C'est une excellente description de ton projet ! Je comprends bien les fonctionnalités que tu souhaites implémenter. Nous pouvons commencer à structurer le projet étape par étape. Voici un plan détaillé pour t'aider à démarrer l'implémentation :

---

### \*\*Plan du Projet\*\*

#### \*\*1. Structure de Base\*\*

- \*\*Installation de Laravel 11\*\* : Si ce n'est pas déjà fait, installe Laravel 11.

- \*\*Configuration de la base de données\*\* : Configure ton fichier `.env` pour te connecter à ta base de données.

- \*\*Authentification\*\* : Utilise le système d'authentification de Laravel pour gérer les connexions des utilisateurs (étudiants et administrateurs).

#### \*\*2. Modèles et Migrations\*\*

Nous allons créer les modèles et les migrations nécessaires pour gérer les différentes entités :

- \*\*Utilisateurs\*\* : `User` (étudiants et administrateurs).

- \*\*Universités\*\* : `University`.

- \*\*Filières\*\* : `Program`.

- \*\*Documents\*\* : `Document` (avec deux types : public et abonnement).

- \*\*Messages de contact\*\* : `ContactMessage`.

- \*\*Forum\*\* : `ForumPost` et `ForumReply`.

- \*\*Abonnements\*\* : `Subscription` (pour gérer les utilisateurs abonnés).

#### \*\*3. Routes et Contrôleurs\*\*

Nous allons créer des contrôleurs pour gérer les différentes fonctionnalités :

- \*\*Accueil\*\* : `HomeController`.

- \*\*Filières\*\* : `ProgramController`.

- \*\*Bibliothèque\*\* : `LibraryController`.

- \*\*Forum\*\* : `ForumController`.

- \*\*Contact\*\* : `ContactController`.

- \*\*Administration\*\* : `AdminController` (pour gérer les universités, filières, documents, messages, etc.).

#### \*\*4. Vues (Blade Templates)\*\*

Nous allons créer des vues pour chaque fonctionnalité :

- \*\*Accueil\*\* : `home.blade.php`.

- \*\*Filières\*\* : `programs.blade.php`.

- \*\*Bibliothèque\*\* : `library.blade.php`.

- \*\*Forum\*\* : `forum.blade.php`.

- \*\*Contact\*\* : `contact.blade.php`.

- \*\*Administration\*\* : `admin/dashboard.blade.php`, `admin/universities.blade.php`, etc.

#### \*\*5. Fonctionnalités Spécifiques\*\*

- \*\*Connexion des utilisateurs\*\* : Gérer la connexion en tant qu'étudiant ou administrateur.

- \*\*Approbation des administrateurs\*\* : Un administrateur existant doit approuver les nouveaux administrateurs.

- \*\*Gestion des documents\*\* : Deux types de documents (publics et abonnés).

- \*\*Abonnements\*\* : Gérer les utilisateurs abonnés pour accéder aux documents payants.

- \*\*Forum\*\* : Permettre aux utilisateurs de créer des discussions et d'y répondre.

- \*\*Contact\*\* : Permettre aux utilisateurs d'envoyer des messages.

---

### \*\*Étapes d'Implémentation\*\*

#### \*\*Étape 1 : Installation et Configuration\*\*

1. Installe Laravel 11 :

```bash

composer create-project laravel/laravel portail-universites

```

2. Configure ton fichier `.env` avec les informations de ta base de données.

#### \*\*Étape 2 : Authentification\*\*

1. Installe le système d'authentification de Laravel :

```bash

php artisan make:auth

```

2. Modifie la migration `users` pour ajouter un champ `role` (étudiant ou administrateur) et un champ `approved` (pour l'approbation des administrateurs) :

```php

// database/migrations/xxxx\_xx\_xx\_create\_users\_table.php

$table->string('role')->default('student'); // 'student' ou 'admin'

$table->boolean('approved')->default(false); // Pour l'approbation des admins

```

3. Exécute les migrations :

```bash

php artisan migrate

```

#### \*\*Étape 3 : Modèles et Migrations\*\*

1. Crée les modèles et les migrations pour les autres entités :

```bash

php artisan make:model University -m

php artisan make:model Program -m

php artisan make:model Document -m

php artisan make:model ContactMessage -m

php artisan make:model ForumPost -m

php artisan make:model ForumReply -m

php artisan make:model Subscription -m

```

2. Définis les relations dans les modèles. Par exemple :

- Un `User` peut avoir plusieurs `ForumPost`.

- Un `Document` appartient à une `University`.

- Un `Subscription` appartient à un `User`.

