Projet Laravel : Eatup

Rapport du Projet : Développer une application CRUD

Toutes les fichiers source sont disponibles sur le dépôt GitHub associé à l'article.

Par : MAZTAOUI ILIAS

Lien GitHub :

https://github.com/ilias-coder/Eatup

Sommaire :

Remerciement :

Je tiens à vous remercier pour votre initiative pour nous donner un projet par lequel j'ai pu personnellement maitriser plusieurs notions et leurs nécéssités (OOP, MVC, Laravel…).

J'espère que ce travail a été à la hauteur de vos prédictions.

Travail réalisé par : Maztaoui Ilias

Cordialement , Portez-vous bien.

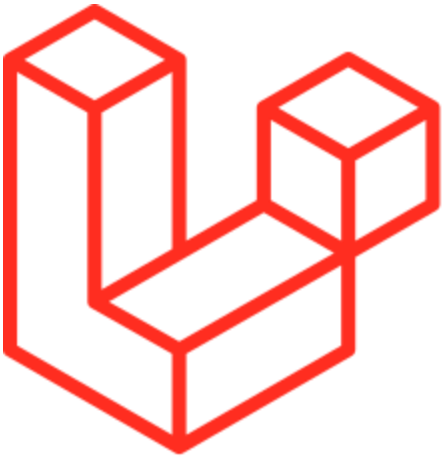
Présentation d'application :

L'application du CRUD, que j'ai développé pendant ce projet s'agit d'un utilisateur qui peut ajouter des recettes qu'on mange (titre de la recette et préciser son type, description), tout en lui gardant la possibilité de supprimer une recette qu'on vient d'ajouter, ou de modifier quelques données qui étaient ajoutées là-dessus.

En ajoutant la possibilité pour se connecter avec un email et un mot de passe, tout en voyant tout d'abord une interface que tout le monde peut la voir, ce sont les recettes activées par un des utilisateurs de l'application. Puis, on positionne un bouton pour que l'utilisateur se dirige vers son profil et ajouter ou activer, modifier une recette… ou de supprimer définitivement une recette supprimée mais qui est gardée si l'utilisateur change d'avis.

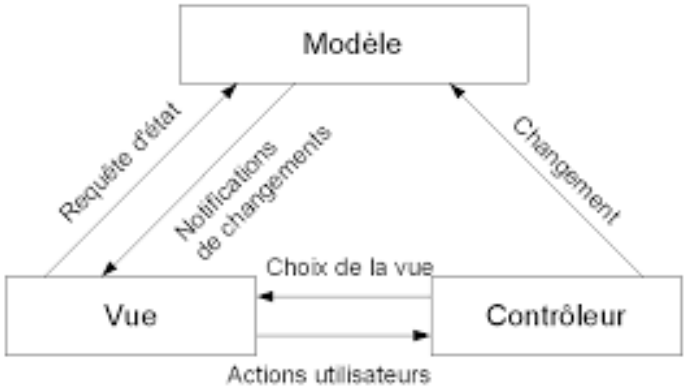
Tout d’abord, la première question qui se pose :

C’est quoi Laravel ?



Laravel est un framework web open-source écrit en PHP1 respectant le principe **MVC (modèle-vue-contrôleur)** et entièrement développé en programmation orientée objet. Laravel est distribué sous licence MIT, avec ses sources hébergées sur GitHub.

Mais un MVC, c'est quoi plus précisement ?



Modèle-vue-contrôleur ou MVC est un motif d'architecture logicielle destiné aux interfaces graphiques lancé en 1978 et très populaire pour les applications web. Le motif est composé de trois types de modules ayant trois responsabilités différentes : les modèles, les vues et les contrôleurs.

Un **modèle (Model)** contient les données à afficher.

Une **vue (View)** contient la présentation de l'interface graphique.

Un **contrôleur (Controller)** contient la logique concernant les actions effectuées par l'utilisateur.

