

TP N° 5

Base de donnée : Migrations

Ista Tinghir

Réalisée par : Groupe 5

Module :M205

Développer Back-End

le :17 mai 2023

Introduction

Une migration permet de créer et de mettre à jour un schéma de base de données. Autrement dit, vous pouvez créer des tables, des colonnes dans ces tables, en supprimer, créer des index... Tout ce qui concerne la maintenance de vos tables peut être pris en charge par cet outil. Vous avez ainsi un suivi de vos modifications

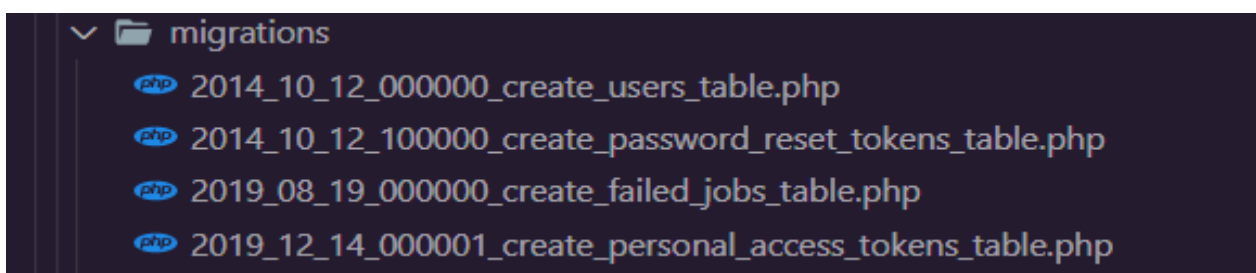
La migration de base de données est l'une des meilleures fonctionnalités fournies par Laravel . Dans le passé, vous devez créer une base de données, une table et des colonnes soit par codage SQL ou en utilisant un outil comme PHPMysqlAdmin.

Tout ce que vous avez à faire est d'écrire quelques lignes de code PHP et Laravel s'occupera du prochain pas.

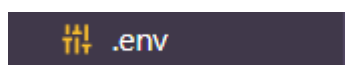
1- Génération de migrations

Vous pouvez utiliser le make:migration Commande artisanale pour générer une migration de base de données. La nouvelle migration sera placée dans votre database/migrations répertoire. Chaque nom de fichier de migration contient un horodatage qui permet à Laravel de déterminer l'ordre des migrations:

En vous rendant dans ce dossier vous découvrez que 4 migrations existent déjà. Une pour la création de la table users et une pour la table password_resets et failed_job, personal_access_tokens



Commencez par créer une base MySQL et informez **.env**, par exemple :



```
11 DB_CONNECTION=mysql
12 DB_HOST=127.0.0.1
13 DB_PORT=3306
14 DB_DATABASE=Stagiaires
15 DB_USERNAME=root
16 DB_PASSWORD=
```

a. La création d'une migration (la création d'une table dans la base de données)

```
php artisan make:migration create_stagiaires_table
```

Le nom de table il doit écrire en plurielle (**stagiaires**) ↕

2- Structure de migration

Une classe de migration contient deux méthodes: up et down. Le up la méthode est utilisée pour ajouter de nouvelles tables, colonnes ou index à votre base de données, tandis que down la méthode doit inverser les opérations effectuées par le up méthode.

```
return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        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.
     */
    public function down(): void
    {
        Schema::dropIfExists('users');
    }
};
```

#	Nom	Type
1	id 	bigint(20)
2	name	varchar(255)
3	email 	varchar(255)
4	email_verified_at	timestamp
5	password	varchar(255)
6	remember_token	varchar(100)
7	created_at	timestamp
8	updated_at	timestamp


```

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();
});