#### \*\*Étape 4 : Routes et Contrôleurs\*\*

1. Crée les contrôleurs :

```bash

php artisan make:controller HomeController

php artisan make:controller ProgramController

php artisan make:controller LibraryController

php artisan make:controller ForumController

php artisan make:controller ContactController

php artisan make:controller AdminController --resource

```

2. Définis les routes dans `routes/web.php` :

```php

// Routes publiques

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

Route::resource('programs', ProgramController::class);

Route::resource('library', LibraryController::class);

Route::resource('forum', ForumController::class);

Route::resource('contact', ContactController::class);

// Routes d'administration

Route::middleware(['auth', 'admin'])->group(function () {

Route::get('/admin', [AdminController::class, 'index'])->name('admin.dashboard');

Route::resource('admin/universities', AdminController::class);

Route::resource('admin/programs', AdminController::class);

Route::resource('admin/documents', AdminController::class);

Route::resource('admin/messages', AdminController::class);

Route::resource('admin/forum', AdminController::class);

});

```

#### \*\*Étape 5 : Vues\*\*

1. Crée les vues dans `resources/views` :

- `home.blade.php`

- `programs.blade.php`

- `library.blade.php`

- `forum.blade.php`

- `contact.blade.php`

- `admin/dashboard.blade.php`

- `admin/universities.blade.php`

- etc.

#### \*\*Étape 6 : Fonctionnalités Avancées\*\*

1. \*\*Approbation des administrateurs\*\* :

- Ajoute une logique dans le `AdminController` pour approuver les nouveaux administrateurs.

2. \*\*Gestion des documents\*\* :

- Ajoute une logique pour vérifier si un utilisateur est abonné avant d'accéder à un document payant.

3. \*\*Abonnements\*\* :

- Crée un système de paiement (via Stripe ou PayPal) pour gérer les abonnements.

---

### \*\*Prochaines Étapes\*\*

Si tu es prêt, nous pouvons commencer par implémenter l'authentification et la structure de base. Ensuite, nous passerons aux fonctionnalités spécifiques comme la gestion des filières, des documents, etc.

Dis-moi si tu veux que nous commencions par une partie en particulier ! 😊

La commande `php artisan make:auth` n'existe plus dans les versions récentes de Laravel (à partir de Laravel 6). Cette commande était utilisée pour générer rapidement un système d'authentification (connexion, inscription, etc.), mais elle a été supprimée pour encourager l'utilisation de packages dédiés comme \*\*Laravel Breeze\*\*, \*\*Laravel Jetstream\*\*, ou \*\*Laravel Fortify\*\*.

Ne t'inquiète pas, nous allons utiliser \*\*Laravel Breeze\*\*, qui est une solution simple et légère pour ajouter l'authentification à ton projet. Voici comment procéder :

---

### \*\*Étape 1 : Installer Laravel Breeze\*\*

1. \*\*Installe Laravel Breeze via Composer\*\* :

Dans ton terminal, exécute la commande suivante :

```bash

composer require laravel/breeze --dev

```

2. \*\*Installe les vues et les routes d'authentification\*\* :

Exécute la commande suivante pour générer les fichiers nécessaires :

```bash

php artisan breeze:install

```

3. \*\*Compile les assets (CSS et JavaScript)\*\* :

Laravel Breeze utilise Tailwind CSS pour le style. Exécute les commandes suivantes pour compiler les assets :

```bash

npm install && npm run dev

```

4. \*\*Exécute les migrations\*\* :

Laravel Breeze ajoute des tables pour gérer l'authentification. Exécute les migrations pour créer ces tables dans ta base de données :

```bash

php artisan migrate

```

---

### \*\*Étape 2 : Tester l'authentification\*\*

1. \*\*Lance le serveur de développement\*\* :

Exécute la commande suivante pour démarrer le serveur Laravel :

```bash

php artisan serve

```

2. \*\*Ouvre ton navigateur\*\* :

Visite `http://localhost:8000`. Tu devrais voir un lien "Log in" ou "Register" en haut à droite de la page.

3. \*\*Inscris-toi et connecte-toi\*\* :

- Clique sur "Register" pour créer un nouveau compte.

- Après t'être inscrit, tu seras automatiquement connecté.

---

### \*\*Étape 3 : Personnaliser l'authentification\*\*

Laravel Breeze génère des vues et des contrôleurs que tu peux personnaliser selon tes besoins.

1. \*\*Vues\*\* :

Les vues d'authentification se trouvent dans `resources/views/auth`. Tu peux les modifier pour adapter le design à ton projet.

2. \*\*Contrôleurs\*\* :

Les contrôleurs d'authentification se trouvent dans `app/Http/Controllers/Auth`. Tu peux les modifier pour ajouter des fonctionnalités personnalisées.

3. \*\*Routes\*\* :

Les routes d'authentification sont définies dans `routes/auth.php`. Tu peux les modifier si nécessaire.

---

### \*\*Étape 4 : Ajouter des rôles (étudiant/admin)\*\*

Pour gérer les rôles (étudiant/admin) et l'approbation des administrateurs, tu peux suivre ces étapes :

1. \*\*Ajoute un champ `role` et un champ `approved` à la table `users`\*\* :

Crée une nouvelle migration pour ajouter ces champs :

```bash

php artisan make:migration add\_role\_and\_approved\_to\_users\_table --table=users

```

Ouvre la migration générée et ajoute les champs :

```php

public function up()

{

Schema::table('users', function (Blueprint $table) {

$table->string('role')->default('student'); // 'student' ou 'admin'

$table->boolean('approved')->default(false); // Pour l'approbation des admins

});

}

```

Exécute la migration :

```bash

php artisan migrate

```

2. \*\*Modifie le modèle `User`\*\* :

