie si l'utilisateur est un gestionnaire de thèmes.
➢ **is.editor** : Vérifie si l'utilisateur est un éditeur.
➢ **can.access.article** : Vérifie si l'utilisateur peut accéder à un article.

➢ can.access.issue : Vérifie si l'utilisateur peut accéder à un numéro.

5. **Public**

☐ JS

❖ **Gestion de la Boîte de Dialogue de Confirmation 'boite.js'**

➢ permet d'afficher une boîte de dialogue de confirmation lorsque l'utilisateur clique sur un bouton de suppression. Si l'utilisateur confirme, le formulaire est soumis. Sinon, la boîte de dialogue est masquée.

- **Fonctionnalités**

- **Interception des Clics :**

➢ Le code écoute les clics sur tous les boutons de type `submit`.

➢ Lorsqu'un bouton est cliqué, le comportement par défaut (soumission du formulaire) est empêché.

➢ Le formulaire associé au bouton est récupéré et stocké dans une variable (`formToSubmit`).

➢ La boîte de dialogue (`confirmDialog`) est affichée.

- Confirmation :

➢ Si l'utilisateur clique sur "Oui" (`confirmYes`), le formulaire est soumis.

➢ Si l'utilisateur clique sur "Non" (`confirmNo`), la boîte de dialogue est masquée et le formulaire est réinitialisé.

❖ **Fonction de Défilement du Slider (`slides.php`)**

➢ permet de faire défiler un slider (carrousel) d'images vers la gauche. Elle déplace la dernière image au début du slider pour créer un effet de défilement infini.

- **Fonctionnalités**

- **Animation de Défilement** :

➢ Le slider est déplacé vers la gauche en utilisant une transformation CSS (`translateX`).

➢ Une transition CSS est appliquée pour créer un effet fluide.

- **Réorganisation des Éléments :**

➢ La dernière image du slider est déplacée au début du conteneur.

➢ Cela permet de créer un effet de défilement infini.

- Réinitialisation :

➢ Après l'animation, la position du slider est réinitialisée pour permettre une nouvelle transition.

6. Views

➔ Article

❖ Formulaire de création d'article (`create.blade.php`):

```
<form action="{{ route('articles.store', ['theme' => $theme->id]) }}"
method="POST" enctype="multipart/form-data">
```

➢ Ce formulaire permet à l'utilisateur de créer un nouvel article.

- **Champs du formulaire :**

➢ **Titre** : Champ de texte pour saisir le titre de l'article.

➢ **Contenu** : Zone de texte pour saisir le contenu de l'article.

➢ **Statut** : Menu déroulant pour sélectionner le statut de l'article (pending, accepted, rejected).

➢ **Image**: Champ de fichier pour téléverser une image associée à l'article.

➢ **Méthode HTTP** : POST avec enctype multipart/form-data pour gérer les fichiers.

➢ **Action** : Le formulaire est soumis à la route articles.store avec l'ID du thème associé.

➢ **CSRF Protection** : Utilisation de @csrf pour protéger contre les attaques CSRF.

- **Statut**

➢ Le statut de l'article est déterminé via une modale qui s'affiche lorsque l'utilisateur clique sur le bouton "Créer l'article". L'utilisateur peut choisir entre trois statuts :

➔ **Accepter** (statut accepted)

➔ **En attente** (statut pending)

➔ **Refuser**(statut rejected)

❖ Formulaire de modification d'article (`edit.blade.php`) :

```
<form action="{{ route('articles.update', $article->id) }}"
method="POST" enctype="multipart/form-data">
```

➢ Ce formulaire permet à l'utilisateur de modifier un article existant.

- **Champs du formulaire :**

➢ **Titre**: Champ de texte pré-rempli avec le titre actuel de l'article.

➢ **Contenu** : Zone de texte pré-remplie avec le contenu actuel de l'article.

➢ **Image** : Champ de fichier pour téléverser une nouvelle image. L'image actuelle est affichée si elle existe.

➢ **Statut** : Menu déroulant pour sélectionner le statut de l'article, avec l'option actuelle pré-sélectionnée.

➢ **Méthode HTTP** : PUT avec enctype multipart/form-data pour gérer les fichiers.