```

- Types de colonnes disponibles

bigIncrements	jsonb	chaîne
bigInteger	lineString	texte
binaire	longText	timeTz
booléen	macAddress	temps
char	moyennes incréments	timestampTz
dateTimeTz	mediumInteger	horodatage
date	mediumText	timestampsTz
date	morphes	horodatages
décimal	multiLineString	minuscules incréments
double	multiPoint	minuscule
enum	multiPolygone	minuscule texte
flottant	nullableMorphs	non signé BigInteger
étranger	nullableTimestamps	unsignedDecimal
foreignIdFor	nullableUlidMorphs	non signé
étranger	nullableUuidMorphs	non signéMediumInteger
étranger	point	non signé SmallInteger
géométrie	polygone	unsignedTinyInteger
géométrie	me souvenir de Token	ulidMorphs
id	ensemble	uuidMorphs
incrément	petits incréments	ulid
entier	smallInteger	uuid
ipAddress	softDeletsTz	année
json	softDelets	

➤ Quelque exemple de type de colonnes

- BigIncrements()

Le `bigIncrements` la méthode crée une incrémentation automatique UNSIGNED BIGINT (clé primaire) colonne équivalente:

```
$table->bigIncrements('id')
```

- **boolean ()**

Le `boolean` la méthode crée un BOOLEAN colonne équivalente

```
$table->boolean('confirmed')
```

- **Text ()**

Le `text` la méthode crée un TEXT colonne équivalente:

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

3- Exécution des migrations

Pour exécuter toutes vos migrations exceptionnelles, exécutez le `migrate` Commande artisanale:

```
php artisan migrate
```

On va maintenant utiliser la migration (méthode **up** de la migration) :

```
PS E:\DDOWFS\Laravel 9\project\Auth\Auth> php artisan migrate
INFO Running migrations.
2023_05_16_174541_create_stagiaires_table ..... 2,209ms DONE
```

Si on regarde maintenant dans la base on trouve la table « stagiaires » avec ces 3 colonnes :

id	Nom et Prenom	Note	created_at	updated_at
----	---------------	------	------------	------------

Si vous avez fait une erreur vous pouvez revenir en arrière avec l'un des **méthodes de Migrations de recul** `rollback` qui annule la dernière migration effectuée (utilisation de la méthode **down** de la migration) :

```
php artisan migrate:rollback
```

```
PS E:\DDOWFS\Laravel 9\project\Auth\Auth> php artisan migrate:rollback
INFO Rolling back migrations.
2023_05_16_174541_create_stagiaires_table ..... 605ms DONE
```

Vous pouvez annuler un nombre limité de migrations en fournissant le `step` option à la `rollback` commande. Par exemple, la commande suivante annulera les cinq dernières migrations:

```
php artisan migrate:rollback --step=5
```

1- Le migrate:reset

la commande annulera toutes les migrations de votre application:

```
php artisan migrate:reset
```

2- Le migrate:refresh

la commande renversera toutes vos migrations, puis exécutera migrate commande. Cette commande recrée efficacement toute votre base de données:

```
php artisan migrate:refresh
```

3- Le migrate:fresh

la commande supprimera toutes les tables de la base de données, puis exécutera migrate commande:

```
php artisan migrate:fresh
```

4- Manipulation des Tables

1- Renommer une table de base de données

Pour renommer une table de base de données existante, utilisez le **rename** méthode:

```
php artisan make:migration rename_stagiaires_table
```

```
public function up(): void
{
    Schema::rename('stagiaires','stg');
}

/**
 * Reverse the migrations.
 */
public function down(): void
{
    Schema::rename('stg','stagiaires');
}
```

2- Suppression d'une table dans la base de données

Pour supprimer une table existante, vous pouvez utiliser le **drop** ou **dropIfExists** méthodes:

```
php artisan make:migration suppri_stagiaires_table
```

```
public function up(): void  
{  
    Schema::drop('stg');  
}
```

5- Modification des colonnes

1. Renommer les colonnes

```
php artisan make:migration renameColumn_Note_in_stagiaires --table=stagiaires
```

```
public function up(): void  
{  
    Schema::table("stagiaires", function (Blueprint $table) {  
        $table->renameColumn('Note', 'nte');  
    });  
}
```

