

Documentation du mini logiciel de gestion de billetterie

Isidore Kpabou

14 Août 2024

Sommaire

Sommaire	2
1 Introduction	5
2 Configuration du fichier .env	5
2.1 Paramètres Généraux	5
2.2 Clé API	5
2.3 Paramètres de Journalisation	5
2.4 4. Paramètres de Base de Données	6
2.5 Paramètres Cache et Sessions	6
2.6 Paramètres Redis	6
2.7 Paramètres d'Email	6
2.8 Authentification Google	7
3 Installation	7
3.1 Explication des commandes	8
3.2 Utilisateur par défaut	9
4 Architecture des Dossiers et Fichiers	9
5 Base de Données	12
5.1 Table users	12
5.2 Table order_intents	12
5.3 Table events	14
5.4 Table orders	15
5.5 Table ticket_types	16
5.6 Table tickets	16
5.7 Table background_images	17
6 Seeders	18
6.1 Seeder Principal : DatabaseSeeder	19
6.2 Détails des Seeders	19
6.2.1 RolesPermissionTableSeeder	19
6.2.2 UserSeeder	20
7 Documentation des Routes : API et Web	21
7.1 Introduction	21
7.2 Routes API	21
7.3 Routes Web	21
7.4 Utilisation des Middlewares	22
7.4.1 Middleware auth	22
7.4.2 Routes sans Middleware	22
7.5 Conclusion	23

8 Travail Réalisé dans les Contrôleurs	23
8.1 Contrôleur API : <code>EventController.php</code>	23
8.2 Explication du <code>EventController</code>	23
8.3 Contrôleur Général : <code>EventController.php</code>	26
8.4 Explication du <code>EventController</code>	27
8.4.1 Namespace et Importations	28
8.4.2 Middlewares	28
8.4.3 Méthode <code>index()</code>	28
8.4.4 Méthode <code>create()</code>	29
8.4.5 Méthode <code>store()</code>	29
8.4.6 Méthode <code>edit()</code>	30
8.4.7 Méthode <code>update()</code>	30
8.4.8 Méthode <code>destroy()</code>	32
8.4.9 Gestion des Tickets Associés	32
8.5 Contrôleur <code>FaonctionExterneController.php</code>	32
8.6 Contrôleur <code>OrderController.php</code>	33
8.7 Contrôleur <code>ProfileController.php</code>	33
8.8 RegisteredUserController	34
8.8.1 Description	34
8.8.2 Méthodes	34
8.8.3 <code>create</code>	34
8.8.4 <code>store</code>	34
8.8.5 Génération du Nom d'utilisateur et Mot de Passe	35
8.8.6 Envoi de l'E-mail de Confirmation	35
8.8.7 Création de l'utilisateur dans la base de données	36
8.8.8 Retour et Redirection	36
8.8.9 Gestion des Erreurs	36
8.9 AuthenticatedSessionController	37
8.9.1 Description	37
8.9.2 Méthodes	37
8.9.3 <code>create</code>	37
8.9.4 <code>store</code>	37
8.9.5 Recherche de l'utilisateur	37
8.9.6 Régénération de la Session	38
8.9.7 Redirection	38
8.9.8 <code>destroy</code>	38
8.9.9 Gestion des Erreurs	38
9 Structure des Ressources de l'Application	39
9.1 Répertoire <code>views</code>	39
9.1.1 Répertoire <code>emails</code>	39
9.1.2 Répertoire <code>events</code>	39
9.1.3 Répertoire <code>layouts</code>	39
9.1.4 Répertoire <code>orders</code>	39
9.1.5 Autres Répertoires	40
9.2 Extending <code>app.layout</code>	40
9.3 Explication du Fichier <code>index.blade.php</code>	40
9.4 Script JavaScript	46

10 Démonstration	50
10.1 Welcome page	50
10.1.1 Bannière Principale	50
10.1.2 Présentation des Événements	50
10.1.3 Pied de page	51
10.2 Liste des Événements	51
10.2.1 Bannière Principale	51
10.2.2 Filtre de Recherche	51
10.2.3 Liste des Événements	52
10.2.4 Procédure d'achat de ticket	52
10.2.5 Pied de page	54
10.3 Page d'Inscription	55
10.3.1 Formulaire d'Inscription	55
10.3.2 Boutons d'Action	55
10.3.3 Illustration Visuelle	56
11 Événements	56
11.1 Publication d'événements	56
11.2 Accéder à la page des événements	56
11.3 Ajouter un nouvel événement	57
11.4 Filtrer les événements	58
12 Gestion des Commandes	59
12.1 Accéder à la page des commandes	59
12.2 Détails de la commande	59
12.3 Gestion des commandes	59
12.4 Rechercher une commande spécifique	60
12.5 Visualiser plus de commandes	60
13 Démonstration : Mise à Jour du Profil Utilisateur	60
13.1 Figure	60
13.2 Explication des fonctionnalités	61
14 Modification des différents images d'arrière plan de login	62
15 Conclusion	62

1 Introduction

Une API (Application Programming Interface) est un ensemble de règles et de protocoles qui permet à différentes applications de communiquer entre elles de manière fluide et structurée. Dans le cadre d'une billetterie événementielle, l'API joue un rôle central en facilitant l'intégration des systèmes de partenaires tels que les points de vente, les plateformes de promotion, et d'autres services. Elle permet à ces partenaires d'accéder aux données d'événements, de passer des commandes, de générer et de gérer des tickets en toute sécurité, garantissant ainsi une expérience utilisateur harmonisée et efficiente.

2 Configuration du fichier .env

Le fichier `.env` contient les paramètres essentiels de l'application. Voici les différents groupes de paramètres que vous devrez configurer.

2.1 Paramètres Généraux

Les paramètres généraux de l'application définissent son nom, son environnement, sa clé de sécurité, et d'autres informations importantes.

Paramètres Généraux

```
APP_NAME="Events"
APP_ENV=local
APP_KEY=base64:9HUwMbTQ3bUE/YhngB+rwr0I00WyTUKMwxtNXeJvAJI=
APP_DEBUG=true
APP_URL=http://localhost
LOGO_PATH=images/logo/logo.png
APP_LOGO="imagesdefaut/default-logo.png"
```

2.2 Clé API

Cette section définit la clé API utilisée pour authentifier les requêtes externes.

Clé API

```
API_KEY=0x-889-y-erta
```

2.3 Paramètres de Journalisation

Ces paramètres contrôlent la gestion des logs et leur niveau de détail.

Journalisation

```
LOG_CHANNEL=stack
LOG_DEPRECATED_CHANNEL=null
LOG_LEVEL=debug
```

2.4 4. Paramètres de Base de Données

Les paramètres de connexion à la base de données permettent à l'application de se connecter à votre serveur de base de données PostgreSQL.

Base de Données

```
DB_CONNECTION=pgsql
DB_HOST=127.0.0.1
DB_PORT=5432
DB_DATABASE=events
DB_USERNAME=isidoredev
DB_PASSWORD=isidoredev
```

2.5 Paramètres Cache et Sessions

Ces paramètres définissent les configurations pour le cache et les sessions de l'application.

Cache et Sessions

```
BROADCAST_DRIVER=log
CACHE_DRIVER=file
FILESYSTEM_DISK=local
QUEUE_CONNECTION=sync
SESSION_DRIVER=file
SESSION_LIFETIME=120
```

2.6 Paramètres Redis

Ces variables sont utilisées pour la connexion à un serveur Redis, souvent utilisé pour la gestion des caches ou des files d'attente.

Redis

```
REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379
```

2.7 Paramètres d'Email

Pour l'envoi d'emails depuis l'application, ces variables configurent le service SMTP de Google.

Email

```
MAIL_MAILER=smtp
MAIL_HOST=smtp.googlemail.com
MAIL_PORT=465
MAIL_USERNAME=davimadevima@gmail.com
MAIL_PASSWORD=wpayahkommetwwcx
MAIL_ENCRYPTION=ssl
MAIL_FROM_ADDRESS="davimadevima@gmail.com"
MAIL_FROM_NAME="${APP_NAME}"
```

2.8 Authentification Google

Ces variables sont utilisées pour configurer l'authentification via Google OAuth.

Google Auth

```
GOOGLE_CLIENT_ID=371807713804-4tisll881mtf.....googleusercontent.com
GOOGLE_CLIENT_SECRET=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
GOOGLE_CLIENT_CALLBACK=http://127.0.0.1:8000/callback/google
```

3 Installation

Cette section détaille les étapes pour installer et configurer correctement le projet. Suivez les commandes ci-dessous pour installer les dépendances, générer les caches, et configurer les liens de stockage.

Commandes d'installation

Les commandes suivantes doivent être exécutées dans l'ordre :

- **composer install** : Installe toutes les dépendances du projet.
- **composer dumpautoload** : Génère l'autoloader des classes du projet.
- **php artisan migrate:fresh --seed** : Exécute les migrations de la base de données et insère des données de test à partir des seeders.
- **php artisan route:cache** : Met en cache les routes pour une meilleure performance.
- **php artisan route:clear** : Efface le cache des routes.
- **php artisan config:cache** : Met en cache les fichiers de configuration.
- **php artisan config:clear** : Efface le cache de la configuration.
- **php artisan view:cache** : Met en cache les vues Blade compilées.
- **php artisan view:clear** : Efface le cache des vues compilées.
- **php artisan optimize:clear** : Efface tous les caches (route, config, vue).
- **php artisan optimize** : Optimise le cache pour une meilleure performance globale.
- **php artisan storage:unlink** : Supprime le lien symbolique du stockage public si présent.
- **php artisan storage:link** : Crée un lien symbolique entre le répertoire de stockage et le répertoire public.

3.1 Explication des commandes

- **composer install** : Cette commande installe toutes les dépendances du projet définies dans le fichier `composer.json`.
- **composer dumpautoload** : Elle génère un fichier `autoload.php` qui permet à PHP de charger automatiquement les classes nécessaires sans inclure manuellement les fichiers.
- **php artisan migrate :fresh --seed** : Cette commande recrée la base de données à partir des migrations et exécute les seeders pour insérer des données initiales.
- **php artisan route :cache** et **php artisan route :clear** : Elles sont utilisées pour optimiser et gérer le cache des routes Laravel.
- **php artisan config :cache** et **php artisan config :clear** : Ces commandes permettent de mettre en cache les configurations de Laravel pour une exécution plus rapide.
- **php artisan view :cache** et **php artisan view :clear** : Elles optimisent et nettoient les vues compilées, réduisant ainsi les temps de chargement.
- **php artisan optimize :clear** et **php artisan optimize** : Ces commandes effacent et recréent les caches pour améliorer la performance globale du projet.
- **php artisan storage :unlink** et **php artisan storage :link** : Utilisées pour gérer le lien symbolique entre les fichiers de stockage et le répertoire public.

3.2 Utilisateur par défaut

Lors de l'exécution des seeders avec la commande `php artisan migrate:fresh --seed`, un utilisateur par défaut est créé. Cet utilisateur peut se connecter à l'application en utilisant soit son email, soit son nom d'utilisateur.

Utilisateur par défaut

Les informations de connexion de l'utilisateur par défaut sont les suivantes :

- **Email** : `dev@isidore.com`
- **Nom d'utilisateur** : `dev_isidore`
- **Mot de passe** : `_123456789`

L'utilisateur peut se connecter à l'application soit en utilisant l'adresse e-mail, soit le nom d'utilisateur, accompagné du mot de passe défini.

4 Architecture des Dossiers et Fichiers

L'architecture de l'application est structurée comme suit :

Structure de Répertoire

```
app/
|----- Console/
|----- Controllers/
| |----- API/
| | |----- EventController.php
| | |----- OrderController.php
| | |----- OrderIntentController.php
| | |----- TicketController.php
| | |----- TicketTypeController.php
| |----- Auth/
| | |----- BackgroundImageController.php
| | |----- DashController.php
| | |----- EventController.php
| | |----- FonctionExterneController.php
| | |----- OrderController.php
| | |----- ProfileController.php
| | |----- RoleController.php
| | |----- SocialiteController.php
| | |----- UsersController.php
|----- Middleware/
|----- Requests/
|----- Models/
|----- Providers/
|----- Rules/

bootstrap/
config/
database/
|----- factories/
|----- migrations/
| |----- 2014_10_12_000000_create_users_table.php
| |----- 2014_10_12_100000_create_password_resets_table.php
| |----- 2019_08_19_000000_create_failed_jobs_table.php
| |----- 2019_12_14_000001_create_personal_access_tokens_table.php
| |----- 2023_10_27_092802_create_permission_table.php
| |----- 2024_07_30_092022_create_background_images_table.php
| |----- 2024_08_13_080127_create_order_intents_table.php
| |----- 2024_08_13_080143_create_events_table.php
| |----- 2024_08_13_080205_create_orders_table.php
| |----- 2024_08_13_080201_create_ticket_types_table.php
| |----- 2024_08_13_080206_create_tickets_table.php
```

Structure de Répertoire

```
|---- seeders/
| |---- DatabaseSeeder.php
| |---- InsertionSeeder.php
| |---- RolesPermissionTableSeeder.php
| |---- UserSeeder.php
public/
resources/
|---- css/
|---- js/
|---- views/
| |---- Accueil/
| | |---- infos_payment.blade.php
| | |---- payment_confirm.blade.php
| | |---- welcome_events.blade.php
| | |---- welcome.blade.php
| |---- auth/
| |---- background_image/
| |---- components/
| |---- dashboard.blade.php
| |---- flash-message.blade.php
| |---- layouts/
| | |---- app.blade.php
| | |---- auth.blade.php
| | |---- welcome.blade.php
| |---- orders/
| | |---- index.blade.php
| |---- parametrage/
| |---- profile/
| |---- roles/
| |---- tickets/
| |---- users/
| |---- vendor/
```

- **app/** : Contient toute la logique de l'application avec les contrôleurs, les middlewares, les requêtes, et les modèles.
- **Console/** : Spécifie les commandes artisan personnalisées.
- **Controllers/** : Divisé en sous-dossiers **API** et **Auth** pour gérer les différentes fonctionnalités du système, comme les événements, les commandes, l'authentification, etc.
- **migrations/** : Contient toutes les migrations de la base de données.
- **seeders/** : Gère les fichiers permettant de pré-remplir la base de données.
- **resources/** : Regroupe les vues, fichiers CSS, JS et d'autres ressources statiques pour l'interface utilisateur.
- **routes/** : Contient les fichiers de routage pour les différentes sections de l'application (API, web, auth, etc.).
- **public/** : Dossier destiné aux ressources accessibles publiquement.
- **tests/** : Contient les tests automatisés de l'application.

Cette architecture est représentative de la structure d'une application Laravel, et vous pouvez ajouter plus de détails ou des sections spécifiques si nécessaire.

5 Base de Données

Cette section décrit la structure des principales tables utilisées dans l'application de gestion de billetterie.

5.1 Table users



```
● ● ●
1 <?php
2
3 use Illuminate\Database\Migrations\Migration;
4 use Illuminate\Database\Schema\Blueprint;
5 use Illuminate\Support\Facades\Schema;
6
7 return new class extends Migration
8 {
9     /**
10      * Run the migrations.
11      */
12     public function up(): void
13     {
14         Schema::create('users', function (Blueprint $table) {
15             $table->id();
16             $table->string('firstname');
17             $table->string('lastname');
18             $table->date('datenaiss')->nullable();
19             // entreprise,adresse,ville
20
21             $table->string('adresse')->nullable();
22             $table->string('ville')->nullable();
23             $table->string('entreprise')->nullable();
24             $table->string('lieu naissance')->nullable();
25             $table->string('indicatiftel')->nullable();
26             $table->string('telephone')->nullable();
27             $table->string('sexe')->nullable();
28             $table->string('username')->unique();
29             $table->string('email')->unique()->nullable();
30             $table->string('password');
31             $table->string('avatar')->nullable();
32
33             $table->rememberToken();
34             $table->timestamps();
35         });
36     }
37
38     /**
39      * Reverse the migrations.
40      */
41     public function down(): void
42     {
43         Schema::dropIfExists('users');
44     }
45 };
46
```

La table `users` stocke les informations des utilisateurs du système. Voici les principales colonnes :

- `id` : Clé primaire.
- `firstname`, `lastname` : Prénom et nom de l'utilisateur.
- `datenaiss` : Date de naissance (facultative).
- `adresse`, `ville`, `entreprise` : Informations supplémentaires sur l'utilisateur.
- `email`, `username` : Adresse email et nom d'utilisateur, uniques.
- `password` : Mot de passe de l'utilisateur.
- `avatar` : Chemin de l'avatar de l'utilisateur.