➢ **Action** : Le formulaire est soumis à la route `articles.update` avec l'ID de l'article.

➢ **CSRF Protection**: Utilisation de @csrf et @method(PUT) pour protéger contre les attaques CSRF et spécifier la méthode HTTP.

❖ **Affichage d'un article avec commentaires (`show.blade.php`):**

```
<form action="{{ route('comments.store', $article->id) }}"
method="POST">
```

➢ Cette vue affiche les détails d'un article ainsi que les commentaires associés.

- **Contenu affiché :**

➢ **Titre** : Le titre de l'article.

➢ **Contenu** : Le contenu de l'article.

➢ **Image** : L'image de l'article est affichée si elle existe.

➢ **Commentaires** : Liste des commentaires associés à l'article, affichant le nom de l'utilisateur et le message du commentaire.

- **Formulaire de commentaire :**

➢ **Message** : Zone de texte pour saisir un nouveau commentaire.

➢ **Méthode HTTP** : POST.

➢ **Action** : Le formulaire est soumis à la route comments.store avec l'ID de l'article.

➢ **CSRF Protection** : Utilisation de @csrf pour protéger contre les attaques CSRF.

❖ **Structure commune:**

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

➢ **Layout** : Toutes les vues étendent le layout layouts.app via @extends('layouts.app') et injectent leur contenu dans la section `content` via @section(content).

```
@section('content')
```

➢ **Conteneur** : Le contenu est encapsulé dans un conteneur Bootstrap pour une mise en page cohérente.

❖ **Routes et contrôleurs :**

➢ **Création d'article** : Le formulaire de création est soumis à articles.store, ce qui suggère l'existence d'une méthode store dans le contrôleur ArticleController.

➢ **Modification d'article** : Le formulaire de modification est soumis à articles.update, ce qui suggère l'existence d'une méthode update dans le contrôleur ArticleController.

➢ **Ajout de commentaire** : Le formulaire de commentaire est soumis à `comments.store`, ce qui suggère l'existence d'une méthode `store` dans le contrôleur CommentController.

❖ **Gestion des fichiers:**

➢ **Téléversement d'images** : Les formulaires de création et de modification d'article permettent de téléverser des images, qui sont stockées dans le dossier storage et servies via la fonction asset(storage/ . $article->image).

❖ **Statut des articles :**

```
<label for="status">Statut :</label>
```

➢ **Statuts disponibles** : pending, accepted, rejected. Ces statuts sont gérés via un menu déroulant dans les formulaires de création et de modification.

❖ **Commentaires :**

➢ **Affichage**: Les commentaires sont affichés sous l'article, avec le nom de l'utilisateur et le message.

➢ **Ajout** : Un formulaire permet aux utilisateurs d'ajouter de nouveaux commentaires.

```
<!-- Formulaire pour ajouter un commentaire -->
    <form action="{{ route('comments.store', $article->id) }}"
method="POST">
```

➔ **auth**

❖ **Connexion (`login.blade.php`) :**

➢ Cette vue permet à l'utilisateur de se connecter à l'application.

- **Contenu :**

➢ **Champ d'email** : Champ de texte pour saisir l'adresse email.

➢ **Champ de mot de passe** : Champ de texte pour saisir le mot de passe.

➢ **Case à cocher "Se souvenir de moi"** : Option pour rester connecté.

➢ **Lien de réinitialisation** : Lien vers la page de réinitialisation du mot de passe.

➢ **Bouton de connexion** : Bouton pour soumettre le formulaire.

➢ **Lien d'inscription** : Lien vers la page d'inscription.

➢ **Méthode HTTP** : POST.

➢ **Action** : Le formulaire est soumis à la route login.

➢ **CSRF Protection** : Utilisation de @csrf pour protéger contre les attaques CSRF.

❖ **Inscription (`register.blade.php`) :**

➢ Cette vue permet à l'utilisateur de s'inscrire à l'application.

- **Contenu :**

➢ **Champ de nom** : Champ de texte pour saisir le nom de l'utilisateur.

➢ **Champ d'email** : Champ de texte pour saisir l'adresse email.

➢ **Champ de mot de passe** : Champ de texte pour saisir le mot de passe.

