

Synthèse des projets LARAVEL

KONDI Abdoul malik

January 2, 2023

Contents

1	Préambule	2
2	Installation d'un projet laravel via composer	3
2.1	Prérequis	3
2.2	Organisation d'un projet laravel.	5
2.2.1	Répertoire App.	5
2.2.2	Autres repertoires.	6
2.2.3	Fichiers de racine.	6
2.2.4	Accessibilité.	7
3	Implémentation du modèle UML.	8
3.1	Les Migrations	8
3.1.1	documents de référence	13
3.2	Les modèles	14
3.2.1	documents de référence	17
3.3	Les Controllers	18
3.3.1	Documents de référence	18
3.4	Les Routes	19
3.4.1	Documents de référence	19
3.5	Les vues	19
3.5.1	Documents de référence	24
4	Les Services Providers	25
4.1	Helpers	25
4.2	Service	27
4.3	Authentification avec breez	28
4.3.1	Documents de référence	30
4.4	Localisation	30
4.5	DOM PDF	31
4.6	Import et Export Excel	33

1 Préambule

Ce document est un résumé du framework PHP LARAVEL (v8). Il est rédigé a partir des travaux que les équipes de la promotion **2020** à l'IFNTI (sokodé) on fourni après leurs stage de fin L2.

2 Installation d'un projet laravel via composer

2.1 Prérequis

Pour utiliser la version 8 du framework PHP LARAVEL il faut avoir:

- Une version de PHP supérieure à 7.2.
- Composer (gestionnaire de dépendance php) [lien d'installation de composer].
- Les extensions PHP suivantes doivent être activées.
 - Version $\geq 7.2.0$,
 - Extension PDO (Le driver pour une connexion avec une base de donnée),
 - Extension **pgsql** ou **mysql** ou autre selon votre SGBD.
 - Extension **mbstring**
 - Extension **opcache**
 - Extension **xml**
 - Extension **curl**
 - Extension **gd**
 - Extension **openssl**
 - Extension **json**
 - Extension **fileinfo**
 - Extension **ctype**
 - Extension **BCMath**
 - Extension **soap**
 - Extension **zip**
 - Extension **intl**
 - Extension **common**
 - Extension **imagick**
 - Extension **imap**

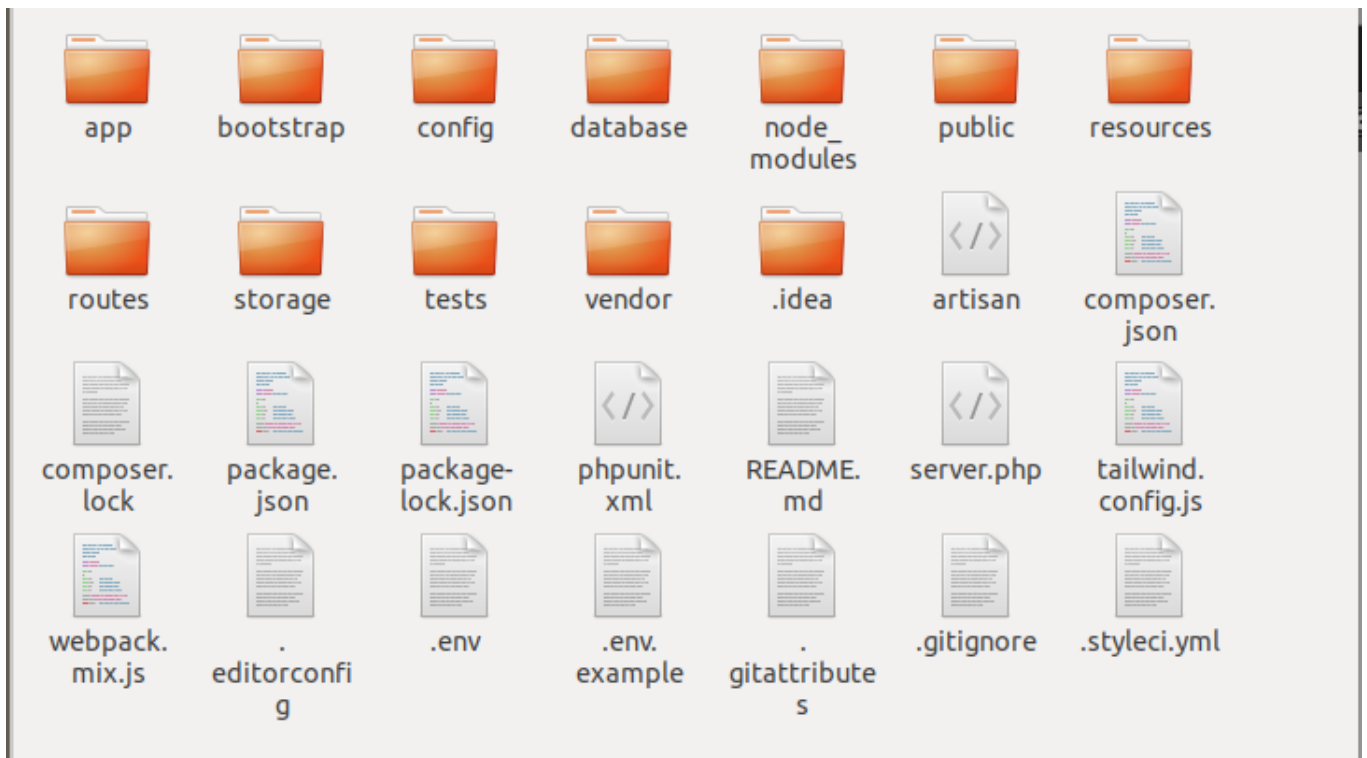
Il y a plusieurs façons de créer une application Laravel. La plus classique consiste à utiliser la commande **create-project** de Composer. voici la syntaxe générale à utiliser.

```
composer create-project laravel/laravel chemin/du/repertoire --prefer-dist "version"
```

Créons une application Laravel 8 qui s'appelle **G_event**.

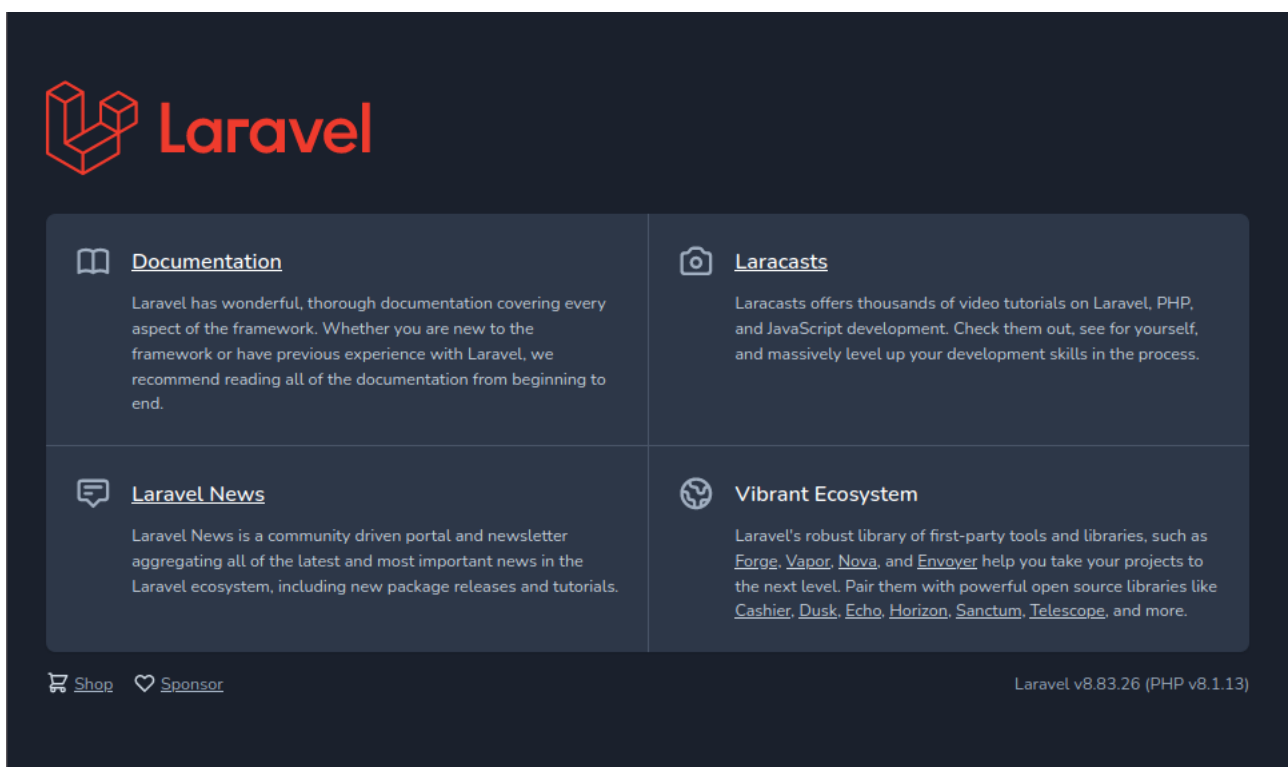
```
composer create-project laravel/laravel G_event --prefer-dist "8.*"
```