Ouvre le modèle `User` dans `app/Models/User.php` et ajoute ces attributs remplissables :

```php

protected $fillable = [

'name', 'email', 'password', 'role', 'approved',

];

```

3. \*\*Crée un middleware pour vérifier les rôles\*\* :

Crée un middleware pour vérifier si un utilisateur est un administrateur approuvé :

```bash

php artisan make:middleware EnsureAdminIsApproved

```

Ouvre le middleware généré dans `app/Http/Middleware/EnsureAdminIsApproved.php` et ajoute la logique suivante :

```php

public function handle($request, Closure $next)

{

if (auth()->check() && auth()->user()->role === 'admin' && !auth()->user()->approved) {

return redirect()->route('home')->with('error', 'Vous devez être approuvé par un administrateur.');

}

return $next($request);

}

```

Enregistre ce middleware dans `app/Http/Kernel.php` :

```php

protected $routeMiddleware = [

// ...

'admin.approved' => \App\Http\Middleware\EnsureAdminIsApproved::class,

];

```

4. \*\*Protège les routes d'administration\*\* :

Dans `routes/web.php`, utilise ce middleware pour protéger les routes d'administration :

```php

Route::middleware(['auth', 'admin.approved'])->group(function () {

Route::get('/admin', [AdminController::class, 'index'])->name('admin.dashboard');

// Ajoute d'autres routes d'administration ici

});

```

---

### \*\*Résumé des Actions\*\*

1. Installe Laravel Breeze pour l'authentification.

2. Personnalise les vues et les contrôleurs selon tes besoins.

3. Ajoute des rôles (étudiant/admin) et un système d'approbation pour les administrateurs.

Si tu as des questions ou besoin d'aide pour une étape spécifique, n'hésite pas à demander ! 😊

Super ! Maintenant que nous avons configuré l'authentification avec \*\*Laravel Breeze\*\* et ajouté la gestion des rôles (étudiant/admin), passons à la suite pour dynamiser ton projet de portail des universités. Voici les étapes suivantes que nous allons aborder :

---

### \*\*Étape 5 : Créer les Modèles et Migrations pour les Fonctionnalités du Projet\*\*

Nous allons créer les modèles et les migrations pour gérer les universités, les filières, les documents, les messages de contact, les discussions du forum, et les abonnements.

1. \*\*Universités\*\* :

```bash

php artisan make:model University -m

```

Ouvre la migration générée et définis les champs :

```php

public function up()

{

Schema::create('universities', function (Blueprint $table) {

$table->id();

$table->string('name');

$table->text('description')->nullable();

$table->timestamps();

});

}

```

2. \*\*Filières\*\* :

```bash

php artisan make:model Program -m

```

Migration :

```php

public function up()

{

Schema::create('programs', function (Blueprint $table) {

$table->id();

$table->string('name');

$table->foreignId('university\_id')->constrained()->onDelete('cascade');

$table->timestamps();

});

}

```

3. \*\*Documents\*\* :

```bash

php artisan make:model Document -m

```

Migration :

```php

public function up()

{

Schema::create('documents', function (Blueprint $table) {

$table->id();

$table->string('title');

$table->string('file\_path');

$table->enum('type', ['public', 'subscription'])->default('public');

$table->foreignId('university\_id')->constrained()->onDelete('cascade');

$table->timestamps();

});

}

```

4. \*\*Messages de Contact\*\* :

```bash

php artisan make:model ContactMessage -m

```

Migration :

```php

public function up()

{

Schema::create('contact\_messages', function (Blueprint $table) {

$table->id();

$table->string('name');

$table->string('email');

$table->text('message');

$table->timestamps();

});

}

```

5. \*\*Forum (Discussions et Réponses)\*\* :

```bash

php artisan make:model ForumPost -m

php artisan make:model ForumReply -m

```

Migration pour `ForumPost` :

```php

public function up()

{

Schema::create('forum\_posts', function (Blueprint $table) {

$table->id();

$table->string('title');

$table->text('content');

$table->foreignId('user\_id')->constrained()->onDelete('cascade');

$table->timestamps();

});

}

```

Migration pour `ForumReply` :

```php

public function up()

{

Schema::create('forum\_replies', function (Blueprint $table) {

$table->id();

$table->text('content');

$table->foreignId('forum\_post\_id')->constrained()->onDelete('cascade');

$table->foreignId('user\_id')->constrained()->onDelete('cascade');

$table->timestamps();

});

}

```

6. \*\*Abonnements\*\* :

```bash

php artisan make:model Subscription -m

```

Migration :

```php

public function up()

{

Schema::create('subscriptions', function (Blueprint $table) {

$table->id();

$table->foreignId('user\_id')->constrained()->onDelete('cascade');

$table->date('start\_date');

$table->date('end\_date');

$table->timestamps();

});

}

```

7. \*\*Exécute les migrations\*\* :

```bash

php artisan migrate

```

---

### \*\*Étape 6 : Créer les Contrôleurs\*\*

Nous allons créer des contrôleurs pour gérer les différentes fonctionnalités.