➢ **Champ de confirmation du mot de passe** : Champ de texte pour confirmer le mot de passe.

➢ **Bouton d'inscription** : Bouton pour soumettre le formulaire.

➢ Lien de connexion: Lien vers la page de connexion.

➢ **Méthode HTTP** : POST.
➢ **Action** : Le formulaire est soumis à la route register.
➢ **CSRF Protection** : Utilisation de @csrf pour protéger contre les attaques CSRF.

❖ **Vérification de l'email (`verify.blade.php`) :**

➢ Cette vue demande à l'utilisateur de vérifier son adresse email.

- **Contenu :**

➢ **Message de statut** : Affiche un message de succès si un nouveau lien de vérification a été envoyé.
➢ **Message de vérification** : Demande à l'utilisateur de vérifier son email.
➢ **Formulaire de renvoi** : Formulaire pour demander un nouveau lien de vérification.
➢ **Méthode HTTP** : POST.
➢ **Action** : Le formulaire est soumis à la route verification.resend.
➢ **CSRF Protection** : Utilisation de `@csrf` pour protéger contre les attaques CSRF.

☐ **Passwords**

❖ **Confirmation du mot de passe (`confirm.blade.php`):**

➢ Cette vue permet à l'utilisateur de confirmer son mot de passe avant d'accéder à une section sécurisée de l'application.

- **Contenu :**

➢ Message : Demande à l'utilisateur de confirmer son mot de passe.

★ **Formulaire :**

➢ Champ de mot de passe : Champ de texte pour saisir le mot de passe.
➢ **Bouton de confirmation** : Bouton pour soumettre le formulaire.
➢ **Lien de réinitialisation** : Lien vers la page de réinitialisation du mot de passe si l'utilisateur l'a oublié.
➢ **Méthode HTTP** : POST.
➢ **Action** : Le formulaire est soumis à la route password.confirm.
➢ **CSRF Protection** : Utilisation de @csrf pour protéger contre les attaques CSRF.

❖ **Demande de réinitialisation du mot de passe (`email.blade.php`):**

Cette vue permet à l'utilisateur de demander un lien de réinitialisation de mot de passe.

- **Contenu :**

➢ **Message de statut** : Affiche un message de succès si un lien de réinitialisation a été envoyé.

★ **Formulaire :**

➢ **Champ d'email** : Champ de texte pour saisir l'adresse email.

➢ **Bouton d'envoi** : Bouton pour envoyer le lien de réinitialisation.

➢ **Méthode HTTP** : POST.

➢ **Action** : Le formulaire est soumis à la route password.email.

➢ **CSRF Protection**: Utilisation de @csrf pour protéger contre les attaques CSRF.

❖ **Réinitialisation du mot de passe (`reset.blade.php`) :**

➢ Cette vue permet à l'utilisateur de réinitialiser son mot de passe.

- **Contenu :**

➢ **Champ d'email** : Champ de texte pré-rempli avec l'adresse email de l'utilisateur.

➢ **Champ de mot de passe** : Champ de texte pour saisir le nouveau mot de passe.

➢ **Champ de confirmation du mot de passe** : Champ de texte pour confirmer le nouveau mot de passe.

➢ **Bouton de réinitialisation** : Bouton pour soumettre le formulaire.

➢ **Méthode HTTP** : POST.

➢ **Action** : Le formulaire est soumis à la route password.update.

➢ **CSRF Protection** : Utilisation de @csrf pour protéger contre les attaques CSRF.

➢ **Token** : Un champ caché contient le token de réinitialisation.

❖ **Structure commune:**

➢ **Layout** : Toutes les vues étendent le layout layouts.app via @extends('layouts.app') et injectent leur contenu dans la section content via @section('content').

➢ **Conteneur** : Le contenu est encapsulé dans un conteneur Bootstrap pour une mise en page cohérente.