Une fois l'installation démarrée, nous avons plus qu'à attendre quelques minutes pour que Composer fasse son travail jusqu'au bout. On verra s'afficher une liste de téléchargements. Au final on se retrouve avec cette architecture :



Laravel est équipé d'un serveur pour le développement qui se lance avec cette commande : **php artisan serve**

On peut vérifier que tout fonctionne bien en allant sur l'URL *http://127.0.0.1:8000*. Normalement nous devons obtenir cette page.



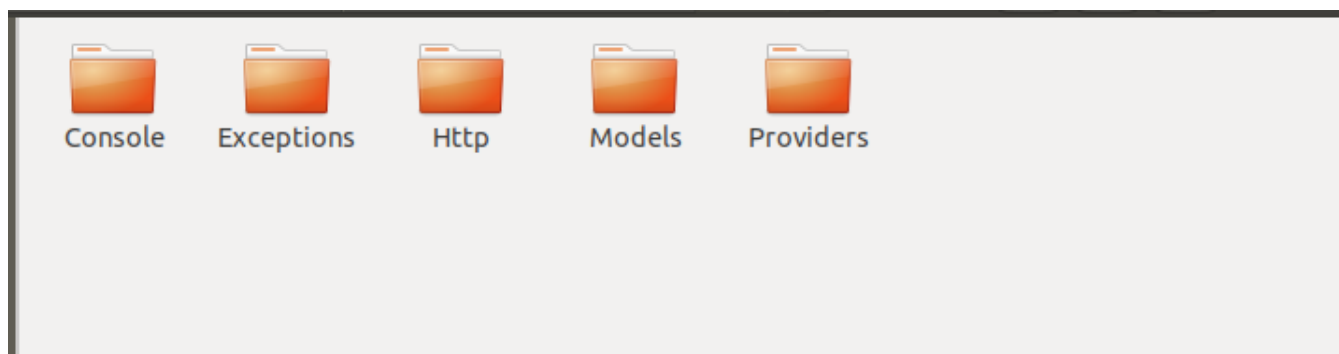
Pour des mises à jour ultérieures il suffit encore d'utiliser Composer avec la commande update : **composer update**

Note : Si vous installez Laravel en téléchargeant directement les fichiers sur Github et en utilisant la commande *composer install*, il vous faut effectuer une action supplémentaire. En effet, dans ce cas la clé de sécurité ne sera pas automatiquement créée et vous allez tomber sur une erreur au lancement. Il faut donc la créer avec la commande. *php artisan key:generate*.

2.2 Organisation d'un projet laravel.

2.2.1 Répertoire App.

Le répertoire **App** est répertoire le plus important de votre projet. C'est lui qui contiendra votre application, c'est à dire tout votre code php (classes, fonctions ...).



Voici donc son contenu :

- **Console** : toutes les commandes en mode console.
- **Exceptions** : pour gérer les erreurs d'exécution.
- **Http** : tout ce qui concerne la communication : contrôleurs, middlewares (il y a 8 middlewares de base qui servent à filtrer les requêtes HTTP) et le kernel.
- **Providers** : tous les fournisseurs de services (providers), il y en a déjà 5 au départ. Les providers servent à initialiser les composants.
- **Models** : le dossier des modèles avec un modèle déjà présent qui concerne les utilisateurs.

2.2.2 Autres repertoires.

Voici une description du contenu des autres dossiers :

- **Bootstrap** : scripts d'initialisation de Laravel pour le chargement automatique des classes, la fixation de l'environnement et des chemins, et pour le démarrage de l'application.
- **Public** : tout ce qui doit être accessible par les utilisateurs (images, CSS, scripts javascript, etc...).
- **Config** : contient toutes les configurations de l'application (authentification, cache, base de données, espaces de noms, emails, systèmes de fichier, session, etc...).
- **Database** : migrations et populations. Le dossier database permet la gestion de la base données. Il contient trois sous-dossier. Les migrations sont des fichiers permettant de décrire votre base de données afin de permettre à Laravel de créer, modifier ou supprimer les tables et les colonnes automatiquement pour vous.
- **Resources** : vues, fichiers de langage et assets (par exemple les fichiers Sass).
- **Routes** : la gestion des urls d'entrée de l'application.
- **Storage** : données temporaires de l'application : vues compilées, caches, clés de session.
- **Tests** : fichiers de tests unitaires.
- **Vendor** : tous les composants de Laravel et de ses dépendances (créé par composer).

2.2.3 Fichiers de racine.

Il y a un certain nombre de fichiers dans la racine dont voici les principaux :

- **artisan** : outil en ligne de commande de Laravel pour des tâches de gestion.
- **composer.json** : fichier de référence de composer.
- **package.json** : fichier de référence de npm pour les assets.
- **phpunit.xml** : fichier de configuration de phpunit (pour les tests unitaires).
- **.env** : fichier pour spécifier l'environnement d'exécution.

Le fichier **.env** contient les mots de passe de vos services ainsi que toutes les données sensibles de votre application (mot de passe de base de données, adresse de la base de données...). **Ce fichier ne doit jamais être partagé.** Afin de connaître les informations à renseigner, il existe un fichier *.env.example* qui contient uniquement des valeurs d'exemple.

Nous verrons tout cela progressivement dans ce cours.

2.2.4 Accessibilité.

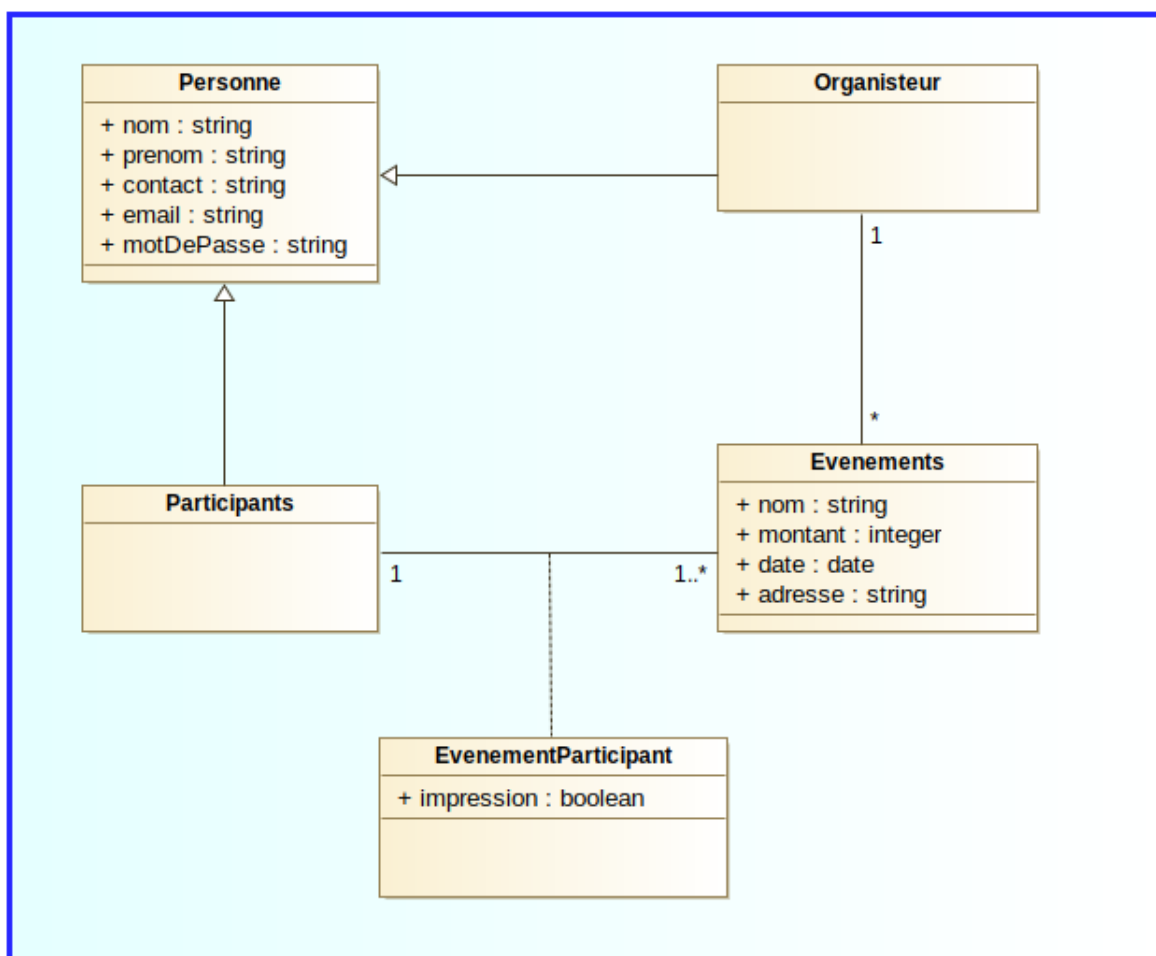
Pour des raisons de sécurité sur le serveur seul le dossier public est accessible. Cette configuration n'est pas toujours possible sur un serveur mutualisé (partagé). Il faut alors modifier un tout petit peu Laravel pour que ça fonctionne. Nous en parlerons peut être dans la partie déploiement.

