Gestion des événements

1. Introduction :

"Aujourd'hui, nous apprenons un sujet important de Laravel à l'aide d'un exemple. J'espère qu'il vous aidera à comprendre le sujet d'aujourd'hui. Dans cet exemple, nous allons montrer comment suivre l'historique de connexion de votre application et stocker les données dans la base de données en utilisant des événements et des Listener."

Qu'est-ce qu'un événement Laravel ?

Les événements Laravel sont un moyen de mettre en œuvre un modèle d'observateur simple de l'activité de votre application. Par exemple, si vous voulez surveiller le moment où l'utilisateur de votre application se connecte, à partir de quelle adresse IP, vous pouvez exécuter une fonction en utilisant des événements. Si vous avez un site de commerce électronique, vous avez parfois besoin de notifier ou d'envoyer un SMS à votre vendeur lorsqu'une nouvelle commande est passée. Ainsi, nous pouvons appeler les événements comme des récepteurs d'action de notre application. Laravel possède des fonctionnalités par défaut pour gérer un événement.

Qu'est-ce qu'un Listener dans Laravel ?

Dans Laravel, le listener est une classe qui exécute les instructions d'un événement. Par exemple, vous voulez envoyer un message de bienvenue à votre client lorsqu'il s'inscrit sur votre site. Dans ce cas, nous pouvons définir un événement qui appelle un listener pour envoyer le mail.

Etape 1 : Enregistrement d'événements et des écouteurs

`App\Providers\EventServiceProvider` inclus dans votre application Laravel fournit un endroit pratique pour enregistrer tous les écouteurs d'événements de votre application. La propriété `listen` contient un tableau de tous les événements (clés) et de leurs listeners (valeurs). Vous pouvez ajouter autant d'événements à ce tableau que votre application le requiert.

Par exemple, nous avons ajouté une autre classe d'événement appelée `VisitHistory` et également un écouteur appelé `StoreVisitProfileHistory`, et nous avons remarqué que nous avons appelé la classe de cette façon `use`

AppEvents\VisitHistory ; et use AppListeners\storeVisitProfileHistory ;, ne vous inquiétez pas, je sais que vous vous demandez si la classe n'existe pas dans notre application, nous allons la générer dans l'étape suivante, vous pouvez ajouter autant d'événements et d'écouteurs que possible comme ceci et même davantage.

app/Providers/EventServiceProvider.php

```
use App\Events\VisitHistory;
use App\Listeners\storeVisitProfileHistory;


/**
 * The event listener mappings for the application.
 *
 * @var array
 */
 protected $listen = [
        Registered::class => [
            SendEmailVerificationNotification::class,
        ],
        VisitHistory::class => [
            StoreVisitProfileHistory::class,
        ]
    ];
```

La commande event:list peut être utilisée pour afficher une liste de tous les événements et les listeners enregistrés par votre application.

Etape 2 : Générer des événements et des écouteurs

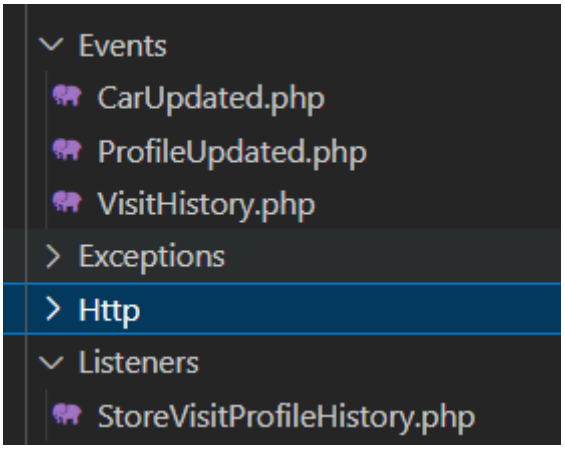
Précédemment, nous avons écrit des classes d'événements et des listeners dans l'EventServiceProvider, donc pour le générer en une seule fois, nous lançons cette commande

```
php artisan event:generate
```

```
PS C:\xampp\htdocs\tp1> php artisan event:generate

   INFO  Events and listeners generated successfully.
```

Cette commande générera automatiquement tous les événements et les récepteurs trouvés dans le fournisseur d'événements,



Etape 3 : Ecrire la classe Events et Listener

Rappelez-vous que ce que nous essayons de faire est de stocker tous les visites des profils de notre application dans un tableau, donc cliquez sur app/Events/VisitHistory.php et modifiez-le comme suit :

app/Events/VisitHistory.php

```php
<?php

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 VisitHistory
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    /**
     * Create a new event instance.
     *
```

```
     * @return void
     */
    public $student;
    public function __construct($student)
    {
        $this->student = $student;
    }

    /**
     * Get the channels the event should broadcast on.
     *
     * @return \Illuminate\Broadcasting\Channel|array
     */
    public function broadcastOn()
    {
        return new PrivateChannel('channel-name');
    }
}
```

Dans le code ci-dessus, l'événement accepte **$student** qui est l'information de l'étudiant, et il le passera à l'écouteur (listener).

