

Création d'un gestionnaire de membres de club sportif

Les données

Nous allons commencer la partie M du modèle **MVC** en voyant les **migrations**, les **factories** et les **seedings**...

Grâce à Laravel, aucune raison cependant de mettre les mains dans phpmyadmin ou dans des lignes de code SQL pour créer et peupler ses tables : ce sont les rôles des migrations et des seeders.

- ➔ Les migrations sont des représentations, des schémas des tables en base de données.
- ➔ Les seeders nous permettent de peupler notre base de données avec des données fictives. Cela nous permet de nous éviter d'inventer des faux utilisateurs à la main pour nos tests...

Une fois les migrations créées, on n'aura plus qu'à utiliser artisan pour créer nos tables en base de données automatiquement.

Les migrations

Les migrations se trouvent dans  database > migrations


Laravel propose déjà 3 migrations qui correspondent à des tables dont on a très souvent besoin dans chaque application :

- Les **utilisateurs** : Laravel peut créer pour nous une table "users" qui contiendra nos utilisateurs. Nous la modifierons pour parfaire nos besoins
- Les **mots de passe oubliés** : Laravel peut créer pour nous une table "password_resets" qui enregistrera les demandes de mots de passe oubliés
- Les **jobs** : Laravel peut créer pour nous une table "jobs" (nous ne l'utiliserons pas). Elle sert dans le cas où nous souhaitons réaliser de manière différée des opérations coûteuses en temps.
Super pratique dans le cas d'envoi d'emails de masse par exemple ou de traitement d'image : les jobs vont permettre de gérer ces tâches sans bloquer l'utilisateur

Les attributs de l'entité que nous allons gérer : les utilisateurs

id : l'identifiant unique de l'utilisateur name : le nom et prénom de l'utilisateur email : l'email de l'utilisateur password : le mot de passe de l'utilisateur

⚠ Dans des projets plus avancés, il est essentiel de réaliser **une analyse préalable** grâce à des outils de modélisation comme UML pour concevoir les classes ou MERISE (Français) pour concevoir des bases de données.

Nous allons débiter par la modification de la migrations associées aux  **users**, qui actuellement, contient ce code :

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

class CreateUsersTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $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');
    }
}
```

👁 On peut remarquer deux méthodes : `up()` et `down()`. La première sert à dire quoi faire pour créer la table et l'autre pour la détruire.

C'est à dire que c'est dans la méthode `up()` que l'on va indiquer les colonnes (ou champs) que l'on va créer dans notre table "users".

La classe **Schema** nous offre les méthodes permettant de créer les colonnes de notre table avec des fonctions comme `string`, `boolean`, `enum`, `text`... qui représentent les types en base de données¹.

Dans notre table `users`, on aura donc :

- Un champ `id`, de type `bigInteger`, auto-incrémenté. C'est une clé primaire
- Un champ `name`, `varchar`
- Un champ `email`, `varchar`, `unique`
- Un champ `password`, `varchar`

- `rememberToken()` crée un `varchar nullable` `remember_token` qui stockera un identifiant de connexion

¹ <https://laravel.com/docs/8.x/migrations#available-column-types>

- timestamp('email_verified_at') crée un champ de type timestamp qui stockera la date à laquelle l'adresse email a été vérifiée (ce n'est pas obligatoire, c'est juste par défaut et disponible).

Il y a aussi la méthode timestamps(), qui crée automatiquement deux champs : created_at et updated_at.

RQ : Si nous avons besoin d'une nouvelle table, nous utiliserions la commande :

`php artisan make:migration create_nomdelatable_table`

Cette commande crée le fichier de migration dans `database > migrations`

et contient déjà la base de notre fichier : les "use" utiles ainsi que la déclaration de la classe et les méthodes `up` et `down`.

Il ne nous resterait plus qu'à compléter la méthode `up()` qui est l'équivalent grossier d'un `CREATE TABLE AS` en SQL.

Nous n'avons plus qu'à demander à artisan de créer les tables dans notre base de données. Pour ça, rien de plus simple : `php artisan migrate`

👍 C'est gagné ? Tu peux aller directement regarder dans la base de données qui a été créé si tu es curieux ;)

Arthur, l'apprenti développeur : Mince, j'ai une erreur Illuminate\Database\QueryException SQLSTATE[HY000] [2002] Connection refused...

Cela signifie que les données de connexion que tu as renseignées dans le `.env` ne sont pas correctes, il faut que tu les vérifies ;)

Il se peut également qu'une erreur apparaisse si tu utilises une version de **MySQL < 5.7.7** ou **MariaDB < 10.2.2**. En effet, Laravel utilise par défaut l'encodage `utf8mb4`. Pour corriger ça, rendez-vous dans `le fichier app > providers > AppServiceProvider.php`,

Rajouter : `use Illuminate\Support\Facades\Schema;`

puis dans la méthode `boot()` ajouter



`Schema::defaultStringLength(191);`


⚠ il ne faut pas que le serveur artisan soit lancé ! vous risqueriez d'avoir une erreur ! (pour stopper : touches `ctrl+c`)

Les factories et les seedings

Passons maintenant au peuplement de notre base de données.

On peut générer des données fictives pour tester notre application sans s'embêter grâce aux factories et aux seedings.

Comme les migrations, nous allons aller dans le dossier  database pour cette partie. Puis, dans  factories.

Un premier fichier est déjà là pour les  Users, regardons son contenu.