5.2 Table order_intents

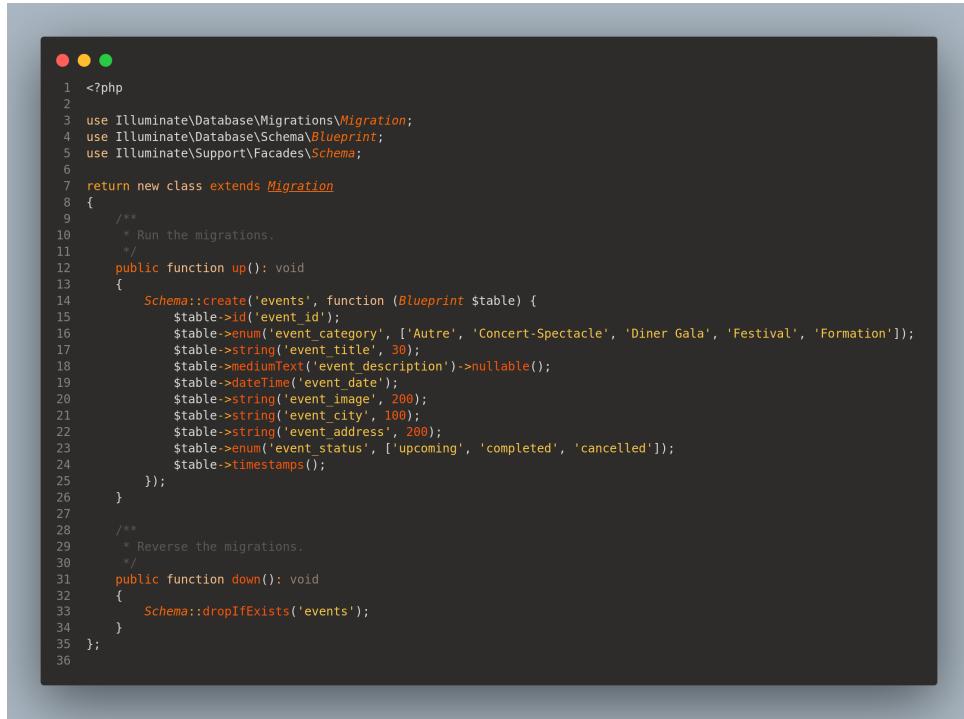
La table `order_intents` stocke les intentions de commande avant qu'elles ne soient validées. Voici les principales colonnes :



```
1 <?php
2
3 use Illuminate\Database\Migrations\Migration;
4 use Illuminate\Database\Schema\Blueprint;
5 use Illuminate\Support\Facades\Schema;
6
7 return new class extends Migration
8 {
9     /**
10      * Run the migrations.
11      */
12     public function up(): void
13     {
14         Schema::create('users', function (Blueprint $table) {
15             $table->id();
16             $table->string('firstname');
17             $table->string('lastname');
18             $table->date('datenaiss')->nullable();
19             // entreprise,adresse,ville
20
21             $table->string('adresse')->nullable();
22             $table->string('ville')->nullable();
23             $table->string('entreprise')->nullable();
24             $table->string('lieu_naissance')->nullable();
25             $table->string('indicatiftel')->nullable();
26             $table->string('telephone')->nullable();
27             $table->string('sexe')->nullable();
28             $table->string('username')->unique();
29             $table->string('email')->unique()->nullable();
30             $table->string('password');
31             $table->string('avatar')->nullable();
32
33             $table->rememberToken();
34             $table->timestamps();
35         });
36     }
37
38     /**
39      * Reverse the migrations.
40      */
41     public function down(): void
42     {
43         Schema::dropIfExists('users');
44     }
45 };
46
```

- `order_intent_id` : Clé primaire.
- `order_intent_price`, `order_intent_type` : Prix et type de l'intention de commande.
- `user_email`, `user_phone` : Informations de contact de l'utilisateur.
- `expiration_date` : Date d'expiration de l'intention de commande.

5.3 Table events

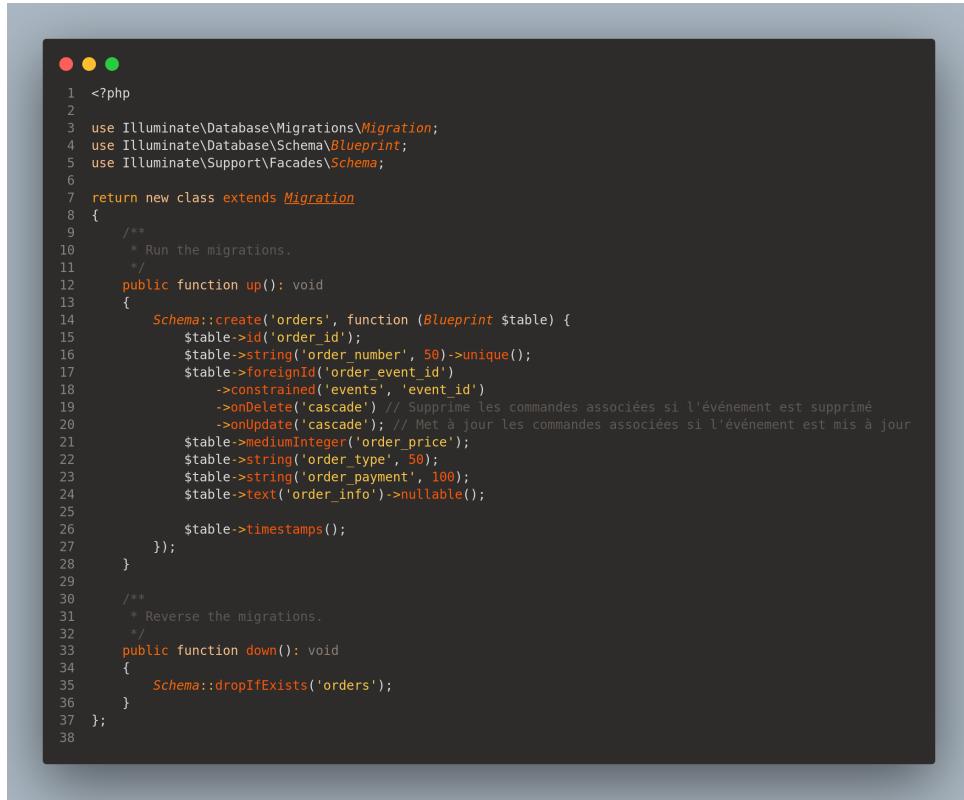


```
1 <?php
2
3 use Illuminate\Database\Migrations\Migration;
4 use Illuminate\Database\Schema\Blueprint;
5 use Illuminate\Support\Facades\Schema;
6
7 return new class extends Migration
8 {
9     /**
10      * Run the migrations.
11      */
12     public function up(): void
13     {
14         Schema::create('events', function (Blueprint $table) {
15             $table->id('event_id');
16             $table->enum('event_category', ['Autre', 'Concert-Spectacle', 'Diner Gala', 'Festival', 'Formation']);
17             $table->string('event_title', 30);
18             $table->mediumText('event_description')->nullable();
19             $table->dateTime('event_date');
20             $table->string('event_image', 200);
21             $table->string('event_city', 100);
22             $table->string('event_address', 200);
23             $table->enum('event_status', ['upcoming', 'completed', 'cancelled']);
24             $table->timestamps();
25         });
26     }
27
28     /**
29      * Reverse the migrations.
30      */
31     public function down(): void
32     {
33         Schema::dropIfExists('events');
34     }
35 };
36
```

La table **events** stocke les informations sur les événements disponibles dans la billetterie. Voici les principales colonnes :

- **event_id** : Clé primaire.
- **event_category**, **event_title**, **event_description** : Catégorie, titre et description de l'événement.
- **event_date** : Date de l'événement.
- **event_city**, **event_address** : Ville et adresse de l'événement.
- **event_status** : Statut de l'événement (**upcoming**, **completed**, **cancelled**).

5.4 Table orders

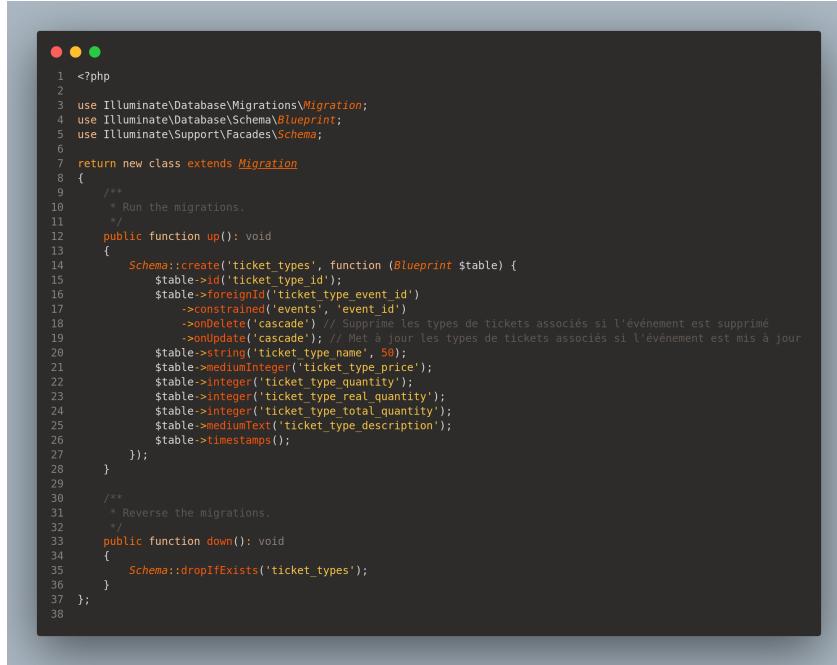


```
1 <?php
2
3 use Illuminate\Database\Migrations\Migration;
4 use Illuminate\Database\Schema\Blueprint;
5 use Illuminate\Support\Facades\Schema;
6
7 return new class extends Migration
8 {
9     /**
10      * Run the migrations.
11      */
12     public function up(): void
13     {
14         Schema::create('orders', function (Blueprint $table) {
15             $table->id('order_id');
16             $table->string('order_number', 50)->unique();
17             $table->foreignId('order_event_id')
18                 ->constrained('events', 'event_id')
19                 ->onDelete('cascade') // Supprime les commandes associées si l'événement est supprimé
20                 ->onUpdate('cascade'); // Met à jour les commandes associées si l'événement est mis à jour
21             $table->mediumInteger('order_price');
22             $table->string('order_type', 50);
23             $table->string('order_payment', 100);
24             $table->text('order_info')->nullable();
25
26             $table->timestamps();
27         });
28     }
29
30     /**
31      * Reverse the migrations.
32      */
33     public function down(): void
34     {
35         Schema::dropIfExists('orders');
36     }
37 };
38
```

La table `orders` stocke les commandes effectuées par les utilisateurs. Voici les principales colonnes :

- `order_id` : Clé primaire.
- `order_number` : Numéro unique de la commande.
- `order_event_id` : Référence vers l'événement associé à la commande.
- `order_price`, `order_type`, `order_payment` : Informations de prix, type et mode de paiement.

5.5 Table ticket_types

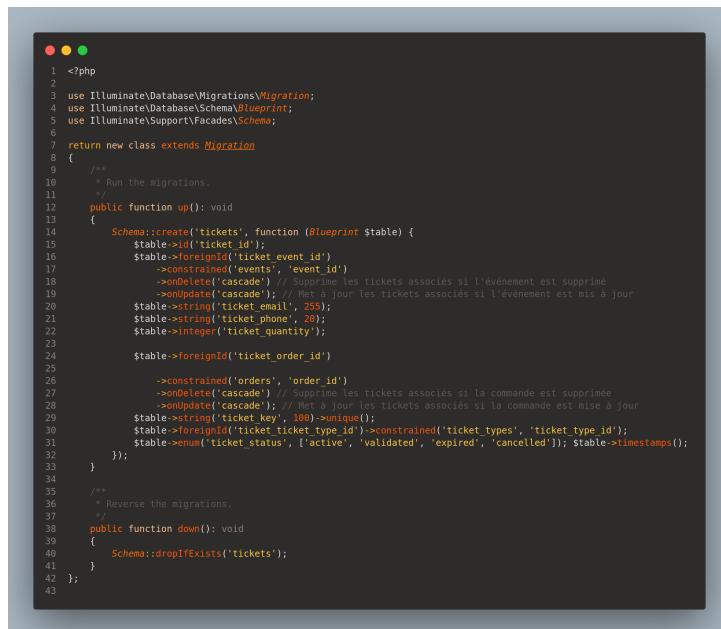


```
1 <?php
2
3 use Illuminate\Database\Migrations\Migration;
4 use Illuminate\Database\Schema\Blueprint;
5 use Illuminate\Support\Facades\Schema;
6
7 return new class extends Migration
8 {
9     /**
10      * Run the migrations.
11      */
12     public function up(): void
13     {
14         Schema::create('ticket_types', function (Blueprint $table) {
15             $table->id('ticket_type_id');
16             $table->foreignId('ticket_type_event_id')
17                 ->constrained('events', 'event_id')
18                 ->onDelete('cascade'); // Supprime les types de tickets associés si l'événement est supprimé
19                 ->onUpdate('cascade'); // Met à jour les types de tickets associés si l'événement est mis à jour
20             $table->string('ticket_type_name', 50);
21             $table->mediumInteger('ticket_type_price');
22             $table->integer('ticket_type_quantity');
23             $table->integer('ticket_type_real_quantity');
24             $table->integer('ticket_type_total_quantity');
25             $table->mediumText('ticket_type_description');
26             $table->timestamps();
27         });
28     }
29
30     /**
31      * Reverse the migrations.
32      */
33     public function down(): void
34     {
35         Schema::dropIfExists('ticket_types');
36     }
37 };
38
```

La table `ticket_types` contient les différents types de tickets pour chaque événement.
Voici les principales colonnes :

- `ticket_type_id` : Clé primaire.
- `ticket_type_event_id` : Référence vers l'événement associé.
- `ticket_type_name`, `ticket_type_price`, `ticket_type_quantity` : Nom, prix et quantité de tickets disponibles.

5.6 Table tickets



```
1 <?php
2
3 use Illuminate\Database\Migrations\Migration;
4 use Illuminate\Database\Schema\Blueprint;
5 use Illuminate\Support\Facades\Schema;
6
7 return new class extends Migration
8 {
9     /**
10      * Run the migrations.
11      */
12     public function up(): void
13     {
14         Schema::create('tickets', function (Blueprint $table) {
15             $table->id('ticket_id');
16             $table->foreignId('ticket_event_id')
17                 ->constrained('events', 'event_id')
18                 ->onDelete('cascade'); // Supprime les tickets associés si l'événement est supprimé
19                 ->onUpdate('cascade'); // Met à jour les tickets associés si l'événement est mis à jour
20             $table->string('ticket_email', 255);
21             $table->string('ticket_phone', 20);
22             $table->integer('ticket_quantity');
23
24             $table->foreignId('ticket_order_id')
25
26                 ->constrained('orders', 'order_id')
27                 ->onDelete('cascade'); // Supprime les tickets associés si la commande est supprimée
28                 ->onUpdate('cascade'); // Met à jour les Tickets associés si la commande est mise à jour
29             $table->string('ticket_key', 100)->unique();
30             $table->foreignId('ticket_ticket_type_id')->constrained('ticket_types', 'ticket_type_id');
31             $table->enum('ticket_status', ['active', 'validated', 'expired', 'cancelled']); $table->timestamps();
32         });
33     }
34
35     /**
36      * Reverse the migrations.
37      */
38     public function down(): void
39     {
40         Schema::dropIfExists('tickets');
41     }
42 };
43
```

La table `tickets` stocke les informations sur les tickets individuels. Voici les principales colonnes :

- `ticket_id` : Clé primaire.
- `ticket_event_id`, `ticket_order_id` : Références vers l'événement et la commande associés.
- `ticket_key` : Clé unique du ticket.
- `ticket_status` : Statut du ticket (`active`, `validated`, `expired`, `cancelled`).