1. \*\*Universités\*\* :

```bash

php artisan make:controller UniversityController --resource

```

2. \*\*Filières\*\* :

```bash

php artisan make:controller ProgramController --resource

```

3. \*\*Bibliothèque\*\* :

```bash

php artisan make:controller LibraryController --resource

```

4. \*\*Contact\*\* :

```bash

php artisan make:controller ContactController --resource

```

5. \*\*Forum\*\* :

```bash

php artisan make:controller ForumController --resource

```

6. \*\*Administration\*\* :

```bash

php artisan make:controller AdminController --resource

```

---

### \*\*Étape 7 : Définir les Routes\*\*

Dans `routes/web.php`, définis les routes pour chaque contrôleur.

Exemple :

```php

// Routes publiques

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

Route::resource('universities', UniversityController::class);

Route::resource('programs', ProgramController::class);

Route::resource('library', LibraryController::class);

Route::resource('contact', ContactController::class);

Route::resource('forum', ForumController::class);

// Routes d'administration

Route::middleware(['auth', 'admin.approved'])->group(function () {

Route::get('/admin', [AdminController::class, 'index'])->name('admin.dashboard');

Route::resource('admin/universities', AdminController::class);

Route::resource('admin/programs', AdminController::class);

Route::resource('admin/documents', AdminController::class);

Route::resource('admin/messages', AdminController::class);

Route::resource('admin/forum', AdminController::class);

});

```

---

### \*\*Étape 8 : Créer les Vues\*\*

Crée les vues dans `resources/views` pour chaque fonctionnalité.

Exemple de structure :

```

resources/views/

├── auth/

├── layouts/

├── universities/

│ └── index.blade.php

├── programs/

│ └── index.blade.php

├── library/

│ └── index.blade.php

├── contact/

│ └── index.blade.php

├── forum/

│ └── index.blade.php

└── admin/

└── dashboard.blade.php

```

---

### \*\*Étape 9 : Implémenter les Fonctionnalités\*\*

1. \*\*Universités\*\* :

- Affiche la liste des universités.

- Permets aux administrateurs d'ajouter, modifier ou supprimer des universités.

2. \*\*Filières\*\* :

- Affiche la liste des filières par université.

- Permets aux administrateurs de gérer les filières.

3. \*\*Bibliothèque\*\* :

- Affiche les documents publics et ceux réservés aux abonnés.

- Permets aux administrateurs d'ajouter ou de supprimer des documents.

4. \*\*Contact\*\* :

- Permets aux utilisateurs d'envoyer des messages.

- Affiche les messages reçus dans l'interface d'administration.

5. \*\*Forum\*\* :

- Permets aux utilisateurs de créer des discussions et d'y répondre.

- Donne aux administrateurs la possibilité de modérer les discussions.

6. \*\*Abonnements\*\* :

- Gère les abonnements pour accéder aux documents payants.

---

### \*\*Prochaines Étapes\*\*

Une fois que tu as terminé ces étapes, nous pouvons passer à :

- L'implémentation des fonctionnalités spécifiques (comme l'approbation des administrateurs, la gestion des abonnements, etc.).

- L'ajout de tests pour vérifier que tout fonctionne correctement.

- Le déploiement du projet.

Dis-moi si tu veux que nous approfondissions une partie en particulier ! 😊

Très bien ! Nous allons adapter le modèle `University` pour inclure les champs que tu as mentionnés. Voici comment procéder :

---

### \*\*Étape 1 : Modifier la Migration `universities`\*\*

Ouvre la migration que tu as créée pour la table `universities` (située dans `database/migrations/xxxx\_xx\_xx\_create\_universities\_table.php`) et ajoute les champs supplémentaires.

Voici à quoi ressemblera la migration mise à jour :

```php

public function up()

{

Schema::create('universities', function (Blueprint $table) {

$table->id();

$table->string('name');

$table->text('description')->nullable(); // Description : Koudougou

$table->string('address')->nullable(); // Adresse : BP 190

$table->string('website')->nullable(); // Site Web : Visitez le site

$table->string('region')->nullable(); // Région

$table->string('phone')->nullable(); // Téléphone

$table->text('student\_life')->nullable(); // Vie Étudiante

$table->text('achievements')->nullable(); // Travaux Réalisés

$table->timestamps();

});

}

```

---

### \*\*Étape 2 : Exécuter la Migration\*\*

Si tu as déjà exécuté la migration précédente, tu dois d'abord la rollback pour la modifier. Sinon, tu peux directement exécuter la migration.

1. \*\*Rollback de la migration\*\* (si elle a déjà été exécutée) :

```bash

php artisan migrate:rollback

```

2. \*\*Exécute la migration mise à jour\*\* :

```bash

php artisan migrate

```

---

### \*\*Étape 3 : Mettre à Jour le Modèle `University`\*\*

Ouvre le modèle `University` dans `app/Models/University.php` et ajoute les champs remplissables (`fillable`) pour permettre l'assignation de masse.