Ce motif est utilisé par de nombreux frameworks pour applications web tels que **Laravel**, Ruby on Rails, Grails, ASP.NET MVC, Spring, Struts, Symfony, Apache Tapestry, ou AngularJS.

En général on résume en disant que le modèle gère la base de données, la vue produit les pages HTML et le contrôleur fait tout le reste. Dans Laravel :

- le modèle correspond à une table d'une base de données. C'est une classe qui étend la classe Model qui permet une gestion simple et efficace des manipulations de données et l'établissement automatisé de relations entre tables.
- le contrôleur se décline en plusieurs catégories : contrôleur classique, contrôleur RESTfull et contrôleur de ressource (je détaillerai évidemment tout ça dans le cours).
- la vue est soit un simple fichier avec du code HTML, soit un fichier utilisant le système de template Blade de Laravel.

Laravel propose ce modèle mais ne l'impose pas. Nous verrons d'ailleurs qu'il est parfois judicieux de s'en éloigner parce qu'il y a des tas de chose qu'on arrive pas à caser dans ce modèle. Par exemple si je dois envoyer des emails où vais-je placer mon code ? En général ce qui se produit est l'inflation des contrôleurs auxquels on demande des choses pour lesquelles ils ne sont pas faits.

Pourquoi Laravel :

- **Constitution de Laravel**

Laravel, créé par Taylor Otwel, initie une nouvelle façon de concevoir un framework en utilisant ce qui existe de mieux pour chaque fonctionnalité. Par exemple toute application web a besoin d'un système qui gère les requêtes HTTP. Plutôt que de réinventer quelque chose le concepteur de Laravel a tout simplement utilisé celui de **Symfony** en l'étendant pour créer un système de routage efficace. De la même manière l'envoi des emails se fait avec la bibliothèque **SwiftMailer**. En quelque sorte Otwel a fait son marché parmi toutes les bibliothèques disponibles. Nous verrons dans ce cours comment cela est réalisé. Mais Laravel ce n'est pas seulement le regroupement de bibliothèques existantes, c'est aussi de nombreux composants originaux et surtout une orchestration de tout ça.

Vous allez trouver dans Laravel :

- un système de routage perfectionné (RESTFul et ressources),
- un créateur de requêtes SQL et un ORM performants,
- un moteur de template efficace,
- un système d'authentification pour les connexions,
- un système de validation,
- un système de pagination,
- un système de migration pour les bases de données,
- un système d'envoi d'emails,
- un système de cache,
- une gestion des sessions...

Et bien d'autres choses encore que nous allons découvrir ensemble. Il est probable que certains éléments de cette liste ne vous évoque pas grand chose, mais ce n'est pas important pour le moment, tout cela deviendra plus clair au fil des chapitres.

- **Utilisation du meilleur du php :**

Plonger dans le code de Laravel c'est recevoir un cours de programmation tant le style est clair et élégant et le code merveilleusement organisé. La version actuelle de Laravel est la 5.1, elle nécessite au minimum la version 5.5.9 de PHP. Pour aborder de façon efficace ce framework il serait souhaitable que vous soyez familiarisé avec ces notions :

- **Les espaces de noms** : c'est une façon de bien ranger le code pour éviter des conflits de nommage. Laravel utilise cette possibilité de façon intensive. Tous les composants sont rangés dans des espaces de noms distincts, de même que l'application créée.
- **Les fonctions anonymes** : ce sont des fonctions sans nom (souvent appelées closures) qui permettent d'améliorer le code. Les utilisateurs de Javascript y sont habitués. Les utilisateurs de PHP un peu moins parce qu'elle y sont plus récentes. Laravel les utilise aussi de façon systématique.
- **Les méthodes magiques** : ce sont des méthodes qui n'ont pas été explicitement décrites dans une classe mais qui peuvent être appelées et résolues.
- **Les interfaces** : une interface est un contrat de constitution des classes. En programmation objet c'est le sommet de la hiérarchie. Tous les composants de Laravel sont fondés sur des interfaces. La version 5 a même vu appraraître un lot de contrats pour étendre de façon sereine le framework.
- **Les traits** : c'est une façon d'ajouter des propriétés et méthodes à une classe sans passer par l'héritage, ce qui permet de passer outre certaines limitation de l'héritage simple proposé par défaut par PHP.

