

TP 1 : Préparer l'environnement de Laravel

1. Installation complète de Laravel (Composer, commandes PHP Artisan)

Pour travailler avec Laravel, vous avez besoin de :

- a. PHP et un serveur Local (Apache), permettant l'exécution du PHP, je vous recommande d'installer Xampp.

XAMPP est mis à disposition par le projet à but non lucratif Apache Friends. Les versions avec PHP 8.1.12 est téléchargeable gratuitement sous www.apachefriends.org/fr/download.html

Les extensions PDO, Tokenizer, OpenSSL et Mbstring de PHP doivent être activées. Dans le fichier de configuration de PHP `php.ini`, décommenter les lignes ci-dessous en enlevant le point virgule au début des lignes :

;extension=php_mbstring.dll

;extension=php_openssl.dll

- ## b. Composer, un gestionnaire de dépendances pour PHP

Pour installer Composer, il suffit de télécharger un installateur <https://getcomposer.org/download/> et téléchargez Composer-Setup.exe.

Vérifiez lors de l'installation que le chemin par défaut vers PHP est bien **C:\xampp\php**

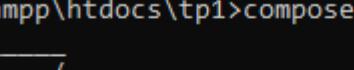
Page 1 of 1

, car Composer est un fichier PHP et a besoin d'être exécuté

- c. IDE et Editeurs de code (Visual Studio Code)
 - d. Navigateurs Web : Laravel est compatible sur tous les navigateurs récents (Chrome, Firefox ou Safari)

Pour vérifier que tout fonctionne exécuter composer sur la ligne de commande comme suit:

```
C:\xampp\htdocs\tp1>composer
```



```
Composer version 2.4.4 2022-10-27 14:39:29
```

Puis la commande `php --version`

```
C:\xampp\htdocs\tp1> php --version
PHP 8.1.12 (cli) (built: Oct 25 2022 18:16:21) (ZTS Visual C++ 2019 x64)
Copyright (c) The PHP Group
Zend Engine v4.1.12, Copyright (c) Zend Technologies
```

2. Création d'un premier projet

Après avoir installé Xampp et Composer, vous pouvez créer un nouveau projet Laravel via la commande Composer `create-project`.

Dirigez-vous vers votre dossier à la racine de votre serveur www ou htdocs et exécuter la commande.

```
C:\xampp\htdocs>composer create-project laravel/laravel tp1
```

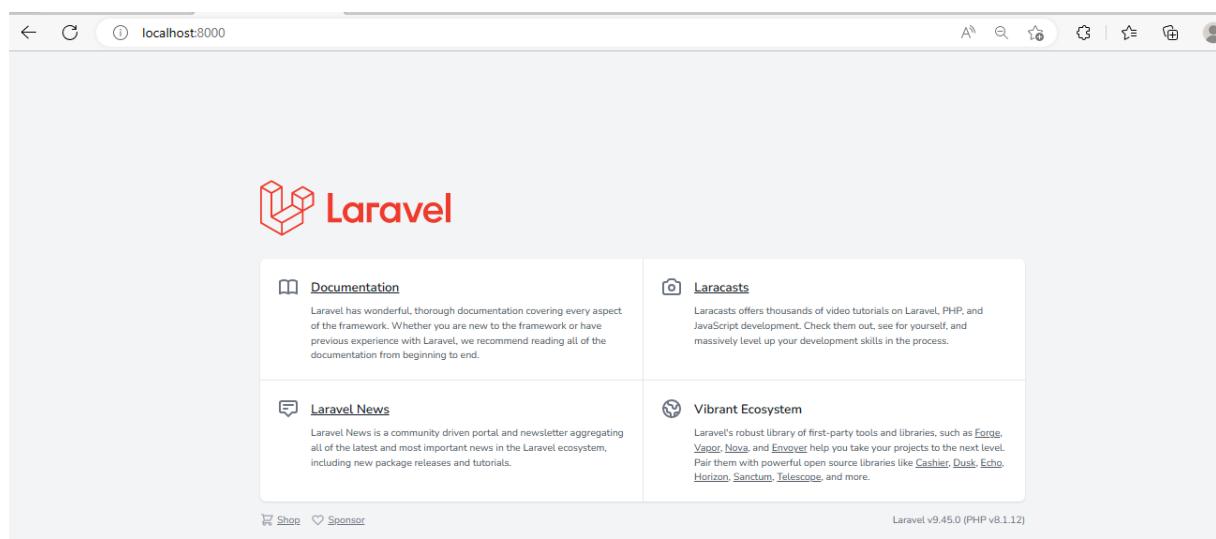
Ici nous avons demandé à composer de créer une installation Laravel dans le dossier «tp1». Cette commande installera automatiquement la dernière version stable de Laravel (9.45.0).

3. Lancement du serveur Laravel

Après la création du projet, démarrez le serveur de développement local de Laravel à l'aide de la commande `serve` de la CLI Artisan de Laravel :

```
php artisan serve
```

Une fois que vous avez démarré le serveur de développement Artisan, votre application sera accessible dans votre navigateur Web à l'adresse `http://localhost:8000`. Ensuite, vous êtes prêt à faire vos prochains pas dans l'écosystème Laravel. Bien entendu, vous voudrez peut-être aussi configurer une base de données.



Gestion du routage

1. Objectifs

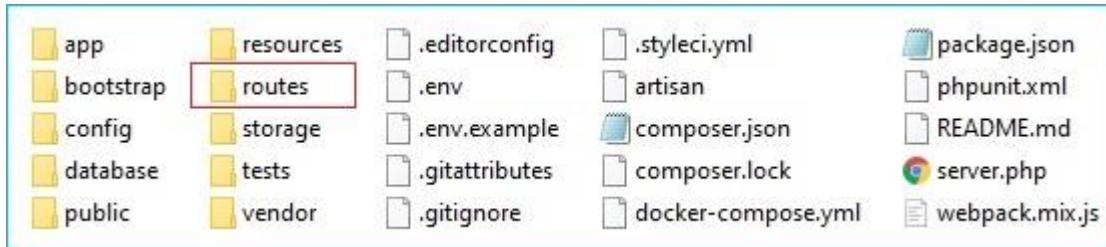
- Connaitre le système de routage Laravel

2. Qu'est-ce que le routage sous Laravel

- Laravel fournit un système de routes simple. Déclarer une route permet de lier une URI (identifiant de ressource uniforme, autrement dit la partie de l'adresse qui suit le nom de domaine) à un code à exécuter.
- Le routage est un moyen de créer une URL de requête pour votre application. La meilleure chose à propos du routage Laravel est que vous êtes libre de définir vos routes comme vous le souhaitez.
- Le routage est l'un des composants clés du framework Laravel, c'est simplement un mécanisme qui effectue le mappage de vos requêtes vers une action de contrôleur spécifique.
- Toutes les routes Laravel sont définies dans le fichier situé sous forme de fichier /routes/web.php , qui est automatiquement chargé par le framework
- Dans Laravel, toutes les demandes sont mappées avec des routes, toutes les routes sont créées dans le dossier racine. Pour votre application Web, vous pouvez définir des itinéraires liés à l'application dans le fichier web.php tandis que tous les itinéraires pour l'API sont définis dans le fichier api.php.

3. Présentation

- Lorsque Laravel est installé et configuré, vous aurez tous les fichiers dont vous avez besoin dans votre dossier pour commencer un projet Laravel.
- Lorsqu'on affiche une page Web Laravel dans le navigateur, l'URL ne se terminera pas par le nom d'un fichier .html ni .php. Plutôt, il s'agira d'une suite de mots ressemblant à des dossiers et sous-dossiers, comme par exemple <http://127.0.0.1:8000/inscription>.
- Toutes les routes d'application sont enregistrées dans le fichier web.php . Ce fichier indique à Laravel les URI auxquels il doit répondre et le contrôleur associé lui donnera un appel particulier.



- Par défaut, Laravel s'installe avec un dossier de routes, pour gérer divers besoins des itinéraires, dans son répertoire racine, ce dossier contient quatre fichiers, api.php (utilisé pour gérer les routes de l'API), channels.php , console.php et web.php (gère les routes normales).
- Pour créer une route, il faut ainsi appeler la classe Route avec la méthode HTTP souhaitée (get par exemple). Indiquez à cette méthode l'URI concernée et le retour à afficher pour le visiteur comme dans l'exemple ci-dessous.
- Fichier routes/web.php

```
Route::get('/', function () {
    return 'Welcome to Laravel Site.';
});
```

- Dans cet article, nous nous concentrerons principalement sur le web.php, car c'est là que tous nos itinéraires vont vivre.
- Le routage dans Laravel comprend trois catégories:
 - Routage de base
 - Routes nommés
 - Paramètres de Route

4. Routage de base

- Route basique**
- Une route Laravel très basique est simplement exprimée comme suit:

```
Route::get('/', function () {
    return 'Welcome to Laravel Site.';
});
```

- Ici, “Route” est une classe qui a une méthode statique “get” qui renvoie une méthode “view” qui présente une page Web.
- Lorsqu'un utilisateur visite la page d'accueil, il est redirigé vers la page “Welcome” page. Ce “welcome” est en fait un fichier PHP: “welcome.blade.php“.

- Il a été stocké par défaut dans le dossier “views” pendant l’installation de Laravel. Lorsque nous discutons du concept du moteur de modèle “view” et “blade”, vous comprendrez parfaitement ce qui se passe dans la couche de présentation.
- Plusieurs routes et paramètre de route comme suit:
 - À l’installation Laravel a une seule route qui correspond à l’URL de base composée uniquement du nom de domaine. Voyons maintenant comment créer d’autres routes.
 - Imaginons que nous ayons trois pages qui doivent être affichées avec ces URL :
 1. <http://ista.ma/1>
 2. <http://ista.ma/2>
 3. <http://ista.ma/3>

```
Route::get('1', function() { return 'Je suis la page 1 !';
});
Route::get('2', function() { return 'Je suis la page 2 !';
});
Route::get('3', function() { return 'Je suis la page 3 !';
});
```

- **afficher des pages “view”**



- Si vous ouvrez le dossier Route, vous verrez quatre fichiers.
- Ceux-ci incluent :
 - api.php,
 - channels.php,
 - console.php
 - web.php
- C’est ce fichier web.php que nous voulons examiner maintenant.
- Les routes utilisées sont définies dans le fichier routes\web.php.
- Par défaut, nous avons accès à une route pour la page d’accueil qui est écrite sous cette forme :

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

- Comme Laravel est explicite vous pouvez déjà deviner à quoi sert ce code :
 - **Route** : on utilise le routeur, appel statique à la classe qui gère le système de routage ;
 - **get** : on regarde si la requête a la méthode “**get**” ;
 - ‘/’ : on regarde si l’URL comporte uniquement le nom de domaine ;
 - ‘**return**’ : retourne une vue (view) à partir du fichier “**welcome**”.

o **Création de nouvelles routes**

- Il est possible et même nécessaire d’ajouter d’autres routes pour diriger les demandes vers la bonne méthode d’un contrôleur ou encore la fonction de rappel (callback function), comme c’est le cas dans le code précédent, vers la bonne vue.

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

o **Verbes de route**

- Vous avez peut-être remarqué que nous utilisons **Route :: get** dans nos définitions d’itinéraire.
- Ça signifie nous disons à **Laravel** de ne faire correspondre ces routes que lorsque la requête HTTP utilise la méthode **GET**.
- Mais que se passe-t-il s’il s’agit d’un formulaire **POST**, ou peut-être d’un **JavaScript** envoyant **PUT** ou **DELETE**.
- Il existe quelques autres options pour les méthodes permettant d’appeler une définition d’itinéraire, comme illustré ci-dessous:

```
Route::get('/', function () {
    return 'Hello, World!';
});
Route::post('/', function () {});
Route::put('/', function () {});
Route::delete('/', function () {});
Route::any('/', function () {});
```

```
Route::match(['get', 'post'], '/', function () {});
```

- **GET:** Ceci est principalement utilisé pour récupérer les données du serveur, sans modifier les données et les renvoyer à l'utilisateur.
- **POST:** Cela nous permet de créer de nouvelles ressources ou données sur le serveur, bien que cela puisse être utilisé uniquement pour envoyer des données, pour un traitement ultérieur (cela pourrait être la validation des données, comme dans le processus de connexion). Il est considéré comme plus sûr que GET pour l'envoi de données sensibles.
- **PUT:** Fonctionne comme le POST dans le sens où il vous permet d'envoyer des données au serveur, généralement pour mettre à jour une ressource existante au lieu de la créer. Vous le mettez essentiellement.