3 Implémentation du modèle UML.

Laravel dans son fonctionnement implémente le pattern MVC (Modèle Vue Controller). Le tableau ci-dessus représente mieux cela.

MVC	LARAVEL
Model	Migration + Model
Vue	Vue
Controlleur	Route + Controller

Dans le cadre du cours et du projet que nous réaliserons, voici le modèle UML que nous utiliserons :



3.1 Les Migrations

Maintenant que l'application **G_event** a été créée nous allons maintenant créer une base de données appelée **gevent**. Une fois que c'est fait nous allons connecter notre application **LARAVEL** à la base de donnée **gevent**.

Pour le faire rendons nous au niveau du fichier **.env**. Il s'agit du fichier dans lequel nous configurons l'environnement de l'application **LARAVEL** (on peut y spécifier la base de données utilisée, le serveur de messagerie de notre choix, et bien d'autres choses).

Une fois que le fichier ouvert, modifiez les valeurs des constantes : **DB_CONNECTION**, **DB_HOST**, **DB_PORT**, **DB_DATABASE**, **DB_USERNAME**, **DB_PASSWORD** comme ceci :

```
DB_CONNECTION=pgsql
DB_HOST=127.0.0.1
DB_PORT=5432
DB_DATABASE=gschool
DB_USERNAME=votre_nom_utilisateur_postgres (Ex : Tamba)
DB_PASSWORD=votre_mot_de_passe_postgres (EX : 123456789)
```

Une fois que c'est fait lancer la commande **php artisan migrate** et relancer la commande **php artisan serve**.

NB : A chaque fois que vous modifier le fichier .env, redémarrez toujours le serveur LARAVEL.

Maintenant que notre application est connectée à une base de donnée, nous pouvons y créer des migrations.

Une migration est un pattern nous permettant de créer des tables dans une base de donnée tout en étant dans l'application : c'est une représentation en classe PHP des commandes d'un script SQL.

Pour créer une migration sur LARAVEL on utilise la commande suivante:

```
php artisan make:migration create_migration_names_table
```

NB:

- Le nom de la migration doit se terminer par un **s**. C'est une convention de LARAVEL. Le sens de cette convention est de dire que nous avons plusieurs objet dans une table.

Maintenant que vous connaissez la commande php qui vous permet de créer des migrations je vous propose de créer les migrations correspondant à notre modèle UML.

NB:

- Nous allons utiliser la migration users comme étant la migration du modèle UML personne. Le pourquoi sera expliqué par la suite.
- Lorsque nous avons une relation ***-*** entre deux modèles, cela est implémenté en LARAVEL par une migration intermédiaire (comme en **SQL**). Le nom de cette dernière est composé des noms des deux modèles avec un **s** à la fin de chacune d'elle. Encore une fois c'est la convention qui l'oblige.

Une fois que les migration créées voici leur contenu.

- users

```
<?php

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

```

return new class extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->bigIncrements('id_user');
            $table->string('nom');
            $table->string('prenom');
            $table->string('contact');
            $table->string('email')->unique();
            $table->timestamp('email_verified_at')->nullable();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('users');
    }
};

```

- organisateur

```

<?php

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

class CreateOrganisateursTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()

```

```

{
    Schema::create('organisateurs', function (Blueprint $table) {
        $table->bigIncrements('id_organisateur');
        $table->bigInteger("personne");
        $table->timestamps();
        $table->foreign("personne")->references("id_user")->on("users");

    });
}

/**
 * Reverse the migrations.
 *
 * @return void
 */
public function down()
{
    Schema::dropIfExists('organisateurs');
}
}

```

- evenements

```

<?php

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

return new class extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('evenements', function (Blueprint $table) {
            $table->bigIncrements("id_evenement");
            $table->string('nom');
            $table->float('montant');
            $table->date('date'); // date et heure
            $table->json('adresse');
            $table->bigInteger('organisateur');
            $table->timestamps();
            $table->foreign("organisateur")->references("id_user")->on("users");

        });
    }
}

```

```

/**
 * Reverse the migrations.
 *
 * @return void
 */
public function down()
{
    Schema::dropIfExists('evenements');
}
};

```

- participants

```

<?php

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

return new class extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('participants', function (Blueprint $table) {
            $table->bigIncrements('id_participant');
            $table->bigInteger("personne");
            $table->timestamps();
            $table->foreign("personne")->references("id_user")->on("user");
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('participants');
    }
};

```

- `evenements_participants`

```
<?php

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

return new class extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('evenements_participants', function (Blueprint $table) {
            $table->foreignId("id_evenement");
            $table->foreignId("id_participant");
            $table->enum("impression", array(0,1));
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('evenements_participants');
    }
};
```

Nous pouvez constater qu'une migration est une classe **PHP** qui hérite de la classe **Migration** et qu'elle contient deux méthodes : **up** et **down**.

- **up** est la méthode dans la quelle nous allons indiquer ce qui se passe lorsqu'on lance la migration.
- **down** est la méthode dans la quelle nous indiquons le directive à exécuter lorsque la migration est annulée.

3.1.1 documents de référence

- <https://laravel.com/docs/8.x/migrations>
- <https://laravel.com/docs/8.x/seeding>

3.2 Les modèles

Un modèle en LARAVEL implémente l'ORM (Object-Relational Mapper). L'ORM permet à un modèle de communiquer directement avec la base de donnée.

Pour créer un model en LARAVEL on utilise la commande suivante:

```
php artisan make:model NomModel
```

NB: Il n'y a pas de s à la fin du nom d'un modèle et le nom d'un modèle doit commencer par une lettre majuscule.

Maintenant que vous connaissez la commande PHP qui vous permet de créer un modèle je vous propose de créer les modèles correspondant à nos migrations.

Le modèle User existe déjà c'est pour cela nous l'avons pas créer. Vous vous demandez sûrement pourquoi nous n'avons pas créer de modèle pour la migration association ?

La réponse à cette que la méthode que nous allons utiliser ici au niveau du modèle pour régler ce problème ne nécessite pas de modèle association.

Voici le contenu de nos modèles.

- User

```
<?php

namespace App\Models;

// use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Laravel\Sanctum\HasApiTokens;

class User extends Authenticatable
{
    use HasApiTokens, HasFactory, Notifiable;

    /**
     * The attributes that are mass assignable.
     *
     * @var array<int, string>
     */
    protected $fillable = [
        'nom',
        'prenom',
        'contact',
        'email',
        'password',
    ];

    protected $primaryKey = 'id_user';
```

```

/**
 * The attributes that should be hidden for serialization.
 *
 * @var array<int, string>
 */
protected $hidden = [
    'password',
    'remember_token',
];