➢ **Utilisation du POO (Programmation orienté objet) :**

Laravel est fondamentalement orienté objet. La POO est un design pattern qui s'éloigne radicalement de la programmation procédurale. Avec la POO tout le code est placé dans des classes qui découlent d'interfaces qui établissent des contrats de fonctionnement. Avec la POO on manipule des objets.

Avec la POO la responsabilité du fonctionnement est répartie dans des classes alors que dans l'approche procédurale tout est mélangé. Le fait de répartir la responsabilité évite la duplication du code qui est le lot presque forcé de la programmation procédurale. Laravel pousse au maximum cette répartition en utilisant l'injection de dépendance.

L'utilisation de classes bien identifiées, dont chacune a une rôle précis, pilotées par des interfaces claires, dopées par l'injection de dépendances : tout cela crée un code élégant, efficace, lisible, facile à maintenir et à tester. C'est ce que Laravel propose. Alors vous pouvez évidemment greffer là dessus votre code approximatif, mais vous pouvez aussi vous inspirer des sources du framework pour améliorer votre style de programmation.

En écrivant en Git :

composer require Laravel/ui

php artisan ui vue – auth

la commande ci-dessous, pour pouvoir l'utiliser, il faudra installer node js :

npm install && npm run dev

php artisan migrate

….

Recipe.php

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Recipe extends Model
{
    //
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = [
        'title', 'body', 'type','status'
    ];

    public function user()
    {
        return $this->belongsTo(\App\User::class);
    }
```

L'attribut $fillable attend un tableau contenant title, body, type et status par create et edit méthodes.

La fonction user détermine que la classe recette appartient à l'utilisateur, donc il faut spécifier dans la classe user en User.php que l'utilisateur a beaucoup de recettes. (Voir la dernière méthode ci-dessous)

User.php :

```php
<?php

namespace App;

use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = [
        'name', 'email', 'password',
    ];

    /**
     * The attributes that should be hidden for arrays.
     *
     * @var array
     */
    protected $hidden = [
        'password', 'remember_token',
    ];

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

```
    public function recipes()
    {
        return $this->hasMany(\App\Recipe::class);
    }
}
```

Ajout du resource controller :

php artisan make :controller RecipeController -r -m Recipe

Et maintenant en Web.php

On poura écrire :

Route ::resource('Recipes',RecipesController') ;

Pour avoir les méthodes de CRUD, il suffit de se repérer à la documentation de Laravel ;

Actions Handled By Resource Controller

Verb	URI	Action	Route Name
GET	`/photos`	index	photos.index
GET	`/photos/create`	create	photos.create
POST	`/photos`	store	photos.store
GET	`/photos/{photo}`	show	photos.show
GET	`/photos/{photo}/edit`	edit	photos.edit

Verb	URI	Action	Route Name
PUT/PATCH	`/photos/{photo}`	update	photos.update
DELETE	`/photos/{photo}`	destroy	photos.destroy

Ajout de la possibilité d'avoir une notification (email) par votre mail :

En .env : On change MAIL_MAILER=smtp,et on le remplace par : **MAIL_MAILER=log**

Puis en Git Bash : on écrit **php artisan config:cache**

On va à **web.php**, on ajoute une route :

```
Route::get('/email','DashboardController@email')->name('dashboard.email');
```

Pour pouvoir tester notre fonction, on utilisera **mailtrap.io** et on créera un compte pour pouvoir tester, puis on choisira l'option laravel pour que la site nous donne la configuration nécessaire qu'on devra ajouter à la fin de notre **.env**

```
MAIL_MAILER=smtp
MAIL_HOST=smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=92c1ceba2cd16a
MAIL_PASSWORD=471700d32bf5dd
MAIL_ENCRYPTION=tls
```