```
<?php


namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;
use Illuminate\Support\Str;

class UserFactory extends Factory
{
    /**
     * Define the model's default state.
     *
     * @return array
     */
    public function definition()
    {
        return [
            'name' => $this->faker->name(),
            'email' => $this->faker->unique()->safeEmail(),
            'email_verified_at' => now(),
            'password' => '$2y$10$92IXUNpkjO0rQQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi', // password
            'remember_token' => Str::random(10),
        ];
    }

    /**
     * Indicate that the model's email address should be unverified.
     *
     * @return \Illuminate\Database\Eloquent\Factories\Factory
     */
    public function unverified()
    {
        return $this->state(function (array $attributes) {
            return [
                'email_verified_at' => null,
            ];
        });
    }
}
```

Ce fichier, comme son nom l'indique, est un Factory. Il permet de créer un utilisateur bidon en base de données grâce à **Faker** : <https://github.com/fzaninotto/Faker>

 attention, depuis le 21 octobre 2020, le créateur a arrêté son développement. Cela n'empêche cependant pas son bon fonctionnement.

➔ Il faudra analyser les fichiers disponibles sous Laravel 9 !

Laravel intègre par défaut une entité User prête à l'emploi comme nous l'avons dit précédemment.

Ensuite, on a une **méthode** « définition » qui permet d'indiquer quelles valeurs renseignées aux colonnes de notre table users.

Faker permet de générer des string aléatoires, des nombres aléatoires...

🔑 Nous avons cependant besoin d'au moins un utilisateur qui sera administrateur.

⚠ Dans notre base de données, aucune colonne ne nous permet de savoir si quelqu'un est administrateur.

Comment rajouter/modifier/supprimer une colonne quand on a déjà réalisé les migrations

Il faut créer une nouvelle migration.

⚠ Ne surtout pas modifier l'ancienne car elle ne sera pas prise en compte si on fait `php artisan migrate`, à moins de tout supprimer et de tout recommencer...

La solution : **`php artisan make:migration add_admin_column_to_users_table`**.

```
PS G:\01-BTS-SIO\02-MODULES\Bloc1-an2-S1\GestionClub\GestionClub> php artisan make:migration add_admin_column_to_users_table
Created Migration: 2022_10_11_180244_add_admin_column_to_users_table
PS G:\01-BTS-SIO\02-MODULES\Bloc1-an2-S1\GestionClub\GestionClub>
```

Son contenu sera le suivant :

Rajout de la ligne `$table->boolean('admin')->default(false);`

```
<?php


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

class AddAdminColumnToUsersTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::table('users', function (Blueprint $table) {
            $table->boolean('admin')->default(false);
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::table('users', function (Blueprint $table) {
            //
        });
    }
}
```

Il suffit de faire **`php artisan migrate`** une nouvelle fois et le tour est joué ;)

Modification du UserFactory

Maintenant nous pouvons modifier notre fichier  UserFactory pour intégrer le fait que nous voulons un utilisateur administrateur

⚠ En général, les factories sont utiles pour générer **plusieurs insertions en base de données**, pas pour un seul utilisateur. Cette utilisation convient simplement à ce TP...

Rajouter : `'admin' => true,`