Cliquez sur app/Listeners/storeVisitProfileHistory.php, c'est ici que nous allons écrire la logique principale du stockage de l'historique des visites, à l'intérieur de la méthode **handle**, ajoutez le code suivant

app/Listeners/storeVisitProfileHistory.php

```
<?php

namespace App\Listeners;

use Carbon\Carbon;
use App\Events\VisitHistory;
use Illuminate\Support\Facades\DB;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

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

```
        //
    }

    /**
     * Handle the event.
     *
     * @param  \App\Events\VisitHistory  $event
     * @return void
     */
    public function handle(VisitHistory $event)
    {
        $current_timestamp = Carbon::now();

        $userinfo = $event->student;

        $saveHistory = DB::table('visit_history')->insert(
            ['name' => $userinfo->name, 'email' => $userinfo->mail,
'created_at' => $current_timestamp, 'updated_at' => $current_timestamp]
        );

    }
}
```

N'oubliez pas non plus d'appeler la façade **Carbon** et **DB** avant la classe.

```
use Illuminate\Support\Facades\DB;
use Carbon\Carbon;
```

À partir de l'écouteur, nous essayons d'ajouter le nom, l'adresse électronique, l'heure de création et l'heure de mise à jour à une table **visit_history**, c'est-à-dire que lorsqu'on visite un profil d'un étudiant, il récupère ces informations et les stocke dans la table.

Etape 4 : Créer la table et migrer

Créer le fichier de migration

```
PS C:\xampp\htdocs\tp1> php artisan make:migration create_visit_history_table --create=visit_history

  INFO  Migration [2023_03_25_132512_create_visit_history_table] created successfully.
```

Ajouter vos colonnes au fichier de migration

Databese/migrations/xxx_create_visit_history_table.php

```php
<?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('visit_history', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('email');
            $table->timestamps();
        });
    }

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

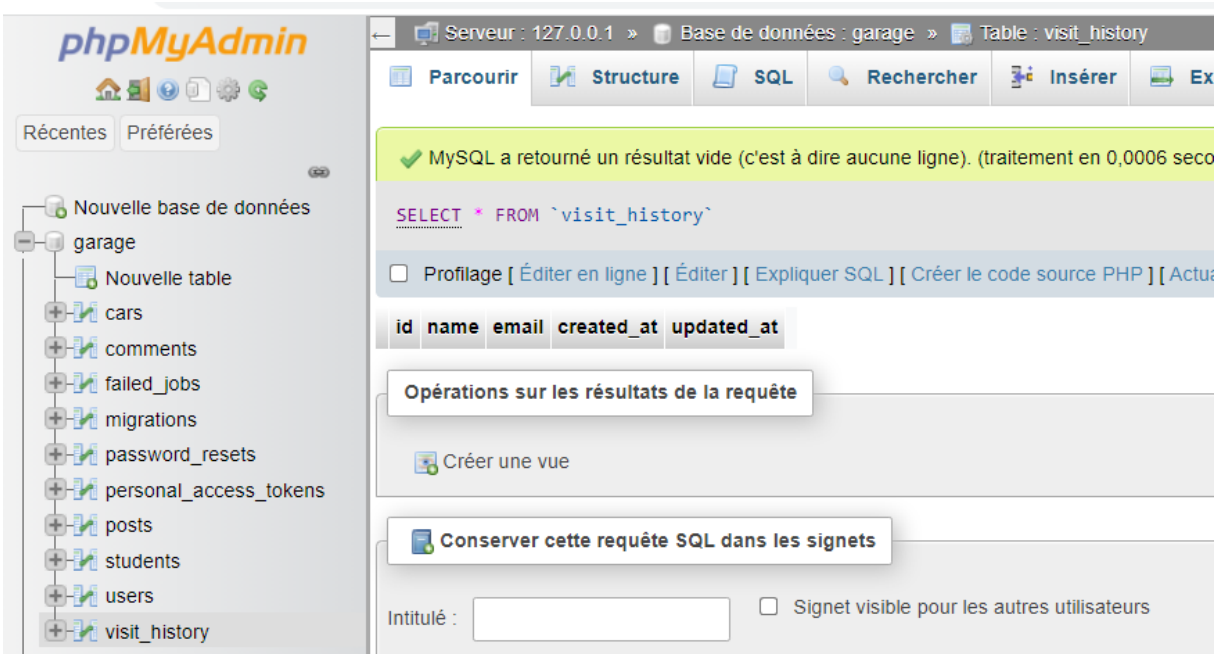
Lancer la migration

```
PS C:\xampp\htdocs\tp1> php artisan migrate --path=database\migrations\2023_03_25_132512_create_visit_history_table.p
hp

  INFO  Running migrations.

  2023_03_25_132512_create_visit_history_table ..................................................... 380ms DONE
```