Puis, en terminal du Git Bash : **php artisan make :Mail NotifyAdmin,** qu'on le retrouvera dans notre dossier **app** dans un nouveau dossier **mail.**

Notre fichier **NotifyAdmin.php** devient :

```
<?php

namespace App\Mail;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;

class NotifyAdmin extends Mailable
{
    use Queueable, SerializesModels;

    public $title;

    /**
     * Create a new message instance.
     *
     * @return void
     */
```

```
    public function __construct($title)
    {
        //
        $this->title=$title;
    }

    /**
     * Build the message.
     *
     * @return $this
     */
    public function build()
    {
        return $this->view('emails.notifyAdmin') ->to('user1@gmail.com');
;
    }
}
```

On ajoute un dossier en **views** qui s'appelle emails comme mentionné dans la fonction **build()** et on crée le fichier **notifyAdmin.php**

```
<h1> A new recipe was added !</h1>

<p>Recipe: {{$title}} was added</p>
```

Puis, on se déplace vers **DashboardController.php** :

On ajoute au début :

```
use Illuminate\Support\Facades\Mail;
use App\Mail\NotifyAdmin;
use App\MAil\NewRecipeNotification;
```

Et, on crée une nouvelle fonction :

```
public function email()
    {
        Mail::send(new NotifyAdmin('Title of the recipe'));

    }
```

En Git Bash :

php artisan make:mail NewRecipeNotification --markdown=emails.newRecipeNotification

Ce qui nous crée NewRecipeNotification.blade.php avec contenu:

```
@component('mail::message')
# Introduction

A new recipe was added with {{$title}}!

@component('mail::button', ['url' => ''])
Button Text
@endcomponent

Thanks,<br>
{{ config('app.name') }}
@endcomponent
```

```
php artisan vendor:publish --tag=laravel-mail
```

```
ce qui nous copie :Directory
[\vendor\laravel\framework\src\Illuminate\Mail\resources\views] To
[\reources\views\vendor\mail]
```

on fait une modification en RecipesController.php

```
use Illuminate\Support\facades\Mail;
use App\Mail\NewRecipeNotification;


public function store(RecipeRequest $request)
    {
        //
        $recipe=auth()->user()->recipes()->create($request->all());
        Mail::send(new NotifyAdmin($story->title));

        return redirect()->route('Recipes.index')-
>with('status','Recipe created successfully !');
    }
```

Maintenant, on ajoutera la possibilité d'avoir des évènements et des abonnés .

Alors, on commence par ajouter la façade en en RecipesController.php

```
use Illuminate\Support\facades\Log;
```

On modifie maitenant notre méthode store, qui devient :

```
public function store(RecipeRequest $request)
    {
        //
        $recipe=auth()->user()->recipes()->create($request->all());

        Mail::send(new NotifyAdmin($story->title));
        Log::info('A recipe with title: ' .$recipe->title .'was added');

        return redirect()->route('Recipes.index')-
>with('status','Recipe created successfully !');
    }
```

C'est le temps d'ajouter l'évènement (création d'une recette). En Git, on tape :

```
php artisan make:event RecipeCreated
```

Ceci nous crée un nouveau dossier qui s'appelle (event) avec un nouveau fichier (RecipeCreated) contenant :

```
<?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 RecipeCreated
{
    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 le modifie pour le rendre :

```
public $title;

    /**
     * Create a new event instance.
     *
     * @return void
     */
    public function __construct($title)
    {
        //
        $this->title = $title;
    }
```

Et, en RecipesController.php

```
use App\Events\RecipeCreated;
```

```
public function store(RecipeRequest $request)
    {
        //
        $recipe=auth()->user()->recipes()->create($request->all());