5. Redirection

Si vous définissez une route qui redirige vers un autre URI, vous pouvez utiliser la méthode `Route::redirect`. Cette méthode fournit un raccourci pratique pour que vous n'ayez pas à définir une route ou un contrôleur complet pour effectuer une simple redirection :

```
Route::redirect('/ici', '/là-bas');
```

6. Affichage des routes

Si votre itinéraire ne doit renvoyer qu'une vue, vous pouvez utiliser la méthode `Route::view`. Comme la méthode `redirect`, cette méthode fournit un raccourci simple pour que vous n'ayez pas à définir une route ou un contrôleur complet. La méthode `view` accepte un URI comme premier argument et un nom de vue comme second argument. De plus, vous pouvez fournir un tableau de données à transmettre à la vue en tant que troisième argument facultatif :

```
Route::view('/welcome', 'welcome');
```

```
Route::view('/welcome', 'welcome', ['name' => 'steve']);
```

7. Liste des routes

La commande Artisan `route:list` peut facilement fournir une vue d'ensemble de toutes les routes définies par votre application :

```
php artisan route:list
```

8. Les paramètres des routes

- Partant de l'exemple précédent on peut se poser une question : est-il vraiment indispensable de créer trois routes alors que la seule différence tient à peu de choses : une valeur qui change ?
- On peut utiliser un paramètre pour une route qui accepte des éléments variables en utilisant des accolades. L'exemple devient alors:

```
Route::get('{n}', function($n) {  
    return 'Je suis la page ' . $n . ' !';  
});
```

- Les paramètres des routes sont toujours entre crochets {} et doivent être composés de caractères alphabétiques. Ne peut pas contenir les caractères “-” , peut utiliser le caractère “_” Substitution de caractères “-” .
- Les paramètres de routage sont injectés dans le contrôleur ou la fonction de rappel en fonction de leur ordre, plutôt que par les paramètres de la fonction de rappel ou du contrôleur.

o Paramètres obligatoires

- Par exemple, si vous souhaitez capturer l'ID utilisateur dans l'URL lorsque vous souhaitez obtenir les détails de l'utilisateur, vous pouvez vous référer aux pratiques suivantes:

```
Route::get('user/{id}', function ($id) {  
    return 'User ' . $id;  
});
```

- Plusieurs paramètres de routage peuvent également être capturés, par exemple:

```
Route::get('posts/{post}/comments/{comment}', function  
($postId, $commentId) {  
    //  
});
```

o Paramètres facultatifs

- Parfois, vous devez spécifier des paramètres de routage, mais les paramètres ne sont pas obligatoires, “?” est utilisé à ce moment. Le caractère est marqué après le nom du paramètre pour garantir que la valeur par défaut est donnée à la variable correspondante de l'itinéraire.

```
Route::get('user/{name?}', function ($name = null) {
    return $name;
});
Route::get('user/{name?}', function ($name = 'Tom') {
    return $name;
});
```

9. Routes nommées

- Il est parfois utile de nommer une route, par exemple pour générer une **URL** ou pour effectuer une **redirection**.
 - L'avantage d'utiliser une route nommée est que si nous changeons l'**URL** d'une route à l'avenir, nous n'aurons pas besoin de changer l'**URL** ou les redirections pointant vers cette route si nous utilisons une route nommée.
 - Les routes nommées sont créées en utilisant la méthode `name`
- La syntaxe (méthode name)**
- Laravel fournit un moyen simple de nommer vos routes par lequel nous n'avons pas à coder en dur l'URI dans nos vues Blade.
 - Vous pouvez déterminer le nom de l'itinéraire à l'aide de la méthode du nom dans l'exemple d'itinéraire ci-dessous.

```
Route::view('/home', 'pages.home')->name('home');
```

Générer une URL à l'aide d'une route nommée

- Pour générer une URL en utilisant le nom de la route
 - ``
- Si vous utilisez le nom de la route pour la redirection

```
Route::view('/login', 'pages/home')->name('home');

Route::get('/', function () {
    return redirect()->route('home');

});
```

- Si la route nommée définit des paramètres, vous pouvez passer les paramètres comme deuxième argument à la fonction route.

```
Route::get('/user/{id}/profile', function ($id) {
    return view('users.profile', [
        'id' => $id
    ]);
})->name('profile');
```

Les paramètres donnés seront automatiquement insérés dans l'URL générée dans leurs positions correctes :

```
<a href="{{route('profile', ['id' => 1])}}>home</a>
```

- Si vous transmettez des paramètres supplémentaires dans le tableau :

```
Route::get('/user/{id}/profile', function ($id)
{
    // ...
})->name('profile');
```

- Ces paires clé/valeur seront automatiquement ajoutées à la chaîne de requête de l'URL générée :

```
<a href="{{route('profile', ['id' => 1, 'photos' => 'yes'])}}>
    My Profile </a>
// /user/1/profile?photos=yes
```

10. Groupe des routes

- Les routes peuvent être regroupées pour éviter la répétition du code.
- Les groupes de routage vous permettent de partager des attributs de routage, tels que le middleware, sur un grand nombre de routages sans avoir à définir ces attributs sur chaque routage individuel.
- Disons que tous les URI avec un préfixe de /admin utilisent un certain middleware appelé admin et qu'ils vivent tous dans l'espace de noms App\Http\Controllers\Admin .
- Une manière propre de représenter cela en utilisant des groupes de routage est la suivante:

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

Route::group([
    'namespace' => 'Admin',
```

```

'middleware' => 'auth',
'prefix' => 'admin'
], function () {

Route::view('/home', 'pages.home',[ 'name' => 'steve' ])->name('home');
Route::get('/login/{id?}',function($id=null){
    return view('pages.home', [
        'name' => $id
    ]);
})->name('login');

});

```

11. Applications

o Exercice 01

1. Créer la liste des routes pour répondre aux URLs suivants :

- <http://localhost/laravel/public/>
- <http://localhost/laravel/public/login>
- <http://localhost/laravel/public/page/1>

```

Route::get('/', function()
{
    return 'accueil';
});

Route::get('login', function()
{
    return 'login';
});

Route::get('page/1', function()
{
    return 'page1';
});

```

2. Ajouter une route /test
3. Ajouter un paramètre qui sera affiché : /test/param
4. Utiliser une vue pour cette route
5. Lister les routes avec la commande artisan

o Exercice 02

Soit la route suivante :

```
Route::get('article/{n}', function($n) {
    return 'Je suis l\'article numéro ' . $n . ' !';
})->where('n', '[0-9]+');
```

1. Une seule URL ci-dessous est interceptée par cette route, laquelle ?

- .../article
- .../article/1254
- .../11
- .../article/un

Gestion du routage

1. Liaison modèle de route

Lorsque vous injectez un ID de modèle dans une route ou une action de contrôleur, vous allez souvent interroger la base de données pour récupérer le modèle qui correspond à cet ID. Laravel route model binding fournit un moyen pratique d'injecter automatiquement les instances de modèle directement dans vos routes. Par exemple, au lieu d'injecter l'ID d'un utilisateur, vous pouvez injecter l'instance entière du modèle User qui correspond à l'ID donné.

Créer un modèle

- Tous les Models de l'application Laravel sont stockés dans le dossier **App**.
- Le modèle peut être créé simplement en utilisant la commande `make: model` artisan comme suit:
- Syntaxe: **php artisan make:model Student**

```
<?php

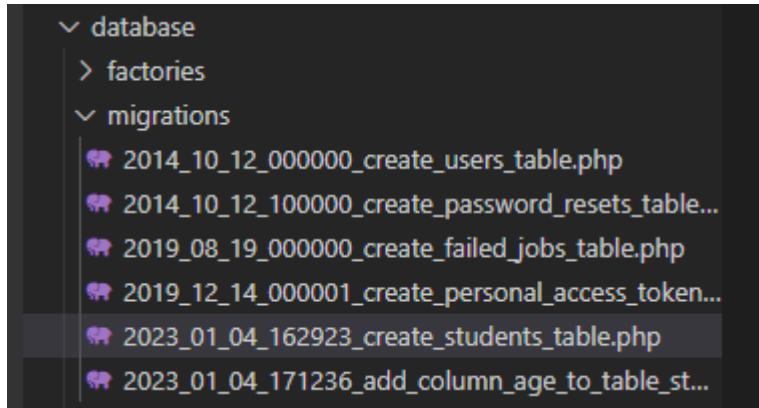
namespace App\Models;

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

class Student extends Model
{
    use HasFactory;
}
```

- Après créer une migration pour définir les propriétés de modèle **Student** en utilisant la commande :

php artisan make:migration create_students_table



- Commencer par définir les champs de votre base de données pour notre table **students** dans notre fichier de migration comme suit:

```
<?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('students', function (Blueprint $table) {
            $table->id();
            $table->string("name");
            $table->string("mail");
            $table->integer("phone");
            $table->string("section");
            $table->string("image");
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    }
```

```

    {
        Schema::dropIfExists('students');
    }
};
```

- Utilisez PHPMyAdmin pour créer une Base de données dont le nom “school”.
- Configurez la base de données dans votre projet Laravel ([.env](#))

```

.env
1 APP_NAME=Laravel
2 APP_ENV=local
3 APP_KEY=base64:XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
4 APP_DEBUG=true
5 APP_URL=http://localhost
6
7 LOG_CHANNEL=stack
8 LOG_DEPRECATIONS_CHANNEL=null
9 LOG_LEVEL=debug
10
11 DB_CONNECTION=mysql
12 DB_HOST=127.0.0.1
13 DB_PORT=3306
14 DB_DATABASE=school
15 DB_USERNAME=root
16 DB_PASSWORD=
17
18 BROADCAST_DRIVER=log
19 CACHE_DRIVER=file
20 FILESYSTEM_DISK=local
21 QUEUE_CONNECTION=sync
22 SESSION_DRIVER=file
23 SESSION_LIFETIME=120
24
```

- Lancez la migration : **php artisan migrate**
- Vous devez ainsi vous retrouver avec quatre tables dans votre base.

○ Liaison Implicite

Laravel résout automatiquement les modèles Eloquent définis dans les routes ou les actions de contrôleur dont les noms de variables à indication de type correspondent à un nom de segment de route. Par exemple :

```

use App\Models\Student;

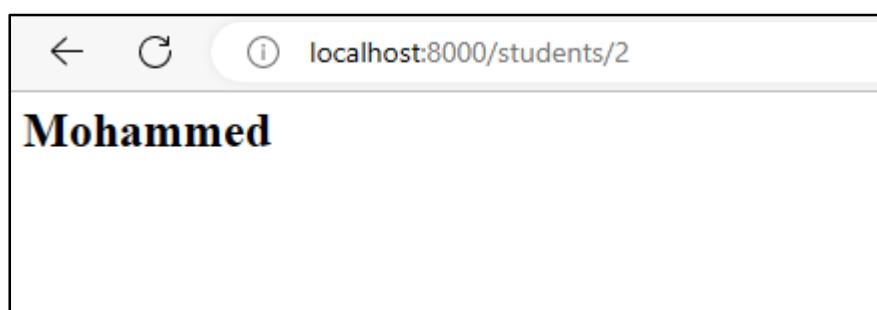
Route::get('students/{student}', function (Student $student)
{
    return $student->name;
});
```

Puisque la variable \$user est signalée comme étant le modèle Eloquent `App\Models\Student` et que le nom de la variable correspond au segment `{student}` Laravel injectera automatiquement l'instance de modèle dont l'ID correspond à la valeur correspondante de l'URI de la requête. Si une instance de modèle correspondante n'est pas trouvée dans la base de données, une réponse HTTP 404 sera automatiquement générée.

Bien sûr, la liaison implicite est également possible en utilisant les méthodes du contrôleur. Encore une fois, notez que le segment `{student}` correspond à la variable `$student` dans la fonction de rappel qui contient une indication de type `App\Models\Student`.

Comme nous avons lié tous `{student}` paramètres `{student}` au modèle `App\Student`, une instance `Student` sera injectée dans la route. Ainsi, par exemple, une demande de `students/1` injectera l'instance d'utilisateur de la base de données dont l'identifiant est 1.

Si une instance de modèle correspondante n'est pas trouvée dans la base de données, une réponse HTTP 404 sera automatiquement générée

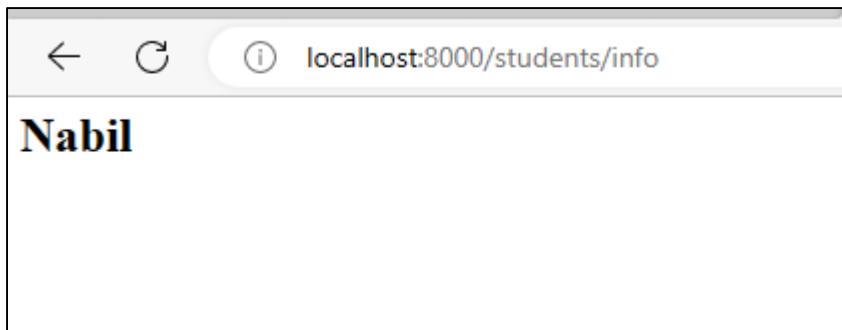


Personnalisation de la clé

Parfois, vous pouvez souhaiter résoudre des modèles Eloquent en utilisant une colonne **section** autre que **id**. Pour ce faire, vous pouvez spécifier la colonne dans la définition du paramètre de la route :