❖ **Routes et contrôleurs :**

➢ **Confirmation du mot de passe** : Le formulaire est soumis à `password.confirm`, ce qui suggère l'existence d'une méthode `confirm` dans le contrôleur `Auth\ConfirmPasswordController`.
➢ **Demande de réinitialisation** : Le formulaire est soumis à `password.email`, ce qui suggère l'existence d'une méthode `sendResetLinkEmail` dans le contrôleur `Auth\ForgotPasswordController`.
➢ **Réinitialisation du mot de passe** : Le formulaire est soumis à `password.update`, ce qui suggère l'existence d'une méthode `reset` dans le contrôleur `Auth\ResetPasswordController`.
➢ **Connexion** : Le formulaire est soumis à `login`, ce qui suggère l'existence d'une méthode `login` dans le contrôleur `Auth\LoginController`.
➢ **Inscription** : Le formulaire est soumis à `register`, ce qui suggère l'existence d'une méthode `register` dans le contrôleur `Auth\RegisterController`.
➢ **Vérification de l'email** : Le formulaire est soumis à `verification.resend`, ce qui suggère l'existence d'une méthode `resend` dans le contrôleur `Auth\VerificationController.`

➔ **history**
❖ (`history.blade.php`) :

➢ Cette vue permet à l'utilisateur de consulter son historique d'articles.

• **Structure :**
➢ Étend le layout principal (`layouts.app`)
➢ Contenu placé dans une section `content`
➢ Utilise une classe container pour le conteneur principal

• **Formulaire de filtrage :**
➢ **Méthode HTTP** : GET
➢ **Action**: Route history.index
➢ **Champs** :
* **Filtre par date** : Input de type date
* **Filtre par thème** : Select avec liste des thèmes disponibles
➢ **Bouton** : "Filtrer" pour appliquer les filtres

• **Affichage de l'historique :**
➢ Liste non ordonnée contenant les entrées

➢ Pour chaque entrée :
 * Titre de l'article
 * Contenu de l'article
 * Date et heure de consultation (format jj/mm/aaaa hh:mm)
 * Lien vers l'article complet
➢ Message "Aucun historique disponible" si la liste est vide

• **Fonctionnalités techniques :**
➢ Utilisation de @forelse/@empty pour la gestion conditionnelle
➢ Exploitation des relations entre modèles (article, theme)
➢ Conservation des valeurs de filtrage via request()

➔ **theme**

❖ **Créer un Nouveau Thème (`create.blade.php`) :**
➢ Cette vue est une page permettant aux utilisateurs de créer un nouveau thème. Elle comprend les éléments suivants :

➢ **Mise en Page** : La vue étend la mise en page principale (`layouts.app`).

• **Contenu Principal :**
➢ **Conteneur de Formulaire** : Une division principale avec une classe theme-form pour styliser le formulaire.
➢ **Titre** : Un titre `Créer un nouveau thème`.
★ **Formulaire** : Un formulaire permettant de soumettre les informations pour créer un nouveau thème.
➢ Action : Le formulaire envoie les données à la route themes.store via la méthode POST.
➢ **CSRF** : Protection CSRF incluse pour sécuriser les soumissions de formulaire.
➢ **Champs du Formulaire** :
- Nom du Thème : Un champ texte obligatoire pour le nom du thème.
- Description : Une zone de texte pour la description du thème.
- Image : Un champ de fichier pour télécharger une image.
➢ **Bouton de Soumission** : Un bouton pour soumettre le formulaire et créer le thème.

❖ la Liste des Thèmes (`index.blade.php`) :

➢ Cette vue affiche une liste de thèmes disponibles dans l'application. Elle comprend les éléments suivants :

- **Contenu Principal :**

➢ **Conteneur Principal** : Une division principale contenant la liste des thèmes.

➢ **Titre** : Un titre `Liste des thèmes`.

- **Bouton pour Créer un Nouveau Thème :**

➢ **Bouton** : Un bouton `Créer un nouveau thème` qui renvoie à la route `themes.create`.

- **Liste des Thèmes :**

➢ **Boucle `@foreach`** : Parcourt chaque thème et affiche ses détails.

➢ **Nom du Thème** : Un titre pour le nom du thème.

➢ **Description** : Une description du thème.

➢ **Image** : Si le thème a une image, elle est affichée avec une largeur de 200 pixels.