        Mail::send(new NotifyAdmin($story->title));
```

```
        Log::info('A recipe with title: ' .$recipe->title .'was added');

        event(new RecipeCreated($recipe->title));

        return redirect()->route('Recipes.index')-
>with('status','Recipe created successfully !');
    }
```

Maintenant, on va supprimer ces deux lignes :

```
        Mail::send(new NotifyAdmin($story->title));
        Log::info('A recipe with title: ' .$recipe->title .'was added');
```

Mais, on va créer un listener pour eux ; En Git : on ajoute :

```
php artisan make:listener SendNotification -e RecipeCreated
```

```
php artisan make:listener WriteLog -e RecipeCreated
```

En Http, on trouve notre nouveau dossier Listeners avec nos fichiers qu'on vient de générer par Git :

SendNotification.php

```
<?php

namespace App\Listeners;

use App\Events\RecipeCreated;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;

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

```
    }

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

Et WriteLog.php :

```
<?php

namespace App\Listeners;

use App\Events\RecipeCreated;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;

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

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

```
        //
    }
}
```

Mainitenant, Les deux lignes qu'on vient de supprimer auparavant du RecipesController.php, on va les ajouter à ces deux fichiers ;

En SendNotification.php :

```
use Illuminate\Support\facades\Mail;
use App\Mail\NewRecipeNotification;

public function handle(RecipeCreated $event)
    {
        //
        Mail::send(new NotifyAdmin($event->title));
    }
```

Et en WriteLog.php :

```
use Illuminate\Support\facades\Log;

public function handle(RecipeCreated $event)
    {
        //
        Log::info('A recipe with title: ' .$recipe->title .'was added');
    }
```

Maintenant, on se déplace vers EventServiceProvider.php :

```
class EventServiceProvider extends ServiceProvider
{
    /**
     * The event listener mappings for the application.
     *
     * @var array
     */
    protected $listen = [
        Registered::class => [
            SendEmailVerificationNotification::class,
        ],
        'App\events\RecipeCreated' =>
        [
            'App\Listeners\writeLog',
            'App\Listeners\SendNotification',
```

```
        ],
    ];
}
```

Maintenant, de la même manière de la création du dernier événement, on ajoutera un nouvel évènement abonné(subsciber) dont le contenu sera :

```
php artisan make:event RecipeEdited
```

```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 RecipeEdited
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    public $title;
    /**
     * Create a new event instance.
     *
     * @return void
     */
    public function __construct($title)
    {
        //
        return $this->title=$title;
    }

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

```
}
```

Maintenant, en RecipesController.php :

On fera une inclusion de cet événement ;

```
use App\Events\RecipeEdited;
```

Mais, on créera pas un listener cette fois, mais on créera en dossier listeners un nouveau fichier :

RecipeEventSubsciber.php

```
<?php
namespace App\Listeners;

use Illuminate\Support\Facades\Log;

class RecipeEventSubscriber {


    public function handleRecipeCreated($event) {

        Log::info('Inside Subscriber. A Recipe with title ' . $event-
>title . ' was added');
    }

    public function handleRecipeEdited($event) {

        Log::info('Inside Subscriber. A Recipe with title ' . $event-
>title . ' was edited');
    }

    public function subscribe( $events) {

        $events->listen(
            'App\Events\RecipeCreated',
            'App\Listeners\RecipeEventSubscriber@handleRecipeCreated'
        );

        $events->listen(
            'App\Events\RecipeEdited',
            'App\Listeners\RecipeEventSubscriber@handleRecipeEdited'
        );
```

```
    }
}
```

Maintenant, on se déplace encrore une foir vers EventServiceProvider.php

```
<?php

namespace App\Providers;

use Illuminate\Auth\Events\Registered;
use Illuminate\Auth\Listeners\SendEmailVerificationNotification;
use Illuminate\Foundation\Support\Providers\EventServiceProvider as ServiceProvid
er;
use Illuminate\Support\Facades\Event;

class EventServiceProvider extends ServiceProvider
{
    /**
     * The event listener mappings for the application.
     *
     * @var array
     */
    protected $listen = [
        Registered::class => [
            SendEmailVerificationNotification::class,
        ],
        'App\events\RecipeCreated' =>
        [
            //'App\Listeners\writeLog',
            'App\Listeners\SendNotification',