```
use App\Models\Student;

Route::get('students/{student :section}', function (Student $student)
{
    return $student->name;
});
```



1.1. Liaison Explicite

Pour enregistrer une liaison explicite, utilisez la méthode du modèle du routeur pour spécifier la classe d'un paramètre donné. Vous devez définir vos liaisons de modèle explicites dans la méthode de démarrage de la classe RouteServiceProvider

```
use App\Models\User;
use Illuminate\Support\Facades\Route;

/**
 * Define your route model bindings, pattern filters, etc.
 *
 * @return void
 */
public function boot()
{
    Route::model('user', User::class);

    // ...
}
```

Ensuite, définissez une route qui contient un paramètre {user} :

```
use App\Models\User;

Route::get('/users/{user}', function (User $user) {
    //
});
```

Comme nous avons lié tous {user} paramètres {user} au modèle App\User, une instance User sera injectée dans la route. Ainsi, par exemple, une demande de [profile/1](#) injectera l'instance d'utilisateur de la base de données dont l'identifiant est 1. Si une instance de modèle correspondante n'est pas trouvée dans la base de données, une réponse HTTP 404 sera automatiquement générée

2. Routes de repli (Fallback Routes)

En utilisant la méthode **Route::fallback**, vous pouvez définir une route qui sera exécutée lorsqu'aucune autre route ne correspond à la requête entrante. En règle générale, les requêtes non traitées renvoient automatiquement une page "404" via le gestionnaire d'exceptions de votre application. Cependant, comme vous définissez généralement la route de repli dans votre fichier **routes/web.php**, tous les middlewares du groupe middleware Web s'appliqueront à cette route.

```
Route::fallback(function () {
    return 'Error';
});
```

3. Usurpation (spoofing) de la méthode du formulaire

Les formulaires HTML ne prennent pas en charge les actions PUT, PATCH ou DELETE. Ainsi, lors de la définition de routes appelées PUT, PATCH ou DELETE à partir d'un formulaire HTML, vous devrez ajouter un champ masqué **_method** au formulaire. La valeur envoyée avec le champ **_method** sera utilisée comme méthode de requête HTTP :

```
<form action="/example" method="POST">
    <input type="hidden" name="_method" value="PUT">
    <input type="hidden" name="_token" value="{{ csrf_token() }}>
</form>
```

Pour plus de commodité, vous pouvez utiliser la directive Blade `@method` pour générer le champ `_method` de saisie :

```
<form action="/example" method="POST">
    @method('PUT')
    @csrf
</form>
```

1. Accès à la route courante

Vous pouvez utiliser les méthodes **current**, **currentRouteName** et **currentRouteAction** sur la façade Route pour accéder aux informations sur la route traitant la requête entrante.

Pour Obtenir le nom de la route dans le fichier Blade.

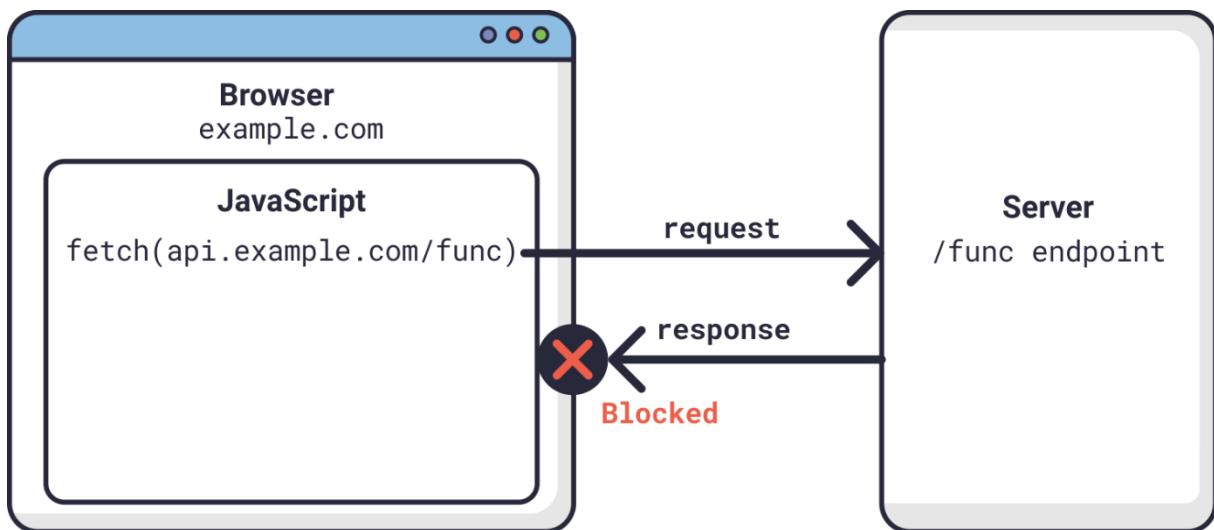
```
<body>
  <h1> Home page</h1>
  <h1>Route Name : {{Route::currentRouteName()}}</h1>

  <h1>Current Route Action : {{Route::currentRouteAction()}}</h1>

</body>
```

2. Cross-origin resource sharing (CORS)

Le « *Cross-origin resource sharing* » (CORS) ou « partage des ressources entre origines multiples » (en français, moins usité) est un mécanisme qui consiste à ajouter des en-têtes HTTP afin de permettre à un agent utilisateur d'accéder à des ressources d'un serveur situé sur une autre origine que le site courant. Un agent utilisateur réalise une requête HTTP **multi-origine** (*cross-origin*) lorsqu'il demande une ressource provenant d'un domaine, d'un protocole ou d'un port différent de ceux utilisés pour la page courante.



Prenons un exemple de requête multi-origine : une page HTML est servie depuis <http://domaine-a.com> contient un élément `` src ciblant <http://domaine-b.com/image.jpg>. Aujourd'hui, de nombreuses pages web chargent leurs ressources (feuilles CSS, images, scripts) à partir de domaines séparés (par exemple des CDN (*Content Delivery Network* en anglais ou « Réseau de diffusion de contenu »)).

5. Mise en cache des routes

Lors du déploiement de votre application en production, vous devez tirer parti du cache de routage de Laravel. L'utilisation du cache de route réduira considérablement le temps nécessaire pour enregistrer toutes les routes de votre application. Pour générer un cache de route, exécutez la commande Artisan **route:cache** :

```
php artisan route:cache
```

- Après avoir exécuté cette commande, votre fichier de routes en cache sera chargé à chaque requête. N'oubliez pas que si vous ajoutez de nouvelles routes, vous devrez générer un nouveau cache de routes. Pour cette raison, vous ne devez exécuter la commande **route:cache** pendant le déploiement de votre projet.
- Vous pouvez utiliser la commande **route:clear** pour vider le cache de route :

```
php artisan route:clear
```

Pour mettre en cache votre fichier de routes, vous devez utiliser toutes les routes de contrôleurs et de ressources (pas de fermetures de routes). Si votre application n'utilise aucune fermeture de route, vous pouvez exécuter **php artisan route:cache**. Laravel sérialisera les résultats de vos fichiers de routes. Si vous voulez supprimer le cache, exécutez **php artisan route:clear**.

Utilisation des Middleware

1. Introduction

Le Middleware fournit un mécanisme pratique pour inspecter et filtrer les requêtes HTTP entrantes dans votre application. Par exemple, Laravel inclut un middleware qui vérifie que l'utilisateur de votre application est authentifié. Si l'utilisateur n'est pas authentifié, le middleware le redirige vers l'écran de connexion de votre application. Cependant, si l'utilisateur est authentifié, le middleware permettra à la requête de se poursuivre dans l'application.

Des middlewares supplémentaires peuvent être écrits pour effectuer une variété de tâches autres que l'authentification. Par exemple, un middleware de journalisation peut enregistrer toutes les demandes entrantes dans votre application. Plusieurs middlewares sont inclus dans le framework Laravel, notamment des middlewares pour l'authentification et la protection CSRF. Tous ces middlewares sont situés dans le répertoire **app/Http/Middleware**.

2. Définition

Pour créer un nouvel middleware, utilisez la commande

```
php artisan make:middleware EnsureTokenIsValid
```

Cette commande va placer une nouvelle classe **EnsureTokenIsValid** dans votre répertoire **app/Http/Middleware**. Dans ce middleware, nous n'autoriserons l'accès à la route que si l'entrée du **token** fourni correspond à une valeur spécifiée. Sinon, nous redirigerons les utilisateurs vers l'URI **home** :

```
<?php

namespace App\Http\Middleware;

use Closure;

class EnsureTokenIsValid
{
    /**
     * Handle an incoming request.

```

```

*
* @param  \Illuminate\Http\Request  $request
* @param  \Closure  $next
* @return mixed
*/
public function handle($request, Closure $next)
{
    if ($request->input('token') !== 'my-secret-token') {
        return redirect()->route('welcome');
    }

    return $next($request);
}

```

Comme vous pouvez le voir, si le **token** donné ne correspond pas à notre **token** secret, le middleware renverra une redirection HTTP au client ; sinon, la requête sera transmise plus loin dans l'application. Pour transmettre la requête plus loin dans l'application (en permettant au middleware de "passer"), vous devez appeler la callback **\$next** avec la requête **\$request**.

Il est préférable d'envisager le middleware comme une série de "couches" que les requêtes HTTP doivent traverser avant d'atteindre votre application. Chaque couche peut examiner la requête et même la rejeter entièrement.

```

Route::get('students/{student}', function (Student $student) {
    return '<h1>' . $student->name . '</h1>';
})->name('student')->middleware('token');

```

Ce middleware vérifie si le paramètre **token** passé dans l'URL correspond bien à la chaîne **'my-secret-token'**. Sinon on se redirige vers la page d'accueil.

Exemple : Visitez l'URL suivante pour tester le Middleware avec des paramètres localhost:8000/students/1?token=my-secret-token

3. Enregistrement

3.1 Middleware Globale

Si vous souhaitez qu'un middleware s'exécute lors de chaque requête HTTP adressée à votre application, répertoriez la classe middleware dans la propriété **\$middleware** de votre classe **app/Http/Kernel.php**.