5.7 Table `background_images`



```
1 <?php
2
3 use Illuminate\Database\Migrations\Migration;
4 use Illuminate\Database\Schema\Blueprint;
5 use Illuminate\Support\Facades\Schema;
6
7 return new class extends Migration
8 {
9     /**
10      * Run the migrations.
11      */
12     public function up(): void
13     {
14         Schema::create('background_images', function (Blueprint $table) {
15             $table->id();
16             $table->string('image_path_welcome')->nullable();
17             $table->string('image_path_login')->nullable();
18             $table->string('background_images_path_login')->nullable();
19
20             $table->timestamps();
21         });
22     }
23
24     /**
25      * Reverse the migrations.
26      */
27     public function down(): void
28     {
29         Schema::dropIfExists('background_images');
30     }
31 };
32 
```

La table `background_images` stocke les images d'arrière-plan utilisées dans l'application. Voici les principales colonnes :

- `id` : Clé primaire.
- `image_path_welcome`, `image_path_login` : Chemins des images pour la page d'accueil et la page de connexion.

List of relations			
Schema	Name	Type	Owner
public	background_images	table	isidoredev
public	events	table	isidoredev
public	failed_jobs	table	isidoredev
public	migrations	table	isidoredev
public	model_has_permissions	table	isidoredev
public	model_has_roles	table	isidoredev
public	order_intents	table	isidoredev
public	orders	table	isidoredev
public	password_reset_tokens	table	isidoredev
public	permissions	table	isidoredev
public	personal_access_tokens	table	isidoredev
public	role_has_permissions	table	isidoredev
public	roles	table	isidoredev
public	ticket_types	table	isidoredev
public	tickets	table	isidoredev
public	users	table	isidoredev

(16 rows)

FIGURE 1 – Liste des tables dans ma base après migration

6 Seeders

```

1 <?php
2
3 namespace Database\Seeders;
4
5 // use Illuminate\Database\Console\Seeds\WithoutModelEvents;
6 use Illuminate\Database\Seeder;
7
8 class DatabaseSeeder extends Seeder
9 {
10     /**
11      * Seed the application's database.
12      *
13      * @return void
14     */
15     public function run()
16     {
17         $this->call(RolesPermissionTableSeeder::class);
18         $this->call(UserSeeder::class);
19
20     }
21 }
22

```

```

1 <?php
2
3 namespace Database\Seeders;
4
5 use Illuminate\Database\Console\Seeds\WithoutModelEvents;
6 use Illuminate\Database\Seeder;
7 use App\Models\User;
8 use Spatie\Permission\Models\Role;
9 use Spatie\Permission\Models\Permission;
10 use HasApiTokens, HasFactory, Notifiable, HasRoles;
11 use App\Models\Vendeur;
12
13
14 class UserSeeder extends Seeder
15 {
16     /**
17      * Run the database seeds.
18     */
19
20
21     public function run(): void
22     {
23         // Create 1 admin
24         $user = User::create([
25             'firstname' => 'dev_isidore',
26             'lastname' => 'dev isidore',
27             'telephone' => '90101134',
28             'sexe' => 'M',
29             'datenaiss' => '1990-01-01',
30             'email' => 'dev@isidore.com',
31             'lieu_naissance' => 'Abidjan',
32             'username' => 'dev isidore',
33             'password' => bcrypt('_123456789'),
34         ]);
35         $role = Role::create(['name' => 'Developpeur']);
36
37         $permissions = Permission::pluck('id', 'id')->all();
38
39         $role->syncPermissions($permissions);
40
41         $user->assignRole([$role->id]);
42
43         $user_id=$user->id;
44
45
46
47
48
49
50
51     }
52
53
54 }
55

```

Les seeders permettent d'initialiser les données de base dans votre application, notamment les rôles, les permissions et les utilisateurs. Ces données sont essentielles pour le bon fonctionnement des fonctionnalités liées à la gestion des utilisateurs et des permissions.

6.1 Seeder Principal : DatabaseSeeder

Le **DatabaseSeeder** est le point d'entrée pour l'exécution des différents seeders dans votre application. Il appelle les seeders spécifiques suivants pour peupler la base de données :

- **RolesPermissionTableSeeder** : Crée les rôles et les permissions nécessaires.
- **UserSeeder** : Crée les utilisateurs initiaux et leur assigne des rôles.

6.2 Détails des Seeders

6.2.1 RolesPermissionTableSeeder

Ce seeder initialise les rôles et les permissions suivants :

- Permissions liées aux événements : `events-list`, `events-create`, `events-edit`, `events-delete`.

```

1  <?php
2
3  namespace Database\Seeders;
4
5  use Illuminate\Database\Console\Seeds\WithoutModelEvents;
6  use Illuminate\Database\Seeder;
7  use Spatie\Permission\Models\Role;
8  use Spatie\Permission\Models\Permission;
9
10 use HasApiTokens, HasFactory, Notifiable, HasRoles;
11
12 use App\Models\User;
13
14
15 class RolesPermissionTableSeeder extends Seeder
16 {
17     public function run(): void
18     {
19         $permissions = [
20             // events
21             'events-list',
22             'events-create',
23             'events-edit',
24             'events-delete',
25             // commandes
26             'commandes-list',
27             'commandes-create',
28             'commandes-edit',
29             'commandes-delete',
30         ];
31
32         // Create permissions
33         foreach ($permissions as $permission) {
34             Permission::create(['name' => $permission]);
35         }
36
37         // Create roles
38         $roleAdmin = Role::create(['name' => 'Admin']);
39
40         // Define permissions for each role
41         $permissionsAdmin = [
42             'events-list',
43             'events-create',
44             'events-edit',
45             'events-delete',
46             'commandes-list',
47             'commandes-create',
48             'commandes-edit',
49             'commandes-delete',
50         ];
51
52         // Assign permissions to roles
53         $roleAdmin->syncPermissions($permissionsAdmin);
54
55         $roleUserClient = Role::create(['name' => 'UserClient']);
56         // Define permissions for each role
57         $permissionsUserClient = [
58             'events-list',
59             'events-create',
60             'events-edit',
61             'events-delete',
62             'commandes-list',
63             'commandes-create',
64             'commandes-edit',
65             'commandes-delete',
66         ];
67
68         // Assign permissions to roles
69         $roleUserClient->syncPermissions($permissionsUserClient);
70
71         // Create admin user
72         $userAdmin = User::create([
73             'firstname' => 'admin',
74             'lastname' => 'admin',
75             'telephone' => '901407134',
76             'sexe' => 'M',
77             'datenaiss' => '1990-01-01',
78             'email' => 'admin@sirius.com',
79             'lieu_naissance' => 'Abidjan',
80             'username' => 'admin',
81             'password' => bcrypt('123456789'),
82         ]);
83
84         // Assign roles to users
85         $userAdmin->assignRole($roleAdmin);
86
87     }
88
89 }
90
91
92 }
93
94

```

- Permissions liées aux commandes : `commandes-list`, `commandes-create`, `commandes-edit`, `commandes-delete`.

Les rôles créés sont :

- `Admin` : A accès à toutes les permissions.
- `UserClient` : A accès aux permissions liées aux événements et commandes.

Un utilisateur administrateur est également créé et assigné au rôle `Admin`.

6.2.2 UserSeeder

Ce seeder crée un utilisateur développeur initial avec les informations suivantes :

- Prénom : `dev_isidore`

- Nom : `dev_isidore`
- Téléphone : 90101134
- Sexe : M
- Date de naissance : 1990-01-01
- Email : `dev@isidore.com`
- Lieu de naissance : Abidjan
- Nom d'utilisateur : `dev_isidore`

Cet utilisateur se voit attribuer le rôle de Développeur avec toutes les permissions.

7 Documentation des Routes : API et Web

7.1 Introduction

Dans mon application **Events**, que j'ai développée, les routes jouent un rôle crucial en définissant comment les requêtes HTTP sont dirigées vers les contrôleurs pour traiter les demandes des utilisateurs. Mon application utilise principalement deux types de routes : les routes **API** pour les communications via des services web et les routes **Web** pour la navigation utilisateur standard.

7.2 Routes API

Les routes API dans **Events** sont définies dans le fichier `routes/api.php`. Elles sont chargées dans le groupe de middleware `api`, ce qui leur confère un ensemble de fonctionnalités spécifiques aux services web, telles que la gestion des formats JSON et l'authentification via des tokens.

Exemple de Routes API

```
Route::middleware('auth:sanctum')->get('/user', function (Request $request) {
    return $request->user();
});

Route::get('/filter-events', [EventController::class, 'filter_evenements'])
->name('events.filter');
Route::get('/events/{eventId}/tickets', [EventController::class,
    'getTicketsByEvent'])->name('event.tickets');
```

7.3 Routes Web

Les routes Web dans mon application **Events** sont définies dans le fichier `routes/web.php`. Elles sont chargées dans le groupe de middleware `web`, qui inclut des fonctionnalités comme les sessions, la protection CSRF, et les cookies.

Exemple de Routes Web

```
Route::get('/', [FaonctionExterneController::class, 'welcome'])
->name('welcome');
Route::get('/listeevents', [FaonctionExterneController::class, 'listeEvents'])
->name('listeevents');

Route::middleware('auth')->group(function () {
    Route::get('/dashboard', [DashController::class, 'index'])
    ->name('dashboard');
    Route::get('/listeorders', [OrderController::class, 'index'])
    ->name('listeorders');
});
```

7.4 Utilisation des Middlewares

Les **middlewares** sont des filtres intermédiaires qui inspectent et modifient les requêtes HTTP avant qu'elles ne soient traitées par le contrôleur ou après qu'une réponse ait été envoyée au client. Dans **Events**, j'ai utilisé les middlewares pour diverses tâches telles que l'authentification, la gestion des sessions, et la protection CSRF.

7.4.1 Middleware auth

Le middleware **auth** est souvent utilisé pour protéger les routes qui nécessitent que l'utilisateur soit authentifié. Si une route est protégée par le middleware **auth**, l'utilisateur doit être connecté pour y accéder. Cela est crucial pour sécuriser les parties de l'application réservées aux utilisateurs enregistrés.

Exemple de Routes avec Middleware auth

```
Route::middleware('auth')->group(function () {
    Route::get('/profile', [ProfileController::class, 'edit'])
    ->name('profile.edit');

    Route::patch('/profile', [ProfileController::class, 'update'])
    ->name('profile.update');

    Route::delete('/profile', [ProfileController::class, 'destroy'])
    ->name('profile.destroy');
});
```

7.4.2 Routes sans Middleware

Certaines routes dans **Events** n'ont pas besoin de middleware, notamment celles qui sont accessibles à tous les utilisateurs, comme les pages publiques (page d'accueil, liste des événements, etc.). Ces routes ne nécessitent pas de protection spécifique et peuvent être accessibles sans authentification.

Exemple de Routes sans Middleware

```
Route::get('/', [FaonctionExterneController::class, 'welcome'])
->name('welcome');

Route::get('/listeevents', [FaonctionExterneController::class,
'listeEvents'])->name('listeevents');
```

7.5 Conclusion

Les routes, en combinaison avec les middlewares, permettent de contrôler l'accès aux différentes parties de mon application **Events**, garantissant ainsi la sécurité et la bonne gestion des ressources. Les routes API et Web sont organisées pour répondre aux besoins spécifiques de chaque type de client ou d'utilisateur, et les middlewares assurent que seules les personnes autorisées peuvent accéder à certaines fonctionnalités.

8 Travail Réalisé dans les Contrôleurs

Dans le cadre de la gestion des événements, plusieurs contrôleurs ont été développés pour répondre aux différentes fonctionnalités requises par le système. Ces contrôleurs sont répartis entre ceux se trouvant dans le dossier **API** et ceux en dehors, chacun ayant des responsabilités spécifiques.

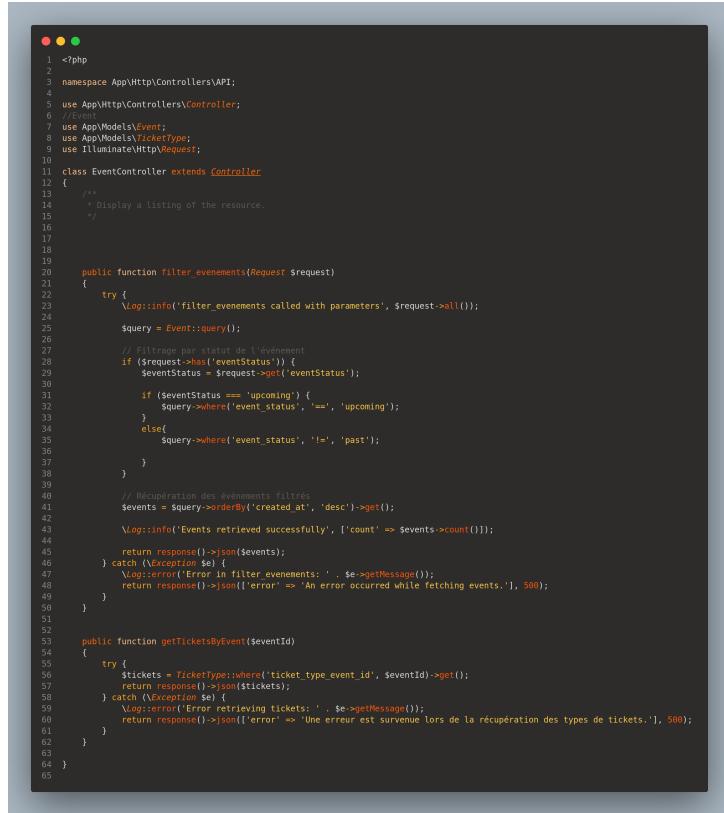
8.1 Contrôleur API : **EventController.php**

Le contrôleur **EventController.php** dans l'API gère principalement les fonctionnalités suivantes :

- **Filtrage des événements :**
 - La méthode `filter_evenements` permet de filtrer les événements en fonction de leur statut (`upcoming` ou `past`). Les événements sont récupérés et triés par date de création avant d'être renvoyés sous forme de réponse JSON.
 - Cette méthode intègre également des logs pour tracer les appels et les résultats obtenus, ce qui facilite le suivi et le débogage.
- **Récupération des types de tickets pour un événement :**
 - La méthode `getTicketsByEvent` récupère les différents types de tickets associés à un événement spécifique en utilisant son `eventId`. Les données sont également renvoyées sous forme de réponse JSON.
 - Des mécanismes de gestion des exceptions sont en place pour capturer et loguer toute erreur survenant lors de la récupération des types de tickets.