        ],

    ];
    protected $subscribe = [

        'App\Listeners\RecipeEventSubscriber'
    ];

    /**
     * Register any events for your application.
     *
     * @return void
     */
    public function boot()
```

```
    {
        parent::boot();

        //
    }
}
```

On trouve une problématique, c'est que quand on supprime une recette, alors cette suppression est permanente ; pour régler cela et pour garder la traçabilité on fait :

```
php artisan make:migration add_softdelete_to_Recipes --table=Recipes
```

Remarque:

On aimera créer une interface d'admin :

Donc, pour atteindre cela, on doit faire

```
$ php artisan make:migration add_type_to_users --table users
```

Le Contenu du fichier qu'on vient de générer est :

```
<?php

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

class AddTypeToUsers extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::table('users', function (Blueprint $table) {
            //
            $table->integer('type')->default(2);
        });
    }

    /**
     * Reverse the migrations.
```

```
     *
     * @return void
     */
    public function down()
    {
        Schema::table('users', function (Blueprint $table) {
            //
            $table->dropColumn('type');
        });
    }
}
```

On fait la migration avec : php artisan migrate en Git !

Cette opération nous ajouter une nouvelle colonne qui s'appelle 'type'

Maintenant, on se dirigera vers app.blade.php et on ajoutera ces lignes qui permettront d'ajouter deux possibilités en cliquant sur le nom d'user, avec une restriction sur les recettes supprimées qui peuvent être vus que par l'admin.

```
<a class="dropdown-item"href="{{route('Recipes.index')}}">Recipes</a>
@if (Auth::user()->type == 1)
     <a class="dropdown-item"href="#">Deleted Recipes</a>
 @endif
```

Maintenant en Web.php, on ajoute cette route :

```
Route::namespace('Admin')->prefix('admin')->group(function()
{
    Route::get('/deleted_recipes','RecipesController@index')-
>name('admin.recipes.index');


});
```

On ajoute un nouveau controlleur en Git :

`php artisan make:controller Admin/RecipesController`

dont le contenu (`RecipesController`) dans (`Admin`) sera:

```
<?php

namespace App\Http\Controllers\Admin;
```

```
use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use App\Recipe;

class RecipesController extends Controller
{
    //
    public function index()
    {
        $recipes = Recipe::onlyTrashed()
         ->with('user')

      ->paginate(7);
            return view('admin.recipes.index',[
                'recipes'=>$recipes
            ]);
    }
}
```

Puis, on crée en dossier views un nouveau dossier admin, dedans on crée un nouveau dossier recipes dont on créera dedans un nouveau fichier nommé index.blade.php dont le contenu est :

```
@extends('layouts.app')

@section('content')
<div class="container">
    <div class="row justify-content-center">
        <div class="col-md-8">
            <div class="card">
                <div class="card-header">{{ __('Deleted Recipes') }}
                </div>

                <div class="card-body">
                    <table class="table">
                        <thead>
                            <tr>
                                <th>Title</th>
                                <th>Type</th>
                                <th>User</th>

                            </tr>
                        </thead>
                        <tbody>
                            @foreach ($recipes as $recipe)
```

```
                            <tr>
                                <td>{{$recipe->title}}</td>
                                <td>{{$recipe->type}}</td>
                                <td>{{$recipe->user->name}}</td>
                            </tr>
                            @endforeach
                        </tbody>
                    </table>
                    {{$recipes->links()}}


                </div>
            </div>
        </div>
    </div>
</div>
@endsection
```

Ensuite,on finit en app.blade.php :

```
<a class="dropdown-
item"href="{{route('admin.recipes.index')}}">Deleted Recipes</a>
```

Problématique:

On peut accéder aux données d'un URL : donc faille de sécurité :

Veillons à essayer de trouver une solution :

En web.php :