- **Bouton pour Voir les Articles** : Un bouton Voir les articles qui renvoie à la route `themes.show` pour afficher les articles du thème.
- **Bouton pour Créer un Article** : Un bouton Créer un article qui renvoie à la route `articles.create` pour créer un nouvel article pour ce thème.
- **Bouton pour Voir les Statistiques** : Un lien pour voir les statistiques du thème via la route `themes.statistics`.
- **bouton pour voir les notifications des articles**:un bouton redirige vers la routes thermes.articles_status avec l'ID du thème pour afficher les notifications ou le statut des articles

❖ les Articles d'un Thème (`show.blade.php`) :

➢ Cette vue affiche la liste des articles associés à un thème spécifique. Elle comprend les éléments suivants :

- **Contenu Principal :**

➢ **Conteneur Principal** : Une division principale contenant la liste des articles.
➢ **Titre** : Un titre `Articles du thème : {{ $theme->name }}` qui affiche le nom du thème.

- **Liste des Articles :**

➢ **Boucle `@foreach`**: Parcourt chaque article du thème et affiche ses détails.
➢ **Titre de l'Article** : Un titre pour le nom de l'article.
➢ **Contenu** : Un paragraphe pour le contenu de l'article.
➢ **Image** : Si l'article a une image, elle est affichée avec une largeur de 200 pixels.

- **Boutons d'Action pour Chaque Article :**

★ **Boutons :**

➢ **Lire l'Article** : Un bouton Lire l'article qui renvoie à la route `articles.show` pour afficher l'article complet.
➢ **Éditer l'Article** : Un bouton Éditer qui renvoie à la route `articles.edit` pour modifier l'article.
➢ **Supprimer l'Article** : Un bouton Supprimer qui utilise un formulaire pour envoyer une demande de suppression via la méthode DELETE, protégé par CSRF.

❖ **Statistiques pour un Thème (`statistics.blade.php`) :**

➢ Cette vue affiche les statistiques d'un thème spécifique dans l'application. Elle comprend les éléments suivants :

- **Contenu Principal :**

➢ **Conteneur Principal** : Une division principale contenant les statistiques.
➢ **Titre** : Un titre Statistiques pour le thème : {{ $theme->name }} affichant le nom du thème.

- **Statistiques des Articles :**

➢ **Liste d'Articles**: Une liste non ordonnée affichant
➢ **Total d'articles** : Nombre total d'articles.
➢ **Articles en cours** : Nombre d'articles en cours.
➢ **Articles publiés** : Nombre d'articles publiés.
➢ **Articles refusés** : Nombre d'articles refusés.

- **Statistiques des Abonnés :**

➢ **Abonnés** : Paragraphe affichant le nombre total d'abonnés.

- **Statistiques des Commentaires :**

➢ **Commentaires** : Paragraphe affichant le nombre total de commentaires.

- **Bouton de Retour :**

➢ **Bouton** : Un bouton Retour au thème qui renvoie à la route `themes.show` pour revenir à la vue du thème.

❖ `articles-statuts.blade.php`:

- **Articles en attente :**

➢ Cette section liste les articles dont le statut est "pending".

➢ Pour chaque article, le titre et le contenu sont affichés, ainsi qu'un indicateur visuel (en orange) indiquant que l'article est en attente.

➢ Un formulaire permet de modifier le statut de l'article en choisissant entre "accepted", "rejected", ou "pending".

- **Articles acceptés :**

➢ Cette section liste les articles dont le statut est "accepted".

➢ Pour chaque article, le titre est affiché avec un indicateur visuel (en vert) indiquant que l'article a été accepté.

➢ Un formulaire permet de modifier le statut de l'article en choisissant entre "rejected" ou "pending".

- **Articles refusés :**

➢ Cette section liste les articles dont le statut est "rejected".

➢ Pour chaque article, le titre est affiché avec un indicateur visuel (en rouge) indiquant que l'article a été refusé.

➢ Un formulaire permet de modifier le statut de l'article en choisissant entre "accepted" ou "pending".

- **Bouton de retour :**

➢ Un bouton permet de revenir à la liste des thèmes.

- **Fonctionnalités**

➢ **Modification du statut** : Chaque article est accompagné d'un formulaire qui permet de changer son statut. Les boutons du formulaire envoient une requête PATCH à la route `articles.update_status` avec l'ID de l'article et le nouveau statut.