8.2 Explication du **EventController**

Ce contrôleur, nommé **EventController**, gère les opérations liées aux événements et aux types de tickets. Il se trouve dans le namespace **API**, ce qui indique qu'il est conçu pour une API.



```

1  <?php
2
3  namespace App\Http\Controllers\API;
4
5  use App\Http\Controllers\Controller;
6  //Event
7  use App\Models\Event;
8  use App\Models\TicketType;
9  use Illuminate\Http\Request;
10
11 class EventController extends Controller
12 {
13     /**
14      * Display a listing of the resource.
15     */
16
17
18
19
20    public function filter_evenements(Request $request)
21    {
22        try {
23            \Log::info('filter_evenements called with parameters', $request->all());
24
25            $query = Event::query();
26
27            // Filtrage par statut de l'événement
28            if ($request->has('eventstatus')) {
29                $eventstatus = $request->get('eventstatus');
30
31                if ($eventstatus === 'upcoming') {
32                    $query->where('event_status', '=', 'upcoming');
33                } else{
34                    $query->where('event_status', '!=', 'past');
35                }
36            }
37
38        }
39
40        // Récupération des événements filtrés
41        $events = $query->orderBy('created_at', 'desc')->get();
42
43        \Log::info('Events retrieved successfully', ['count' => $events->count()]);
44
45        return response()->json($events);
46    } catch (\Exception $e) {
47        \Log::error("Error in filter evenements: " . $e->getMessage());
48        return response()->json(['error' => 'An error occurred while fetching events.'], 500);
49    }
50
51
52    public function getTicketsByEvent($eventId)
53    {
54        try {
55            $tickets = TicketType::where('ticket_type_event_id', $eventId)->get();
56            return response()->json($tickets);
57        } catch (\Exception $e) {
58            \Log::error("Error in retrieving tickets: " . $e->getMessage());
59            return response()->json(['error' => 'Une erreur est survenue lors de la récupération des types de tickets.'], 500);
60        }
61    }
62
63
64    }
65
66}

```

`namespace App\Http\Controllers\API;`

Cette ligne définit le namespace du contrôleur, indiquant qu'il fait partie des contrôleurs spécifiques à l'API dans l'application Laravel.

```

use App\Http\Controllers\Controller;
use App\Models\Event;
use App\Models\TicketType;
use Illuminate\Http\Request;

```

Les déclarations `use` permettent d'inclure des classes spécifiques qui seront utilisées dans ce contrôleur :

```

use App\Http\Controllers\Controller;
use App\Models\Event;
use App\Models\TicketType;
use Illuminate\Http\Request;

```

- : Importation de la classe `Controller`, qui est la classe de base pour tous les contrôleurs dans Laravel.

- Importation du modèle `Event`, qui représente la table des événements dans la base de données.

- Importation du modèle `TicketType`, qui représente la table des types de tickets.

- Importation de la classe `Request`, qui est utilisée pour gérer les requêtes HTTP entrantes.

```
/*
 * Display a listing of the resource.
 */
```

Cette partie est un commentaire qui peut décrire la fonction qui suit, mais dans ce cas, elle est laissée vide.

```
public function filter_evenements(Request $request)
```

Cette fonction, `filter_evenements`, est utilisée pour filtrer les événements en fonction de certains paramètres fournis dans la requête.

```
try {
    \Log::info('filter_evenements called with parameters', $request->all());
```

Le bloc `try` commence ici pour capturer toute exception qui pourrait se produire lors de l'exécution de la fonction. La méthode `\Log::info` enregistre dans les logs que la fonction a été appelée, ainsi que les paramètres reçus dans la requête.

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

Cette ligne initialise une nouvelle requête sur le modèle `Event`.

```
// Filtrage par statut de l'événement
if ($request->has('eventStatus')) {
    $eventStatus = $request->get('eventStatus');
```

Cette section vérifie si la requête contient un paramètre `eventStatus`. Si c'est le cas, la valeur de ce paramètre est récupérée.

```
if ($eventStatus === 'upcoming') {
    $query->where('event_status', '==', 'upcoming');
}
else{
    $query->where('event_status', '!=', 'past');
}
```

Ici, un filtrage est appliqué à la requête en fonction de la valeur de `eventStatus`. Si la valeur est `upcoming`, seuls les événements à venir sont récupérés. Sinon, les événements passés sont exclus.

```
// Récupération des événements filtrés
$events = $query->orderBy('created_at', 'desc')->get();
```

Les événements filtrés sont ensuite récupérés de la base de données, triés par date de création en ordre décroissant.

```
\Log::info('Events retrieved successfully', ['count' => $events->count()]);
```

Un log est généré pour indiquer que les événements ont été récupérés avec succès, avec le nombre d'événements récupérés.

```
return response()->json($events);
```

La réponse est renvoyée sous forme de JSON avec la liste des événements filtrés.

```
} catch (\Exception $e) {
    \Log::error('Error in filter_evenements: ' . $e->getMessage());
    return response()->json(['error' => 'An error occurred while fetching events.']),
}
```

En cas d'erreur, celle-ci est enregistrée dans les logs, et une réponse JSON avec un message d'erreur et un code HTTP 500 est renvoyée.

```
public function getTicketsByEvent($eventId)
```

Cette fonction, `getTicketsByEvent`, est utilisée pour récupérer les types de tickets associés à un événement spécifique.

```
try {
    $tickets = TicketType::where('ticket_type_event_id', $eventId)->get();
    return response()->json($tickets);
} catch (\Exception $e) {
    \Log::error('Error retrieving tickets: ' . $e->getMessage());
    return response()->json(['error' => 'Une erreur est survenue lors de la récupération des billets.']),
}
```

Comme pour la fonction précédente, elle est entourée d'un bloc `try-catch`. La requête récupère tous les types de tickets associés à l'événement `eventId`, et retourne la liste en JSON. En cas d'erreur, un message d'erreur est logué, et une réponse JSON avec un message d'erreur est renvoyée.

8.3 Contrôleur Général : EventController.php

Le contrôleur général `EventController.php` est en charge de la gestion complète des événements via une interface utilisateur :

- **Affichage des événements :**

- La méthode `index` affiche une liste paginée des événements disponibles, avec la possibilité de filtrer les événements par date via un intervalle de dates.
- Un tableau de données (DataTables) est utilisé pour rendre la présentation des données plus interactive.

- **Création et mise à jour des événements :**

- Les méthodes `create` et `store` permettent de créer un nouvel événement, incluant la gestion des fichiers pour les images et la création de types de tickets associés à l'événement.
- La méthode `edit` récupère les détails d'un événement existant pour les afficher dans un formulaire d'édition, tandis que `update` permet de modifier et de mettre à jour ces informations.
- Les statuts des événements sont automatiquement mis à jour en fonction de la date de l'événement (par exemple, un événement passé est marqué comme `completed`).

- **Suppression des événements :**

- La méthode `destroy` supprime un événement de la base de données et redirige l'utilisateur vers la liste des événements avec un message de succès.

8.4 Explication du EventController

```

1 <?php
2 namespace App\Http\Controllers;
3 use App\Models\Event;
4 use Carbon\Carbon;
5 use Illuminate\Http\Request;
6 use Yajra\Datatables\Facades\Datatables;
7 class EventController extends Controller
8 {
9     public function __construct()
10    {
11        $this->middleware('permission:events-list|events-create|events-edit|events-delete', ['only' => ['index','show']]);
12        $this->middleware('permission:events-create', ['only' => ['create','store']]);
13        $this->middleware('permission:events-edit', ['only' => ['edit','update']]);
14        $this->middleware('permission:events-delete', ['only' => ['destroy']]);
15    }
16    public function index(Request $request)
17    {
18        if ($request->ajax())
19        {
20            $data = Datatables::of($data)
21                ->addColumn('action', function ($row) {
22                    $editUrl = route('events.billets.edit', $row->event_id);
23                    $showUrl = route('events.billets.show', $row->event_id);
24                    $deleteUrl = route('events.billets.destroy', $row->event_id);
25
26                    return '';
27                })
28                ->addIndexColumn()
29                ->rawColumns(['action'])
30                ->make(true);
31        }
32        return view('events.index');
33    }
34    public function create()
35    {
36        return view('events.create');
37    }
38    public function store(Request $request)
39    {
40        $validatedData = $request->validate([
41            'event_category' => 'required|string',
42            'event_start_date' => 'required|date|before:' . now(),
43            'event_end_date' => 'nullable|date|after:' . now(),
44            'event_image' => 'required|image|max:2048',
45            'event_address' => 'required|string|max:100',
46            'event_address2' => 'nullable|string|max:200',
47            'ticket_type_name' => 'required|array',
48            'ticket_type_price' => 'required|array',
49            'ticket_type_description' => 'nullable|array',
50            'ticket_type_quantity' => 'required|array',
51            'ticket_type_real_quantity' => 'nullable|array',
52            'ticket_type_description.' => 'nullable|string',
53            'ticket_type_price.' => 'nullable|float'
54        ]);
55
56        // Handle file upload
57        if ($request->hasFile('event_image')) {
58            $imagePath = $request->file('event_image')->store('event_images', 'public');
59            $validatedData['event_image'] = $imagePath;
60        }
61
62        //event_date
63        // Calcul de la date d'événement à partir de la date de début et de la durée
64        $event_date = Carbon::createFromFormat('Y-m-d\TH:i', $validatedData['event_date']);
65        $event->status = $event_date->isPast() ? 'completed' : 'upcoming';
66        $validatedData['event_status'] = $event->status;
67
68        // Create event
69        $event = Event::create($validatedData);
70
71        // calcul de la somme des deux quantités ticket_type_real_quantity + ticket_type_real_quantity
72        // Calcul de la quantité totale de tickets
73        // Total quantity = validatedData[ticket_type_real_quantity];
74
75        foreach ($validatedData['ticket_type_name'] as $index => $name) {
76            $event->ticketTypes()->create([
77                'ticket_type_name' => $name,
78                'ticket_type_price' => $validatedData['ticket_type_price'][$index],
79                'ticket_type_quantity' => $validatedData['ticket_type_quantity'][$index],
80                'ticket_type_description' => $validatedData['ticket_type_description'][$index] ?? '',
81                'ticket_type_real_quantity' => $validatedData['ticket_type_real_quantity'][$index],
82                'ticket_type_total_quantity' => $validatedData['ticket_type_quantity'][$index], // Calcul correct de la quantité totale
83            ]);
84        }
85
86        return redirect()->route('events.billets.index')->with('success', 'Événement créé avec succès');
87    }
88
89    public function edit($id)
90    {
91        $event = Event::findOrFail($id);
92        return view('events.edit', compact('event'));
93    }
94    public function update(Request $request, $id)
95    {
96        $validatedData = $request->validate([
97            'event_category' => 'required|string',
98            'event_start_date' => 'required|date|before:' . now(),
99            'event_end_date' => 'nullable|date|after:' . now(),
100            'event_image' => 'nullable|image|max:2048',
101            'event_address' => 'required|string|max:100',
102            'event_address2' => 'nullable|string|max:200',
103            'ticket_type_name' => 'required|array',
104            'ticket_type_price' => 'required|array',
105            'ticket_type_description' => 'nullable|array',
106            'ticket_type_quantity' => 'required|array',
107            'ticket_type_description.' => 'nullable|string',
108            'ticket_type_price.' => 'nullable|float',
109            'ticket_type_real_quantity.' => 'required|integer',
110        ]);
111
112        // Update event status based on date
113        $event_date = Carbon::createFromFormat('Y-m-d\TH:i', $validatedData['event_date']);
114        $event->status = $event_date->isPast() ? 'completed' : 'upcoming';
115        $validatedData['event_status'] = $event->status;
116
117        // Update event
118        $event = Event::findOrFail($id);
119
120        // ticketTypes()>delete(); // Delete old ticket types
121        foreach ($validatedData['ticket_type_name'] as $index => $name) {
122            $event->ticketTypes()->where('ticket_type_name', $name)
123                ->delete();
124            $ticket_type_price => $validatedData['ticket_type_price'][$index];
125            $ticket_type_quantity => $validatedData['ticket_type_quantity'][$index];
126            $ticket_type_description => $validatedData['ticket_type_description'][$index] ?? '';
127            $ticket_type_real_quantity => $validatedData['ticket_type_real_quantity'][$index];
128        }
129
130        return redirect()->route('events.billets.index')->with('success', 'Événement mis à jour avec succès');
131    }
132
133    public function destroy($id)
134    {
135        $event = Event::findOrFail($id);
136        $event->delete();
137
138        return redirect()->route('events.billets.index')->with('success', 'Événement supprimé avec succès');
139    }
140}

```

Le contrôleur EventController est responsable de la gestion des opérations relatives aux événements dans l'application. Ce contrôleur utilise plusieurs fonctionnalités et ser-

vices offerts par Laravel, comme les middlewares, les validations de formulaire, et les opérations CRUD sur les modèles Event.

8.4.1 Namespace et Importations

```
namespace App\Http\Controllers;

use App\Models\Event;
use Carbon\Carbon;
use Illuminate\Http\Request;
use Yajra\DataTables\Facades\DataTables;
```

Le contrôleur est défini dans le namespace. Il importe plusieurs classes et fonctions nécessaires, notamment :

- **Event** : Le modèle représentant les événements dans la base de données.
- **Carbon** : Une bibliothèque pour manipuler les dates et les heures.
- **Request** : Pour manipuler les requêtes HTTP entrantes.
- **DataTables** : Pour faciliter la gestion des tableaux de données dans les vues.

8.4.2 Middlewares

```
public function __construct()
{
    $this->middleware('permission:events-list|events-create|events-edit|events-delete');
    $this->middleware('permission:events-create', ['only' => ['create', 'store']]);
    $this->middleware('permission:events-edit', ['only' => ['edit', 'update']]);
    $this->middleware('permission:events-delete', ['only' => ['destroy']]);
}
```

Le constructeur du contrôleur applique des middlewares pour restreindre l'accès aux différentes actions en fonction des permissions de l'utilisateur. Par exemple, seules les personnes ayant la permission `events-create` peuvent accéder aux méthodes `create` et `store`.

8.4.3 Méthode `index()`

```
public function index(Request $request)
{
    if ($request->ajax()) {
        $data = Event::query();

        if ($request->filled('from_date') && $request->filled('to_date')) {
            $fromDate = Carbon::createFromFormat('Y-m-d', $request
                ->from_date)->startOfDay();
            $toDate = Carbon::createFromFormat('Y-m-d', $request->to_date)
                ->endOfDay();

            $data->whereBetween('created_at', [$fromDate, $toDate]);
        }
    }
}
```

```

    }

    return DataTables::of($data)
        ->addColumn('action', function ($row) {
            // code pour les boutons d'action (édition, affichage,
            suppression)
        })
        ->addIndexColumn()
        ->rawColumns(['action'])
        ->make(true);
}

return view('events.index');
}

```

La méthode `index` gère la liste des événements. Si la requête est de type AJAX, elle applique des filtres (par date) et retourne les résultats au format JSON pour être utilisés par DataTables. Sinon, elle retourne la vue `events.index`.

8.4.4 Méthode `create()`

```

public function create()
{
    return view('events.create');
}

```

La méthode `create` retourne la vue qui permet de créer un nouvel événement.

8.4.5 Méthode `store()`

```

public function store(Request $request)
{
    $validatedData = $request->validate([
        'event_category' => 'required|string',
        'event_title' => 'required|string|max:30',
        'event_description' => 'nullable|string',
        'event_date' => 'required|date',
        'event_image' => 'required|image|max:2048',
        'event_city' => 'required|string|max:100',
        'event_address' => 'required|string|max:200',
        'ticket_type_name' => 'required|array',
        'ticket_type_name.*' => 'required|string|max:50',
        'ticket_type_price' => 'required|array',
        'ticket_type_price.*' => 'required|integer',
        'ticket_type_quantity' => 'required|array',
        'ticket_type_quantity.*' => 'required|integer',
        'ticket_type_description' => 'nullable|array',
        'ticket_type_description.*' => 'nullable|string',
    ]);
}

```

```

if ($request->hasFile('event_image')) {
    $imagePath = $request->file('event_image')->store('event_images', 'public');
    $validatedData['event_image'] = $imagePath;
}

$event_date = Carbon::createFromFormat('Y-m-d\TH:i',
    $validatedData['event_date']);
$event_status = $event_date->isPast() ? 'completed' : 'upcoming';

$validatedData['event_status'] = $event_status;

$event = Event::create($validatedData);

foreach ($validatedData['ticket_type_name'] as $index => $name) {
    $event->ticketTypes()->create([
        'ticket_type_name' => $name,
        'ticket_type_price' => $validatedData['ticket_type_price'][$index],
        'ticket_type_quantity' => $validatedData['ticket_type_quantity'][$index],
        'ticket_type_description' => $validatedData['ticket_type_description'][$index],
        'ticket_type_real_quantity' => $validatedData['ticket_type_quantity'][$index],
        'ticket_type_total_quantity' => $validatedData['ticket_type_quantity'][$index]
    ]);
}

return redirect()->route('events_billets.index')->with('success', 'Événement créé');
}

```

La méthode `store` valide les données de la requête, gère le téléchargement de l'image de l'événement, détermine le statut de l'événement (en fonction de la date) et crée un nouvel événement avec les types de tickets associés.