Si vous avez un error , vous devez installer le **doctrine/dbal package** à l'aide du gestionnaire de packages Composer. La bibliothèque Doctrine DBAL est utilisée pour déterminer l'état actuel de la colonne et pour créer les requêtes SQL nécessaires pour apporter les modifications demandées à votre colonne:

```
composer require doctrine/dbal
```

2. Modifier le type et les attributs

Le **change** la méthode vous permet de modifier le type et les attributs des colonnes existantes. Par exemple, vous souhaitez peut-être augmenter [la taille d'un string colonne](#). Pour voir le change méthode en action, augmentons la taille de la name colonne de 25 à 50. Pour ce faire, nous définissons simplement le nouvel état de la colonne, puis appelons le **change méthode**:

```
php artisan make:migration change_string_in_stagiaires --table=stagiaires
```

```
public function up(): void  
{  
    Schema::table('stagiaires', function (Blueprint $table) {  
        $table->text('Note')->change();  
    });  
}
```

3. supprimer une colonne

Pour supprimer une colonne, vous pouvez utiliser le **dropColumn** méthode sur le générateur de schéma:

```
php artisan make:migration dropColumn_Note_in_stagiaires --table=stagiaire
```

```
public function up(): void
{
    Schema::table('stagiaires', function (Blueprint $table) {
        $table->dropColumn('Note');
    });
}
```

Vous pouvez supprimer plusieurs colonnes d'une table en passant un tableau de noms de colonnes à dropColumn méthode:

```
public function up(): void
{
    Schema::table('users', function (Blueprint $table) {
        $table->dropColumn(['Note', 'Nome', '....']);
    });
}
```

6- Index

1- Création d'index

Le constructeur de schémas Laravel prend en charge plusieurs types d'index. L'exemple suivant crée un nouveau email colonne et spécifie que ses valeurs doivent être uniques. Pour créer l'index, nous pouvons enchaîner le unique méthode sur la définition de colonne:

```
public function up(): void
{
    Schema::table('stagiaires', function (Blueprint $table) {
        $table->string('email')->unique();
    });
}
```

Ou

```
public function up(): void
{
    Schema::table('stagiaires', function (Blueprint $table) {
        $table->unique('email');
    });
}
```

2- Types d'index disponibles

La classe de plan du constructeur de schémas de Laravel fournit des méthodes pour créer chaque type d'index pris en charge par Laravel. Chaque méthode d'index accepte un deuxième argument facultatif pour spécifier le nom de l'index. S'il est omis, le nom sera dérivé des noms de la table et

de la colonne (s) utilisés pour l'index, ainsi que du type d'index. Chacune des méthodes d'index disponibles est décrite dans le tableau ci-dessous:

Commande	La description
<code>\$table->primary(['id', 'parent_id']);</code>	Ajoute des clés composites.
<code>\$table->unique('email');</code>	Ajoute un index unique.
<code>\$table->index('state');</code>	Ajoute un index.
<code>\$table->fullText('body');</code>	Ajoute un index de texte intégral (MySQL / PostgreSQL).
<code>\$table->primary(['id', 'parent_id']);</code>	Ajoute des clés composites.

3- Contraintes clés étrangères

Laravel prend également en charge la création de contraintes clés étrangères, qui sont utilisées pour forcer l'intégrité référentielle au niveau de la base de données. Par exemple, définissons un **user_id** colonne sur le **posts** tableau qui fait référence au id colonne sur un users tableau:

```
public function up(): void
{
    Schema::table('posts', function (Blueprint $table) {
        $table->foreignId('user_id')->constrained();
    });
}
```

constrained la méthode utilisera des conventions pour déterminer le nom de la table et de la colonne référencés. Si le nom de votre table ne correspond pas aux conventions de Laravel, vous pouvez spécifier le nom de la table en le transmettant comme argument au constrained méthode:

```
$table->foreign('user_id')->references('id')->on('users');
```

Vous pouvez également spécifier l'action souhaitée pour les propriétés "**on delete**" et "**on update**" de la contrainte:

```
$table->foreignId('user_id')
    ->constrained()
    ->onUpdate('cascade')
    ->onDelete('cascade');
```

Méthode	La description
<code>\$table->cascadeOnUpdate();</code>	Les mises à jour devraient tomber en cascade.
<code>\$table->restrictOnUpdate();</code>	Les mises à jour doivent être restreintes.
<code>\$table->cascadeOnDelete();</code>	Les suppressions devraient cascade.
<code>\$table->restrictOnDelete();</code>	Les suppressions doivent être restreintes.