```
protected $middleware = [
    // \App\Http\Middleware\TrustHosts::class,
    \App\Http\Middleware\TrustProxies::class,
    \App\Http\Middleware\EnsureTokenIsValid::class,
    \Illuminate\Http\Middleware\HandleCors::class,
    \App\Http\Middleware\PreventRequestsDuringMaintenance::class,
    \Illuminate\Foundation\Http\Middleware\ValidatePostSize::class,
    \App\Http\Middleware\TrimStrings::class,
    \Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull::class,
];
```

3.2 Affectation de middleware aux routes

Si vous souhaitez affecter un **middleware** à des routes spécifiques, vous devez d'abord affecter au middleware une **clé** dans le fichier **app/Http/Kernel.php** de votre application. Par défaut, la propriété **\$routeMiddleware** de cette classe contient des entrées pour le **middleware** inclus avec **Laravel**. Vous pouvez ajouter votre propre middleware à cette liste et lui attribuer une clé de votre choix :

```
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'auth.session' => \Illuminate\Session\Middleware\AuthenticateSession::class,
    'cache.headers' => \Illuminate\Http\Middleware\SetCacheHeaders::class,
    'can' => \Illuminate\Auth\Middleware\Authorize::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'password.confirm' => \Illuminate\Auth\Middleware\RequirePassword::class,
    'signed' => \App\Http\Middleware\ValidateSignature::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
    'verified' => \Illuminate\Auth\Middleware\EnsureEmailIsVerified::class,
    'token' => \App\Http\Middleware\EnsureTokenIsValid::class,
];
}
```

3.3 Groupe de Middlewares

Parfois, vous souhaiterez peut-être regrouper plusieurs middlewares sous une seule clé pour faciliter leur affectation aux routes. Vous pouvez accomplir cela en utilisant la propriété **\$middlewareGroups** de votre noyau HTTP.

Prêt à l'emploi, Laravel est livré avec des groupes **middlewares Web** et **API** qui contiennent des middlewares communs que vous voudrez peut-être appliquer à vos routes **Web** et **API**. N'oubliez pas que ces groupes de middleware sont automatiquement appliqués par le fournisseur de services **App\Providers\RouteServiceProvider** de votre application aux routes dans vos fichiers de route Web et API correspondants :

```
protected $middlewareGroups = [
    'web' => [
        \App\Http\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
        \Illuminate\Session\Middleware\StartSession::class,
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,
        \App\Http\Middleware\VerifyCsrfToken::class,
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
    ],
    'api' => [
        //
\ Laravel\Sanctum\Http\Middleware\EnsureFrontendRequestsAreStateful::class,
        'throttle:api',
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
    ],
];
```

4. Activation du middleware

Le middleware peut être activé sur n'importe quelle route. Dans le fichier **routes/web.php**, nous pouvons utiliser la fonction **->middleware()** pour demander à Laravel d'activer le middleware :

```
Route::get('/profile', function () {
    //
})->middleware('auth');
```

Vous pouvez affecter plusieurs middlewares à la route en transmettant un tableau de noms de middlewares à la méthode middleware :

```
Route::get('/', function () {
    //
})->middleware(['first', 'second']);
```

5. L'exclusion de middleware

Lors de l'attribution d'un **middleware** à un groupe de routes, vous devrez peut-être parfois empêcher le middleware de s'appliquer sur une route individuelle au sein du groupe. Vous pouvez accomplir cela en utilisant la méthode **withoutMiddleware**

```
Route::get('/profile', function () {
    //
})->withoutMiddleware('token');
```

La méthode **withoutMiddleware** ne peut supprimer que le middleware de route et ne s'applique pas au middleware global.

6. Paramétrage

Le **middleware** peut également recevoir des paramètres supplémentaires. Par exemple, si votre application doit vérifier que l'utilisateur authentifié a un "rôle"

donné avant d'effectuer une action donnée, vous pouvez créer un middleware **EnsureUserHasRole** qui reçoit un nom de rôle comme argument supplémentaire. Des paramètres middleware supplémentaires seront transmis au middleware après l'argument **\$next** :

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;

class EnsureUserHasRole
{
    /**
     * Handle an incoming request.
     *
     * @param \Illuminate\Http\Request $request
     * @param \Closure(\Illuminate\Http\Request):
     * (\Illuminate\Http\Response|\Illuminate\Http\RedirectResponse)
     * $next
     * @return \Illuminate\Http\Response|\Illuminate\Http\RedirectResponse
     */
    public function handle(Request $request, Closure
    $next,$role)

    {
        if ($role=="admin"){
            return $next($request);
        }
        else

            return redirect()->route('welcome');

    }
}
```

Les paramètres du **middleware** peuvent être spécifiés lors de la définition de la route en séparant le nom du middleware et les paramètres par un **:** .

Plusieurs paramètres doivent être délimités par des virgules :

```
Route::get('/profile', function () {  
    return "my profile";})->middleware('role:admin') ;
```

7. Terminable Middleware

Un **middleware terminable** exécute une tâche après l'envoi de la réponse au navigateur. Ceci peut être accompli en créant un middleware avec la méthode **terminate** dans le middleware. **Le middleware terminable** doit être enregistré dans le middleware global. La méthode **terminate** recevra deux arguments : **\$request** et **\$response**. La méthode **terminate** peut être créée comme indiqué dans le code suivant.

Exemple

Etape 1 : Créez **TerminateMiddleware** en exécutant la commande suivante.

```
php artisan make:middleware TerminateMiddleware
```

Etape 2 : Copiez le code suivant dans le **TerminateMiddleware** nouvellement créé à l'adresse **app/Http/Middleware/TerminateMiddleware.php**

```
<?php  
  
namespace App\Http\Middleware;  
use Closure;  
  
class TerminateMiddleware {  
    public function handle($request, Closure $next) {  
        echo "Executing statements of handle method of  
TerminateMiddleware.";  
        return $next($request);  
    }  
  
    public function terminate($request, $response) {  
        echo "<br>Executing statements of terminate method of  
TerminateMiddleware.";  
    }  
}
```

Etape 3 : Enregistrez le **TerminateMiddleware** dans le fichier **app\Http\Kernel.php**. Ajoutez la ligne surlignée en jaune dans ce fichier pour enregistrer **TerminateMiddleware**.

```
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'auth.session' => \Illuminate\Session\Middleware\AuthenticateSession::class,
    'cache.headers' => \Illuminate\Http\Middleware\SetCacheHeaders::class,
    'can' => \Illuminate\Auth\Middleware\Authorize::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'password.confirm' => \Illuminate\Auth\Middleware\RequirePassword::class,
    'signed' => \App\Http\Middleware\ValidateSignature::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
    'verified' => \Illuminate\Auth\Middleware\EnsureEmailIsVerified::class,
    'token' => \App\Http\Middleware\EnsureTokenIsValid::class,
    ''terminate'=>\App\Http\Middleware\TerminateMiddleware::class
];
}
```

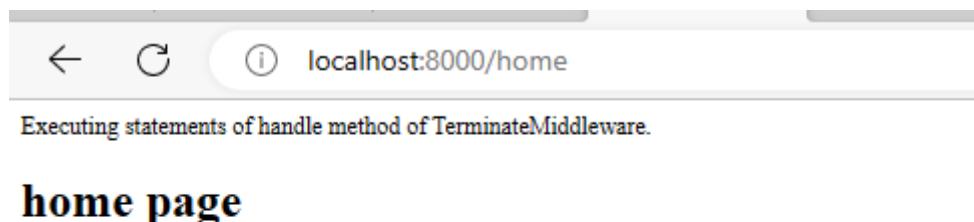
Etape 4 : Ajoutez la ligne de code suivante dans le fichier **app/routes/web.php**.

```
Route::get('/home', function () {
    return '<h1>home page</h1>';
})->name('welcome')->middleware('terminate');
```

Etape 5 : Visitez l'URL suivante pour tester le Terminable Middleware.

localhost:8000/home

La sortie apparaîtra comme indiqué dans l'image suivante.



Executing statements of handle method of TerminateMiddleware.

home page

Executing statements of terminate method of TerminateMiddleware.

Protection CSRF

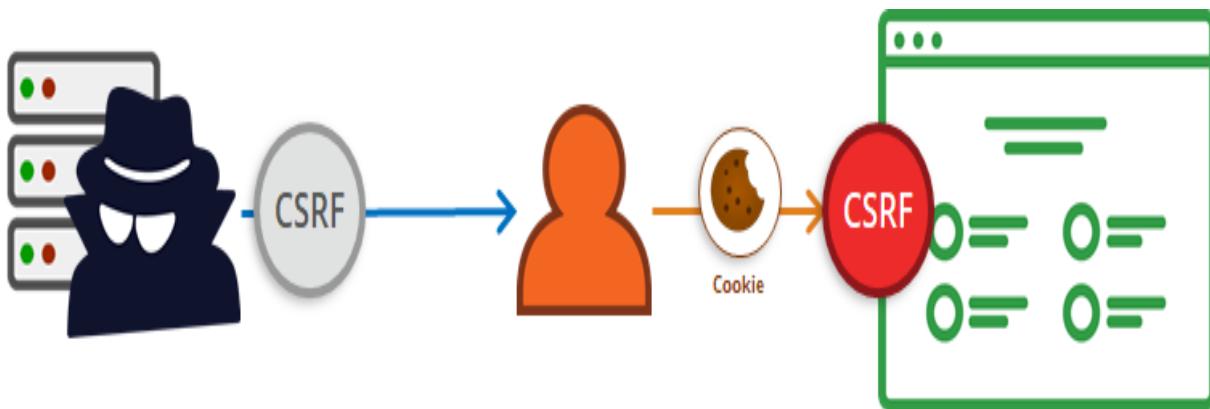
Sécuriser les formulaires avec Laravel

1. Objectifs

- Connaitre les méthodes de sécurisation d'un formulaire avec Laravel.
- Etre capable de sécuriser un formulaire laravel.

2. Présentation

- Un formulaire est un vecteur d'attaque potentiel pour une personne malveillante.
- Laravel propose par défaut un mécanisme de défense face aux attaques de type **CSRF** (Cross-Site Request Forgery).
- Dans ce formulaire on va voir comment protéger très facilement une application Laravel contre les attaques CSRF (Cross Site Request Forgery)
- La forme complète de CSRF est la falsification de requêtes intersites. Il s'agit d'un type d'attaque en ligne dans lequel l'attaquant envoie des demandes en tant qu'utilisateur autorisé à un système en obtenant des informations d'accès d'un utilisateur particulier de ce système et effectue différents types d'activités malveillantes en utilisant l'identité de cet utilisateur. L'impact de cette attaque dépend des priviléges de la victime sur le système.
- Si la victime est un utilisateur normal, cela n'affectera que les données personnelles de la victime. Mais si la victime est l'administrateur du système, l'attaquant peut endommager l'ensemble du système.



3. Fonctionnement du CSRF

- La falsification de requêtes intersites (CSRF) est un type d'attaque effectué par l'attaquant pour envoyer des requêtes à un système avec l'aide d'un utilisateur autorisé auquel le système fait confiance.
- Laravel fournit une protection contre les attaques CSRF en générant un jeton **CSRF**. Ce jeton CSRF est généré automatiquement pour chaque utilisateur. Ce jeton n'est rien d'autre qu'une chaîne aléatoire gérée par l'application Laravel pour vérifier les demandes des utilisateurs.
- Cette protection de jeton CSRF peut être appliquée à n'importe quel formulaire HTML dans l'application Laravel en spécifiant un champ de formulaire caché du jeton CSRF.
- Une des manières efficaces de se prémunir contre les attaques **CSRF** est d'accompagner chaque formulaire à protéger d'un jeton (en anglais “token”) d'identification de session.
- Pour ce faire, lorsqu'un visiteur se connecte sur votre site, vous enregistrez en session une chaîne de caractère aléatoire.
- Cette même chaîne est récupérée en session et passée au formulaire, sous la forme d'un champ de type “**hidden**“.
- Lorsque le formulaire est posté, votre système de validation doit vérifier que le champ caché contient la même valeur que celle enregistrée en session.

- Si ça n'est pas le cas, ça signifie que quelqu'un a essayé de soumettre le formulaire depuis un autre endroit que votre site. Il faut alors annuler le traitement de ce formulaire.

Laravel inclut un **plug-in CSRF** intégré, qui génère des jetons pour chaque session utilisateur active. Ces jetons vérifient que les opérations ou requêtes sont envoyées par l'utilisateur authentifié concerné.

4. La mise en œuvre

- CSRF est implémenté dans les formulaires HTML déclarés dans les applications Web. Vous devez inclure un jeton CSRF validé masqué dans le formulaire, afin que le middleware de protection CSRF de Laravel puisse valider la demande.
- La protection **CSRF** peut être implémentée dans Laravel à l'aide de n'importe quel formulaire HTML avec une forme cachée de jeton CSRF et la demande de l'utilisateur est validée à l'aide du middleware CSRF **VerifyCsrfToken**.
- L'une des options suivantes peut être utilisée pour générer un jeton **CSRF**.