/**
 * The attributes that should be cast.
 *
 * @var array<string, string>
 */
protected $casts = [
    'email_verified_at' => 'datetime',
];

public function organisateur()
{
    return $this->hasOne(Organisateur::class, "personne");
}

public function participant()
{
    return $this->hasOne(Participant::class, "personne");
}
}

```

- Organisateur

```

<?php

namespace App\Models;

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

class Organisateur extends Model
{
    use HasFactory;
    protected $fillable = ['personne'];
    protected $primaryKey = 'id_organisateur';

    public function user()
    {

```

```

        return $this->belongsTo(User::class,"personne");
    }

    public function evenements()
    {
        return $this->hasMany(Eventement::class, "organisateur");
    }
}

```

- Participant

```

<?php

namespace App\Models;

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

class Participant extends Model
{
    use HasFactory;
    protected $fillable = ['personne'];
    protected $primaryKey = 'id_participant';

    public function user()
    {
        return $this->belongsTo(User::class,"personne");
    }

    public function evenements(){
        return $this->belongsToMany(Eventement::class, "evenements_participants",
            "id_evenement", "id_participant")
            ->withPivot(array("impression"));
    }
}

```

- Evenement

```

<?php

namespace App\Models;

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

class Evenement extends Model
{
    use HasFactory;

    protected $fillable = [

```



```

        'nom',
        'montant',
        'organisateur',
        'date',
        'adresse',
    ];

    protected $casts = [
        'adresse' => 'array'
    ];

    protected $dates = [
        'date',
    ];

    protected $primaryKey = 'id_evenement';

    public function organisateur()
    {
        return $this->belongsTo(Organisateur::class, "organisateur");
    }

    public function participants(){
        return $this->belongsToMany(Participant::class, "evenements_participants",
            "id_evenement", "id_participant")
            ->withPivot(array("impression"));
    }
}

```

Explication :

- La variable \$fillable permet d'indiquer les colonnes à accepter lors de la création de l'objet.
- La variable \$primaryKey permet d'indiquer la clé primaire de la relation.
- La relation 1 - 1 est implémenter avec **hasOne** et **belongsTo**
- La relation 1 - * est implémenter avec **hasMany** et **belongsTo**
- La relation * - * est implémenter avec **belongsToMany** et **belongsToMany**

3.2.1 documents de référence

- <https://laravel.com/docs/8.x/eloquent>

- <https://laravel.com/docs/8.x/eloquent-relationships>

3.3 Les Controllers

Un contrôleur est une classe PHP qui permet de relier plusieurs vues à un modèle. On peut énumérer sept (7) méthodes se trouvant dans un contrôleur à savoir :

index, create, store, show, edit, update, delete.

Utilisation et utilité de chaque méthodes.

Verb	URI	Action	Route Name	Signification
GET	/photos	index	photos.index	Affiche la liste des objets Photo
GET	/photos/create	create	photos.create	Affiche un formulaire d'enregistrement
POST	/photos	store	photos.store	enregistrement d'un objet Photo
GET	/photos/photo	show	photos.show	Affiche les détails d'un objet Photo
GET	/photos/photo/edit	edit	photos.edit	Affiche un formulaire d'édition
PUT/PATCH	/photos/photo	update	photos.update	Met à jour un objet Photo
DELETE	/photos/photo	destroy	photos.destroy	Supprime un objet Photo

Pour créer un contrôleur on utilise la commande suivante:

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

Je vous laisse donc créer les contrôleurs correspondant à nos modèle. C'est fais ? si telle n'est pas le cas voici ce qu'il fallait faire.

- EvenementController

```
php artisan make:controller EvenementController
```

- participantController

```
php artisan make:controller participantController
```

Maintenant comment accéder à ces méthodes via le navigateur ? Pour le faire il vas falloir comprendre la notion de route.

3.3.1 Documents de référence

- <https://laravel.com/docs/8.x/controllers>

3.4 Les Routes

Une route est le moyen par lequel un contrôleur peut communiquer avec une vue.

Pour créer une route on se rend dans le fichier **route/web.php** et on n'y ajoute une ligne sous la forme suivante.

```
route:methodeHttp(uri, methodeController)->name(nom);
```

Explication :

- **uri** est un suffixe qui sera ajouté à l'url de base (*http://127.0.0.1/uri*) pour former une nouvelle url menant à la méthode indiqué dans la route.
- La **methodeHttp** est la moyen par le quelle nous accédons à la méthode (get, post, put ...).
- Avec le nom de la route nous pouvons l'utiliser n'importe où dans un fichier **php**.

Exemple de route :

```
Route::get('/', function () {  
    return "Hello";  
});
```

```
Route::get('/', function () {  
    return view('welcome');  
});
```

```
Route::post('enregistrement_abonne',  
'App\Http\Controllers\AbonneController@store')->name('storeAbonne');
```

```
Route::get('lire_pdf/{ouvragesElectronique}/lecture',  
[LivreNumeriqueController::class, 'readPdf'])->name('lirePDF');
```

```
Route::put('mise_a_jour_des_abonnes/{abonne}', 'App\Http\Controllers\  
AbonneController@update')->name('updateAbonne');
```

3.4.1 Documents de référence

- <https://laravel.com/docs/8.x/routing>

3.5 Les vues

Sous Laravel, pour afficher correctement une page Web dans le navigateur, il faut utiliser une vue. C'est la vue qui est en charge de générer le code HTML. Elle utilisera pour cela, en plus des balises HTML, des directives et instructions que le moteur d'affichage **Blade** met à sa disposition.

Blade est un langage de template comme Jindja2. Essayons d'afficher la liste des événements. Pour le faire

je vous propose d'aller dans la documentation de laravel partie **Blade Templates** pour pouvoir réaliser cette page.

Maintenant que c'est fait nous allons passer à la réalisation de toute nos vues. Pour cela nous allons créer deux répertoires dans le dossier ressource/vue qui se nomme respectivement événements et participants.

Dans chaque répertoire nous allons créer quatre fichiers (index, create, edit, show) avec l'extension **.blade.php**. Comme l'indique la figure suivante.

Nous allons éditer ensemble les fichiers du dossier événements. Je vous laisse faire seuls l'édition des fichiers du répertoire participant.

- **create.blade.php** :

```
<html>
<head>
    <meta charset="utf-8">
    <title>Liste</title>
</head>
<body>
    <main>
        <h1>Formulaire d'enregistrement d'un événements.</h1>
        <form method="post" action="{{ route('enregistrementEvenement') }}">
            @csrf
            <div>
                <label>Nom</label>
                <input type="text" name="nom">
            </div>
            <div>
                <label>Montant</label>
                <input type="number" name="montant">
            </div>
            <div>
                <label>Date</label>
                <input type="datetime-local" name="date">
            </div>
            <div>
                <label>Adresse</label>
                <table>
                    <thead>
                        <th>Ville</th>
                        <th>Quartier</th>
                        <th>lieu</th>
                    </thead>
                    <tbody>
                        <tr>
                            <td>
                                <input type="text" name="ville">
                            </td>
                            <td>
                                <input type="text" name="quartier">
                            </td>
                            <td>
```

```

        <input type="text" name="lieu">
    </td>
</tr>
</tbody>
</table>
</div>
<div>
    <label>Organisateur</label>
    <select name="organisateur">
        <option value="">Sélectionner</option>
        @foreach($organisateurs as $organisateur)
            <option value="{{ $organisateur->user->id_user }}">
                {{ $organisateur->user->nom }}</option>
        @endforeach
    </select>
</div>
<input type="submit" value="enregister">
</form>
</main>
</body>
</html>

```

- edit.blade.php :

```

<html>
<head>
    <meta charset="utf-8">
    <title>Liste</title>
</head>
<body>
<main>
    <h1>Edition de l' événements {{ $evenement->id_evenement }}.</h1>
    <form method="post" action="{{ route('miseAJourEvenement', $evenement) }}">
        @csrf
        @method('put')
        <div>
            <label>Nom</label>
            <input type="text" name="nom"
                value="{{ $evenement->nom }}">
        </div>
        <div>
            <label>Montant</label>
            <input type="number" name="montant"
                value="{{ $evenement->montant }}">
        </div>
        <div>
            <label>Date</label>
            <input type="datetime-local" name="date"
                value="{{ $evenement->date }}">
        </div>
    </form>

```