7- Événements

Les événements de Laravel fournissent une implémentation simple du modèle d'observateur, vous permettant de vous abonner et d'écouter pour divers événements qui se produisent dans votre application. Les classes d'événements sont généralement stockées dans le `app/Events` répertoire, tandis que leurs auditeurs sont stockés dans **app/Listeners**. Ne vous inquiétez pas si vous ne voyez pas ces répertoires dans votre application car ils seront créés pour vous lorsque vous générerez des événements et des auditeurs à l'aide des commandes de la console Artisan.

Les événements servent de moyen idéal pour découpler divers aspects de votre application, car un seul événement peut avoir plusieurs auditeurs qui ne dépendent pas les uns des autres. Par exemple, vous souhaitez peut-être envoyer une notification Slack à votre utilisateur chaque fois qu'une commande est expédiée. Au lieu de coupler votre code de traitement des commandes à votre code de notification Slack, vous pouvez augmenter

un `App\Events\OrderShipped` événement qu'un auditeur peut recevoir et utiliser pour envoyer une notification Slack.

1- Définition des événements

Une classe d'événements est essentiellement un conteneur de données qui contient les informations relatives à l'événement. Par exemple, supposons qu'un événement reçoive un **App\Events\OrderShippedORM** éloquent objet:

```
<?php

namespace App\Events;

use App\Models\Order;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Queue\SerializesModels;

class OrderShipped
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    /**
     * Create a new event instance.
     */
    public function __construct(
        public Order $order,
    ) {}
}
```

Comme vous pouvez le voir, cette classe d'événements ne contient aucune logique. Il s'agit d'un conteneur pour l'instance achetée. Le trait utilisé par l'événement sérialisera gracieusement tous les modèles Eloquent si l'objet événement est sérialisé à l'aide de la fonction PHP, par exemple lors de l'utilisation **App\Models\OrderSerializesModelserializeécouteurs** en file d'attente.

2- Définition des écouteurs

Ensuite, jetons un coup d'œil à l'écouteur pour notre exemple d'événement. Les écouteurs d'événements reçoivent des instances d'événement dans leur méthode. Les commandes et Artisan importent automatiquement la classe d'événement appropriée et indiquent l'événement sur la méthode. Dans le cadre de la méthode, vous pouvez effectuer toutes les actions nécessaires pour répondre à l'événement : **handleevent:generatemade:listenerhandlehandle**

```
<?php

namespace App\Listeners;

use App\Events\OrderShipped;

class SendShipmentNotification
{
    /**
     * Create the event listener.
     */
    public function __construct()
    {
        // ...
    }

    /**
     * Handle the event.
     */
    public function handle(OrderShipped $event): void
    {
        // Access the order using $event->order...
    }
}
```

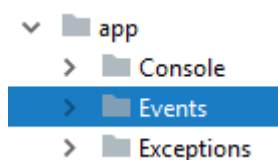
Laravel permet de gérer avec facilité les événements comme on va le voir dans cet exemple.

On a une application et des utilisateurs. Les utilisateurs doivent s'inscrire (on dit aussi s'abonner) auprès de l'application. Lorsque l'application change d'état il notifie tous ses abonnés. Un utilisateur peut aussi se désinscrire, il ne recevra alors plus les notifications.