➢ **Protection CSRF** : Les formulaires incluent un jeton CSRF pour protéger contre les attaques de type Cross-Site Request Forgery.

➢ **Gestion des états vides** : Si aucune catégorie d'articles (en attente, acceptés, refusés) ne contient d'articles, un message approprié est affiché.

➔ **editor**

❖ **dashboard.blade.php :**

➢ Cette vue est structurée pour présenter les statistiques récupérées par le contrôleur `EditorController`. Voici une analyse détaillée du code :

- **Structure de Base**

➢ **`@extends('layouts.app')`**: La vue étend un layout principal nommé `layouts.app`, ce qui permet de réutiliser une structure commune (en-tête, pied de page, etc.).

➢ `@section('content')` : Cette directive définit une section nommée `content` qui sera injectée dans le layout principal.

- **Titre Principal**

➢ Un titre principal est affiché pour indiquer la page actuelle (tableau de bord de l'éditeur).

- **Statistiques des Articles**

➢ **Titre**: Statistiques des Articles.

➢ **Liste**: Une liste non ordonnée (`<ul>`) est utilisée pour afficher les statistiques des articles.

➢ **Total d'articles** : Affiche le nombre total d'articles (`{{ $totalArticles }}`).

➢ **Articles en attente**: Affiche le nombre d'articles en attente (`{{ $articlesPending }}`).

➢ **Articles acceptés**: Affiche le nombre d'articles acceptés (`{{ $articlesAccepted }}`).

➢ **Articles refusés** : Affiche le nombre d'articles refusés (`{{ $articlesRejected }}`).

- **Statistiques des Abonnés**

➢ **Titre** : Statistiques des Abonnés
➢ **Paragraphe** : Affiche le nombre total d'abonnés (`{{ $totalSubscribers }}`).

- **Statistiques des Thèmes**

➢ **Titre** :Statistiques des Thèmes
➢ Liste: Une liste non ordonnée (`<ul>`) est utilisée pour afficher les statistiques des thèmes.
➢ **Total de thèmes** : Affiche le nombre total de thèmes (`{{ $totalThemes }}`).

- **Top 5 Thèmes avec le plus d'articles**

➢ **Titre**: Top 5 Thèmes avec le plus d'articles
➢ **Liste** : Une liste non ordonnée (`<ul>`) est utilisée pour afficher les 5 thèmes les plus actifs.
➢ **Boucle `@foreach`** : Parcourt la variable `$themesWithMostArticles` (transmise par le contrôleur).
➢ **Pour chaque thème, affiche** :
- Le nom du thème (`{{ $theme->name }}`).
- Le nombre d'articles associés (`{{ $theme->articles_count }}`)

- **Top 5 Thèmes les plus populaires**

➢ **Titre** : Top 5 Thèmes les plus populaires</h3>`.
➢ **Liste** : Une liste non ordonnée (`<ul>`) est utilisée pour afficher les 5 thèmes les plus populaires.
- **Boucle `@foreach` : Parcourt la variable** `$themesWithMostSubscribers` (transmise par le contrôleur).

➢ **Pour chaque thème, affiche** :
- Le nom du thème (`{{ $theme->name }}`).
- Le nombre d'abonnés associés (`{{ $theme->subscribers_count }}`).

➔ home.blade.php :

- **Structure de Base**

➢ **`@extends('layouts.app')`** : Le template étend un layout principal nommé `layouts.app`. Cela permet de réutiliser une structure de base commune à plusieurs pages.

➢ **`@section('content')`** : Cette directive définit une section nommée `content` qui sera injectée dans le layout principal.

- **Barre de Navigation (`<nav class="navbar">`)**

➢ **Logo** : Le logo est composé d'une icône, d'une image (`log.png`), et d'un texte ("Tech Horizons").

- **Liens de Navigation :**

➢ **Home** : Redirige vers la page d'accueil (`/home`).

➢ **All Themes** : Redirige vers la page des thèmes (`/themes`).

➢ **About Us** : Redirige vers la page "À propos" (`/about-us`).