8.4.6 Méthode `edit()`

```

public function edit($id)
{
    $event = Event::findOrFail($id);
    return view('events.edit', compact('event'));
}

```

La méthode `edit` récupère un événement spécifique par son `id` et retourne la vue pour l'éditer.

8.4.7 Méthode `update()`

```

public function update(Request $request, $id)
{

```

```

$validatedData = $request->validate([
    'event_category' => 'required|string',
    'event_title' => 'required|string|max:30',
    'event_description' => 'nullable|string',
    'event_date' => 'required|date',
    'event_image' => 'nullable|image|max:2048',
    'event_city' => 'required|string|max:100',
    'event_address' => 'required|string|max:200',
    'ticket_type_name' => 'required|array',
    'ticket_type_name.*' => 'required|string|max:50',
    'ticket_type_price' => 'required|array',
    'ticket_type_price.*' => 'required|integer',
    'ticket_type_quantity' => 'required|array',
    'ticket_type_quantity.*' => 'required|integer',
    'ticket_type_description' => 'nullable|array',
    'ticket_type_description.*' => 'nullable|string',
    'ticket_type_real_quantity' => 'required|array',
    'ticket_type_real_quantity.*' => 'required|integer',
]);
];

$event = Event::findOrFail($id);

if ($request->hasFile('event_image')) {
    $imagePath = $request->file('event_image')->store('event_images', 'public');
    $validatedData['event_image'] = $imagePath;
}

$event_date = Carbon::createFromFormat('Y-m-d\TH:i', $validatedData['event_date'])
$event_status = $event_date->isPast() ? 'completed' : 'upcoming';
$validatedData['event_status'] = $event_status;

$event->update($validatedData);

$event->ticketTypes()->delete();
foreach ($validatedData['ticket_type_name'] as $index => $name) {
    $event->ticketTypes()->create([
        'ticket_type_name' => $name,
        'ticket_type_price' => $validatedData['ticket_type_price'][$index],
        'ticket_type_quantity' => $validatedData['ticket_type_quantity'][$index],
        'ticket_type_description' =>
            $validatedData['ticket_type_description'][$index] ?? '',
        'ticket_type_real_quantity' =>
            $validatedData['ticket_type_real_quantity'][$index],
        'ticket_type_total_quantity' =>
            $validatedData['ticket_type_real_quantity'][$index],
    ]);
}

```

```

    }

    return redirect()->route('events_billets.index')->with('success',
        'Événement mis à jour avec succès');
}

```

La méthode `update` permet de modifier un événement existant. Elle suit une logique similaire à celle de la méthode `store`, avec en plus la gestion de la suppression des anciens types de tickets avant de les recréer.

8.4.8 Méthode `destroy()`

```

public function destroy($id)
{
    $event = Event::findOrFail($id);
    $event->delete();

    return redirect()->route('events_billets.index')->with('success',
        'Événement supprimé avec succès');
}

```

La méthode `destroy` supprime un événement spécifique de la base de données.

8.4.9 Gestion des Tickets Associés

Les méthodes `store()` et `update()` gèrent non seulement les événements, mais aussi les types de tickets associés. Chaque événement peut avoir plusieurs types de tickets, chacun avec son propre nom, prix, quantité, et description. Ces informations sont gérées par des relations dans le modèle `Event`.

En résumé, le contrôleur `EventController` joue un rôle clé dans la gestion des événements au sein de l'application, en s'assurant que toutes les opérations sont effectuées de manière sécurisée et efficace grâce à l'utilisation des middlewares, des validations, et des services offerts par Laravel.

8.5 Contrôleur `FaonctionExterneController.php`

Ce contrôleur gère des fonctionnalités plus spécifiques liées à la gestion des tickets et des paiements :

- **Affichage et sélection des événements :**
 - Les méthodes `welcome` et `listeEvents` affichent la liste des événements disponibles sur la page d'accueil.
- **Processus de paiement :**
 - La méthode `showPaymentPage` présente la page de sélection de tickets pour un événement spécifique, tandis que `processPaymentInfo` traite les informations de paiement en générant une intention de commande avec une expiration automatique de 48 heures.
 - La méthode `showPaymentConfirmPage` affiche la page de confirmation du paiement, où le montant total et les détails de la commande sont récapitulés avant validation.

— **Confirmation et traitement du paiement :**

- La méthode `processPaymentConfirm` enregistre définitivement la commande en générant un numéro unique pour chaque commande, puis crée le ticket associé à cette commande.
- Enfin, la méthode `downloadTicket` permet de télécharger le ticket généré sous forme de fichier PDF.

8.6 Contrôleur `OrderController.php`

Le `OrderController` est responsable de la gestion des commandes dans l'application. Il permet notamment d'afficher, filtrer et supprimer les commandes. Voici les principales fonctionnalités développées dans ce contrôleur :

— **Affichage et filtrage des commandes :**

- La méthode `index` permet d'afficher une liste de commandes. Si la requête est effectuée en AJAX, elle récupère les commandes avec leurs événements associés en utilisant la relation `event`.
- Un filtrage par date de création des commandes est possible en fournissant les dates `from_date` et `to_date`. Ces dates sont converties en objets Carbon pour permettre un filtrage précis entre le début de la première date et la fin de la seconde.
- Les données sont ensuite formatées et présentées sous forme de tableau interactif (DataTables), avec une colonne supplémentaire pour les actions, notamment la suppression de la commande.

— **Suppression des commandes :**

- La méthode `destroy` permet de supprimer une commande spécifique en utilisant son identifiant. Après la suppression, l'utilisateur est redirigé vers la liste des commandes avec un message de confirmation de la suppression.

8.7 Contrôleur `ProfileController.php`

Le `ProfileController` gère les opérations liées à la visualisation, la mise à jour et la suppression des profils utilisateurs. Voici les principales méthodes de ce contrôleur :

— **Affichage du formulaire de profil :**

- La méthode `edit` affiche le formulaire d'édition du profil utilisateur. Elle récupère les informations de l'utilisateur connecté via la requête et les passe à la vue `profile.edit`.

— **Mise à jour des informations du profil :**

- La méthode `update` permet de mettre à jour les informations du profil de l'utilisateur, incluant la gestion de l'avatar et la validation du mot de passe.
- Si l'utilisateur télécharge une nouvelle image de profil (`profile_picture`), l'ancienne est supprimée avant de stocker la nouvelle.
- Le numéro de téléphone est également normalisé pour supprimer les espaces et le convertir en entier.
- Si l'utilisateur change son adresse email, le champ `email_verified_at` est réinitialisé à `null`.

- Si un nouveau mot de passe est fourni, il est validé et mis à jour après vérification du mot de passe actuel et confirmation du nouveau mot de passe.
- **Suppression du compte utilisateur :**
 - La méthode `destroy` supprime le compte utilisateur après vérification du mot de passe. L'utilisateur est ensuite déconnecté, et sa session est invalidée avant d'être redirigé vers la page d'accueil.

8.8 RegisteredUserController

8.8.1 Description

Le `RegisteredUserController` est un contrôleur qui gère la création de nouveaux utilisateurs. Il est principalement utilisé pour afficher le formulaire d'inscription, traiter les données envoyées par ce formulaire, créer un nouvel utilisateur, lui assigner un rôle, et envoyer un e-mail de confirmation.

8.8.2 Méthodes

8.8.3 `create`

Description

La méthode `create` est responsable de l'affichage du formulaire d'inscription.

Retourne

`Illuminate-View-View` - La vue `auth.register` contenant le formulaire d'inscription.

8.8.4 `store`

Description

La méthode `store` traite la requête d'inscription. Elle valide les données saisies, crée un compte utilisateur, génère un mot de passe, envoie un e-mail de confirmation, et redirige l'utilisateur vers la page de connexion.

Paramètres

`Request $request` - La requête HTTP contenant les données du formulaire d'inscription.

Validation

Les règles de validation sont appliquées sur les champs suivants :

- `firstname` : requis, chaîne, max 255 caractères.
- `lastname` : requis, chaîne, max 255 caractères.
- `ville` : requis, chaîne, max 255 caractères.
- `adresse` : requis, chaîne, max 255 caractères.
- `entreprise` : requis, chaîne, max 255 caractères.
- `email` : requis si présent, chaîne, e-mail valide, unique dans la table `users`.

8.8.5 Génération du Nom d'utilisateur et Mot de Passe

Génération

Un nom d'utilisateur est généré en utilisant les deux premières lettres du prénom et du nom de famille de l'utilisateur, combinées à 4 caractères aléatoires générés par `Str::random(4)`. Le mot de passe est généré de manière similaire.

Variables générées

- `$username` : Nom d'utilisateur généré.
- `$password` : Mot de passe généré.

8.8.6 Envoi de l'E-mail de Confirmation

Envoi d'e-mail

Un e-mail est envoyé à l'utilisateur contenant son nom d'utilisateur et son mot de passe. Le contenu de l'e-mail est généré à partir de la vue `emails.accuser_ouverture`.

Bloc try-catch Le bloc `try-catch` est utilisé pour capturer et gérer les erreurs potentielles lors de l'envoi de l'e-mail :

Gestion des erreurs lors de l'envoi d'e-mail

- `Swift_TransportException` : Capturée en cas de problème de transport d'e-mail, comme un problème de configuration du serveur de messagerie ou du domaine. Si cette exception est levée, l'utilisateur est redirigé en arrière avec un message d'erreur spécifique : `Nous ne parvenons pas à envoyer le message. Il y a peut-être un problème avec votre domaine ou serveur de messagerie.`
- `Exception` : Toutes les autres exceptions générales sont capturées ici. En cas d'erreur inattendue, un message d'erreur générique est renvoyé à l'utilisateur : `Une erreur inconnue s'est produite lors de l'envoi du code. Veuillez réessayer plus tard.`

8.8.7 Création de l'utilisateur dans la base de données

Base de Données

Les informations de l'utilisateur sont enregistrées dans la table `users` avec les champs suivants :

- `firstname` : Prénom de l'utilisateur.
- `lastname` : Nom de famille de l'utilisateur.
- `username` : Nom d'utilisateur généré.
- `indicatiftel` : Indicatif téléphonique du pays.
- `telephone` : Numéro de téléphone.
- `email` : Adresse e-mail.
- `password` : Mot de passe hashé.

Assignation de Rôle

Le rôle `UserClient` est assigné à l'utilisateur nouvellement créé en utilisant la bibliothèque `spatie/laravel-permission`.

8.8.8 Retour et Redirection

Retourne

`Illuminate\Http\RedirectResponse` - Redirige l'utilisateur vers la page de connexion avec un message de succès : `Votre compte a été créé avec succès. Veuillez vous connecter.`

8.8.9 Gestion des Erreurs

La gestion des erreurs est une partie cruciale de la méthode `store`. Les erreurs sont capturées à plusieurs niveaux pour assurer que l'utilisateur reçoive un retour approprié en cas de problème, qu'il s'agisse d'un problème de validation, d'un problème d'envoi d'e-mail, ou d'un autre type d'erreur.

Exemple d'Erreur de Validation

L'adresse e-mail est déjà utilisée par un autre compte.

Exemple d'Erreur d'Envoi d'E-mail

Nous ne parvenons pas à envoyer le message. Il y a peut-être un problème avec votre domaine ou serveur de messagerie.

Exemple d'Erreur Générale

Une erreur inconnue s'est produite lors de l'envoi du code. Veuillez réessayer plus tard.

8.9 AuthenticatedSessionController

8.9.1 Description

Le `AuthenticatedSessionController` est un contrôleur qui gère l'authentification des utilisateurs. Il est responsable de l'affichage de la vue de connexion, de la gestion des demandes d'authentification, et de la destruction de la session utilisateur lors de la déconnexion.

8.9.2 Méthodes

8.9.3 create

Description

La méthode `create` est responsable de l'affichage de la vue de connexion.

Retourne

`Illuminate\View\View` - La vue `auth.login` contenant le formulaire de connexion.

8.9.4 store

Description

La méthode `store` traite les demandes d'authentification entrantes. Elle authentifie l'utilisateur, régénère la session pour éviter les attaques de fixation de session, et redirige l'utilisateur vers la page d'accueil avec son nom d'utilisateur.

Paramètres

`LoginRequest $request` - La requête HTTP contenant les données de connexion de l'utilisateur.

8.9.5 Recherche de l'utilisateur

Recherche dans la base de données

La méthode utilise le nom d'utilisateur fourni pour rechercher l'utilisateur correspondant dans la table `users`. Si un utilisateur est trouvé, les informations sont disponibles pour une utilisation ultérieure.

Validation de l'authentification

Après la recherche de l'utilisateur, la méthode `authenticate` du `LoginRequest` est appelée pour authentifier les informations de connexion fournies par l'utilisateur.

8.9.6 Régénération de la Session

Sécurité

Une fois l'authentification réussie, la session de l'utilisateur est régénérée pour protéger contre les attaques de fixation de session.

8.9.7 Redirection

Redirection

La méthode redirige ensuite l'utilisateur vers la page d'accueil définie dans `RouteServiceProvider::HOME` et passe le nom d'utilisateur à la vue.

8.9.8 destroy

Description

La méthode `destroy` est responsable de la déconnexion de l'utilisateur. Elle déconnecte l'utilisateur, invalide la session actuelle, régénère le jeton CSRF et redirige l'utilisateur vers la page d'accueil.

Paramètres

`Request $request` - La requête HTTP associée à la session utilisateur.

Étapes de la Déconnexion

- **Déconnexion** : La méthode appelle `Auth::guard('web')->logout()` pour déconnecter l'utilisateur.
- **Invalidation de la session** : La session actuelle est invalidée pour s'assurer que l'utilisateur ne peut plus l'utiliser.
- **Régénération du jeton CSRF** : Un nouveau jeton CSRF est généré pour protéger contre les attaques CSRF.
- **Redirection** : L'utilisateur est redirigé vers la page d'accueil.

Retourne

`Illuminate\Http\RedirectResponse` - Redirige l'utilisateur vers la page d'accueil après la déconnexion.

8.9.9 Gestion des Erreurs

Gestion des erreurs lors de l'authentification

La méthode `store` gère implicitement les erreurs de connexion à travers la méthode `authenticate()` du `LoginRequest`. Si les informations d'authentification sont incorrectes, une exception est levée et un message d'erreur est renvoyé à l'utilisateur.

9 Structure des Ressources de l'Application

Dans cette section, nous allons explorer la structure des fichiers au sein du répertoire `resources/views` de votre application Laravel. Ce répertoire contient les vues (templates Blade) utilisées par l'application pour rendre les interfaces utilisateur.

9.1 Répertoire `views`

Le répertoire `views` contient l'ensemble des fichiers Blade qui composent les différentes vues de l'application. Chaque sous-répertoire organise les vues en fonction de leur domaine d'application spécifique.

9.1.1 Répertoire `emails`

Le répertoire `emails` contient des vues dédiées à l'envoi d'e-mails. Un exemple est le fichier `accuser_ouverture.blade.php`, qui pourrait être utilisé pour envoyer une notification de confirmation d'ouverture.

9.1.2 Répertoire `events`

Le répertoire `events` regroupe les vues qui gèrent les événements. Il contient trois fichiers principaux :

- `create.blade.php` : Vue pour la création d'un événement.
- `edit.blade.php` : Vue pour l'édition d'un événement existant.
- `index.blade.php` : Vue qui liste tous les événements.

9.1.3 Répertoire `layouts`

Le répertoire `layouts` contient les modèles de mise en page globale de l'application. Ces modèles sont utilisés pour définir la structure générale des pages, incluant des sections telles que le header, le footer, et la barre latérale.

- `app.blade.php` : Le modèle principal de l'application, qui inclut la structure de base (header, footer, etc.).
- `auth.blade.php` : Modèle spécifique pour les pages d'authentification.
- `welcome.blade.php` : Vue pour la page d'accueil ou la page de bienvenue.

Sous-répertoire `partials` : Ce sous-répertoire organise les fragments de vue réutilisables, tels que les composants d'interface utilisateur communs :

- `admin_dash.blade.php` : Partie du tableau de bord admin.
- `left_sidebar.blade.php` : Barre latérale gauche utilisée dans plusieurs vues.
- `principal_header.blade.php` : En-tête principal de l'application.