```

<div>
  <label>Adresse</label>
  <table>
    <thead>
      <th>Ville</th>
      <th>Quartier</th>
      <th>lieu</th>
    </thead>
    <tbody>
      <tr>
        <td>
          <input type="text" name="ville"
            value="{{ $evenement->adresse["ville"] }}">
        </td>
        <td>
          <input type="text" name="quartier"
            value="{{ $evenement->adresse["quartier"] }}">
        </td>
        <td>
          <input type="text" name="lieu"
            value="{{ $evenement->adresse["lieu"] }}">
        </td>
      </tr>
    </tbody>
  </table>
</div>
<div>
  <label>Organisateur</label>
  <select name="organisateur">
    <option value="">Sélectionner</option>
    @foreach($organisateurs as $organisateur)
      <option value="{{ $organisateur->user->id_user }}"
        {{ $evenement->organisateur == $organisateur->user->id_user ?
          "selected" : "" }}>{{ $organisateur->user->nom }}</option>
    @endforeach
  </select>
</div>
<input type="submit" value="Mettre à jour">
</form>
</main>
</body>
</html>

```

C'est bien beau tous ça mais cela ne marchera pas. Pourquoi ? Car nous avons pas encore défini le corps des méthodes du contrôleur **EvenementController**.

Remplissez les comme ceci. Pour plus d'information allez voir la documentation (partie contrôler).

```

public function index()
{
    return view('evenement.index')->with(['evenements' => Evenement::all(),]);
}

```

```

public function create()
{
    $organisateurs = Organisateur::all();
    return view('evenement.create', compact('organisateurs'));
}

public function store(Request $request)
{
    $request->validate([
        'nom' => 'required',
        'montant' => 'required',
        'date' => 'required',
        'ville' => 'required',
        'quartier' => 'required',
        'lieu' => 'required',
        'organisateur' => 'required',
    ]);

    $request->adresse = array(
        'ville' => $request->ville,
        'quartier' => $request->quartier,
        'lieu' => $request->lieu,
    );

    $request->datetime = explode("T", $request->date);
    $request->date = $request->datetime[0];
    $request->time = $request->datetime[1];

    Evenement::create([
        'nom' => $request->nom,
        'montant' => $request->montant,
        'date' => Carbon::createFromFormat("Y-m-d H:i",
            $request->date." ".$request->time),
        'adresse' => $request->adresse,
        'organisateur' => User::all()->where('id_user', '=',
            $request->organisateur)->first()->id_user,
    ]);
}

public function edit(Evenement $evenement)
{
    $organisateurs = Organisateur::all();
    return view('evenement.edite',
        compact(['evenement', 'organisateurs']));
}

public function update(Request $request, Evenement $evenement)
{
    $request->validate([
        'nom' => 'required',
        'montant' => 'required',
        'date' => 'required',
    ]);
}

```

```

        'ville' => 'required',
        'quartier' => 'required',
        'lieu' => 'required',
        'organisateur' => 'required',
    ]);

    $request->adresse = array(
        'ville' => $request->ville,
        'quartier' => $request->quartier,
        'lieu' => $request->lieu,
    );
    $request->datetime = explode("T", $request->date);
    $request->date = $request->datetime[0];
    $request->time = $request->datetime[1];

    //dd(Carbon::createFromFormat("Y-m-d H:i",
    $request->date." ".$request->time));

    $evenement->nom = $request->nom;
    $evenement->montant = $request->montant;
    $evenement->adresse = $request->adresse;
    $evenement->date = Carbon::createFromFormat("Y-m-d H:i",
    $request->date." ".$request->time);
    $evenement->organisateur = $request->organisateur;
    $evenement->save();

    return redirect()->route('listeEvenements');
}

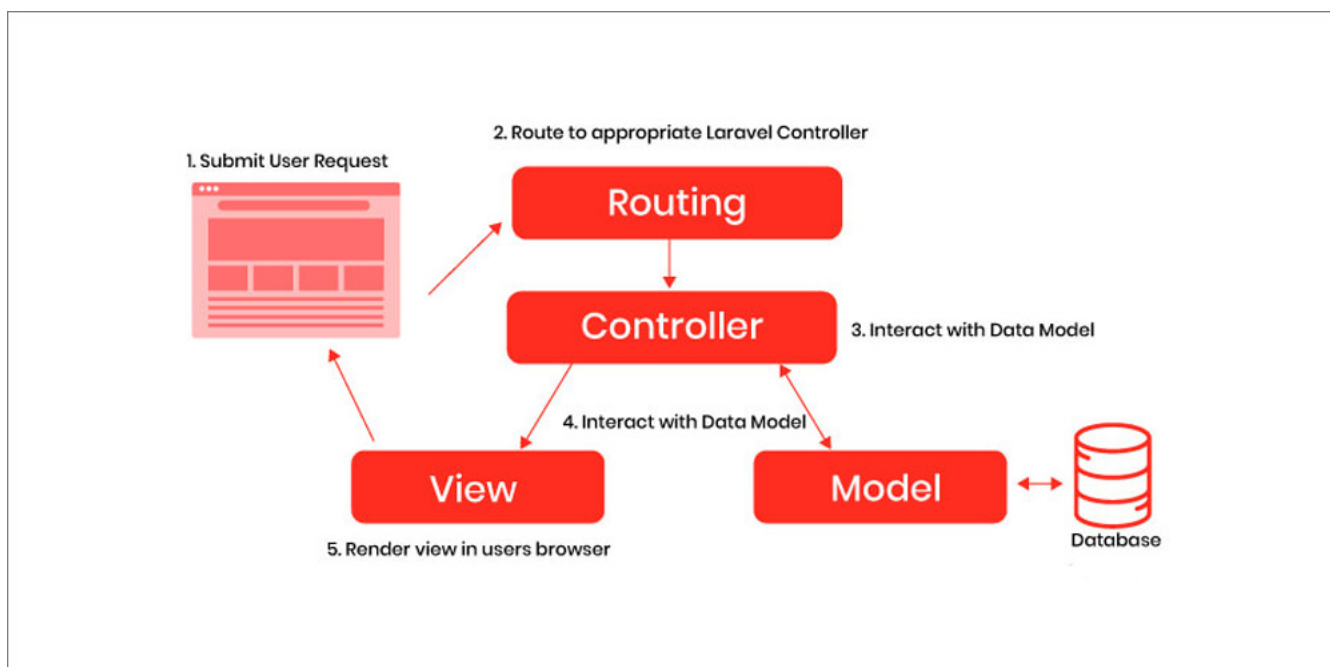
public function destroy(Evenement $evenement)
{
    $participants = $evenement->participants;
    foreach ($participants as $participant){
        $evenement->participants()
        ->detach($participant->id_participant);
    }
    $evenement->delete();
    return redirect()->route('listeEvenements');
}

```

3.5.1 Documents de référence

- <https://laravel.com/docs/8.x/validation>
- <https://laravel.com/docs/8.x/views>

Images récapitulatives:



4 Les Services Providers

4.1 Helpers

Les helpers sont des méthodes statiques permettant d'aider les vues. Ils sont accessibles n'importe où dans l'application laravel.

Vous pouvez remarquer que jusqu'à présent l'adresse de l'événement n'est pas précise. Nous souhaitons afficher l'adresse sous cette forme : "À "+ville+", "+quartier+" vers le "+lieu. Comment faire ? C'est justement le rôle d'un helper.

Pour pouvoir utiliser les helpers dans un projet laravel il va vous falloir installer le module package **helper**. La commande suivante permet de faire.

```
composer require mercuryseries/laravel-helpers
```

Une fois que c'est fait nous allons créer le répertoire **Helpers** dans le répertoire **app/**. Ensuite nous allons créer la classe **EvenementHelper** dans le répertoire précédemment créé. Cette classe aura une méthode statique **afficherAdresse**. Je vous laisse essayer de remplir ce fichier avant de continuer. C'est fait ? Voici ce qu'il faut faire.