1. @csrf

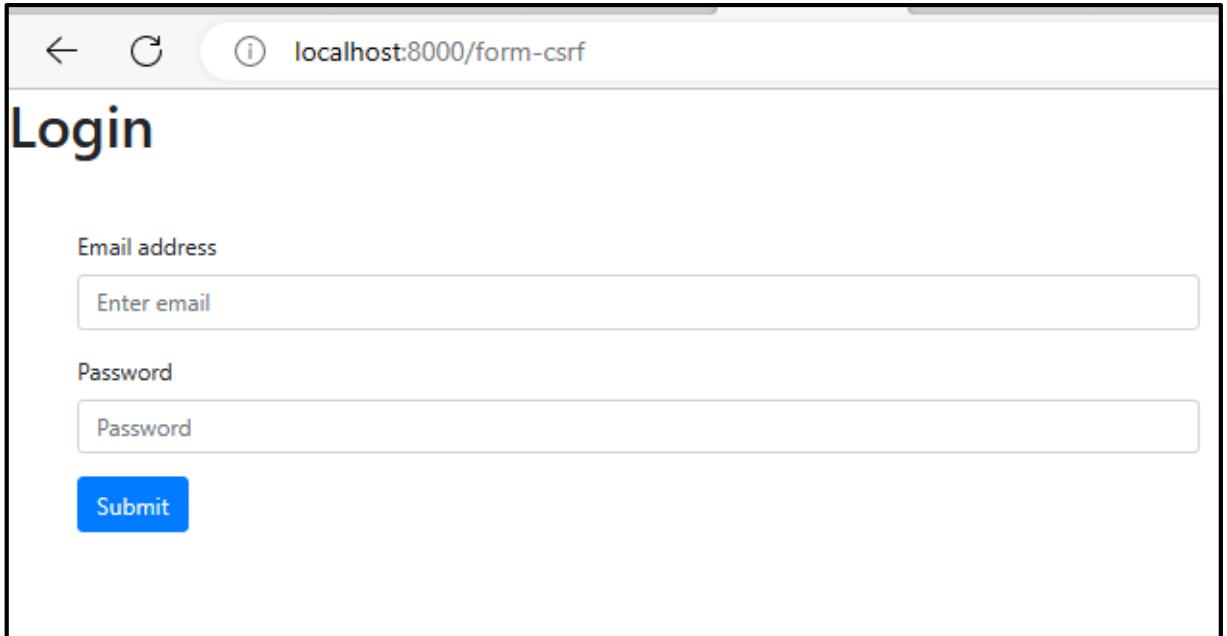
- C'est une directive de Blade pour générer un champ de jeton qui sera utilisé pour la vérification. Il génère un champ de saisie masqué.
- Lorsque la directive blade **@csrf** est utilisée, une balise *input* de type='hidden' est ajoutée au formulaire.
- La valeur sera le jeton CSRF qui sera envoyé dans le cadre des données du formulaire lorsque le formulaire est soumis.

▪ Activité

- Créez un fichier de vue Laravel nommé **form-csrf.blade.php** avec le code HTML suivant où la directive @csrf est utilisée pour générer le jeton CSRF.

```
<form action="{{route('details')}}" method="POST" >
  @csrf
  <label>Email address</label>
  <input name="email" type="email">
  <label>Password</label>
  <input type="password" name="password" >
```

```
<button type="submit">Submit</button>  
</form>
```



localhost:8000/form-csrf

Login

Email address

Password

Submit

- Dans le fichier **web.php** pour charger le fichier de vue dans le navigateur. Lorsque l'utilisateur donnera **form-csrf** après l'URL de base (<http://127.0.0.1:8000/form-csrf>), il recherchera le fichier **form-csrf.blade.php** dans le dossier de vue du projet Laravel.

```
Route::get('/form-csrf', function () {return  
view('form-csrf');});
```

- Si vous inspectez la page après l'exécution, vous obtiendrez la sortie comme ci-dessous. Ici, un champ masqué avec la valeur est généré automatiquement par la directive **@csrf**.

```

▼<div class="container mt-5">
  ▼<form action>
    <input type="hidden" name="_token" value="3bv3Jkk1BnsafdLNyxUe0gXCreX99ztI120f5QNB" > == $0
    ▷<div class="row">...</div>
    ▷<div class="row">...</div>
    <input type="submit" name="send" value="Submit" class="btn btn-primary btn-block">
  </form>
</div>

```

2. csrf_token ()

- Cette fonction peut être utilisée dans la balise **meta** et le champ de saisie masqué du formulaire HTML. Il génère une chaîne aléatoire en tant que jeton CSRF.
- Mettez à la place de **@csrf** dans la vue Laravel nommé **form-csrf.blade.php** la fonction `csrf_token ()` est utilisée pour générer le jeton CSRF. Cette fonction est utilisée comme valeur de l'attribut `value` du champ masqué et est utilisée avec deux accolades.
- **Syntaxe**

```

<form method = "POST">
  <input type = "hidden" name = "_ token" value =
  "{{csrf_token ()}}">
  .....
  .....
</form>

```

- Si vous inspectez la page après l'exécution, vous obtiendrez la même sortie de l'exemple précédent.
 - Veuillez noter que le bouton d'envoi inclut des fonctionnalités dans la section fonction de rappel. Il est montré ci-dessous:

```

Route::post('/details', function (Request $req) {
  return $req->all();
})->name('details');

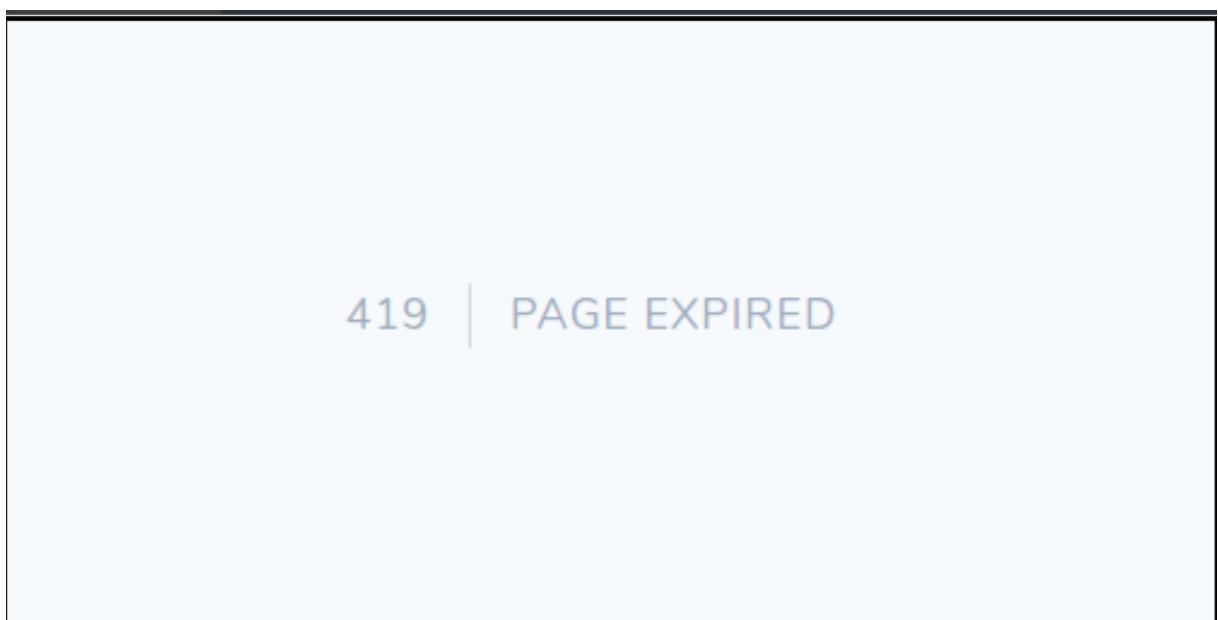
```

- La sortie obtenue renverra JSON avec un jeton comme indiqué ci-dessous:



```
{"_token": "hfliQ8jp09Hd77hhw9rzPYCB73xu2W8AQsb1s0kV", "email": null, "password": null}
```

Si le jeton CSRF n'est pas présent dans la demande de formulaire envoyée ou s'il semble invalide, Laravel envoie un message d'erreur "Page Expired" avec un code d'état 419.



3. Exclure les URI de la protection CSRF

Parfois, vous souhaiterez peut-être exclure un ensemble d'URI de la protection CSRF. En règle générale, vous devez placer ces types de routes en dehors du groupe web de middleware **App\Providers\RouteServiceProvider** qui s'applique à toutes les routes du fichier **routes/web.php**. Cependant, vous pouvez également exclure les routes en ajoutant leurs URI à la propriété `$except` du middleware **VerifyCsrfToken** :

```
<?php  
namespace App\Http\Middleware;
```

```
use Illuminate\Foundation\Http\Middleware\VerifyCsrfToken as  
Middleware;  
  
class VerifyCsrfToken extends Middleware  
{  
    /**  
     * The URIs that should be excluded from CSRF  
     * verification.  
     *  
     * @var array<int, string>  
     */  
    protected $except = [  
        '/details'  
    ];  
}
```

Maintenant essayez de poster à nouveau le formulaire en supprimant `@csrf`.

Manipulation des contrôleurs :

1. Intérêt des contrôleurs

Au lieu de définir toute votre logique de gestion des requêtes comme des fermetures dans vos fichiers de routage, vous pouvez organiser ce comportement à l'aide de classes "**Controller**". Les contrôleurs peuvent regrouper la logique de gestion des demandes associées dans une seule classe. Par exemple, une classe **StudentController** peut gérer toutes les demandes entrantes liées aux étudiants, y compris l'affichage, la création, la mise à jour et la suppression. Par défaut, les contrôleurs sont stockés dans le répertoire **app/Http/Controllers**.

2. Crédit du contrôleur

```
php artisan make:controller <controller-name>
```

Remplacez ce **<nom-du-contrôleur>** dans la syntaxe ci-dessus par votre contrôleur (**StudentController**).

Un exemple de code de contrôleur de base ressemble à ceci, et vous devez le créer dans un répertoire tel que **app/Http/Controller/StuentController.php** :

```
<?php

namespace App\Http\Controllers;

use App\Models\Student;
use Illuminate\Http\Request;

class StudentController extends Controller
{
    public function getStudents(){
        return Student::select('id','name')->get();
    }

    public function show($id){
        $user=Student::findOrFail($id);

        return $user->name;
    }
}
```

Le contrôleur que vous avez créé peut être invoqué à partir du fichier **routes.php** en utilisant la syntaxe ci-dessous.

Vous pouvez définir une route vers cette méthode de contrôleur comme suit : Lorsqu'une requête entrante correspond à l'URI de route spécifié, la méthode `show` de la classe **App\Http\Controllers\UserController** est appelée et les paramètres de route sont transmis à la méthode.

```
Route::get('show/{id}', [StudentController::class, 'show']);
```

TP : Créer le CRUD d'un Modèle (Student)

Ce TP a pour objectif la mise en place les opérations CRUD (Create, Read, Update, Delete) avec upload d'image dans un projet Laravel.

#Base de données, migration et modèle du CRUD

Pour présenter un **Student**, nous avons besoin des informations suivantes dans une table de la base de données, appelons cette table « **students** » :

- Un identifiant : `$student->id`
- Un nom : `$student->name`
- Un E-mail : `$student->mail`
- Une image de profile : `$student->picture`
- Une section: `$student->section`
- La date de création et de mise à jour : `$student->created_at` et `$student->updated_at`

Le schéma de ces informations doit être décrit dans [une migration](#). Nous pouvons générer le **modèle** et la migration en exécutant la commande *artisan* suivante :

```
php artisan make:model Student -m
```

Le paramètre « -m » permet de générer la migration « ..._create_students_table.php » du modèle « Student.php ». Ce qui nous donne deux fichiers :

- **/app/Models/Student.php** : Le modèle qui représente la table d'articles « posts »
- **/database/migrations/..._create_students_table.php** : La migration où décrire le schéma de la table « students »

Décrivons le schéma de la table « students » dans la fonction `up()` de la migration **/database/migrations/..._create_students_table.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('students', function (Blueprint $table) {
            $table->id();
            $table->string(name);
            $table->string(picture);
            $table->string(section);
            $table->string(mail);
            $table->timestamps();
        });

    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
}
```

```
public function down()
{
    Schema::dropIfExists('students');
}
};
```

Pour importer (migrer) la table « students » dans la base de données, on exécute la commande *artisan* suivante :

```
php artisan migrate
```

#Le contrôleur du CRUD

Pour gérer les actions ou opérations du CRUD (students), nous avons besoin d'un contrôleur, appelons-le « **StudentController** ».

Pour générer le contrôleur **StudentController.php** en l'associant au modèle **app/Models/Student.php**, on exécute la commande *artisan* suivante :

```
php artisan make:controller StudentController
```

Ce qui nous donne le code suivant au fichier **/app/Http/Controllers/StudentController.php** :

```
<?php

namespace App\Http\Controllers;

use App\Models\Student;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Validator;

class StudentController extends Controller
{
    public function index() { }

    public function create() { }
```

```
public function store(Request $request) { }

public function show(Student $student) { }

public function edit(Student $student) { }

public function update(Request $request, Student $student) { }

public function destroy(Student $student) { }

}
```

Nous allons décrire ces méthodes après qu'on ait parlé de routes du CRUD.

#Les routes du CRUD

Pour faire pointer les routes du CRUD, nommons ces routes « **students.*** » (students.index, students.create, students.edit, ...), aux actions du contrôleur **StudentController**, il suffit d'insérer 7 routes au fichier **/routes/web/php** :