Nous avons maintenant notre tableau dans la base de données



Etape 5 : Appeler l'événement

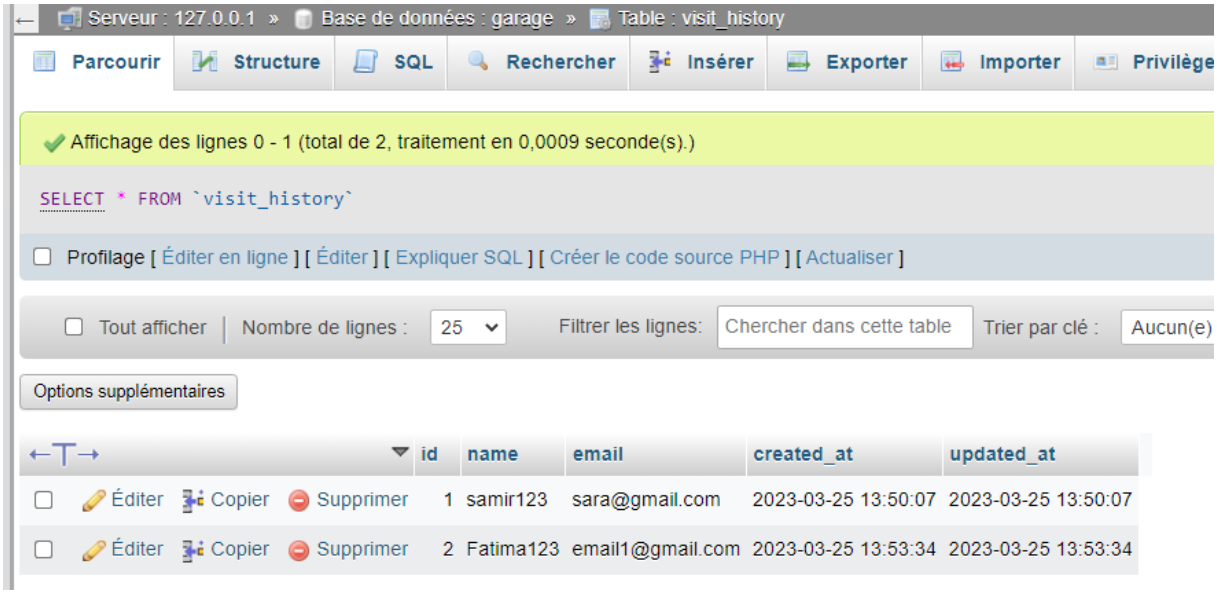
C'est la dernière étape, nous devons appeler l'événement dans le StudentController.

app/Http/controllers/Studentcontroller.php

```
public function show(Request $request,Student $student)
   {

     event(new VisitHistory($student));
      return view('students.profile', ['student' => $student]);
```

Voici le résultat, lorsque je j'accède deux fois à des profils différents, l'événement est déclenché et l'écouteur enregistre l'historique.


Serveur : 127.0.0.1 » Base de données : garage » Table : visit_history
Parcourir
Structure
SQL
Rechercher
Insérer
Exporter
Importer
Privilège
Affichage des lignes 0 - 1 (total de 2, traitement en 0,0009 seconde(s).)
SELECT * FROM `visit_history`
Profilage [Éditer en ligne] [Éditer] [Expliquer SQL] [Créer le code source PHP] [Actualiser]
Tout afficher
Nombre de lignes : 25
Filtrer les lignes: Chercher dans cette table
Trier par clé : Aucun(e)
Options supplémentaires
id name email created_at updated_at
Éditer Copier Supprimer 1 samir123 sara@gmail.com 2023-03-25 13:50:07 2023-03-25 13:50:07
Éditer Copier Supprimer 2 Fatima123 email1@gmail.com 2023-03-25 13:53:34 2023-03-25 13:53:34