```
<?php

namespace App\Helpers;

class EvenementHelper
{

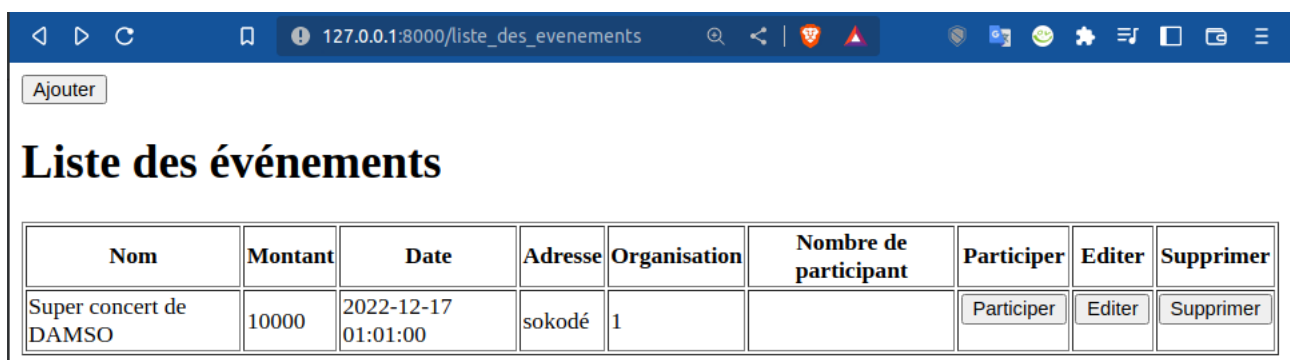
    public static function afficherAdresse(array $array)
    {
        return "À ".$array['ville'].", ".$array['quartier']. " vers le ".$array['lieu'];
    }
}
```

Comment l'utiliser dans nos vues ?

Pour l'utiliser au niveau des vues (dans notre cas dans la vue evenement/index) on importe l'helper comme ceci.

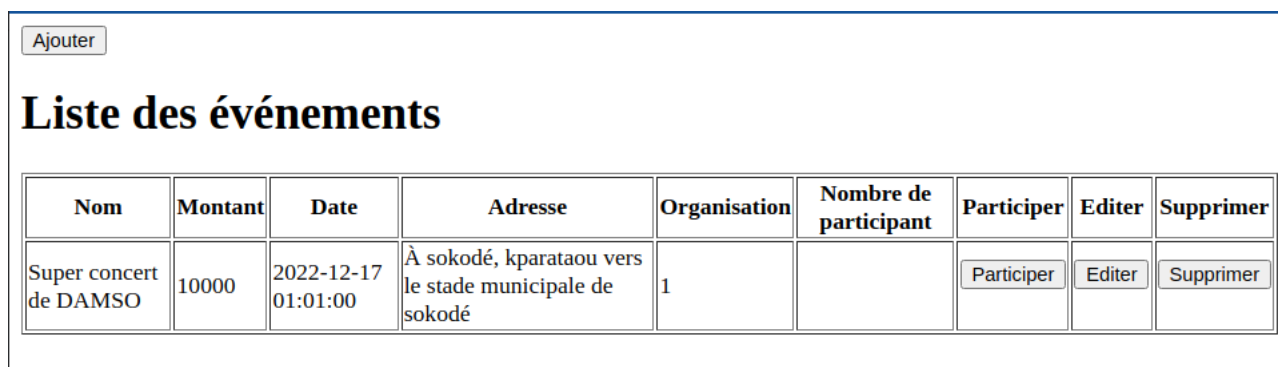
```
<td>{{ \App\Helpers\EvenementHelper::afficherAdresse($event->adresse) }}</td>
```

Avant l'helper :



Nom	Montant	Date	Adresse	Organisation	Nombre de participant	Participer	Editer	Supprimer
Super concert de DAMSO	10000	2022-12-17 01:01:00	sokodé	1		Participer	Editer	Supprimer

Après l'helper :



Nom	Montant	Date	Adresse	Organisation	Nombre de participant	Participer	Editer	Supprimer
Super concert de DAMSO	10000	2022-12-17 01:01:00	À sokodé, kparataou vers le stade municipale de sokodé	1		Participer	Editer	Supprimer

4.2 Service

Les services sont comme les helpers mais eux ils sont axés côté modèle.

Vous pouvez remarquer que pour valider les formulaires aux niveau des méthodes **store et update** du contrôleur **evenementController** on écrit les mêmes instructions. Et vous le savez bien dupliquer du code c'est mal. Les services sont là en partie pour nous aider à régler cela.

Pour utiliser un service il n'y a pas de module à installer, on crée seulement le répertoire **Service** dans **app/**. Une fois le répertoire créé, on crée une classe service et on y définit le plus souvent des méthodes static. Nous allons donc créer une classe php qui s'appelle **EvenementService** dans la quelle nous allons créer la méthode static qui prend la variable **\$request** en paramètre comme ceci.

```
<?php

namespace App\Service;

use Illuminate\Http\Request;

class EvenementService
{
    public static function validateFromRequest(Request $request)
    {
        $request->validate([
            'nom' => 'required',
            'montant' => 'required',
            'date' => 'required',
            'ville' => 'required',
            'quartier' => 'required',
            'lieu' => 'required',
            'organisateur' => 'required',
        ]);
    }
}
```

Comment l'utiliser dans nos contrôleurs ?

Pour l'utiliser au niveau des contrôleurs (dans notre cas dans le contrôleur **evenementController**) on utilise le service comme ceci.

```
public function store(Request $request)
{
    EvenementService::validateFromRequest($request);

    $request->adresse = array(
        'ville' => $request->ville,
        'quartier' => $request->quartier,
        'lieu' => $request->lieu,
    );

    $request->datetime = explode("T", $request->date);
    $request->date = $request->datetime[0];
    $request->time = $request->datetime[1];

    Evenement::create([
```

```

        'nom' => $request->nom,
        'montant' => $request->montant,
        'date'=> Carbon::createFromFormat("Y-m-d H:i", $request->date." ".$request->time),
        'adresse' => $request->adresse,
        'organisateur' => User::all()->where('id_user', '=', $request->organisateur)->first(),
    ]);

    return redirect()->route('listeEvenements');
}

.....
.....
.....

public function update(Request $request, Evenement $evenement)
{
    EvenementService::validateFromRequest($request);

    $request->adresse = array(
        'ville' => $request->ville,
        'quartier' => $request->quartier,
        'lieu' => $request->lieu,
    );
    $request->datetime = explode("T", $request->date);
    $request->date = $request->datetime[0];
    $request->time = $request->datetime[1];

    //dd(Carbon::createFromFormat("Y-m-d H:i", $request->date." ".$request->time));

    $evenement->nom = $request->nom;
    $evenement->montant = $request->montant;
    $evenement->adresse = $request->adresse;
    $evenement->date = Carbon::createFromFormat("Y-m-d H:i", $request->date." ".$request->time);
    $evenement->organisateur = $request->organisateur;
    $evenement->save();

    return redirect()->route('listeEvenements');
}

```

4.3 Authentification avec breez

Il existe deux systèmes d'authentification avec LARAVEL à savoir **UI** et **BREEZ**. Mais dans ce document nous allons voir **BREEZ**. Pour installer breez dans laravel voici comment s'y prendre.

Lancer les commandes suivante mais avant sauvegarder vos routes autre par que dans l'application LARAVEL car Breez les supprimera.

```
composer require laravel/breeze:1.9.2
php artisan breeze:install
```

```
npm install
npm run dev
```

```
php artisan migrate
```

Une fois breez installer, reporter vos route dans le fichier web.php. Si vous voulez qu'un utilisateur soit authentifié avant d'accéder au tableau de bord voici ce qu'il faut faire.

```
Route::group(['middleware' => ['auth']], function () {  
    Route::get('/dashboard', function () {  
        return view('dashboard');  
    }->name('dashboard');  
});
```

Preuve:

Maintenant vous allez vous demander sûrement comment gérer les rôles et les permissions ? Ne vous inquiétez pas. Pour gérer les rôles et les permissions nous allons utiliser LARAVEL spatie. Pour l'installer lancer la commande suivante.

```
composer require spatie/laravel-permission
```

Après l'installation ajouter dans le fichier config/app.php la ligne suivante :

```
Spatie\Permission\PermissionServiceProvider::class,
```

Maintenant il faut publier le service provider comme ceci:

```
php artisan vendor:publish --provider="Spatie\Permission\PermissionServiceProvider"
```

Enfin nettoyez le cache et lancer à nouveau les migrations.

```
php artisan config:clear  
php artisan migrate
```

A cette étape on peut commencer à utiliser spatie mais nous allons ajouter trois lignes dans la variable *routeMiddleware* comme ceci.