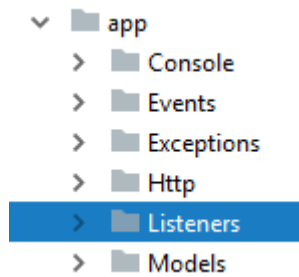
Exemple :

1- Organisation du code

- des classes **Event** placées dans le dossier **app/Events** :



- des classes **Listener** placées dans le dossier **app/Listeners** :



Remarque

Ces dossiers n'existent pas dans l'installation de base, ils sont ajoutés dès qu'on crée la première classe avec Artisan qui dispose de commandes pour le faire :

2- Les évènements déjà présents dans laravel

Il n'y a pas que les événements que vous allez créer. Vous pouvez utiliser les événements existants déjà dans le framework qui en propose au niveau de l'authentification :

- ✓ Registered
- ✓ CurrentDeviceLogout
- ✓ OtherDeviceLogout
- ✓ Attempting
- ✓ Authenticated
- ✓ Login
- ✓ Failed
- ✓ Logout
- ✓ Lockout
- ✓ PasswordReset
- ✓ Validated
- ✓ Verified

On trouve aussi des événements avec Eloquent :

- ✓ retrieved
- ✓ creating
- ✓ created
- ✓ updating
- ✓ updated
- ✓ saving
- ✓ saved
- ✓ deleting
- ✓ deleted
- ✓ restoring
- ✓ restored

Pour tous ces cas vous n'avez donc pas à vous inquiéter du déclencheur qui existe déjà.

3- Une application de test

Pour nos essais créez une nouvelle application Laravel 9 :

```
composer create-project laravel/laravel laravel9
```

Créez une base, renseignez correctement le fichier **.env**. Installez Breeze et lancez les migrations

```
composer require laravel/breeze --dev
```

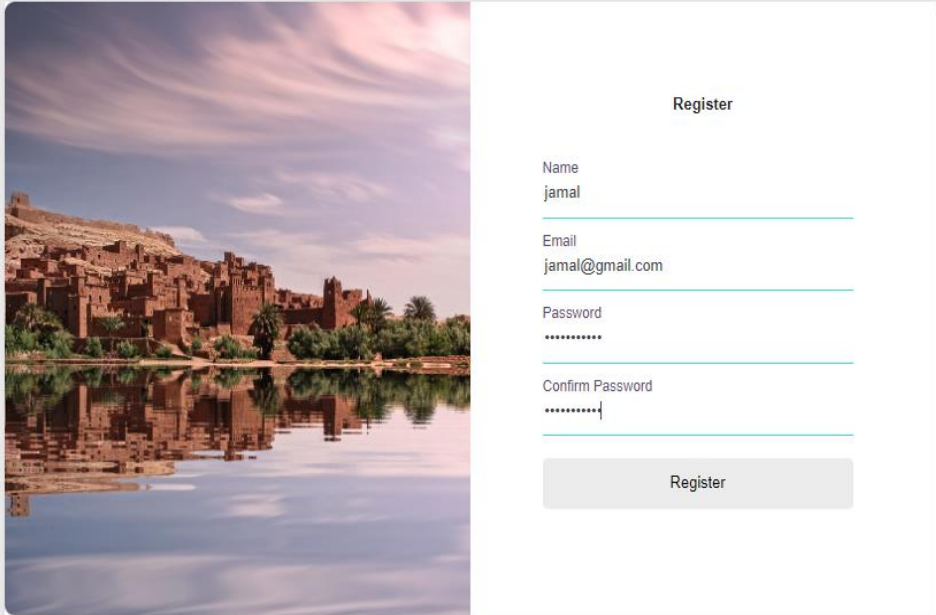
```
php artisan breeze:install
```

```
npm install
```

```
npm run dev
```

```
php artisan migrate
```

Créez un utilisateur à partir du formulaire d'enregistrement :



Register

Name
jamal

Email
jamal@gmail.com

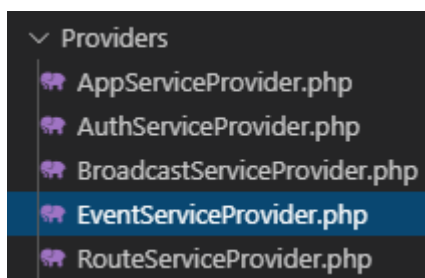
Password

Confirm Password

Register

3- Le fournisseur de service (service provider)