9.1.4 Répertoire `orders`

Le répertoire `orders` gère les vues liées aux commandes. Le fichier `index.blade.php` présente probablement la liste des commandes.

9.1.5 Autres Répertoires

- **parametrage** : Probablement utilisé pour des vues de configuration ou de paramètres.
- **roles** : Pour la gestion des rôles utilisateurs.
- **tickets** : Contient des vues liées à la gestion des tickets, comme `pdf.blade.php` qui génère probablement des tickets en format PDF.
- **users** : Gestion des utilisateurs.
- **vendor** : Peut contenir des vues personnalisées fournies par des packages tiers.

9.2 Extending `app.layout`

Les fichiers Blade au sein de ces répertoires étendent souvent le modèle de mise en page défini dans `app.blade.php` pour assurer une cohérence dans l'interface utilisateur. Par exemple, chaque vue inclut généralement des sections pour le contenu (`@yield('content')`) et les scripts (`@yield('scripts')`), qui sont remplies dans les fichiers spécifiques comme `create.blade.php` ou `index.blade.php`.

En résumé, cette structure permet de maintenir une organisation claire et modulaire des vues, facilitant ainsi la maintenance et l'évolution de l'interface utilisateur de l'application.

9.3 Explication du Fichier `index.blade.php`

Ce fichier `index.blade.php` est utilisé pour afficher une liste d'événements dans une application Laravel. Il comprend des sections qui permettent de structurer la page, de gérer les interactions utilisateur, et de manipuler les données de manière dynamique. Voici une explication détaillée de chaque partie du code :

- **Héritage du Layout (`@extends('layouts.app')`) :**

Le fichier commence par la directive `@extends` qui permet à ce fichier de hériter d'un layout principal situé dans `layouts.app`. Ce layout contient probablement des éléments communs comme l'en-tête, le pied de page et la navigation. Cela permet de réutiliser ces éléments sur plusieurs pages sans dupliquer le code.

- **Définition du Titre de la Page (`@section('title')`) :**

La section `@section('title')` définit le titre de la page qui sera affiché dans l'onglet du navigateur. La fonction `__()` est utilisée ici pour permettre la traduction du titre dans différentes langues. Par exemple :

```
@section('title')
{{ __('Annonces') }}
@endsection
```

Cette section sera remplacée dans le layout principal à l'endroit où `@yield('title')` est utilisé.

- **Breadcrumb Navigation :**

La navigation en fil d'Ariane est créée pour aider les utilisateurs à se situer dans l'application. Elle est construite avec une liste ordonnée (``) contenant des éléments de navigation. Par exemple :

```

<nav aria-label="breadcrumb">
    <ol class="breadcrumb">
        <li class="breadcrumb-item">
            <a href="{{ route('dashboard') }}" class="text-primary">Dashboard</a>
        </li>
        <li class="breadcrumb-item active" aria-current="page">
            {{ __('Liste des Événements') }}
        </li>
    </ol>
</nav>

```

Le lien vers le tableau de bord est dynamique et généré avec `route('dashboard')`, ce qui signifie qu'il redirige vers la route nommée "dashboard" définie dans vos fichiers de routes.

— **Inclusion de Messages Flash (@include('flash-message')) :**

Cette ligne permet d'inclure un fichier partiel nommé `flash-message.blade.php` qui est probablement utilisé pour afficher des notifications ou des messages flash (par exemple, "Opération réussie" après une action).

```
@include('flash-message')
```

— **Boutons d'Action :**

Le bloc suivant contient des boutons permettant à l'utilisateur de filtrer les événements par date ou d'ajouter un nouvel événement. Ces boutons sont stylisés avec Bootstrap, et utilisent des icônes de FontAwesome pour une meilleure présentation :

```

<button class="btn btn-info btn-sm" id="filterBtn">
    <i class="fa fa-filter"></i> {{ __('Filtrer par date') }}
</button>
<a href="{{ route('events_billets.create') }}" class="btn btn-primary btn-sm ml-2">
    <i class="fa fa-plus"></i> {{ __('Ajouter un événement') }}
</a>

```

Le premier bouton ouvre un modal pour filtrer les événements, tandis que le deuxième bouton redirige vers une page de création d'événement.

— **Tableau des Événements :**

Le tableau HTML est utilisé pour afficher les événements. Il est amélioré avec DataTables, une bibliothèque jQuery qui fournit des fonctionnalités avancées telles que le tri, la recherche, et la pagination.

```

<table class="table table-bordered table-hover w-100" id="annoncesTable">
    <thead class="thead-light">
        <tr>
            <th scope="col">#</th>

```

```

        <th scope="col">{{ __('Titre de l\'événement') }}</th>
        <th scope="col">{{ __('Date') }}</th>
        <th scope="col">{{ __('Image') }}</th>
        <th scope="col">{{ __('Lieu') }}</th>
        <th scope="col">{{ __('Adresse') }}</th>
        <th scope="col">{{ __('Statut de l\'événement') }}</th>
        <th scope="col">{{ __('Date de création') }}</th>
        <th scope="col">{{ __('Actions') }}</th>
    </tr>
</thead>
<tbody></tbody>
</table>

```

Le tableau est initialisé avec DataTables dans le script JavaScript en bas du fichier. Les colonnes sont définies pour afficher des informations telles que le titre, la date, l'image, le lieu, et le statut de l'événement.

— Modal de Filtrage :

Le modal permet aux utilisateurs de filtrer les événements par date. Il est déclenché par le bouton "Filtrer par date". Le modal comprend des champs pour sélectionner une date de début et une date de fin :

```

<div id="filterModal" class="modal fade" tabindex="-1" role="dialog"
aria-labelledby="filterModalLabel"
aria-hidden="true">
<div class="modal-dialog" role="document">
    <div class="modal-content">
        <div class="modal-header">
            <h5 class="modal-title" id="filterModalLabel">{{ __('Filtrer par date') }}</h5>
            <button type="button" class="close" data-dismiss="modal"
aria-label="Close">
                <span aria-hidden="true">&times;</span>
            </button>
        </div>
        <div class="modal-body">
            <div class="form-group">
                <label for="startDate" class="col-form-label">{{ __('Date de début') }}</label>
                <input type="date" class="form-control" id="startDate">
            </div>
            <div class="form-group">
                <label for="endDate" class="col-form-label">{{ __('Date de fin') }}</label>
                <input type="date" class="form-control" id="endDate">
            </div>
        </div>
        <div class="modal-footer">

```

```

        <button type="button" class="btn btn-secondary" data-
-dDismiss="modal">{{ __('Fermer') }}</button>
        <button type="button" class="btn btn-primary"
            id="applyFilter">{{ __('Filtrer') }}</button>
    </div>
</div>
</div>
</div>

```

— **JavaScript pour le Filtrage et la Gestion des Données :**

Le script JavaScript gère plusieurs aspects de l'interaction utilisateur avec le tableau et le modal. Il initialise DataTables, montre le modal de filtrage, applique les filtres, et gère la suppression des événements.

```

<script>
var table;

$(document).ready(function() {
    // Afficher le modal de filtre lors du clic sur le bouton de filtre
    $('#filterBtn').on('click', function() {
        $('#filterModal').modal('show');
    });

    // Appliquer le filtre et recharger les données du tableau
    $('#applyFilter').on('click', function() {
        var startDate = $('#startDate').val();
        var endDate = $('#endDate').val();

        table.ajax.url("{{ route('events_billets.index') }}?from_date=" -
            endDate).load();
        $('#filterModal').modal('hide');
    });

    // Initialiser DataTable
    table = $('#annoncesTable').DataTable({
        processing: true,
        serverSide: true,
        ajax: "{{ route('events_billets.index') }}",
        columns: [
            {
                data: 'event_id',
                name: 'event_id'
            },
            {
                data: 'event_title',
                name: 'event_title'
            },
            {
                data: 'event_date',
                name: 'event_date'
            }
        ]
    });
});

```

```

        name: 'event_date'
    },
{
    data: 'event_image',
    name: 'event_image',
    render: function(data, type, row) {
        var imageUrl = "{{ asset('storage/') }}/" + data
        ;
        return '';
    }
},
{
    data: 'event_city',
    name: 'event_city'
},
{
    data: 'event_address',
    name: 'event_address'
},
{
    data: 'event_status',
    name: 'event_status'
},
{
    data: 'created_at',
    name: 'created_at'
},
{
    data: null,
    render: function(data, type, row) {
        return '
<div class="d-flex gap-2">

<form action="{{
route('events_billets.destroy', ['id' =>
'__id__']) }}" method="POST"
class="d-inline">
@csrf
@method('DELETE')
<button type="submit" class="btn btn-sm btn-outline-danger delete-btn">
<i class="fa fa-trash"></i>
</button>
</form>
</div>
'.replace(/__id__/g, row.event_id);
    }
}

```

```

        }
    ],
    order: [
        [1, 'asc']
    ]
});

// Confirmation de suppression avec SweetAlert
$('#annoncesTable').on('click', '.delete-btn', function(event) {
    event.preventDefault(); // Empêche la soumission immédiate du formulaire

    var form = $(this).closest('form');

    Swal.fire({
        title: 'Confirmation',
        text: 'Êtes-vous sûr de vouloir supprimer cet Evenement ?',
        icon: 'warning',
        showCancelButton: true,
        confirmButtonColor: '#3085d6',
        cancelButtonColor: '#d33',
        confirmButtonText: 'Oui, supprimer!',
        cancelButtonText: 'Annuler'
    }).then((result) => {
        if (result.isConfirmed) {
            form.submit();
        }
    });
});

```

- **Filtrer les Événements** : Lorsqu'un utilisateur sélectionne des dates de début et de fin, puis clique sur "Filtrer", les données du tableau sont rechargées avec les événements qui se déroulent dans cette plage de dates. Le filtre est appliqué en mettant à jour l'URL utilisée par DataTables pour récupérer les données.
- **Confirmation de Suppression** : Lorsqu'un utilisateur clique sur le bouton de suppression d'un événement, une boîte de dialogue SweetAlert apparaît pour demander une confirmation. Si l'utilisateur confirme, le formulaire de suppression est soumis et l'événement est supprimé.
- **Sécurisation des Actions avec CSRF** : Les formulaires de suppression utilisent un jeton CSRF (@csrf) et la méthode HTTP DELETE (@method('DELETE')) pour protéger contre les attaques CSRF (Cross-Site Request Forgery). Ce mécanisme garantit que seules les demandes provenant du site web légitime sont acceptées par le serveur.

```
<form action="{{ route('events_billets.destroy', ['id' => '__id__']) }}" method="DELETE">
```

```

@csrf
@method('DELETE')
<button type="submit" class="btn btn-sm btn-outline-danger delete-btn">
    <i class="fa fa-trash"></i>
</button>
</form>

```

9.4 Script JavaScript

Ce script JavaScript gère plusieurs aspects dynamiques de la page, y compris l'affichage du tableau des événements, le filtrage par date, et la suppression d'un événement avec confirmation.

```

<script>
var table;

$(document).ready(function() {
    // Afficher le modal de filtre lors du clic sur le bouton de filtre
    $('#filterBtn').on('click', function() {
        $('#filterModal').modal('show');
    });

    // Appliquer le filtre et recharger les données du tableau
    $('#applyFilter').on('click', function() {
        var startDate = $('#startDate').val();
        var endDate = $('#endDate').val();

        table.ajax.url("{{ route('events_billets.index') }}?from_date=" +
            startDate + "&to_date=" + endDate).load();
        $('#filterModal').modal('hide');
    });

    // Initialiser DataTable
    table = $('#annoncesTable').DataTable({
        processing: true,
        serverSide: true,
        ajax: "{{ route('events_billets.index') }}",
        columns: [
            { data: 'event_id', name: 'event_id' },
            { data: 'event_title', name: 'event_title' },
            { data: 'event_date', name: 'event_date' },
            {
                data: 'event_image',
                name: 'event_image',
                render: function(data, type, row) {
                    var imageUrl = "{{ asset('storage/') }}/" + data;
                    return '';
    }
},
{ data: 'event_city', name: 'event_city' },
{ data: 'event_address', name: 'event_address' },
{ data: 'event_status', name: 'event_status' },
{ data: 'created_at', name: 'created_at' },
{
    data: null,
    render: function(data, type, row) {
        return '
            <div class="d-flex gap-2">
                <form action="{{
                    route('events_billets.destroy', ['id' => '__id__'])
                    class="d-inline">
                    @csrf
                    @method('DELETE')
                    <button type="submit" class="btn btn-sm
                        btn-outline-danger delete-btn">
                        <i class="fa fa-trash"></i>
                    </button>
                </form>
            </div>
            '.replace(/__id__/g, row.event_id);
    }
}
],
order: [[1, 'asc']]
});

// Confirmation de suppression avec SweetAlert
$('#annoncesTable').on('click', '.delete-btn', function(event) {
    event.preventDefault(); // Empêche la soumission immédiate du formulaire

    var form = $(this).closest('form');

    Swal.fire({
        title: 'Confirmation',
        text: 'Êtes-vous sûr de vouloir supprimer cet Evenement ?',
        icon: 'warning',
        showCancelButton: true,
        confirmButtonColor: '#3085d6',
        cancelButtonColor: '#d33',
        confirmButtonText: 'Oui, supprimer!',
        cancelButtonText: 'Annuler'
    }).then((result) => {
        if (result.isConfirmed) {

```

```

        form.submit();
    }
});
});
});
});
</script>

```

— **Initialisation de la Variable `table` :**

```
var table;
```

La variable `table` est utilisée pour stocker l'instance de `DataTable`, qui permet de gérer le tableau de manière dynamique.

— **Fonction `$(document).ready()` :**

```

$(document).ready(function() {
    // Code à exécuter lorsque le document est prêt
});

```

Cette fonction est exécutée lorsque le DOM est complètement chargé. Elle initialise les interactions dynamiques de la page.

— **Affichage du Modal de Filtrage :**

```

$('#filterBtn').on('click', function() {
    $('#filterModal').modal('show');
});

```

Lorsque l'utilisateur clique sur le bouton "Filtrer par date", le modal de filtrage s'affiche.

— **Application du Filtre :**

```

$('#applyFilter').on('click', function() {
    var startDate = $('#startDate').val();
    var endDate = $('#endDate').val();

    table.ajax.url("{{ route('events_billets.index') }}?from_date=" +
        startDate + "&to_date=" + endDate).load();
    $('#filterModal').modal('hide');
});

```

Cette section applique le filtre sur le tableau en fonction des dates sélectionnées, puis recharge les données du tableau.

— **Initialisation de `DataTable` :**

```

table = $('#annoncesTable').DataTable({
    processing: true,
    serverSide: true,
    ajax: "{{ route('events_billets.index') }}",
    columns: [
        { data: 'event_id', name: 'event_id' },
        { data: 'event_title', name: 'event_title' },
        { data: 'event_date', name: 'event_date' },
        ...
    ],
    order: [[1, 'asc']]
});

```

DataTable est initialisé pour afficher les données des événements sous forme de tableau avec une pagination, un tri, et des fonctionnalités de recherche.

— Affichage des Images :

```

render: function(data, type, row) {
    var imageUrl = "{{ asset('storage/') }}/" + data;
    return '';
}

```

Pour chaque ligne du tableau, cette fonction affiche l'image associée à l'événement en utilisant un élément HTML .