```
<?php

use Illuminate\Support\Facades\Route;

use App\Http\Controllers\PostController;

Route::prefix('students')->group(function () {
    Route::get('/', [StudentController::class, 'index'])->name('students.index');
    Route::get('/create', [StudentController::class, 'create'])->name('students.create');
    Route::delete('/{student}', [StudentController::class, 'destroy'])-
>name('students.destroy');
    Route::get('/{student}', [StudentController::class, 'show'])->name('students.show');
    Route::get('/{student}/edit', [StudentController::class, 'edit'])->name('students.edit');
    Route::post('/', [StudentController::class, 'store'])->name('students.store');
    Route::put('/{student}', [StudentController::class, 'update'])->name('students.update');
});
```

Pour voir les nouvelles routes créées, exécutons la commande *artisan* suivante :

```
php artisan route:list
```

Ce qui nous affiche les 7 routes suivantes :

```
PS C:\xampp\htdocs\tp1> php artisan route:list --name=students

GET|HEAD students ..... students.index > StudentController@index
POST students ..... students.store > StudentController@store
GET|HEAD students/create ..... students.create > StudentController@create
DELETE students/{student} ..... students.destroy > StudentController@destroy
GET|HEAD students/{student} ..... students.show > StudentController@show
PUT students/{student} ..... students.update > StudentController@update
GET|HEAD students/{student}/edit ..... students.edit > StudentController@edit

Showing [7] routes
```

Méthode	URI	Action	Nom de la route
GET	/ students	index	students.index
GET	/ students /create	create	students.create
POST	/ students	store	students.store
GET	/ students /{student}	show	students.show
GET	/ students /{student}/edit	edit	students.edit
PUT	/ students /{student}	update	students.update
DELETE	/ students /{student}	destroy	students.destroy

#Les opérations du CRUD

Revenons sur les méthodes du contrôleur **/app/Http/Controllers/StudentController.php** pour décrire les actions des routes :

1. L'action « index »

La méthode ou l'action **index()** dont la route est nommée « **students.index** » permet d'afficher une liste d'un **Modèle**. La « modèle » dont nous parlons ici est le « **Student** ».

Méthode	URI	Action	Nom de la route
GET	/students	index	students.index

Pour afficher la liste de **Student**, nous devons récupérer tous les étudiants de la base de données puis les transmettre à la vue ([template Blade](#)) **/resources/views/students/index.blade.php** :

```
public function index() {
    //On récupère tous les Student
    $students = Student::all() ;

    // On transmet les Student à la vue
    return view("students.index", compact("students"));
}
```

Pour présenter les étudiants, on parcourt (avec la directive `@forelse` ou `@foreach`) la collection de Student sur la vue **/resources/views/students/index.blade.php** :

```
<body class="container">
    <h1> List Students</h1>
    <a href="{{route('students.create')}}" class="btn btn-primary m-2">Add new Student</a>
    <div class="students">

        @forelse ($students as $student)
        <div class="student">
            <h2><a href="{{ route('students.show', $student) }}" >{{ $student->name }}</a></h2> <br>

            <a href="{{route('students.edit',$student)}}" class="btn btn-success ">Edit</a>

            <form action="{{route('students.destroy',$student)}}" method="POST">
                @method('DELETE')
                @csrf

                <button class="btn btn-primary m-2">Delete</button>
            </form>
        </div>
    @forelse
</div>
```

```

</div>
@empty
<h3>No students available</h3>
@endforelse

</body>

```

Sur cette vue, nous avons ajouté les liens suivants :

- {{ route('students.create') }} pour créer un nouveau Student
- {{ route('students.show', \$student) }} pour afficher un Student \$student
- {{ route('students.edit', \$student) }} pour éditer un Student \$student
- {{ route('students.destroy', \$student) }} pour supprimer un Student \$student
-

Et le formulaire pour supprimer un Student \$student :

```

<form method="POST" action="{{ route('students.destroy', $student) }}" >
    <!-- CSRF token -->
    @csrf
    <!-- <input type="hidden" name="_method" value="DELETE"> -->
    @method("DELETE")
    <input type="submit" value="x Remove" >
</form>

```

2. L'action « create »

La méthode ou l'action **create()** dont la route est nommée « **students.create** » permet de présenter un formulaire pour créer une nouvelle ligne :

Méthode	URI	Action	Nom de la route
GET	/students/create	create	students.create

Pour créer un nouveau Student, nous allons retourner la vue **/resources/views/students/create.blade.php** où nous allons placer le formulaire de création d'un nouveau Student:

```
public function create() {
    // On retourne la vue "/resources/views/students/create.blade.php"
    return view("students.create");
}
```

// le formulaire de création d'un nouveau Student

```
<body class="container">
    <div class="success">    @if (Session::has('success'))
        {{Session::get('success')}}
    @endif </div>

<h1>Add new student</h1>

<form action="{{route('students.store')}}" method="POST" class="myform"
enctype="multipart/form-data">
    @method("POST")
    @csrf
    <div class="form-group">
        <label for="name">Name:</label>
        <input id="name" class="form-control" name="name" placeholder="name"
value="{{old('name')}}" >
        @error('name')
            <div class="alert alert-danger">
                {{$message}}
            </div>
        @enderror
    </div>
```

```

<div class="form-group mt-3">
    <label for="mail">E-mail:</label>
    <input name="mail" class="form-control" id="mail" placeholder="mail"
value="{{old('mail')}}">
    @error('mail')
        <div class="alert alert-danger">
            {{$message}}
        </div>
    @enderror
</div>
<div class="form-group mt-3">
    <label for="picture">Picture:</label>
    <input class="form-control" type="file" name="picture"
id="picture" value="{{old('picture')}}">
    <!-- Le message d'erreur pour "picture" -->
    @error('picture')
        <div class="alert alert-danger">
            {{$message}}
        </div>
    @enderror
</div>
<div class="form-group mt-3">
    <label for="section">Section:</label>
    <input class="form-control" id="section" name="section" placeholder="section"
value="{{old('section')}}">
    @error('section')
        <div class="alert alert-danger">
            {{$message}}
        </div>
    @enderror
</div>

<div class="text-center">
    <button class="btn btn-primary mt-3">Add new Student</button>
</div>

</form>

<a href="{{route('students.index')}}" class="btn btn-success w-20 ">Back</a>
</body>

```

L'action du formulaire pointe vers la route nommée « **students.store** ».

3. L'action « store »

La méthode ou l'action **store(Request \$request)** dont la route est nommée « **students.store** » permet d'enregistrer une nouvelle ligne :

Méthode	URI	Action	Nom de la route
POST	/students	store	students.store

Nous allons procéder de la manière suivante pour enregistrer un nouveau Student à partir de données qui proviennent du formulaire de la route « **students.create** » :

1. Valider les informations envoyées (Voir [validation](#) Laravel)
2. Uploader l'image de profile
3. Enregistrer les informations du Student dans la table « students »
4. Retourner vers la liste de student : la route « **students.index** » ou bien sur la même page vous affichez un message de succès.

Implémentons ce processus :

```
public function store(Request $req)
{
    // 1. La validation
    $rules= [
        'name' => ['required', 'max:255'],
        'mail' => ['required', 'email', 'max:255'],
        "picture" => 'bail|required|image|max:1024',
        'section' => ['required', 'max:255'],
    ];
    $messages= [
        'name.required' =>"name is required",
        'picture.image' =>"please upload an image",
    ];

    $validatedData = Validator::make($req->all(),$rules,$messages);
    if($validatedData->fails())
        return redirect()->back()->withErrors($validatedData)->withInput();
    // 2. On upload l'image dans "/storage/app/public/students"
    $path_image = $req->picture->store("students" , "public");
    // 3. On enregistre les informations du Student
    $student=new Student;
    $student->name=$req->name;
    $student->mail=$req->mail;
    $student->picture= $path_image;
```

```
$student->section=$req->section;
$student->save();
// 4. On retourne vers la même page avec un message de réussite
return redirect()->back()->with(["success"=>"Student has added successfully"]);
```

```
}
```

Notez-bien

1. Pour ne pas tomber sur l'exception **MassAssignmentException**, nous devons indiquer les propriétés `$fillable` du modèle **/app/Http/Models/Post.php** :

```
<?php

namespace App\Models;

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

class Student extends Model
{
    use HasFactory;

    protected $fillable = [ "name", "mail", "picture" , "section" ];
}
```

2. Pour uploader l'image dans le répertoire **/storage/app/public/students**, nous devons configurer le driver du système de fichiers en « public » au fichier **.env** :

```
FILESYSTEM_DRIVER=public
```

Puis créer un lien symbolique **/public/storage/** connecté à **/storage/app/public/** en exécutant la commande *artisan* suivante :

```
php artisan storage:link
```

4. L'action « show »

La méthode ou l'action `show(Student $student)` dont la route est nommée « `students.show` » permet d'afficher les détails d'un spécifié :

Méthode	URI	Action	Nom de la route
GET	/students/{student}	show	students.show

Pour afficher un Student enregistré, nous le récupérons à partir de son identifiant puis le transmettons à la vue `/resources/views/students/show.blade.php` :

```
public function show(Student $student) {
    return view("students.show", compact("student"));
}
```

Nous pouvons présenter un Student de la manière suivante sur la vue

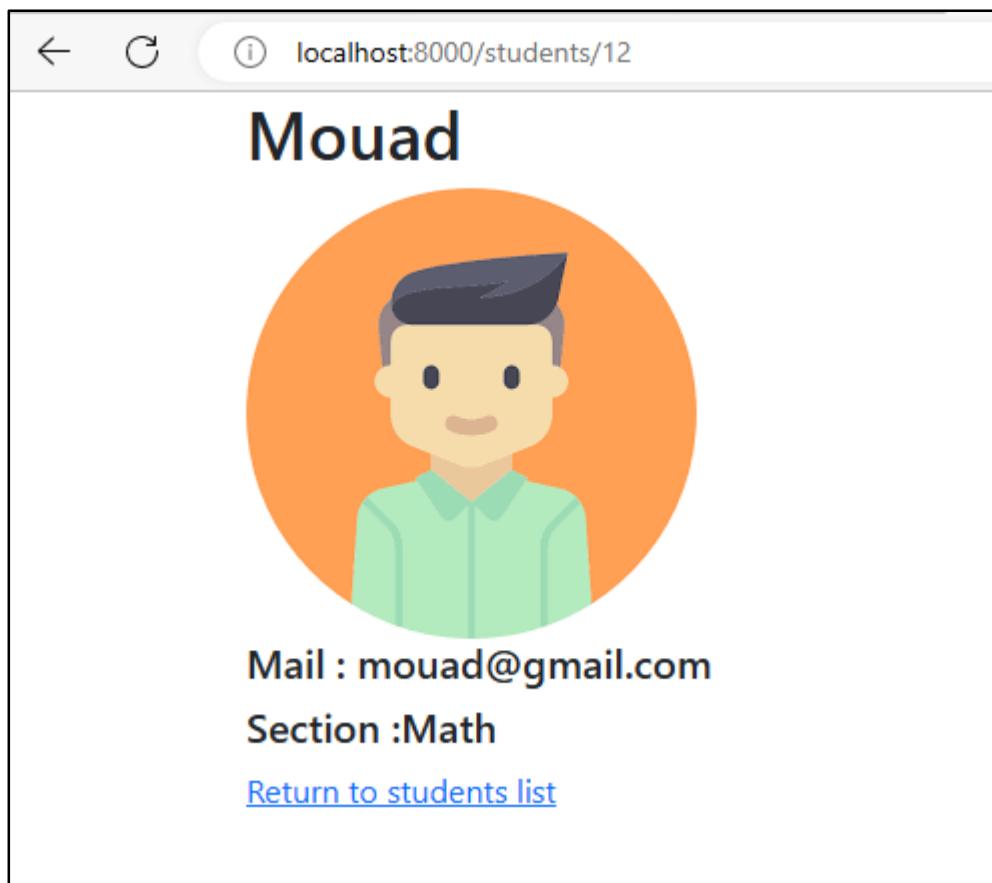
`/resources/views/students/show.blade.php` :

```
<body class="container">

    <h1>{{ $student->name }}</h1>

    <h5>Mail : {{$student->mail}}</h5>
    <h5>Section :{{$student->section}} </h5>
    <p><a href="{{ route('students.index') }}" title="Retourner aux étudiants">Return to students list</a></p>
</body>
```



5. L'action « edit »

La méthode ou l'action **edit(Student \$student)** dont la route est nommée « **students.edit** » permet d'afficher le formulaire où éditer (modifier) un enregistrement spécifié :

Méthode	URI	Action	Nom de la route
GET	/students/{student}/edit	edit	students.edit

Pour éditer un Student, nous le récupérons à partir de son identifiant puis le transmettons à la vue **/resources/views/students/edit.blade.php** :