Voici à quoi ressemblera le modèle mis à jour :

```php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;

use Illuminate\Database\Eloquent\Model;

class University extends Model

{

use HasFactory;

protected $fillable = [

'name',

'description', // Description : Koudougou

'address', // Adresse : BP 190

'website', // Site Web : Visitez le site

'region', // Région

'phone', // Téléphone

'student\_life', // Vie Étudiante

'achievements', // Travaux Réalisés

];

}

```

---

### \*\*Étape 4 : Mettre à Jour le Contrôleur `UniversityController`\*\*

Ouvre le contrôleur `UniversityController` dans `app/Http/Controllers/UniversityController.php` et mets à jour les méthodes pour gérer les nouveaux champs.

Par exemple, dans la méthode `store` :

```php

public function store(Request $request)

{

$request->validate([

'name' => 'required|string|max:255',

'description' => 'nullable|string',

'address' => 'nullable|string|max:255',

'website' => 'nullable|url|max:255',

'region' => 'nullable|string|max:255',

'phone' => 'nullable|string|max:20',

'student\_life' => 'nullable|string',

'achievements' => 'nullable|string',

]);

University::create($request->all());

return redirect()->route('universities.index')

->with('success', 'Université créée avec succès.');

}

```

---

### \*\*Étape 5 : Mettre à Jour les Vues\*\*

1. \*\*Formulaire de création/édition\*\* :

Ouvre la vue pour créer ou modifier une université (par exemple, `resources/views/universities/create.blade.php` ou `resources/views/universities/edit.blade.php`) et ajoute les champs supplémentaires.

Exemple de formulaire :

```html

<form action="{{ route('universities.store') }}" method="POST">

@csrf

<div>

<label for="name">Nom de l'université :</label>

<input type="text" name="name" id="name" required>

</div>

<div>

<label for="description">Description :</label>

<textarea name="description" id="description"></textarea>

</div>

<div>

<label for="address">Adresse :</label>

<input type="text" name="address" id="address">

</div>

<div>

<label for="website">Site Web :</label>

<input type="url" name="website" id="website">

</div>

<div>

<label for="region">Région :</label>

<input type="text" name="region" id="region">

</div>

<div>

<label for="phone">Téléphone :</label>

<input type="text" name="phone" id="phone">

</div>

<div>

<label for="student\_life">Vie Étudiante :</label>

<textarea name="student\_life" id="student\_life"></textarea>

</div>

<div>

<label for="achievements">Travaux Réalisés :</label>

<textarea name="achievements" id="achievements"></textarea>

</div>

<button type="submit">Créer</button>

</form>

```

2. \*\*Affichage des détails\*\* :

Ouvre la vue pour afficher les détails d'une université (par exemple, `resources/views/universities/show.blade.php`) et ajoute les champs supplémentaires.

Exemple d'affichage :

```html

<h1>{{ $university->name }}</h1>

<p><strong>Description :</strong> {{ $university->description }}</p>

<p><strong>Adresse :</strong> {{ $university->address }}</p>

<p><strong>Site Web :</strong> <a href="{{ $university->website }}" target="\_blank">{{ $university->website }}</a></p>

<p><strong>Région :</strong> {{ $university->region }}</p>

<p><strong>Téléphone :</strong> {{ $university->phone }}</p>

<p><strong>Vie Étudiante :</strong> {{ $university->student\_life }}</p>

<p><strong>Travaux Réalisés :</strong> {{ $university->achievements }}</p>

```

---

### \*\*Étape 6 : Tester les Fonctionnalités\*\*

1. \*\*Crée une nouvelle université\*\* :

- Remplis le formulaire de création avec les nouveaux champs.

- Vérifie que les données sont bien enregistrées dans la base de données.

2. \*\*Affiche les détails d'une université\*\* :

- Vérifie que tous les champs sont correctement affichés dans la vue `show`.

3. \*\*Modifie une université\*\* :

- Vérifie que les champs peuvent être mis à jour via le formulaire d'édition.

---

### \*\*Résumé des Actions\*\*

1. Ajoute les champs supplémentaires à la migration `universities`.

2. Mets à jour le modèle `University` pour inclure les nouveaux champs.

3. Adapte le contrôleur `UniversityController` pour gérer les nouveaux champs.

4. Mets à jour les vues pour afficher et modifier les nouveaux champs.

Si tu as des questions ou besoin d'aide pour une étape spécifique, n'hésite pas à demander ! 😊

Très bien ! Avec la fonction `up()` de la migration `programs` que tu as fournie, nous allons continuer à implémenter les fonctionnalités en tenant compte de la structure de la table `programs`. Voici la suite des étapes pour implémenter les fonctionnalités du projet.

---

### \*\*Étape 9 : Implémenter les Fonctionnalités (Suite)\*\*

#### \*\*1. Relations entre `University` et `Program`\*\*

Une filière (`Program`) peut être associée à plusieurs universités (`University`), et une université peut avoir plusieurs filières. Nous allons donc créer une table pivot pour gérer cette relation many-to-many.