— Confirmation de Suppression avec SweetAlert :

```

$('#annoncesTable').on('click', '.delete-btn', function(event) {
    event.preventDefault(); // Empêche la soumission immédiate du formulaire

    var form = $(this).closest('form');

    Swal.fire({
        title: 'Confirmation',
        text: 'Êtes-vous sûr de vouloir supprimer cet Evenement ?',
        icon: 'warning',
        showCancelButton: true,
        confirmButtonColor: '#3085d6',
        cancelButtonColor: '#d33',
        confirmButtonText: 'Oui, supprimer !',
        cancelButtonText: 'Annuler'
    }).then((result) => {
        if (result.isConfirmed) {
            form.submit();
        }
    });
});

```

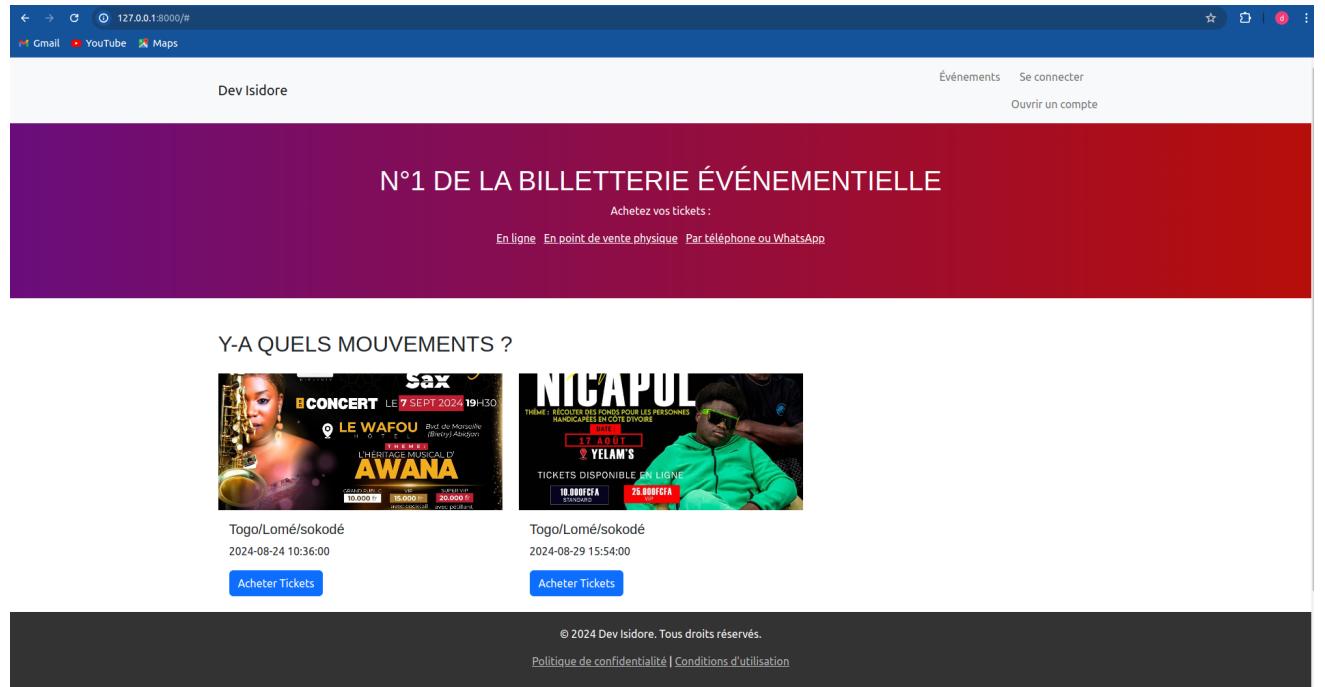
Lorsqu'un utilisateur clique sur le bouton de suppression d'un événement, une boîte de dialogue SweetAlert apparaît pour demander une confirmation avant de soumettre le formulaire de suppression.

Ce fichier `index.blade.php` illustre l'intégration de plusieurs fonctionnalités avancées dans une vue Laravel, offrant une interface utilisateur riche et interactive pour la gestion des événements.

10 Démonstration

10.1 Welcome page

La page d'accueil de notre application, telle que présentée dans la figure, est conçue pour offrir aux utilisateurs un aperçu rapide des événements disponibles et pour faciliter l'achat de tickets.



10.1.1 Bannière Principale

En haut de la page, une bannière colorée présente le slogan **N°1 DE LA BILLETTERIE ÉVÉNEMENTIELLE**. Sous cette bannière, les utilisateurs peuvent voir des options pour acheter des tickets en ligne, en point de vente physique, ou par téléphone/WhatsApp. Cette section est destinée à attirer immédiatement l'attention de l'utilisateur et à lui offrir des options claires pour effectuer des achats.

10.1.2 Présentation des Événements

La section principale de la page présente les événements actuels sous le titre **Y-A QUELS MOUVEMENTS ?**. Chaque événement est affiché avec une image promotionnelle, la date, l'heure, et le lieu de l'événement. Un bouton "Acheter Tickets" est

disponible sous chaque événement pour permettre aux utilisateurs d'accéder directement à la page d'achat.

10.1.3 Pied de page

Le pied de page, visible au bas de l'image, contient les informations de copyright ainsi que des liens vers les politiques de confidentialité et les conditions d'utilisation. Ces éléments sont essentiels pour assurer la transparence et le respect des règles légales.

Cette présentation visuelle est essentielle pour offrir une expérience utilisateur fluide, tout en mettant en valeur les principaux événements et en facilitant l'accès à l'achat de tickets.

10.2 Liste des Événements

La page de liste des événements, comme illustrée à la Figure, permet aux utilisateurs de visualiser les différents événements disponibles sur la plateforme, avec la possibilité de filtrer les résultats en fonction de leurs préférences.

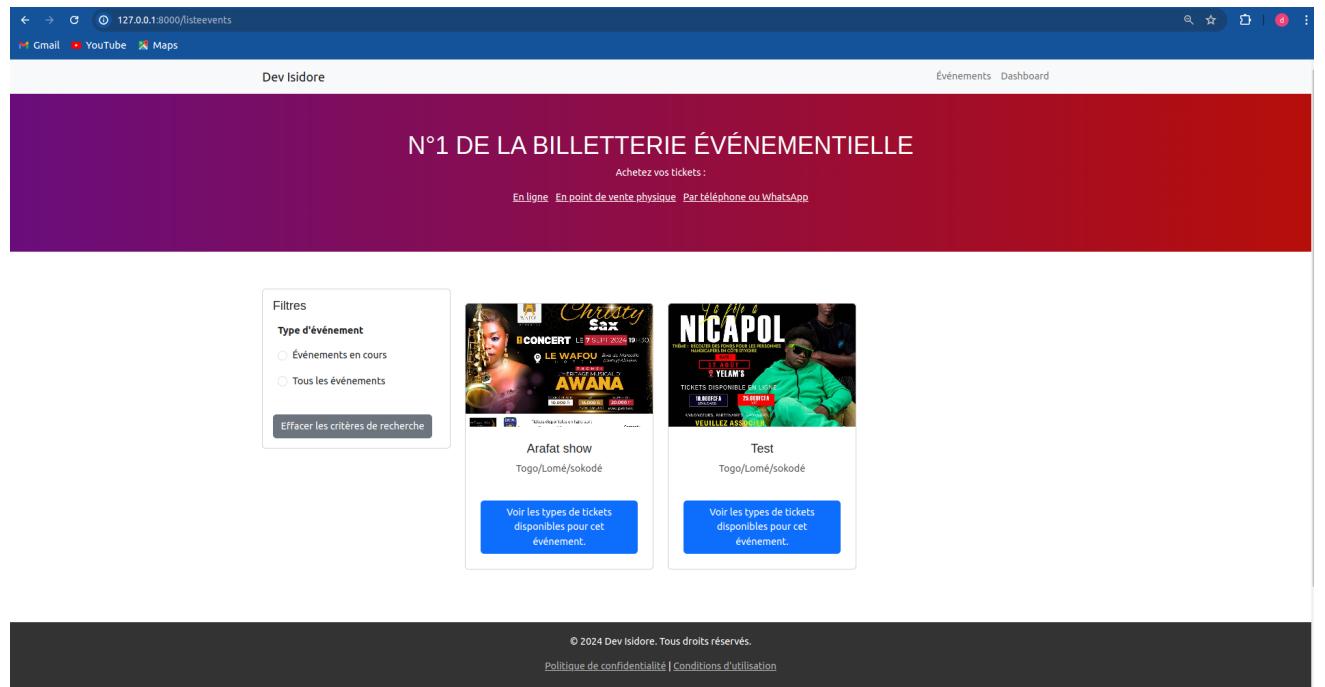


FIGURE 2 – Liste de tous les événements disponibles

10.2.1 Bannière Principale

En haut de la page, la même bannière que celle de la page d'accueil est utilisée, mettant en avant le slogan **N°1 DE LA BILLETTERIE ÉVÉNEMENTIELLE** et les options d'achat de tickets, comme décrit précédemment.

10.2.2 Filtre de Recherche

Sur la partie gauche de la page, un panneau de filtres est disponible pour affiner la recherche. Les utilisateurs peuvent sélectionner le type d'événement qu'ils souhaitent

voir :

- **Événements en cours** : Affiche uniquement les événements actuellement disponibles.
- **Tous les événements** : Affiche la totalité des événements, qu'ils soient à venir ou passés.

Un bouton **Effacer les critères de recherche** est également disponible pour réinitialiser les filtres.

10.2.3 Liste des Événements

Au centre de la page, les événements sont affichés sous forme de cartes individuelles. Chaque carte contient :

- Une image de promotion de l'événement.
- Le titre de l'événement.
- Le lieu (ville/pays).
- Un bouton **Voir les types de tickets disponibles pour cet événement** permettant de consulter et d'acheter des tickets pour cet événement spécifique.

10.2.4 Procédure d'achat de ticket

Lorsque l'on clique sur "Voir les types de tickets", une boîte de dialogue s'ouvre et affiche les types de tickets associés à l'offre, et l'utilisateur n'a plus qu'à choisir le type qu'il souhaite payer :

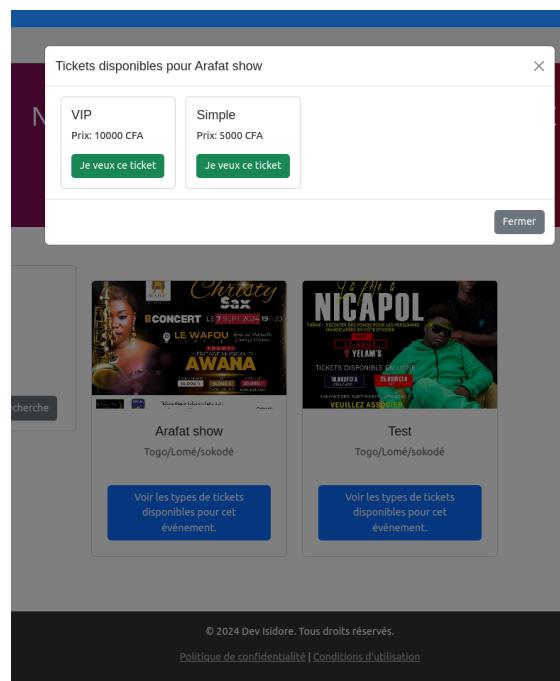


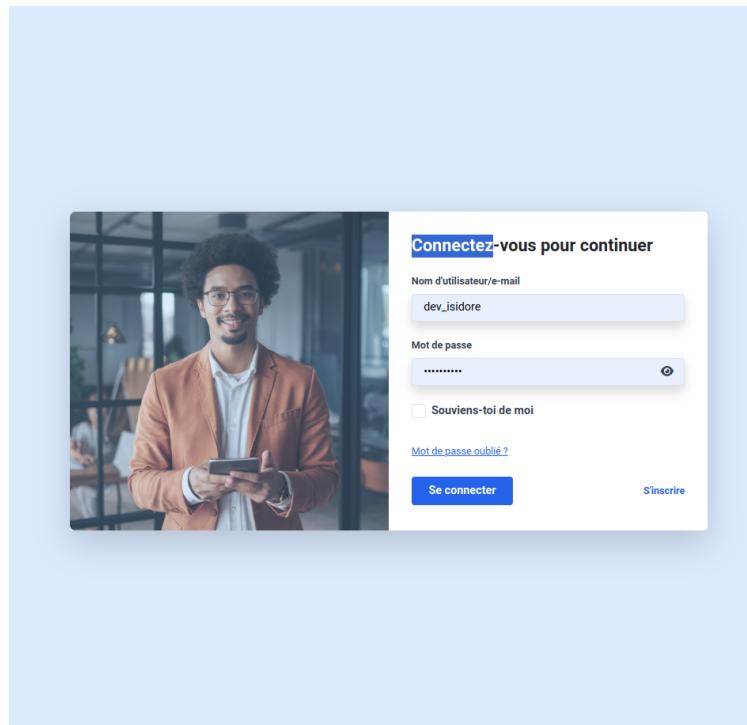
FIGURE 3 – Choix du ticket

Après avoir cliqué, si l'utilisateur est inscrit, il est dirigé vers la page de déclenchement de la commande :

The screenshot shows a web application for ticketing. At the top, there's a navigation bar with 'Dev Isidore' on the left and 'Événements' and 'Dashboard' on the right. Below the header is a purple banner with the text 'N°1 DE LA BILLETTERIE ÉVÉNEMENTIELLE' and 'Achetez vos tickets : En ligne En point de vente physique Par téléphone ou WhatsApp'. The main content area has two sections: one for entering information ('Remplissez vos informations') and another for viewing event details ('Arafat show'). The event details include a thumbnail image of a woman, the name 'Arafat show', the date '2024-08-24 10:30:00', the address 'Togo/Lomé/sokodé', and a price of '5000 CFA'. At the bottom of the page is a footer with copyright information and links to 'Politique de confidentialité' and 'Conditions d'utilisation'.

FIGURE 4 – Déclenchement de la commande

sinon il est redirigé vers la page de connexion : Après s'être connecté, l'utilisateur sera



redirigé vers la page de déclenchement de la commande.

Ensuite, il sera redirigé vers la page de confirmation de paiement

où il devra entrer les différentes informations, notamment le mode de paiement, le type de commande, l'email, et le numéro de téléphone.

Résumé de votre commande

Nombre de tickets: 5

Prix total: 25000 CFA

Informations de Paiement

Mode de paiement: T-Money

Type de Commande: Commande en ligne

Email: Entrez votre email

Téléphone: Entrez votre téléphone

Confirmer le paiement

Christy SAX
CONCERT LE 7 SEPT 2024 19H00
LE WAROU
Arafat show

Date: 2024-08-24 10:36:00

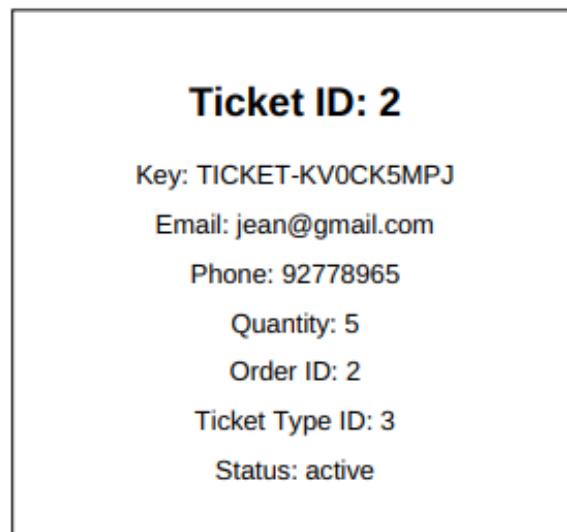
Adresse: Togo/Lomé/sokodé, Togo/Lomé/sokodé

Simple

Prix: 5000 CFA

© 2024 Dev Isidore. Tous droits réservés.
[Politique de confidentialité](#) | [Conditions d'utilisation](#)

Puis le ticket est généré automatiquement :



10.2.5 Pied de page

Comme pour la page d'accueil, le pied de page contient les informations de copyright ainsi que les liens vers les politiques de confidentialité et les conditions d'utilisation.

Cette structure permet une navigation claire et intuitive, facilitant la recherche et la sélection d'événements par les utilisateurs.

10.3 Page d’Inscription

La page d’inscription, illustrée à la Figure, permet aux utilisateurs de créer un compte sur la plateforme en fournissant des informations personnelles et professionnelles.