```
public function edit(Student $student) {  
    return view("students.edit", compact("student"));  
}
```

Comme nous utilisons la vue **edit.blade.php** pour les actions **create()** et **edit()**, adaptons-la. Si un Student **\$student** est transmis à la vue :

- L'action du formulaire se gère par la route nommée « **students.update** »
- On complète les valeurs des inputs du formulaire (« name » et « section » ...) avec les données existantes du student **\$student**
- On affiche l'image de couverture existant.

La vue **/resources/views/students/edit.blade.php** adaptée aux méthodes **create()** et **edit()** :

```
<body class="container">
    @if (Session::has('success'))
        {{Session::get('success')}}
    @endif
<h1>Edit student</h1>

<form action="{{route('students.update',$student)}}" method="POST" class="myform"
enctype="multipart/form-data">

    @method("POST")
    @csrf
    <div class="form-group">
        <label for="name">Name:</label>
        <input id="name" class="form-control" name="name" placeholder="name"
value="{!! isset($student->name) ? $student->name : old('name') !!}">
        @error('name')
            <div class="alert alert-danger">
                {{$message}}
            </div>
        @enderror
    </div>

    <div class="form-group mt-3">
        <label for="mail">E-mail:</label>
        <input name="mail" class="form-control" id="mail" placeholder="mail" value="{!!
isset($student->mail) ? $student->mail : old('mail') !!}">
        @error('mail')
            <div class="alert alert-danger">
                {{$message}}
            </div>
        @enderror
    </div>
</div>
```

```
<div class="form-group mt-3">
    <label for="picture">Picture:</label>
    
    <input class="form-control" type="file" id="picture" name="picture"
placeholder="picture" value="{{ isset($student->picture) ? $student->picture :
old('picture') }}>
    @error('picture')
        <div class="alert alert-danger">
            {{$message}}
        </div>
    @enderror
</div>
<div class="form-group mt-3">
    <label for="section">Section:</label>
    <input id="section" class="form-control" name="section" placeholder="section"
value="{{ isset($student->section) ? $student->section : old('section') }}>
    @error('section')
        <div class="alert alert-danger">
            {{$message}}
        </div>
    @enderror
</div>
<div class="text-center">
    <button class="btn btn-primary mt-3 w-50 text-center">Update</button>
</div>

</form>

</body>
```

← ⏪ ⓘ localhost:8000/students/10/edit

Edit student

Name:

E-mail:

Picture:

VectorStock® Vecteur de VecteurStock.com/12020266

Choisir un fichier Aucun fichier n'a été sélectionné

Section:

Update

6. L'action « update »

La méthode ou l'action **update(Request \$request, Student \$student)** dont la route est nommée « **students.update** » permet de mettre à jour un enregistrement spécifié :

Méthode	URI	Action	Nom de la route
PUT	/students/{student}	update	students.update

Nous allons procéder de la manière suivante pour mettre à jour un Student avec les données qui proviennent du formulaire de la route « **students.edit** » :

1. On valide les informations envoyées
2. Si une nouvelle image est envoyée, on supprime l'ancienne image puis on upload la nouvelle

3. On met à jour les informations du Student spécifié

4. On se redirige vers la route « **students.show** » pour afficher le Student modifié

Implémentons ce processus :

```
public function update(Request $req, Student $student)
{
    // 1. La validation
    $rules= [
        'name' => ['required', 'max:255'],
        'mail' => ['required','email', 'max:255'],
        'section' => ['required', 'max:255'],
    ];
    $messages= [
        'name.required' =>"name is required",
        'mail.email' =>"Email not valid",
    ];
    // dd($req->hasFile("picture"));
    // Si une nouvelle image est envoyée
    if ($req->hasFile("picture")) {
        // On ajoute la règle de validation pour "picture"
        $rules["picture"] = 'bail|required|image|max:1024';
    }

    // 2. On upload l'image dans "/storage/app/public/students"

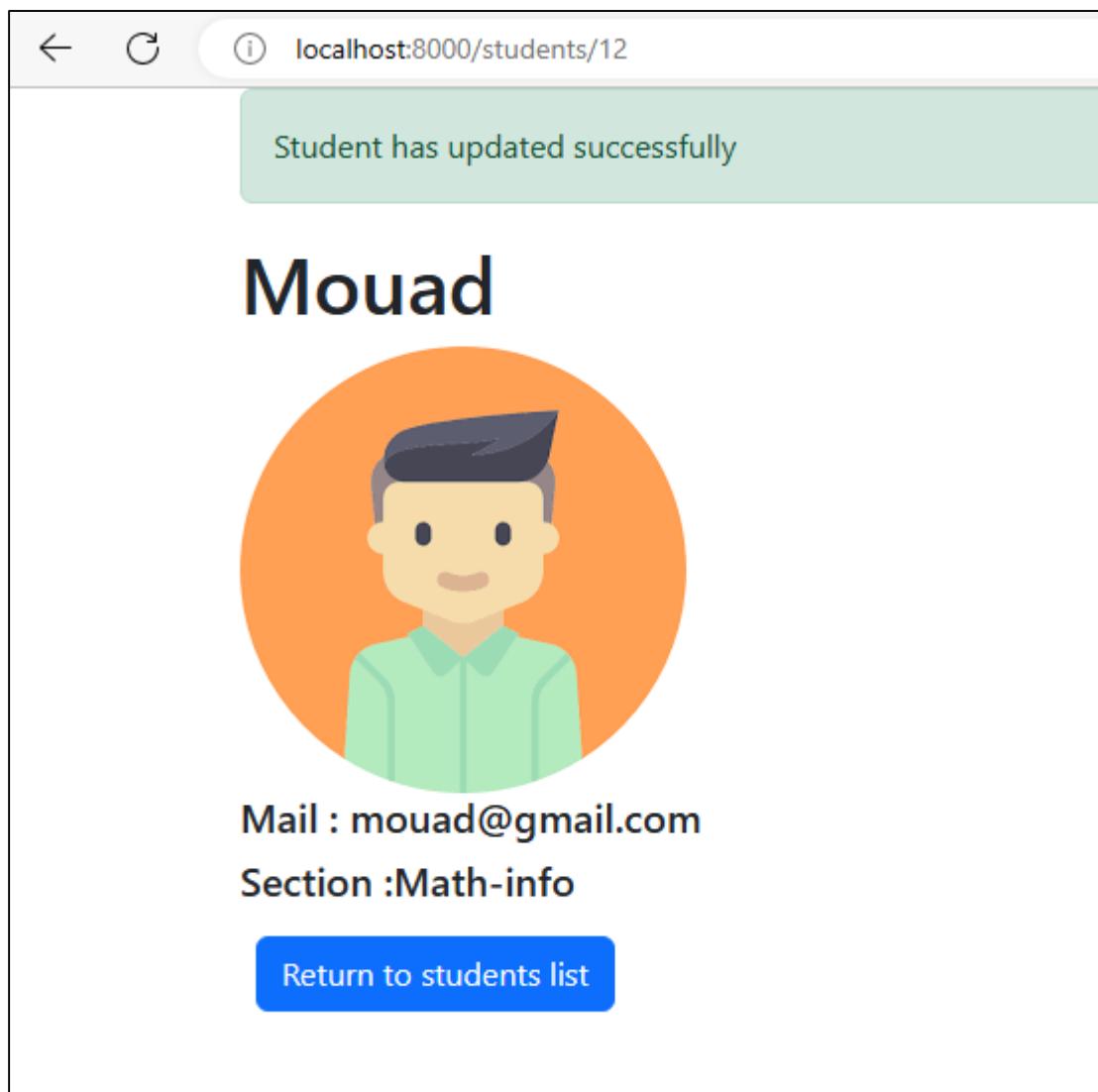
    $validatedData = Validator::make($req->all(),$rules,$messages);
    if($validatedData->fails())
        return redirect()->back()->withErrors($validatedData)->withInput();
    // 2. On upload l'image dans "/storage/app/public/students"
    if ($req->hasFile("picture")) {

        //On supprime l'ancienne image
        Storage::delete($student->picture);

        $chemin_image = $req->picture->store("students","public");
    }
    // 3. On enregistre les informations du Student
    $student->update([
        "name" => $req->name,
        "mail" => $req->mail,
        "picture" =>isset($chemin_image) ? $chemin_image : $student->picture,
        "section"=>$req->section
    ]);
}
```

```
]);
// 4. On retourne vers la même page avec un message de réussite
return redirect()->route('students.show',$student)->with(["success"=>"Student has
updated successfully"]);

}
```



The screenshot shows a web browser window with the URL `localhost:8000/students/12`. A green notification bar at the top says `Student has updated successfully`. Below it, the student's name `Mouad` is displayed in large bold letters. A circular profile picture of a person with short dark hair and a green shirt is shown. Below the name and picture, the student's email `Mail : mouad@gmail.com` and section `Section :Math-info` are listed. At the bottom, a blue button says `Return to students list`.

7. L'action « `destroy` »

La méthode ou l'action `destroy(Student $student)` dont la route est nommée « `students.destroy` » permet de supprimer un Student spécifié :

Méthode	URI	Action	Nom de la route
DELETE	<code>/students/{student}</code>	destroy	<code>students.destroy</code>

Pour supprimer un Student, nous commençons par supprimer son image de couverture du répertoire **/storage/students**, ensuite ses informations dans la table « students » puis on retourne (redirection) à la route « **students.index** » :

```
public function destroy(Student $student)
{
    // On supprime l'image existant
    unlink('storage/' . $student->picture);
    // On supprime les informations du $student de la table "students"
    $student->delete();
    // Redirection route "students.index"
    return redirect()->route('students.index');
}
```

Nous avons terminé l'implémentation du CRUD. Voici un récapitulatif de fichiers édités :

- **/app/Models/Student.php** : Le modèle qui représente un étudiant de la table « students »
- **/database/migrations/..._create_students_table.php** : La migration où décrire le schéma de la table « students »
- **/routes/web.php** : Le fichier où définir les routes du CRUD
- **/app/Http/Controllers/StudentController.php** : Le contrôleur pour pour décrire les actions du CRUD : index, create, store, show, edit, update et destroy
- **/resources/views/students/index.blade.php** : La vue où afficher tous les étudiants
- **/resources/views/ students/edit.blade.php** : La vue où créer et éditer un étudiant
- **/resources/views/ students/show.blade.php** : La vue pour afficher un étudiant
- **/.env** : Le fichier où modifier les configurations de l'application

3. Contrôleur invocable

- Si une action de contrôleur est particulièrement complexe, vous trouverez peut-être pratique de dédier une classe de contrôleur entière à cette action unique. Pour ce faire, vous pouvez définir une seule méthode `__invoke` dans le contrôleur :

```
<?php
namespace App\Http\Controllers;

use App\Models\Student;
use Illuminate\Http\Request;

class StudentController extends Controller
{

    public function __invoke($id){
        $user=Student::findOrFail($id);

        return $user->name;

    }
}
```

- Lors de l'enregistrement d'itinéraires pour des contrôleur à action unique, vous n'avez pas besoin de spécifier une méthode de contrôleur. Au lieu de cela, vous pouvez simplement transmettre le nom du contrôleur au routeur :

```
use App\Http\Controllers\Student;
Route::get('show/{id}', StudentController::class);
```

- Vous pouvez générer un contrôleur invocable en utilisant l'option `--invokable` de la commande Artisan `make:controller` :

```
php artisan make:controller ProvisionServer --invokable
```

4. Contrôleur Middleware

- Un middleware peut être affecté aux routes du contrôleur dans vos fichiers de routes :

```
Route::get('show/{id}', [StudentController::class, 'show'])->name('terminate');
```

- Vous trouverez peut-être pratique de spécifier un middleware dans le constructeur de votre contrôleur. En utilisant la méthode middleware dans le constructeur de votre contrôleur, vous pouvez affecter un middleware aux actions du contrôleur :

```
class StudentController extends Controller

{
    public function __construct()
    {
        //Appliquer ce middleware à toutes les actions du contrôleur
        $this->middleware('terminate');

        //Affecter le middleware à toutes les actions sauf l'action 'show'
        $this->middleware('terminate')->except('show');

        //Affecter le middleware seulement à l'action 'create'
        $this->middleware('terminate')->only('create');

    }
}
```

Les contrôleurs vous permettent également d'enregistrer un middleware à l'aide d'une fermeture. Cela fournit un moyen pratique de définir un middleware en ligne pour un seul contrôleur sans définir une classe complète de middleware :

```
$this->middleware(function ($request, $next)
{
    return $next($request);
});
```

5. Contrôleur de ressources

La route de ressources de Laravel permet aux routes classiques "CRUD" pour les contrôleurs d'avoir une seule ligne de code. Ceci peut être créé rapidement en utilisant la commande `make:controller` (commande Artisan) quelque chose comme ceci".