##### \*\*Créer la table pivot `program\_university`\*\*

1. \*\*Créer une migration pour la table pivot\*\* :

```bash

php artisan make:migration create\_program\_university\_table

```

2. \*\*Définir la structure de la table pivot\*\* :

Ouvre la migration générée dans `database/migrations/xxxx\_xx\_xx\_create\_program\_university\_table.php` et ajoute le code suivant :

```php

public function up()

{

Schema::create('program\_university', function (Blueprint $table) {

$table->id();

$table->foreignId('program\_id')->constrained()->onDelete('cascade');

$table->foreignId('university\_id')->constrained()->onDelete('cascade');

$table->timestamps();

});

}

```

3. \*\*Exécuter la migration\*\* :

```bash

php artisan migrate

```

##### \*\*Définir les relations dans les modèles\*\*

1. \*\*Modèle `University`\*\* :

Ouvre le modèle `University` dans `app/Models/University.php` et ajoute la relation `belongsToMany` :

```php

public function programs()

{

return $this->belongsToMany(Program::class, 'program\_university');

}

```

2. \*\*Modèle `Program`\*\* :

Ouvre le modèle `Program` dans `app/Models/Program.php` et ajoute la relation `belongsToMany` :

```php

public function universities()

{

return $this->belongsToMany(University::class, 'program\_university');

}

```

---

#### \*\*2. Gestion des Filières\*\*

##### \*\*Contrôleur `ProgramController`\*\*

Nous allons mettre à jour le contrôleur `ProgramController` pour gérer les filières en tenant compte de la relation many-to-many avec les universités.

1. \*\*Afficher la liste des filières\*\* :

```php

public function index()

{

$programs = Program::with('universities')->get();

return view('programs.index', compact('programs'));

}

```

2. \*\*Afficher le formulaire de création d'une filière\*\* :

```php

public function create()

{

$universities = University::all();

return view('programs.create', compact('universities'));

}

```

3. \*\*Enregistrer une nouvelle filière\*\* :

```php

public function store(Request $request)

{

$request->validate([

'name' => 'required|string|max:255',

'description' => 'nullable|string',

'entrants\_profile' => 'nullable|string',

'pathways' => 'nullable|string',

'outcomes' => 'nullable|string',

'universities' => 'required|array', // Liste des universités associées

]);

$program = Program::create($request->only([

'name', 'description', 'entrants\_profile', 'pathways', 'outcomes'

]));

$program->universities()->attach($request->universities);

return redirect()->route('programs.index')

->with('success', 'Filière créée avec succès.');

}

```

4. \*\*Afficher le formulaire de modification d'une filière\*\* :

```php

public function edit(Program $program)

{

$universities = University::all();

return view('programs.edit', compact('program', 'universities'));

}

```

5. \*\*Mettre à jour une filière\*\* :

```php

public function update(Request $request, Program $program)

{

$request->validate([

'name' => 'required|string|max:255',

'description' => 'nullable|string',

'entrants\_profile' => 'nullable|string',

'pathways' => 'nullable|string',

'outcomes' => 'nullable|string',

'universities' => 'required|array', // Liste des universités associées

]);

$program->update($request->only([

'name', 'description', 'entrants\_profile', 'pathways', 'outcomes'

]));

$program->universities()->sync($request->universities);

return redirect()->route('programs.index')

->with('success', 'Filière mise à jour avec succès.');

}

```

6. \*\*Supprimer une filière\*\* :

```php

public function destroy(Program $program)

{

$program->universities()->detach();

$program->delete();

return redirect()->route('programs.index')

->with('success', 'Filière supprimée avec succès.');

}

```

##### \*\*Vues\*\*

1. \*\*Liste des filières\*\* (`resources/views/programs/index.blade.php`) :

```html

@foreach ($programs as $program)

<div>

<h2>{{ $program->name }}</h2>

<p>{{ $program->description }}</p>

<p>Universités associées :</p>

<ul>

@foreach ($program->universities as $university)

<li>{{ $university->name }}</li>

@endforeach

</ul>

<a href="{{ route('programs.edit', $program) }}">Modifier</a>

<form action="{{ route('programs.destroy', $program) }}" method="POST">

@csrf

@method('DELETE')

<button type="submit">Supprimer</button>

</form>

</div>

@endforeach

```

2. \*\*Formulaire de création\*\* (`resources/views/programs/create.blade.php`) :

```html

<form action="{{ route('programs.store') }}" method="POST">

@csrf

<div>

<label for="name">Nom de la filière :</label>

<input type="text" name="name" id="name" required>

</div>

<div>

<label for="description">Description :</label>

<textarea name="description" id="description"></textarea>

</div>

<div>

<label for="entrants\_profile">Profil des entrants :</label>

<input type="text" name="entrants\_profile" id="entrants\_profile">

</div>

<div>

<label for="pathways">Parcours :</label>

<input type="text" name="pathways" id="pathways">

</div>

<div>

<label for="outcomes">Débouchés :</label>

<input type="text" name="outcomes" id="outcomes">

</div>

<div>

<label for="universities">Universités associées :</label>

<select name="universities[]" id="universities" multiple required>

@foreach ($universities as $university)

<option value="{{ $university->id }}">{{ $university->name }}</option>

@endforeach

</select>

</div>

<button type="submit">Créer</button>

</form>