```
protected $routeMiddleware = [  
    .....  
    .....  
    .....  
    'role' => \Spatie\Permission\Middlewares\RoleMiddleware::class,  
    'permission' => \Spatie\Permission\Middlewares\PermissionMiddleware::class,  
    'role_or_permission' => \Spatie\Permission\Middlewares\RoleOrPermissionMiddleware::class,  
];
```

Cela nous permettra d'avoir accès aux variables **role**, **permission** et **role_or_permission** dans le fichier **web.php**.

Testons maintenant spatie. Nous allons créer:

- Deux rôles : public et organisateur.
- Deux Utilisateurs : l'un ayant le rôle public et l'autre organisateur.
- L'organisateur pourra voir le nombre de participant et mais le public non.
- De plus l'organisateur pourra supprimer un événement mais le public non.
- Création de deux rôles :

4.3.1 Documents de référence

- <https://laravel.com/docs/8.x/starter-kits#laravel-breeze>
- <https://spatie.be/docs/laravel-permission/v5/installation-laravel>

4.4 Localisation

En LARAVEL quant on parle de localisation il faut penser langue (internationalisation : i18n de l'application). Alors comment adapter notre site en fonction de la langue de l'utilisateur ? Pour le faire, il nous faut installer le package langue de LARAVEL comme suit :

```
composer require bestmomo/laravel5-artisan-language --dev
```

Une fois l'installation terminer, ajouter la ligne suivante dans le fichier `/config/app.php`

```
Bestmomo\ArtisanLanguage\ArtisanLanguageProvider::class,
```

Maintenant nous allons créer un fichier json pour la traduction en anglais de notre application. Lancer la commande suivante.

```
php artisan language:make en
```

Le fichier **en.json** se créer au niveau du répertoire **resources/lang**.

Ajouter la ligne suivante pour changer le texte "Liste des événements" (fr) en "Events list"(en) lorsque la langue change.

```
"Liste des événements" : "Events list"
```

Mais cela ne suffis pas il va falloir écrire tous les textes que nous souhaitons traduire en blade comme ceci.

```
<h1>{{ __('Liste des événements') }}</h1>
```

Et voila le résultat.

Si nous souhaitons retourner en français nous allons modifier la valeur la variable **locale** en fr comme ceci

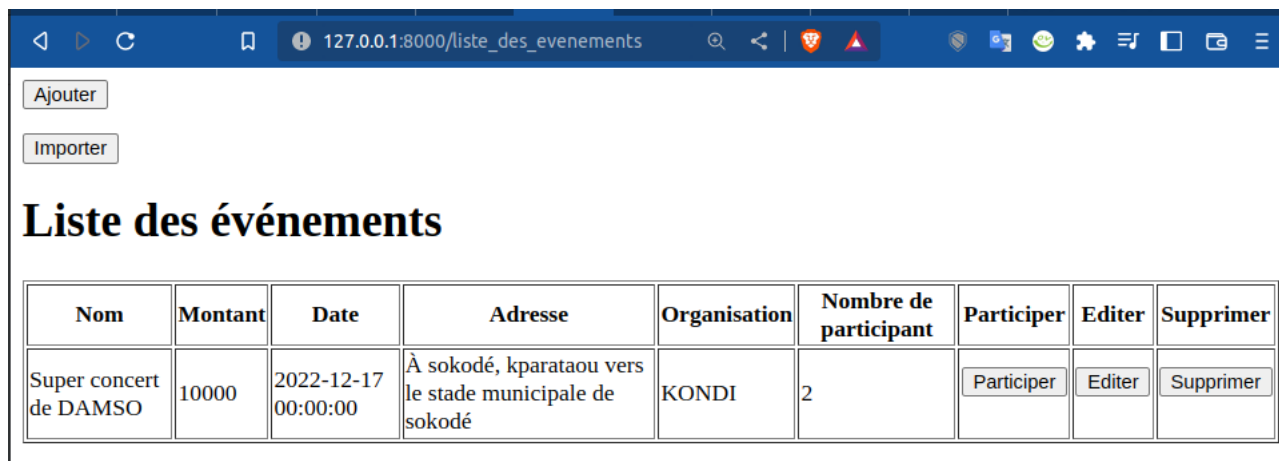
```
'locale' => 'fr'
```

ou utiliser la méthode de setLocale de la class App comme cela

```
App::setLocale('fr')
```

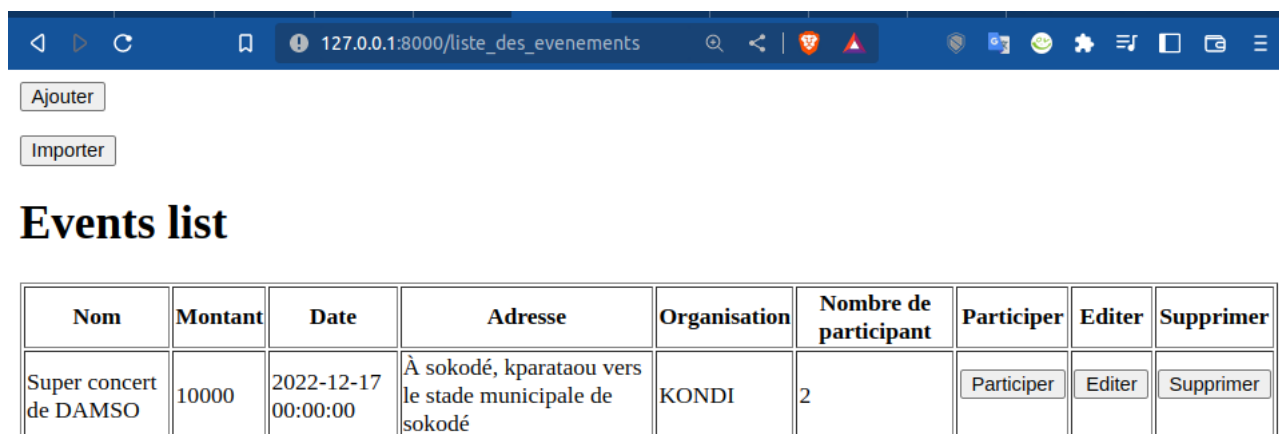
Et voila le résultat.

FR :



Nom	Montant	Date	Adresse	Organisation	Nombre de participant	Participer	Editer	Supprimer
Super concert de DAMSO	10000	2022-12-17 00:00:00	À sokodé, kparataou vers le stade municipale de sokodé	KONDI	2	Participer	Editer	Supprimer

EN :



Nom	Montant	Date	Adresse	Organisation	Nombre de participant	Participer	Editer	Supprimer
Super concert de DAMSO	10000	2022-12-17 00:00:00	À sokodé, kparataou vers le stade municipale de sokodé	KONDI	2	Participer	Editer	Supprimer

4.5 DOM PDF

Il existe plusieurs façon de générer un fichier pdf avec **PHP LARAVEL**. La méthode que nous allons voir ici est l'utilisation du Service **DOM PDF**. Pour l'installer voici ce qu'il faut faire.

```
composer require barryvdh/laravel-dompdf
```

Maintenant que le service est installer, essayons de générer un pdf lorsqu'on clique sur le bouton participer. Pour le faire, nous allons modifier le fichier **evenement/index** comme ceci.

```
<td>{{ $event->organisateur }}</td>
<td></td>
<td>
<form method="post" action="{{ route('generatePdf', $event) }}">
@csrf
<input type="submit" value="Participer">
</form>
</td>
<td>
<form method="get" action="{{route('formulaireEditerEvenement', $event)}}">
@csrf
<input type="submit" value="Editer">
</form>
</td>
```

Nous allons aussi créer la route **generatePdf**

```
Route::post('genarate_pdf/{evenement}', [\App\Http\Controllers\EvenementController::class,
'genererPDF'])->name('generatePdf');
```

Ensuite nous allons créer la méthode generatePDF dans le contrôleur EvenementController.