The figure shows a user interface for a sign-up process. On the left, there is a photograph of a young man with dark curly hair and glasses, wearing an orange blazer over a white shirt, looking at his phone. To the right of the photo is a white rectangular form with a black border. The form has a title "S'inscrire" at the top. It contains six input fields: "Nom" with the value "Bil", "Prénom" with the value "Stone", "Entreprise" with the value "OTR", "Email" with the value "logonedevel@gmail.com", "Ville" with the value "Tchaoudjo", and "Adresse" with the value "Sokodé, Région Centrale, Togo". Below these fields is a small checkbox labeled "J'accepte les Conditions d'utilisation". At the bottom of the form are two buttons: a blue rectangular button on the left containing the text "S'inscrire" and a smaller grey rectangular button on the right containing the text "Se connecter".

FIGURE 5 – Page d’inscription de l’application

10.3.1 Formulaire d’Inscription

Le formulaire d’inscription est divisé en plusieurs champs que l’utilisateur doit remplir :

- **Nom** : Le nom de famille de l’utilisateur.
- **Prénom** : Le prénom de l’utilisateur.
- **Entreprise** : Le nom de l’entreprise où l’utilisateur travaille (facultatif).
- **Email** : L’adresse email de l’utilisateur, qui servira d’identifiant pour se connecter.
- **Ville** : La ville de résidence de l’utilisateur.
- **Adresse** : L’adresse complète de l’utilisateur, incluant la région et le pays.

10.3.2 Boutons d’Action

Deux boutons sont disponibles à la fin du formulaire :

- **S’inscrire** : Pour soumettre le formulaire et créer le compte.
- **Se connecter** : Un lien permettant de se rendre à la page de connexion pour les utilisateurs ayant déjà un compte.

10.3.3 Illustration Visuelle

Sur la gauche du formulaire, une image est affichée pour rendre la page plus accueillante et professionnelle. Cette présentation visuelle améliore l'expérience utilisateur en rendant le processus d'inscription plus engageant.

Une fois que l'utilisateur clique sur "S'inscrire", un email lui est automatiquement envoyé avec les informations de connexion au compte qu'il vient de créer.

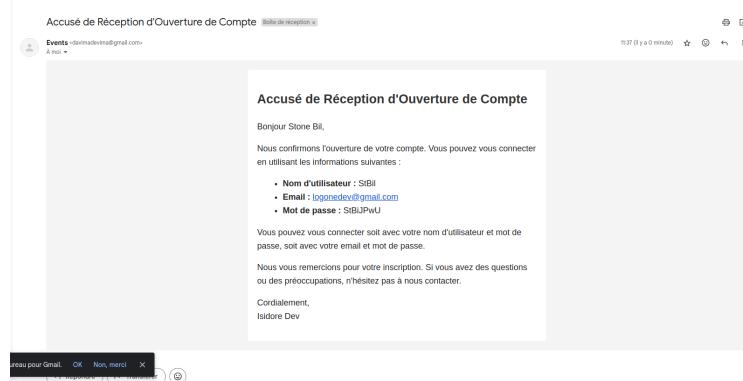


FIGURE 6 – Courriel envoyé depuis l'application

11 Événements

11.1 Publication d'événements

Cette section explique comment ajouter et publier un événement sur la plateforme de gestion d'événements.

#	Titre de l'événement	Date	Image	Lieu	Statut de l'événement	Date de création	Actions	
2	Avril show	2024-08-24 10:36:00		Togo/Lomé/Ikokodé	Togo/Lomé/Ikokodé	upcoming	2024-08-15T07:29:07.000000Z	
1	Test	2024-08-29 15:54:00		Togo/Lomé/Ikokodé	Togo/Lomé/Ikokodé	upcoming	2024-08-14T12:54:49.000000Z	

11.2 Accéder à la page des événements

Pour publier un événement, il faut tout d'abord accéder à la page dédiée à la gestion des événements. Suivez les étapes suivantes :

1. Depuis le tableau de bord, cliquez sur l'onglet **EVENTS**.
2. Vous serez redirigé vers la liste des événements déjà existants.

11.3 Ajouter un nouvel événement

Pour ajouter un nouvel événement :

1. Cliquez sur le bouton **Ajouter un événement** situé en haut à droite de la page des événements.

2. Un formulaire s'affiche avec les champs suivants :

Nous avons 4 étapes :

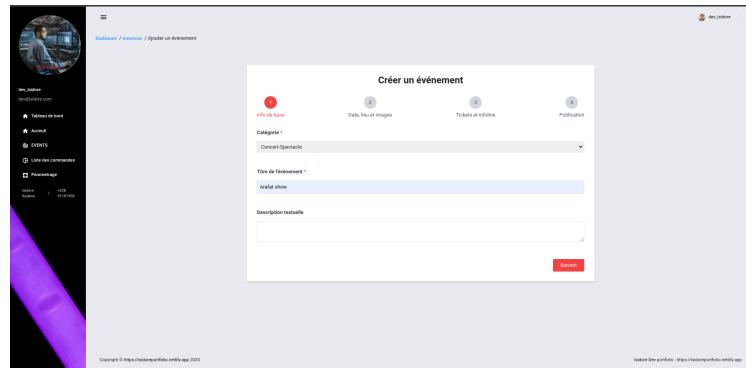


FIGURE 7 – Étape 1

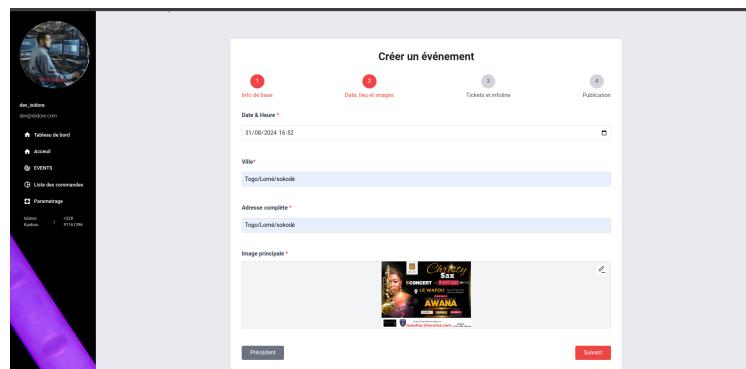


FIGURE 8 – Étape 2

Créer un événement

Informations sur les tickets

Nom du ticket *	VIP
Prix du ticket *	50000
Quantité de tickets *	100
Description	(empty)

Nom du ticket *	Simple
Prix du ticket *	10000
Quantité de tickets *	50
Description	(empty)

Ajouter un autre type de ticket

Précédent Suivant

FIGURE 9 – Étape 3

Créer un événement

Vérifiez les informations de l'événement

Création d'un événement

Titre de l'événement : Arafat show

Description :

Date & Heure : 2024-09-31T18:52

Lieu : Togo/Lome/sakodé

Types de tickets :

Nom du ticket :	VIP
Prix :	50000 F CFA
Quantité :	100
Quantité réelle de tickets :	(empty)
Description :	(empty)

Précédent Publier

FIGURE 10 – Étape 4

3. Une fois le formulaire complété, cliquez sur **publier** pour valider et publier l'événement.

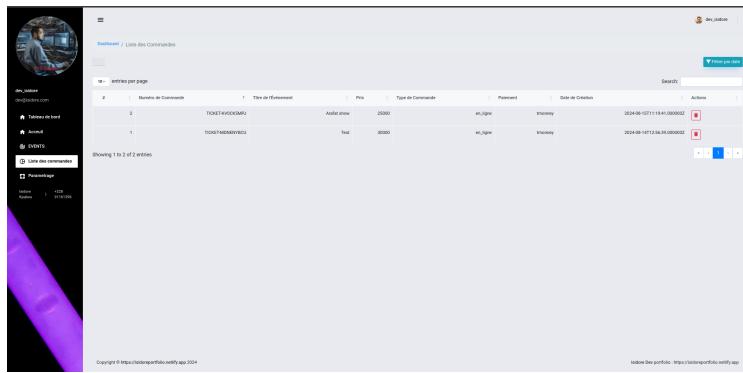
11.4 Filtrer les événements

Pour filtrer les événements par date :

- Utilisez le bouton **Filtrer par date** en haut de la page, à côté du bouton *Ajouter un événement*.
- Sélectionnez la date voulue pour n'afficher que les événements correspondants.

Cette interface vous permet de gérer facilement tous les aspects liés à la publication, la modification, et la suppression des événements à venir.

12 Gestion des Commandes



#	Numéro de Commande	Titre de l'événement	Prix	Type de Commande	Paiement	Date de Crédation	Actions
2	TICKET-KVOCKSMPJ	Autoshow	25000	en_ligne	tmoney	2024-08-10T11:19:41.000Z	
1	TICKET-KVOCKSMPJ	Autoshow	30000	en_ligne	tmoney	2024-08-10T12:04:26.000Z	

Cette section explique comment consulter et gérer les commandes effectuées sur la plateforme.

12.1 Accéder à la page des commandes

Pour visualiser et gérer les commandes, suivez les étapes suivantes :

1. Depuis le tableau de bord, cliquez sur l'onglet **Liste des commandes**.
2. Vous serez redirigé vers une page qui affiche la liste de toutes les commandes existantes.

12.2 Détails de la commande

Dans la page de liste des commandes, les informations suivantes sont disponibles pour chaque commande :

- **Numéro de commande** : Chaque commande est associée à un numéro unique, comme *TICKET-KVOCKSMPJ*.
- **Titre de l'événement** : Le nom de l'événement pour lequel la commande a été passée.
- **Prix** : Le montant total de la commande en monnaie locale (ex. *25000*).
- **Type de commande** : Indique le mode de commande, généralement en ligne (ex. *en_ligne*).
- **Paiement** : Le moyen de paiement utilisé pour la commande (ex. *tmoney*).
- **Date de création** : La date et l'heure auxquelles la commande a été passée.

12.3 Gestion des commandes

Pour chaque commande, deux actions principales peuvent être réalisées depuis la page de liste :

- **Supprimer une commande** : Pour supprimer une commande, cliquez sur l'icône de la poubelle à droite de la commande souhaitée. Une confirmation peut être requise avant suppression définitive.
- **Filtrer les commandes par date** : Vous pouvez utiliser le bouton **Filtrer par date** en haut de la page pour afficher les commandes passées sur une période donnée.

12.4 Rechercher une commande spécifique

Si vous recherchez une commande spécifique :

- Utilisez le champ de recherche situé en haut à droite de la page.
- Entrez des mots-clés comme le numéro de commande ou le titre de l'événement pour filtrer les résultats.

12.5 Visualiser plus de commandes

Si la liste des commandes est longue, utilisez les options de pagination au bas de la page pour naviguer entre les différentes pages de commandes. Vous pouvez aussi ajuster le nombre de commandes affichées par page avec le sélecteur **entries per page**.

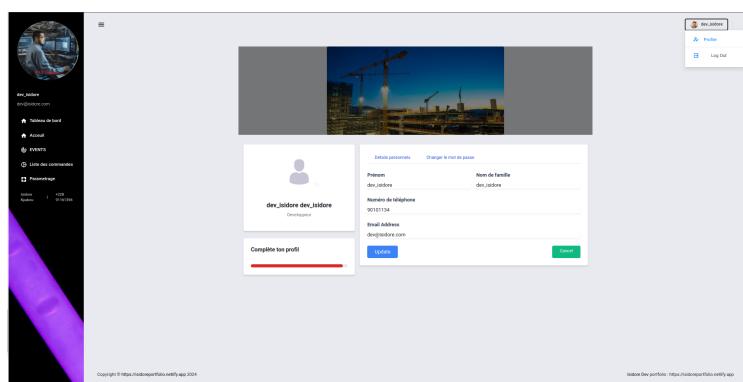
Remarque : La gestion des commandes vous permet de suivre efficacement les achats et paiements relatifs aux événements.

13 Démonstration : Mise à Jour du Profil Utilisateur

Cette section présente l'interface de mise à jour du profil utilisateur, permettant à un utilisateur connecté de modifier ses informations personnelles telles que le prénom, le nom de famille, le numéro de téléphone et l'adresse email.

13.1 Figure

L'image ci-dessous montre l'interface de mise à jour du profil utilisateur. Cette interface est composée de plusieurs éléments importants :



- **Informations Personnelles** : L'utilisateur peut voir son prénom, son nom de famille, son numéro de téléphone ainsi que son adresse email dans les champs correspondants. Il peut mettre à jour ces informations en modifiant le contenu des champs.

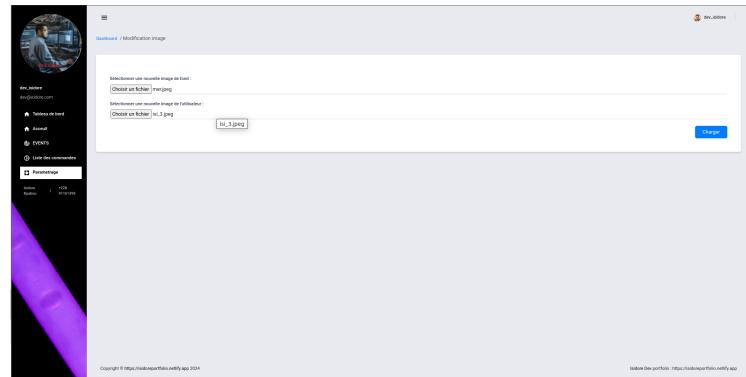
- **Boutons d'action :**
 - **Bouton “Update”** : Permet de sauvegarder les modifications effectuées sur le profil.
 - **Bouton “Cancel”** : Permet d'annuler les modifications sans les enregistrer.
- **Changer le mot de passe** : Une section séparée permet à l'utilisateur de modifier son mot de passe, accessible via l'onglet “Changer le mot de passe”.
- **Option de déconnexion** : Une option “Log Out” permet à l'utilisateur de se déconnecter.

13.2 Explication des fonctionnalités

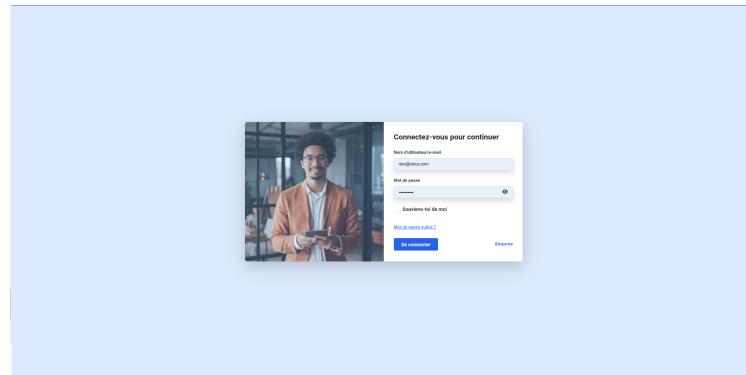
- **Mise à jour du profil** : Cette fonctionnalité permet aux utilisateurs de garder leurs informations à jour, notamment leur numéro de téléphone ou leur email.
- **Sécurité** : L'utilisateur peut modifier son mot de passe pour garantir la sécurité de son compte.

14 Modification des différents images d'arrière plan de login

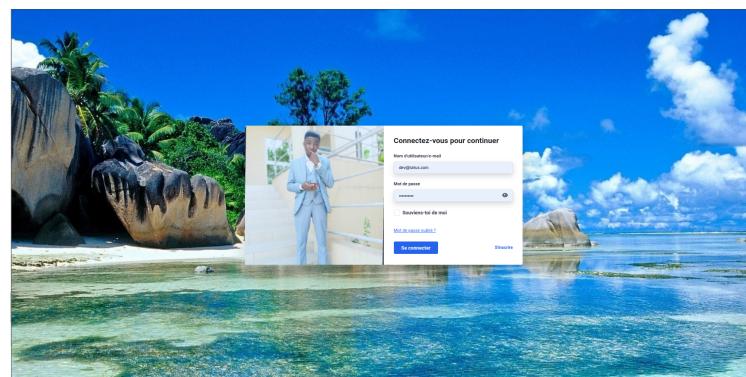
Page de Modification d'image ce trouvant dans la section paramétrage



Avant



Après



15 Conclusion

L'API de billetterie événementielle offre une solution efficace pour intégrer des services tiers, tout en assurant la gestion des événements, des commandes, et des tickets de manière

transparente et sécurisée. Cette documentation vise à fournir toutes les informations nécessaires pour une utilisation optimale de l'API dans un environnement sécurisé et performant.