```
public function definition()
{
    return [
        'name' => $this->faker->name(),
        'email' => $this->faker->unique()->safeEmail(),
        'email_verified_at' => now(),
        'admin' => true,
        'password' =>
'$2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi', // password
        'remember_token' => Str::random(10),
    ];
}
```

Remarque : il faut stocker, pour des raisons évidentes de sécurité, les mots de passe de manière chiffrée ou hashée.

Le password renseigné ici est le résultat du mot "password" hashé avec la méthode de PHP "password_hash" et l'algorithme BCRYPT.

Du coup, quand on voudra essayer de se connecter, n'oublions pas que le mot de passe est tout simplement "password"

Le seeding

Le seeding nous servira simplement à faire tourner notre UserFactory pour **insérer en base de données**.

Rendez-vous dans  seeders pour trouver  DatabaseSeeder et y insérer ce code :

```
public function run()
{
    // \App\Models\User::factory(10)->create();
    \App\Models\User::factory(1)->create();
}
```

Et enfin, dans notre terminal :

php artisan db:seed

Tu peux regarder le contenu de ta base de données pour vérifier qu'un utilisateur a bien été ajouté.

Le modèle User

Depuis Laravel 8, les modèles sont stockés dans le dossier  app -> Models.

Il existe de base un modèle prédéfini « User ».

Regardons son contenu...

```
<?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 = [
        'name',
        'email',
        'password',
    ];

    /**
     * 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',
    ];
}
```

1/ Dans un premier temps, on définit le **namespace**².

Ensuite, on va utiliser **plusieurs classes prédéfinies par Laravel**.

Celles-ci servent à l'authentification, l'utilisation des factories ou les notifications (une fonctionnalité propre à Laravel).

Hormis HasFactory, que nous pourrions utiliser sur nos autres futurs modèles si nous avons renseigné d'autres factories, les classes utilisées supplémentaires sont propres au modèle User.

² Cf. **NAMESPACE** en fin de document

2/ Nous voyons ensuite ce premier attribut protégé nommé **fillable**. Il est extrêmement important.

```
protected $fillable = [
    'name',
    'email',
    'password',
];
```

Cet attribut est lié à la sécurité de notre application.

Dans ce tableau, nous indiquons les colonnes de la table qui peuvent être modifiées lors d'une insertion ou un update en base après soumission d'un formulaire.

Par exemple, si nous créons un nouvel utilisateur, il faut se poser la question "Quelles colonnes vais-je modifier ? "

La réponse, dans le cas de notre projet à ce stade, est la suivante :

- Son nom
- Son email
- Son mot de passe

Mais par contre, on ne va jamais modifier le booléen "admin", le token d'authentification... Autrement dit, un utilisateur lambda ne doit pas pouvoir modifier les colonnes autres que name, email et password lorsqu'il soumet un formulaire.

Cela permet de se protéger dans le cas où une personne malicieuse modifierait les informations soumises à un formulaire en rajoutant par exemple un champ "admin" qu'il passerait à true.

Car dans Laravel, pour insérer des données dans une base de données, on peut simplement demander :

"Fais une insertion dans la table users avec tous les paramètres de la requête POST". Il faut donc pouvoir trier les paramètres souhaités et ceux non souhaités. Fillable sert à ça...

3/ À l'inverse de fillable qui sert lors de requêtes d'insertion ou d'update en base de données, \$hidden va servir lors d'un select.

Ce sont ici les colonnes que l'on ne souhaite pas récupérer lorsqu'on fait un select * from users;

```
protected $hidden = [
    'password',
    'remember_token',
];
```


4/ Nous voyons ensuite un attribut nommé \$casts. Cet attribut sert lors de conversions de type, notamment pour les dates, de sorte que nous puissions utiliser Carbon (une classe de gestion des dates) par exemple.