```

3. \*\*Formulaire de modification\*\* (`resources/views/programs/edit.blade.php`) :

```html

<form action="{{ route('programs.update', $program) }}" method="POST">

@csrf

@method('PUT')

<div>

<label for="name">Nom de la filière :</label>

<input type="text" name="name" id="name" value="{{ $program->name }}" required>

</div>

<div>

<label for="description">Description :</label>

<textarea name="description" id="description">{{ $program->description }}</textarea>

</div>

<div>

<label for="entrants\_profile">Profil des entrants :</label>

<input type="text" name="entrants\_profile" id="entrants\_profile" value="{{ $program->entrants\_profile }}">

</div>

<div>

<label for="pathways">Parcours :</label>

<input type="text" name="pathways" id="pathways" value="{{ $program->pathways }}">

</div>

<div>

<label for="outcomes">Débouchés :</label>

<input type="text" name="outcomes" id="outcomes" value="{{ $program->outcomes }}">

</div>

<div>

<label for="universities">Universités associées :</label>

<select name="universities[]" id="universities" multiple required>

@foreach ($universities as $university)

<option value="{{ $university->id }}" {{ in\_array($university->id, $program->universities->pluck('id')->toArray()) ? 'selected' : '' }}>

{{ $university->name }}

</option>

@endforeach

</select>

</div>

<button type="submit">Mettre à jour</button>

</form>

```

---

#### \*\*3. Gestion des Documents\*\*

##### \*\*Modèle `Document`\*\*

Le modèle `Document` a les champs suivants :

- `title`

- `file\_path`

- `type` (`public` ou `subscription`)

- `program\_id` (clé étrangère vers `Program`)

##### \*\*Relations\*\*

- Un document appartient à une filière : relation `belongsTo` avec le modèle `Program`.

##### \*\*Contrôleur `LibraryController`\*\*

- \*\*Afficher les documents\*\* :

```php

public function index(Program $program)

{

$publicDocuments = $program->documents()->where('type', 'public')->get();

$subscriptionDocuments = $program->documents()->where('type', 'subscription')->get();

return view('library.index', compact('program', 'publicDocuments', 'subscriptionDocuments'));

}

```

- \*\*Ajouter un document\*\* :

```php

public function store(Request $request, Program $program)

{

$request->validate([

'title' => 'required|string|max:255',

'file' => 'required|file|mimes:pdf,doc,docx',

'type' => 'required|in:public,subscription',

]);

$filePath = $request->file('file')->store('documents');

$program->documents()->create([

'title' => $request->title,

'file\_path' => $filePath,

'type' => $request->type,

]);

return redirect()->route('library.index', $program)

->with('success', 'Document ajouté avec succès.');

}

```

- \*\*Supprimer un document\*\* :

```php

public function destroy(Document $document)

{

Storage::delete($document->file\_path);

$document->delete();

return redirect()->route('library.index', $document->program)

->with('success', 'Document supprimé avec succès.');

}

```

##### \*\*Vues\*\*

- \*\*Liste des documents\*\* (`resources/views/library/index.blade.php`) :

```html

<h2>Documents publics</h2>

@foreach ($publicDocuments as $document)

<div>

<h3>{{ $document->title }}</h3>

<a href="{{ Storage::url($document->file\_path) }}" target="\_blank">Télécharger</a>

</div>

@endforeach

<h2>Documents réservés aux abonnés</h2>

@foreach ($subscriptionDocuments as $document)

<div>

<h3>{{ $document->title }}</h3>

@if (auth()->user() && auth()->user()->hasActiveSubscription())

<a href="{{ Storage::url($document->file\_path) }}" target="\_blank">Télécharger</a>

@else

<p>Abonnez-vous pour accéder à ce document.</p>

@endif

</div>

@endforeach

```

---

#### \*\*4. Gestion des Abonnements\*\*

##### \*\*Modèle `Subscription`\*\*

Le modèle `Subscription` a les champs suivants :

- `user\_id` (clé étrangère vers `User`)

- `start\_date`

- `end\_date`

##### \*\*Contrôleur `SubscriptionController`\*\*

- \*\*Souscrire à un abonnement\*\* :

```php

public function store(Request $request)

{

$request->validate([

'start\_date' => 'required|date',

'end\_date' => 'required|date|after:start\_date',

]);

auth()->user()->subscriptions()->create($request->all());

return redirect()->route('subscription.index')

->with('success', 'Abonnement souscrit avec succès.');

}

```

##### \*\*Vues\*\*

- \*\*Formulaire d'abonnement\*\* (`resources/views/subscription/index.blade.php`) :

```html

<form action="{{ route('subscription.store') }}" method="POST">

@csrf

<div>

<label for="start\_date">Date de début :</label>

<input type="date" name="start\_date" id="start\_date" required>

</div>

<div>

<label for="end\_date">Date de fin :</label>

<input type="date" name="end\_date" id="end\_date" required>

</div>

<button type="submit">Souscrire</button>

</form>

```

---

### \*\*Prochaines Étapes\*\*

Une fois que tu as implémenté ces fonctionnalités, nous pouvons passer à :

- L'ajout de tests pour vérifier que tout fonctionne correctement.

- Le déploiement du projet.

Dis-moi si tu as des questions ou besoin d'aide pour une étape spécifique ! 😊

PAIEMENT :  
Pour intégrer **Flutterwave** dans ton projet Laravel, voici les étapes détaillées pour configurer et mettre en place un système de paiement via Flutterwave.