➢ **Actions** :

- Sign up : Bouton pour s'inscrire, redirige vers la page d'inscription (`/register`).
- Login : Bouton pour se connecter, redirige vers la page de connexion (`/login`).

- **Image de Fond (`<div class="background-image">`)**

➢ Contient un titre ("Welcome to Our Website") et un paragraphe invitant les utilisateurs à rester informés des dernières avancées technologiques.

- **Section des Thèmes (`<div class="newbar">`)**

➢ Titre : "Themes".

➢ **Slider de Thèmes** :

- Structure : Un conteneur (`slider-container`) contient une liste (`<ul>`) de thèmes.
- **Thèmes** : Chaque thème est représenté par une image, un titre, un texte de description, et un bouton "Read More".

➢ **Thèmes Inclus** :

- Artificial Intelligence : Image `IAimage.jpg`.
- cybersecurity : Image `cyberImage.jpg`.
- Virtual Reality : Image `VR.jpg`.
- Internet of Things : Image `inter.avif`.

➢ **Boutons de Navigation du Slider** :

- Un bouton avec une image `right.png` est inclus pour naviguer dans le slider. Cependant, la fonction `prevSlide()` est appelée, ce qui suggère une possible erreur (devrait probablement être `nextSlide()`).

7. Routes

❖ Web.php

➢ Il organise les routes pour les différentes fonctionnalités de l'application, telles que l'authentification, la gestion des thèmes, des articles, des commentaires, et des tableaux de bord. Voici une analyse détaillée du fichier :

- **Routes de Base**

➢ **Route d'Accueil** : Affiche la page d'accueil et est nommée `home`.
➢ **Route du Tableau de Bord Abonné** : Affiche le tableau de bord des abonnés. Cependant, il y a une faute de frappe dans l'URL (`/dashbord` au lieu de `/dashboard`).
➢ **Authentification** : Active les routes par défaut de Laravel pour l'authentification (connexion, inscription, etc.).
➢ **Page d'Accueil Authentifiée** : Affiche la page d'accueil après authentification.

- **Routes Protégées par Middleware**

➢ **Page Réservée aux Abonnés** : Accessible uniquement aux utilisateurs ayant le statut d'abonné.
➢ **Page Réservée aux Administrateurs** : Accessible uniquement aux utilisateurs ayant le statut d'administrateur.

- **Routes d'Authentification et d'Abonnement**

➢ **Page d'Abonnement**: Affiche la page d'abonnement.
➢ **Page de Connexion** : Affiche la page de connexion.

- **Routes des Thèmes**

➢ **Liste des Thèmes** : Affiche la liste des thèmes.

- **Création d'un Thème** : Affiche le formulaire de création d'un thème.
- Enregistrement d'un Thème : Enregistre un nouveau thème.
- **Détails d'un Thème** : Affiche les détails d'un thème spécifique.
- **Statistiques d'un Thème** : Affiche les statistiques d'un thème.
- **Statut des Articles d'un Thème** : Affiche le statut des articles associés à un thème.

- **Routes des Articles**
- **Création d'un Article**: Affiche le formulaire de création d'un article pour un thème spécifique.
- **Enregistrement d'un Article** : Enregistre un nouvel article pour un thème spécifique.
- **Détails d'un Article**: Affiche les détails d'un article.
- **Édition d'un Article** : Affiche le formulaire d'édition d'un article.
- **Mise à Jour d'un Article** : Met à jour un article.
- **Suppression d'un Article** : Supprime un article.
- **Mise à Jour du Statut d'un Article** : Met à jour le statut d'un article.

- **Routes des Commentaires**
- **Enregistrement d'un Commentaire**: Enregistre un nouveau commentaire pour un article.

- **Routes du Tableau de Bord de l'Éditeur**
- **Tableau de Bord de l'Éditeur** : Affiche le tableau de bord de l'éditeur.

IV. conclusion

Ce projet Laravel est structuré pour gérer une application web complexe avec des fonctionnalités d'authentification, de gestion de contenu (articles, commentaires, thèmes), d'abonnements, et de gestion des utilisateurs. Les middlewares assurent la sécurité et l'accès aux différentes parties de l'application, tandis que les vues et les contrôleurs gèrent l'interface utilisateur et la logique métier.