Il y a un fournisseur de service dédié aux événements :



Par défaut on a ce code :

```

namespace App\Providers;
use Illuminate\Auth\Events\Registered;
use Illuminate\Auth\Listeners\SendEmailVerificationNotification;
use Illuminate\Foundation\Support\Providers\EventServiceProvider as ServiceProvider;
use Illuminate\Support\Facades\Event;
class EventServiceProvider extends ServiceProvider
{
    /**
     * The event listener mappings for the application.
     *
     * @var array<class-string, array<int, class-string>>
     */
    protected $listen = [
        Registered::class => [
            SendEmailVerificationNotification::class,
        ],
    ];
    /**
     * Register any events for your application.
     *
     * @return void
     */
    public function boot()
    {
        //
    }
    /**
     * Determine if events and listeners should be automatically discovered.
     *
     * @return bool
     */
    public function shouldDiscoverEvents()
    {
        return false;
    }
}

```

On a une propriété **\$listen** pour les abonnements. C'est un simple tableau qui prend l'événement (**event**) comme clé et l'observateur (**listener**) comme valeur. On voit qu'on a déjà l'événement **Registered** qui renvoie à l'observateur **SendEmailVerificationNotification**.

Avec jetstream lorsqu'un utilisateur s'enregistre on peut lui envoyer un email, mais on ne s'était pas posé la question de savoir comment ça se passait en interne. On a ici une réponse, on observe l'événement **Registered** et on lance le code pour la notification avec le listener.

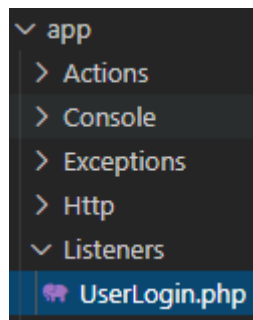
Laravel prévoit de placer les l'observateur (**listeners**) dans un dossier **app/Listeners** (qui n'existe pas au départ) en tant que classes, de même pour les événements (**events**) dans le dossier **Events** (qui n'existe pas au départ).

4- On crée une écoute

On va utiliser l'événement intégré à Laravel qui se déclenche lorsqu'un utilisateur se connecte. La classe correspondante est **Illuminate\Auth\Events\Login**. Puisque l'événement existe on n'a pas à créer une classe **Event**. Par contre il nous faut une écoute (**Listener**) :

```
php artisan make:listener UserLogin
```

La classe **UserLogin** se crée ainsi que le dossier :



Voici le code généré :

```
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class UserLogin
{
    /**
     * Create the event listener.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * Handle the event.
     *
     * @param object $event
     * @return void
     */
    public function handle($event)
    {
        //
    }
}
```

On va établir le lien entre l'événement et l'écoute dans le provider

EventServiceProvider :

```
use Illuminate\Auth\Events\Login;
use App\Listeners\UserLogin;

class EventServiceProvider extends ServiceProvider
{
    protected $listen = [

        Login::class => [UserLogin::class],
    ];
}
```

Donc maintenant dès que utilisateur va se connecter la méthode **handle** de notre écoute va être appelée. Mais qu'y a-t-il dans la variable **\$event** ? On va faire un petit test :

```
Public function handle($event)
{
    dd($event) ;
}
```

Illuminate\Auth\Events>Login {#399 ▼

+guard: "web"

+user: App\Models\User {#1370 ►}

+remember: false

}

Une instance de la classe **Illuminate/Auth/Events/Login** avec plusieurs propriétés, en particulier une instance du modèle de l'utilisateur et un booléen pour le **remember**. On aurait aussi pu trouver ces informations en allant voir directement la classe.

Changez ainsi le code :

```
Public function handle($event)
{
    dd($event->user->name." s'est connecté.");
}
```

Cette fois à la connexion on obtient quelque chose dans ce genre :

"jamal s'est connecté."