```
protected $casts = [
    'email_verified_at' => 'datetime',
];
```


L'authentification

Laravel 8 vient avec un système tout prêt pour l'authentification. Quelques commandes et nous aurons une application rodée pour l'authentification.

Il faut savoir que depuis Laravel 8, le framework propose 3 "starter kits" pour l'authentification :

- [Laravel Breeze](#) 
- [Laravel Jetstream](#) 
- [Laravel Fortify](#) 


Voyons les différences entre chaque :

Laravel Jetstream est un pack très complet intégrant déjà un design travaillé basé sur des technologies puissantes et très récentes (Inertia + VueJS, Livewire + Blade).

Laravel Fortify quant à lui est plus minimaliste, il se contente de nous donner les derrières de l'authentification mais aucune vue, route etc... C'est vraiment le moteur d'une voiture sans châssis, volant, roues...

Laravel Breeze est le kit conseillé pour les débutants, très simple, il donne tout ce qu'il faut pour gérer automatiquement :

- L'inscription
- La connexion
- La déconnexion
- L'oubli de mot de passe

Quelques considérations à lire sur l'utilisation des packages 🤔 😊 :
<https://laravel.sillo.org/cours-laravel-8-la-securite-lauthentification/> 

Breeze reposent sur un « framework UI » autre que Bootstrap celui-ci se nomme **Tailwind**. L'approche de Laravel est moderne et Tailwind repose sur un concept similaire à Bootstrap, mais en se focalisant plus sur un système de class css que l'on va appeler « utilitaire ».

Les deux sont difficilement compatibles et maintenables mais peuvent fonctionner ensemble !

Pour les curieux : <https://developpeur-freelance.io/blog/bootstrap-tailwind/>

L'enjeu de cette partie du TD est de faire cohabiter les deux³ pour nos tests 😊

³ <https://dev.to/tanzimibthesam/how-to-install-both-tailwind-and-bootstrap-5-in-a-laravel-project-393h>

L'installation de Laravel Breeze

Modification de la configuration Bootstrap

Allez dans le dossier  resources/scss/ et créez le fichier  **bootstrap.scss**

```
@import "~bootstrap/scss/bootstrap";
```

Dans le fichier  **app.scss**

```
/* Importation de Font Awesome */
@import "~@fortawesome/fontawesome-free/scss/fontawesome";

// Fonts
@import url('https://fonts.googleapis.com/css?family=Nunito');

/* Importation des styles d'icônes */
@import "~@fortawesome/fontawesome-free/scss/solid";
@import "~@fortawesome/fontawesome-free/scss/regular";
@import "~@fortawesome/fontawesome-free/scss/brands";
```

Dans le fichier  **Webpack.mix.js**

```
mix.js('resources/js/app.js', 'public/js')
    .postCss('resources/css/app.css', 'public/css', [
        // ])
    .sass('resources/scss/bootstrap.scss', 'public/css')
    .sass('resources/scss/app.scss', 'public/css')
mix.version()
.sourceMaps();
```

npm run dev

Tailwind

Tailwind sera installé avec Laravel Breeze par défaut, nous en modifierons la configuration ultérieurement pour qu'elle fonctionne avec Bootstrap.

Laravel Breeze

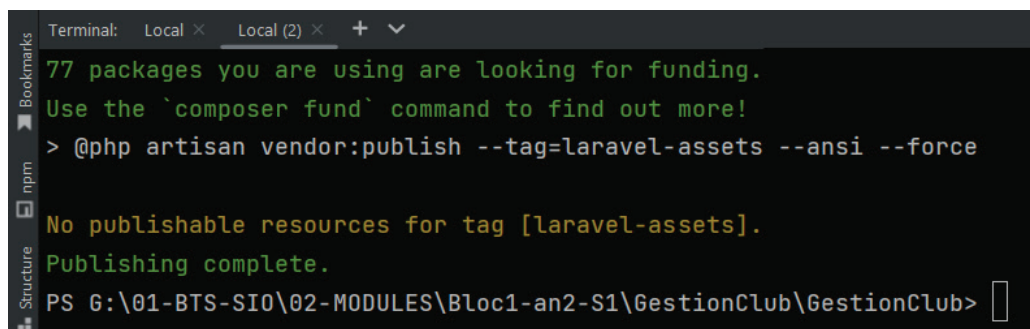
Laravel Breeze donne des fonctionnalités backend pour gérer l'authentification et aussi des vues. Ainsi, nous aurons besoin d'utiliser nos 2 gestionnaires de paquets backend et frontend favoris : composer et npm.

Premièrement, installons la partie back de Laravel Breeze...

⚠ avec cette version de Laravel, il ne faut pas installer le dernier package mais un plus ancien !

⚙ Place-toi à la racine du projet dans ton terminal et tape :

composer require laravel/breeze v1.9.4 --dev