```
Route::namespace('Admin')->prefix('admin')->middleware(['auth'])-
>group(function()
{
    Route::get('/deleted_recipes','RecipesController@index')-
>name('admin.recipes.index');

});
```

Meme si, on peut accéder à partir d'un autre utilisateur :

Pour cela on créera un middleware qui s'appelle CheckAdmin en Git par :

```
php artisan make:middleware CheckAdmin
```

dont le contenu est :

```php
<?php

namespace App\Http\Middleware;

use Closure;

class CheckAdmin
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        if ($request->user()->type !=1)
        {
            abort(404);
        }
        return $next($request);
    }
}
```

En web.php:

```php
use App\Http\Middleware\CheckAdmin;
```

```php
      Route::namespace('Admin')->prefix('admin')->middleware(['auth',CheckAdmin::class])->group(function()

{
    Route::get('/deleted_recipes','RecipesController@index')->name('admin.recipes.index');

});
```

```
Route::namespace('Admin')->prefix('admin')-
>middleware(['auth',CheckAdmin::class])->group(function()
{
    Route::get('/deleted_recipes','RecipesController@index')-
>name('admin.recipes.index');
    Route::put('/recipes/restore/{id}','RecipesController@restore')-
>name('admin.recipes.restore');
    Route::delete('/recipes/delee/{id}','RecipesController@delete')-
>name('admin.recipes.delete');

});
});
```

Maintenant pour ajouter la possiblité d'annuler la suppression ou resupprimer d'une façon permanente une recette :

```
<?php
<?php

namespace App\Http\Controllers\Admin;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use App\Recipe;

class RecipesController extends Controller
{
    //
    public function index()
    {
        $recipes = Recipe::onlyTrashed()
            ->with('user')
            ->paginate(7);
            return view('admin.recipes.index',[
                'recipes'=>$recipes
            ]);
    }
    public function restore($id)
    {
        $recipe=Recipe::withTrashed()->findorfail($id);
        $recipe->restore();
        return redirect()->route('admin.recipes.index')-
>with('status','recipe restored successfully');
    }
```

```
    public function delete($id)
    {
        $recipe=Recipe::withTrashed()->findorfail($id);
        $recipe->forcedelete();
        return redirect()->route('admin.recipes.index')-
>with('status','recipe deleted successfully');
    }
}
```

En index.blade.php qui existe en admin/recipes :

```
@extends('layouts.app')

@section('content')
<div class="container">
    <div class="row justify-content-center">
        <div class="col-md-8">
            <div class="card">
                <div class="card-header">
                    Deleted Recipes:
                </div>

                <div class="card-body">
                    <table class="table">
                        <thead>
                            <tr>
                                <th>Title</th>
                                <th>Type</th>
                                <th>User</th>
                                <th>Action</th>
                            </tr>
                        </thead>
                        <tbody>
                            @foreach( $recipe as $recipe)
                                <tr>
                                    <td>
                                        {{ $recipe->title }}
                                    </td>
                                    <td>
                                        {{ $recipe->type}}
                                    </td>
                                    <td>
                                        {{ $recipe->user->name}}
                                    </td>
```

```
                                    <td>

                                    <form action=" {{ route('admin.recipes.restor
e', [$recipe]) }}" method="POST" style="display:inline-block">
                                                @method('PUT')
                                                @csrf
                                                <button class="btn btn-sm btn-
danger">Restore</button>
                                            </form>
                                    <form action=" {{ route('admin.recipes.delete
', [$recipe]) }}" method="POST" style="display:inline-block">
                                                @method('DELETE')
                                                @csrf
                                                <button class="btn btn-sm btn-
danger">Delete</button>
                                    </form>
                                    </td>
                                </tr>
                            @endforeach
                        </tbody>
                    </table>

                    {{ $recipes->links() }}
                </div>
            </div>
        </div>
    </div>
</div>
@endsection
```