On peut donc ici effectuer tous les traitements qu'on veut, comme comptabiliser les connexions.

5- Découverte automatique des événements

On a vu ci-dessus qu'on a créé un listener pour un événement existant dans Laravel (**Login**) et qu'on a dû déclarer le lien entre événement et listener dans le provider **EventServiceProvider**. Mais il est possible de rendre ce lien automatique, pour ça il faut faire deux choses

- comme Laravel utilise la réflexion il faut un peu l'aider et préciser la classe de l'événement dans la fonction **handle** du listener :

```
use Illuminate\Auth\Events>Login;
...
public function handle(Login $event)
```

- il faut aussi changer la valeur de retour dans cette méthode du provider

EventServiceProvider :

```
/**
 * Determine if events and listeners should be automatically discovered.
 */
public function shouldDiscoverEvents(): bool
{
    return true;
}
```

Maintenant on peut s'éviter l'écriture de la liaison dans la propriété **\$listen** !

6- On crée un événement

Maintenant on va créer un événement et aussi son écoute. On va imaginer qu'on veut savoir quand quelqu'un affiche la page d'accueil et mémoriser son IP ainsi que le moment du chargement de la page, en gros un peu de statistique.

On va commencer par créer une migration :

```
php artisan make:migration create_visits_table --create=visits
```

```
php 2023_05_17_084657_create_stagiaires_table.php
php 2023_05_17_085527_rename_stagiaires_table.php
php 2023_05_17_094133_create_ofppt_table.php
php 2023_05_18_131348_create_visits_table.php
```

Avec ces créations de colonnes :

```
public function up(): void
{
    Schema::create('visits', function (Blueprint $table) {
        $table->id();
        $table->string('ip', 45);
        $table->timestamp('created_at');
    });
}
```

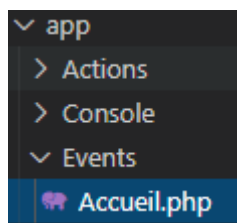
Et on lance la migration :

```
php artisan migrate
```

On crée l'événement :

```
artisan make:event Accueil
```

On le trouve ici avec création du dossier :



Avec ce code par défaut :


```

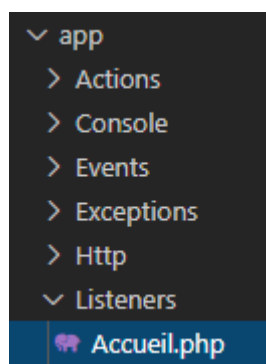
namespace App\Events;
use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Queue\SerializesModels;
class Accueil
{
    use Dispatchable, InteractsWithSockets, SerializesModels;
    /**
     * Create a new event instance.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }
    /**
     * Get the channels the event should broadcast on.
     *
     * @return \Illuminate\Broadcasting\Channel|array
     */
    public function broadcastOn()
    {
        return new PrivateChannel('channel-name');
    }
}

```

On ne va pas toucher à ce code.

On crée aussi l'écoute :

```
php artisan make:listener Accueil
```



On modifie ainsi le code :

```

namespace App\Listeners;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Support\Facades\DB;
use Carbon\Carbon;
use App\Events\Accueil as AccueilEvent;
class Accueil
{
    public function __construct()
    {
        //
    }
    public function handle(AccueilEvent $event)
    {
        DB::table('visits')->insert([
            'ip' => request()->ip(),
            'created_at' => Carbon::now(),
        ]);
    }
}

```





J'utilise directement le Query Builder sans passer par Eloquent.

Il ne reste plus qu'à déclencher cet événement dans la route :

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

Ne oublier pas l'importation de l'événement Accueil : **use App\Events\Accueil**

Maintenant chaque fois que la page d'accueil est ouverte l'événement **Accueil** est déclenché et donc l'écoute **Accueil** en est informée. On sauvegarde alors dans la base l'IP et le moment du chargement :

					id	ip	created_at
<input type="checkbox"/>	 Éditer	 Copier	 Supprimer		2	127.0.0.1	2023-05-18 10:27:29