```
Terminal: Local × Local (2) × + ▾
77 packages you are using are looking for funding.
Use the `composer fund` command to find out more!
> @php artisan vendor:publish --tag=laravel-assets --ansi --force
No publishable resources for tag [laravel-assets].
Publishing complete.
PS G:\01-BTS-SIO\02-MODULES\Bloc1-an2-S1\GestionClub\GestionClub> |
```

Puis : `php artisan breeze:install`

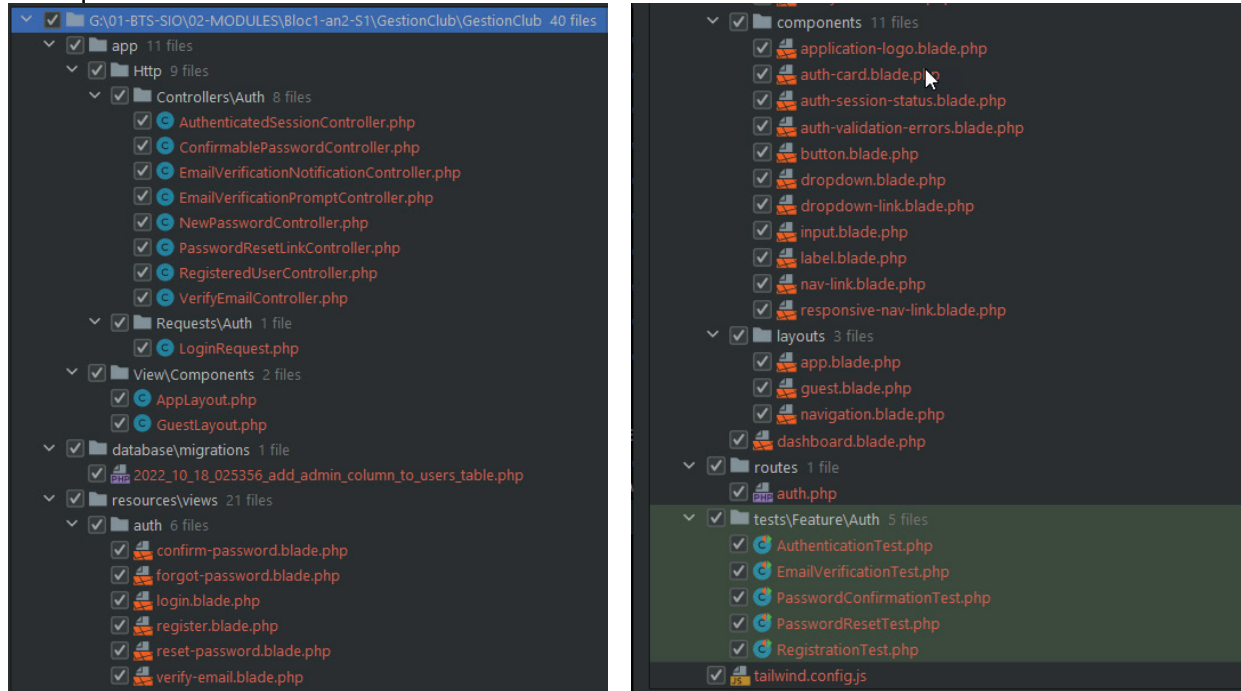
Vous obtenez le message :

« Please execute the "npm install" && "npm run dev" commands to build your assets. »

Utilisons donc npm pour installer les dépendances (⚠ nous compilerons nos assets plus tard)

`npm install`

Une quarantaine de fichiers sont installés :



Breeze insère la route nommée « dashboard » (tableau de bord) puis inclut le fichier `/routes/auth.php` où sont définis ses routes dans le fichier `/routes/web.php` :

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

require __DIR__.'/auth.php';
```

(il écrase vos propres routes, il faudra les recréer !)

Les assets CSS et JavaScript

Nous pouvons voir les modules Node que Breeze apporte au fichier `/package.json` :

La version de autoprefixer n'est pas compatible⁴ !

Pour corriger :

```
npm install autoprefixer@10.4.5 --save-exact
```

```
"devDependencies": {
  "@tailwindcss/forms": "^0.4.0",
  "alpinejs": "^3.4.2",
  "autoprefixer": "^10.4.5",
  "axios": "^0.21",
  "bootstrap": "^5.2.1",
  "laravel-mix": "^6.0.6",
  "lodash": "^4.17.19",
  "postcss": "^8.4.6",
  "resolve-url-loader": "^5.0.0",
  "sass": "^1.55.0",
  "sass-loader": "^12.1.0",
  "tailwindcss": "^3.1.0"
},
```

Pour le style CSS, Tailwind CSS est importé au fichier `/resources/css/app.css` par défaut

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Nous allons modifier ceci pour le fonctionnement avec Bootstrap