```
public function genererPDF(Evenement $evenement){

    //dd($evenement->organisateur());
    $pdf = Pdf::loadView('evenement.ticket', array(
        "evenement" => $evenement
    ));

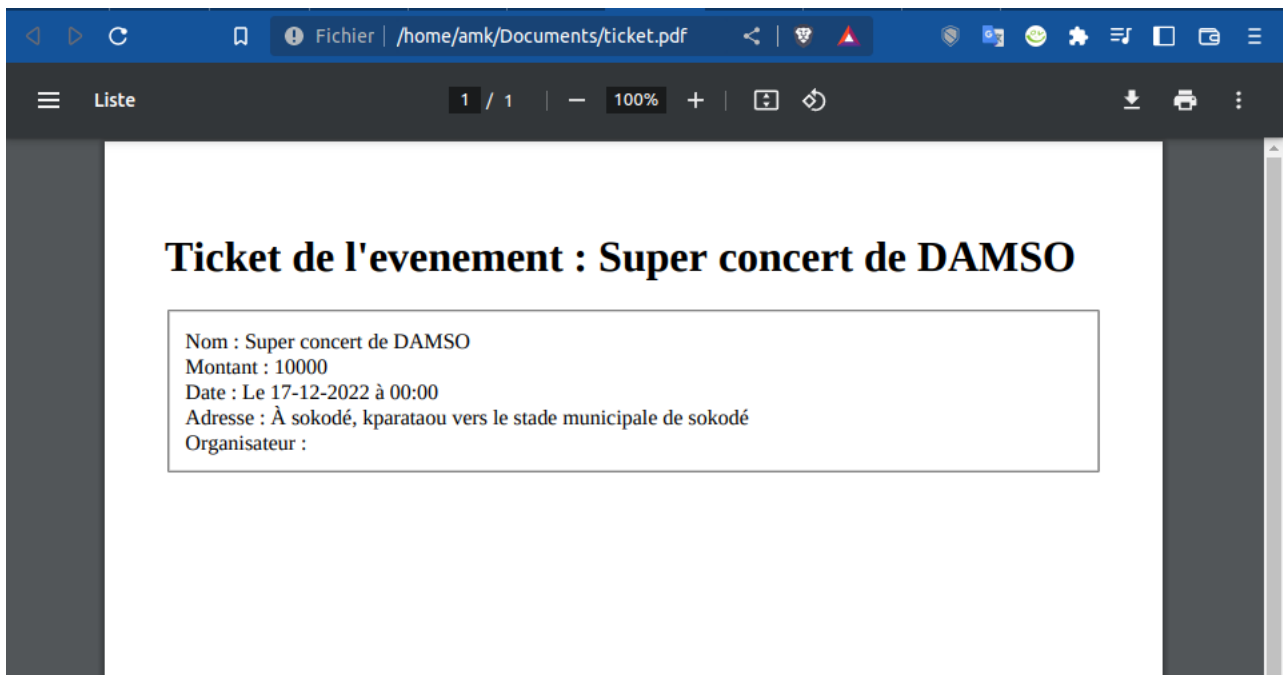
    return $pdf->download('ticket.pdf');
}
```

La méthode **genererPDF** nous permet de télécharger le pdf.

Enfin on créer une vue template **evenement/ticker.blade.php** comme ça.

```
<html>
<head>
    <meta charset="utf-8">
    <title>Liste</title>
</head>
<body>
    <main>
        <h1>Ticket de l'événement : {{ $evenement->nom }}</h1>
        <fieldset>
            <div>
                <label>Nom : </label><label>{{ $evenement->nom }}</label>
            </div>
            <div>
                <label>Montant : </label><label>{{ $evenement->montant }}</label>
            </div>
            <div>
                <label>Date : </label><label>{{ \App\Helpers
                \EvenementHelper::afficherDateEtDate($evenement->date) }}</label>
            </div>
            <div>
                <label>Adresse : </label><label>{{ \App\Helpers
                \EvenementHelper::afficherAdresse($evenement->adresse) }}</label>
            </div>
            <div>
                <label>Organisateur : </label><label>{{-- $evenement->organisateur->nom." ".
                $evenement->organisateur->prenom --}}</label>
            </div>
        </fieldset>
    </main>
</body>
</html>
```

Et voilà le résultat.



4.6 Import et Export Excel

Pour implémenter l'import en des données depuis un fichier Excel vers notre base de données on utilise le module . Voici comment on l'installe.

```
composer require maatwebsite/excel
php artisan vendor:publish --provider="Maatwebsite\Excel\ExcelServiceProvider" --tag=config
```

Si nous voulons importer une liste d'événements, on crée un modèle import correspondant à notre modèle Evenement comme ceci.

```
php artisan make:import EvenementImport --model=Evenement
```

Voici le contenu par défaut de la classe EvenementImport.

```
<?php

namespace App\Imports;

use App\Models\Evenement;
use Maatwebsite\Excel\Concerns\ToModel;

class EvenementImport implements ToModel
{
    /**
     * @param array $row
     *
     * @return \Illuminate\Database\Eloquent\Model|null
     */
    public function model(array $row)
    {
        return new Evenement([
            //
```

```

    });
}
}

```

Voici le format de notre fichier Excel.

	A	B	C	D	E	F
1	NOM	MONTANT	DATE	ADRESSE	ORGANISATEUR	
2	asynconf	25000	13/12/2022 18:00:00	sokodé,komah2,ceg komah	kondi	
3	king bala	13000	13/12/2022 20:00:00	sokodé,didaoure,stade municipale	kondi	
4						
5						
6						

Changer le contenu de la méthode model par ce qui suit.

```
<?php
```

```
namespace App\Imports;
```

```
use App\Models\Evenement;
```

```
use App\Models\User;
```

```
use Carbon\Carbon;
```

```
use Maatwebsite\Excel\Concerns\ToModel;
```

```
class EvenementImport implements ToModel
```

```
{
```

```
/**
```

```
 * @param array $row
```

```
 *
```

```
 * @return \Illuminate\Database\Eloquent\Model|null
```

```
 */
```

```
public function model(array $row)
```

```
{
```

```
    if (! $row ?? "" && $row[0]==null){
```

```
        return null;
```

```
    }
```

```
    if ($row[0]=="NOM"){
```

```
        return null;
```

```
    }
```

```
    $adresse = explode(",", $row[3]);
```

```
    return new Evenement([
```

```
        'nom'=>strtoupper($row[0]),
```

```
        'montant'=>$row[1],
```

```
        'date'=>Carbon::createFromFormat("d/m/Y H:i:s", explode(",", $row[2])[1]),
```

```
        'adresse' => array(
```

```
            'ville' => $adresse[0],
```

```
            'quartier' => $adresse[1],
```

```
            'lieu' => $adresse[2],
```

```
        ),
```

```
        'organisateur' => User::all()->where('nom', strtoupper($row[4]))->first()->id_us
```

```
    ]);
```

```
}
}
```

Maintenant que c'est fait, il nous faut créer deux routes une pour télécharger le fichier excel et l'autre pour charger le contenu du fichier en base de données comme ceci.

```
Route::get('formulaire_import_excel', [\App\Http\Controllers\EvenementController::class, 'importExcel']);
Route::post('import_excel', [\App\Http\Controllers\EvenementController::class, 'importEventStore']);
```

Évidemment nous devons créer les deux méthodes : `importEvent` et `importEventStore`. Voici leur contenu.

```
public function importEvent()
{
    return view('evenement.importExcel');
}

public function importEventStore(Request $request)
{
    $file = $request->file('fichierExcel'); // récupération le fichier
    $path = "fichier." . $file->extension(); // création d'un nom
    $file->storeAs("public/", $path); // stockage du fichier
    Excel::import(new EvenementImport(), "public/" . $path); // importer les données du fichier
    return redirect()->route('listeEvenements') //redirection;
}
```

- La méthode **importEvent** est classique elle retourne le formulaire d'import excel.
- La deuxième méthode **importEventStore** est quant à elle nous permet de stocker puis d'enregistrer les données du fichier excel dans la base. Mais pour que cela marche il faut créer un lien symbolique vers le dossier storage avec la commande suivante.

```
php artisan storage:link
```

Il nous faut aussi définir l'attribut **enctype** de la balise `form` **multipart/form-data** en comme ceci.

```
<html>
<head>
    <meta charset="utf-8">
    <title>Liste</title>
</head>
<body>
<main>
    <h1>Formulaire d'enregistrement d'un événements.</h1>
    <form method="post" action="{{ route('importExcel') }}" enctype="multipart/form-data">
        @csrf
        <div>
            <input type="file" value="" name="fichierExcel">
        </div>
        <input type="submit" value="enregister">
    </form>
</main>
</body>
</html>
```