**1. Installation de la bibliothèque Flutterwave**

Commence par installer la bibliothèque officielle de Flutterwave pour Laravel. Utilise la commande suivante :

composer require kingflamez/laravelrave

Cette bibliothèque simplifie l'intégration de Flutterwave dans Laravel.

**2. Configuration de Flutterwave**

Après avoir installé la bibliothèque, tu dois configurer les clés d'API Flutterwave dans ton fichier .env.

Obtiens tes clés API depuis le [dashboard Flutterwave](https://www.flutterwave.com/).

Ajoute ces lignes à ton fichier .env :

FLUTTERWAVE\_SECRET\_KEY=your\_flutterwave\_secret\_key

FLUTTERWAVE\_PUBLIC\_KEY=your\_flutterwave\_public\_key

**3. Configuration du ServiceProvider**

Après l'installation de la bibliothèque, il faut publier les configurations en exécutant :

php artisan vendor:publish --provider="Kingflamez\Rave\RaveServiceProvider"

Cela va publier le fichier config/rave.php où tu peux vérifier ou ajouter tes clés API Flutterwave.

**4. Création d'une Route et d'un Contrôleur pour le Paiement**

Crée une route pour afficher la page de paiement et une autre pour traiter la demande de paiement.

**a. Route de paiement**

Dans routes/web.php, ajoute les routes pour l'affichage et le traitement du paiement.

use App\Http\Controllers\PaymentController;

Route::get('/checkout', [PaymentController::class, 'checkout'])->name('checkout');

Route::post('/checkout/process', [PaymentController::class, 'processPayment'])->name('checkout.process');

**b. Contrôleur de paiement**

Crée un contrôleur pour gérer la logique du paiement.

php artisan make:controller PaymentController

Dans le contrôleur PaymentController, ajoute la méthode checkout pour afficher le formulaire de paiement et la méthode processPayment pour traiter le paiement.

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use Kingflamez\Rave\Facades\Rave;

use App\Models\Subscription;

class PaymentController extends Controller

{

public function checkout()

{

return view('checkout');

}

public function processPayment(Request $request)

{

$data = [

'tx\_ref' => uniqid(), // Référence de la transaction

'amount' => 1000, // Montant en cents (10€)

'email' => auth()->user()->email,

'currency' => 'NGN', // La devise (ici Naira Nigérian, adapte selon ton besoin)

'order\_id' => uniqid(),

'redirect\_url' => route('checkout.success') // Redirection après paiement

];

// Redirection vers Flutterwave pour le paiement

$response = Rave::initializePayment($data);

if ($response['status'] == 'success') {

return redirect()->route('checkout.success');

} else {

return redirect()->route('home')->with('error', 'Échec du paiement');

}

}

public function success(Request $request)

{

// Vérifie le statut de la transaction

$paymentStatus = Rave::verifyTransaction($request->tx\_ref);

if ($paymentStatus['status'] == 'success') {

// Marque l'abonnement comme payé

Subscription::create([

'user\_id' => auth()->id(),

'plan' => 'premium',

'status' => 'paid',

'start\_date' => now(),

'end\_date' => now()->addMonth(),

]);

return redirect()->route('home')->with('success', 'Paiement réussi !');

}

return redirect()->route('home')->with('error', 'Échec du paiement');

}

}

**5. Création du formulaire de paiement**

Crée la vue pour afficher le formulaire de paiement. Par exemple, dans resources/views/checkout.blade.php :

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Paiement</title>

</head>

<body>

<form action="{{ route('checkout.process') }}" method="POST">

@csrf

<button type="submit">Payer 10€</button>

</form>

</body>

</html>

**6. Test du Paiement**

Lorsque tu cliques sur le bouton pour effectuer le paiement, cela va rediriger l'utilisateur vers la page de paiement de Flutterwave. Une fois que l'utilisateur a effectué le paiement, Flutterwave redirigera vers l'URL spécifiée (redirect\_url).

La méthode success dans ton contrôleur va ensuite vérifier si le paiement a été effectué avec succès et procéder à la mise à jour de l'abonnement.

**7. Personnalisation et Sécurité**

* Assure-toi de valider que l'utilisateur est bien connecté avant de lui permettre d'effectuer un paiement.
* Tu peux personnaliser la page de paiement pour afficher des informations supplémentaires (plan, prix, etc.).
* Utilise des webhooks ou des requêtes périodiques pour valider que la transaction est bien finalisée, même si l'utilisateur ne revient pas immédiatement.

**8. Résumé du Processus**

1. **Utilisateur** va sur la page de paiement.
2. **L'application** crée une requête de paiement et redirige vers Flutterwave.
3. **Utilisateur** effectue le paiement via Flutterwave.
4. **Flutterwave** redirige l'utilisateur vers l'URL de retour (redirect\_url).
5. **L'application** vérifie si le paiement est réussi, puis crée l'abonnement pour l'utilisateur.

Avec ce système, tu peux intégrer un paiement simple et sécurisé dans ton application Laravel avec Flutterwave pour gérer les abonnements des utilisateurs.