Dans **resources/css**, créez un fichier **tailwind.css**

Coupez collez les 3 lignes tailwind de app.css dans ce dernier

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Modification du fichier **Webpack.mix.js**

```
mix.js('resources/js/app.js', 'public/js')
  .postCss('resources/css/tailwind.css', 'public/css', [
    require('tailwindcss'),
    require('autoprefixer')
  ])
  .sass('resources/scss/bootstrap.scss', 'public/css')
  .sass('resources/scss/app.scss', 'public/css')
  .sourceMaps();
mix.version();
```

Alpine.js est importé au fichier `/resources/js/app.js` :

```
import Alpine from 'alpinejs';
window.Alpine = Alpine;
Alpine.start();
```

⁴ Source : <https://stackoverflow.com/questions/72083968/1-warning-in-child-compilations-use-stats-children-true-resp-stats-child>

Modifiez votre template master.blade.php

```
<!DOCTYPE html>
<html lang="{{ str_replace('_', '-', app()->getLocale()) }}">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Club Va'a - @yield('title')</title>
    <link rel="stylesheet" href="{{ asset('css/app.css') }}">
    <link rel="stylesheet" href="{{ asset('css/bootstrap.css') }}">
    <link rel="stylesheet" href="{{ asset('css/tailwind.css') }}">
</head>
<body>

    @include('layout.navbar')
    @include('layout.header')

    <div class="container">

        @yield('content')

    </div>

    @include('layout.footer')

    @yield('script')

    {{--<script src="/js/app.js"></script>--}}
    <script src="{{ asset('js/app.js') }}" defer></script>

</body>
</html>
```

{{ str_replace('_', '-', app()->getLocale()) }} :

ici, on utilise str_replace de PHP pour remplacer les éventuels _ par des - dans la chaîne app()->getLocale().

app() est un helper qui permet d'accéder aux variables de configuration définies dans le fichier app.php. Ici, on va chercher la valeur de "locale". On l'avait remplacée par "fr"

{{ asset('css/app.css') }} :

Le helper asset nous permet de charger les feuilles de style, images et js du dossier public

<title>Club Va'a - @yield('title')</title>

Permet d'afficher le titre spécifié dans la vue

Exemple avec la vue « contact »

```
@extends('layout.master')
@section('title')
    Contact
@endsection
@section('content')

    <h1>contact</h1>

@endsection
```

Tester !

Ouvrez le fichier `resources/views/auth/register.blade.php`
Ajoutez comme il se doit aux bons emplacements

```
@extends("layout.master")
@section('content')
@endsection
```

Testez

<http://127.0.0.1:8000/register>

⚙️ Personnalisez l'ensemble de vos interfaces 😊

Rappels – Mémo

Namespace

Les espaces de nom en PHP sont des sortes de dossiers virtuels qui vont nous servir à encapsuler (c'est-à-dire à isoler) certains éléments de certains autres.

Les espaces de noms vont notamment permettre d'éliminer les conflits possibles entre deux éléments de même nom. Cette ambiguïté autour du nom de plusieurs éléments de même type peut survenir lorsqu'on a défini des fonctions, classes ou constantes personnalisées et qu'on fait appel à des extensions ou des bibliothèques externes PHP.

Ici, vous devez savoir qu'une extension ou une bibliothèque externe est un ensemble de code qui ne fait pas partie du langage nativement mais qui a pour but de rajouter des fonctionnalités au langage de base en proposant notamment par exemple des classes préconstruites pour créer une connexion à une base de données ou pour filtrer des données externes.

Il est possible que certaines classes, fonctions ou constantes d'une extension qu'on va utiliser dans un script possèdent des noms identiques à des classes, fonctions ou constantes personnalisées qu'on a défini dans un script.

Sans définition d'un espace de noms, cela va évidemment créer des conflits, puisque le reste du script utilisant la fonction, classe ou constante ne va pas savoir à laquelle se référer.

Pour prendre un exemple concret, vous pouvez considérer que les espaces de noms fonctionnent comme les dossiers sur notre ordinateur. Alors qu'il est impossible de stocker deux fichiers de même nom dans un même dossier, on va tout à fait pouvoir stocker deux fichiers de même nom dans deux dossiers différents. En effet, dans ce dernier cas, il n'y a plus d'ambiguïté puisque les deux fichiers ont un chemin d'accès et de fait une adresse différente sur notre ordinateur.