```
php artisan make:controller StudentController --resource
```

Le code ci-dessus produira un contrôleur dans l'emplacement `app/Http/Controllers/` avec le nom de fichier `StudentController.php` qui contiendra une méthode pour toutes les tâches disponibles des ressources.

Ensuite, vous pouvez enregistrer une route de ressources qui pointe vers le contrôleur :

```
use App\Http\Controllers\PhotoController;
Route::resource('students', studentController::class);
```

Cette déclaration de route unique crée plusieurs routes pour gérer diverses actions sur la ressource. Le contrôleur généré aura déjà des méthodes pour chacune de ces actions. N'oubliez pas que vous pouvez toujours obtenir un aperçu rapide des routes de votre application en exécutant la commande Artisan `route:list`.

GET HEAD	student	student.index	› StudentController@index
POST	student	student.store	› StudentController@store
GET HEAD	student/create	student.create	› StudentController@create
GET HEAD	student/{student}	student.show	› StudentController@show
PUT PATCH	student/{student}	student.update	› StudentController@update
DELETE	student/{student}	student.destroy	› StudentController@destroy
GET HEAD	student/{student}/edit	student.edit	› StudentController@edit

Actions gérées par le contrôleur de ressources

Verb	URI	Action	Route Name
GET	/student	index	student.index
GET	/student/create	create	student.create

POST	/student	store	student.store
GET	/student/{student}	show	student.show
GET	/student/{student}/edit	edit	student.edit
PUT	/student/{student}	update	student.update
DELETE	/student/{student}	destroy	student.destroy

Personnalisation du comportement du modèle manquant

En règle générale, une réponse HTTP 404 est générée si un modèle de ressource implicitement lié n'est pas trouvé. Cependant, vous pouvez personnaliser ce comportement en appelant la méthode `missing` lors de la définition de votre route de ressource. La méthode `missing` accepte une fermeture qui sera invoquée si un modèle lié implicitement ne peut être trouvé pour aucune des routes de la ressource :

```
Route::resource('student', StudentController::class)->missing(function
(Request $request) {
    return redirect()->route('student.index');
});
```

Les développeurs Laravel ont également la liberté d'enregistrer plusieurs contrôleurs de ressources à la fois en passant un tableau à une méthode de ressource, quelque chose comme ceci :

```
Route::resources([
    'password' => 'PasswordController',
    'picture' => 'DpController'
]);
```

Ressources imbriquées

Parfois, vous devrez peut-être définir des itinéraires vers une ressource imbriquée. Par exemple, une ressource post peut avoir plusieurs commentaires qui peuvent être joints au post. Pour imbriquer les

contrôleurs de ressources, vous pouvez utiliser la notation "point" dans votre déclaration de route :

```
use App\Http\Controllers\PostCommentController;  
  
Route::resource('posts.comments', PostCommentController::class);
```

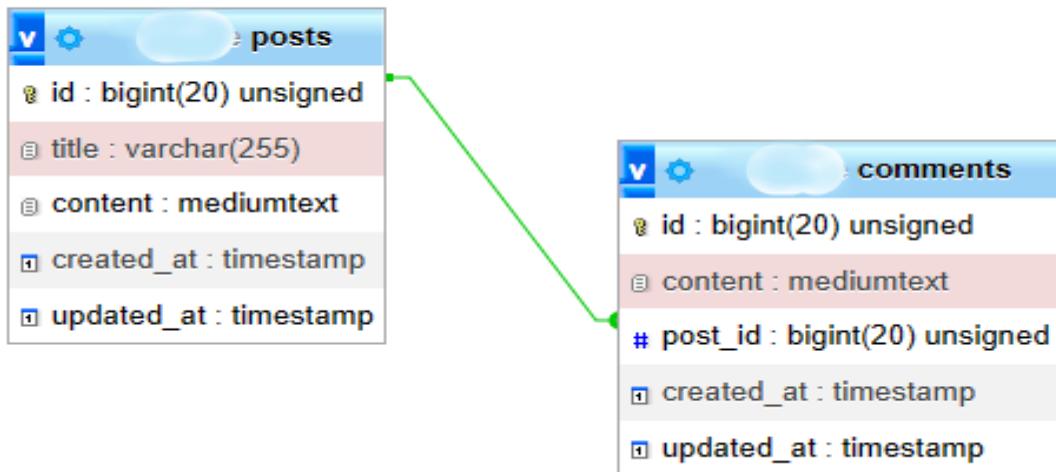
Cette définition de route définira les routes suivantes :

Verb	URI	Action	Route Name
GET	/posts/{post}/comments	index	posts.comments.index
GET	/posts/{post}/comments/create	create	posts.comments.create
POST	/posts/{post}/comments	store	posts.comments.store
GET	/posts/{post}/comments/{comment}	show	posts.comments.show
GET	/posts/{post}/comments/{comment}/edit	edit	posts.comments.edit
PUT	/posts/{post}/comments/{comment}	update	posts.comments.update
DELETE	/posts/{post}/comments/{comment}	destroy	posts.comments.destroy

La méthode **scopeBindings** nous facilitera la vie lorsque nous travaillerons avec des routes qui injectent 2 modèles ayant une relation parent-enfant.

Scénario

Supposons que nous ayons un projet dans lequel les utilisateurs du système peuvent se voir attribuer une ou plusieurs commentaires à un post et dont le schéma de données ressemble à ceci :



posts

	id	title	content	created_at	updated_at
<input type="checkbox"/>	1	my first post	learn css an javascript	NULL	NULL
<input type="checkbox"/>	2	my second post	learn python	NULL	NULL

Comments

	id	content	post_id	created_at	updated_at
<input type="checkbox"/>	1	thanks so much for this course	1	NULL	NULL
<input type="checkbox"/>	2	goog job dear teacher	1	NULL	NULL

Et dans notre backend nous aurions quelque chose comme :

// app/Models/Post.php

```
<?php

namespace App\Models;

use App\Models\Comment;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Factories\HasFactory;

class Post extends Model
{
    use HasFactory;
```

```
public function comments(){
    return $this->hasMany(Comment::class);
}
}
```

// app/Models/Comment.php

```
<?php

namespace App\Models;

use App\Models\Post;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Factories\HasFactory;

class Comment extends Model
{
    use HasFactory;
    public function post(){

        return $this->belongsTo(Post::class);

    }
}
```

//database/migrations/create_posts_table

```
<?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('posts', function (Blueprint $table) {
            $table->id();
            $table->string('title');
            $table->mediumText('content');
            $table->timestamps();
        });
    }

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

```

        });
    }

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

```

//database/migrations/create_comments_table

```

<?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('comments', function (Blueprint $table) {
            $table->id();
            $table->mediumText('content');
            $table->unsignedBigInteger('post_id');
            $table->foreign('post_id')->references('id')->on('posts')-
>onDelete('cascade');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()

```

```
    {
        Schema::dropIfExists('comments');
    }
};
```

// routes/web.php

```
Route::resource('posts.comments', PostCommentController::class);
```

// app/Http/Controllers/PostCommentController.php

```
public function show(Post $post, Comment $comment)
{
    return view('posts.show', compact("comment", "post"));
}

public function edit(Post $post, Comment $comment)
{
    return view('posts.edit', compact('post', 'comment'));
}
```

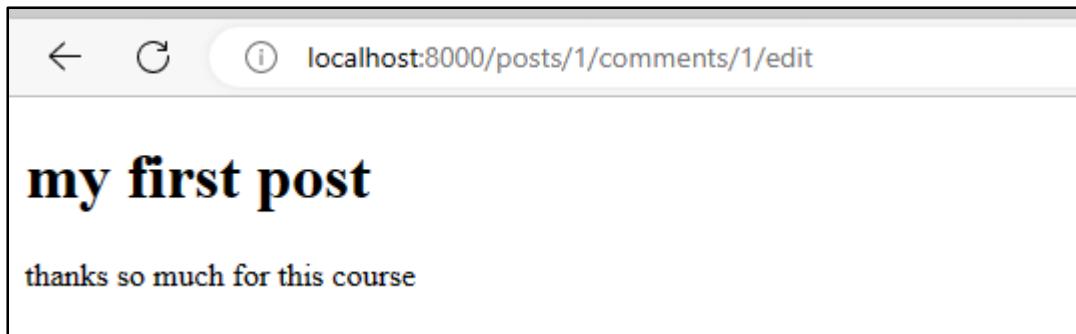
// resources/views/posts/edit.blade.php

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
<body>

    <h1>{{$post->title}}</h1>
    <p>{{$comment->content}}</p>

</body>
</html>
```

Supposons maintenant que nous voulons modifier le commentaire avec l'ID 1 du post avec l'ID 1 :



localhost:8000/posts/1/comments/1/edit

my first post

thanks so much for this course

Jusqu'à ce point, tout semble "normal" et fluide, mais si l'URL saisie était :

<localhost:8000/posts/2/comments/2/edit>

C'est ce que nous aurions fait :



localhost:8000/posts/2/comments/1/edit

my second post

thanks so much for this course

Cela constituerait une incohérence grave pour nos informations car nous éditerions le commentaire d'un autre post. Ce problème se produit parce qu'à aucun moment nous ne validons que l'instance du commentaire est nécessairement un enfant de l'instance du post, nous validons seulement implicitement que chacun existe dans la base de données.

Validation manuelle

Pour s'assurer que seuls les commentaires du post correspondant peuvent être modifiés, nous devons effectuer une validation dans notre contrôleur, qui pourrait par exemple être :

```
public function edit(Post $post, Comment $comment)
{
    abort_if($comment->post_id != $post->id, 404);
    return view('posts.edit', compact('post', 'comment'));
}
```

Validation avec scopeBingdings

Avec la fonction `scopeBindings` dans la définition du chemin, nous pouvons dire à Laravel de définir la portée des liaisons "enfants" même si aucune clé personnalisée n'est fournie :

`// routes/web.php`

```
Route::scopeBindings()->group(function () {  
  
    Route::resource('posts.comments', PostCommentController::class);  
  
});
```

L'utilisation de cette fonction est plus pratique, car elle nous permet de garder notre contrôleur libre de validations supplémentaires lorsque nous utilisons des liens modèle-route dans nos projets.

Localisation des URI de ressource

Par défaut, `Route::resource` créera des URI de ressource en utilisant des verbes anglais et des règles de pluriel. Si vous avez besoin de localiser les verbes d'action `create` et `edit`, vous pouvez utiliser la méthode `Route::resourceVerbs`. Cela peut être fait au début de la méthode `boot` dans votre application `App\Providers\RouteServiceProvider`:

```
public function boot() {  
    Route::resourceVerbs([  
        'create' => 'créer',  
        'edit' => 'éditer',  
    ]);  
    // ...  
}
```

Exemple :

localhost:8000/photos/1/comments/5/éditer

Compléter les contrôleur de ressources

Si vous devez ajouter des routes supplémentaires à un contrôleur de ressources au-delà de l'ensemble de routes de ressources par défaut, vous devez définir ces routes avant votre appel à la méthode `Route::resource` ; sinon, les routes définies par la méthode `resource` peuvent involontairement prendre le pas sur vos routes supplémentaires:

```
use App\Http\Controller\PhotoController;     Route::get('/student/event',  
[StudentController::class, 'event']);  
Route::resource('student', StudentController::class);
```


Manipulation de contrôleur de ressource :

1. Contrôleur invocable

- Si une action de contrôleur est particulièrement complexe, vous trouverez peut-être pratique de dédier une classe de contrôleur entière à cette action unique. Pour ce faire, vous pouvez définir une seule méthode `__invoke` dans le contrôleur :

```
<?php
namespace App\Http\Controllers;

use App\Models\Student;
use Illuminate\Http\Request;

class StudentController extends Controller
{

    public function __invoke($id){
        $user=Student::findOrFail($id);

        return $user->name;

    }
}
```

- Lors de l'enregistrement d'itinéraires pour des contrôleur à action unique, vous n'avez pas besoin de spécifier une méthode de contrôleur. Au lieu de cela, vous pouvez simplement transmettre le nom du contrôleur au routeur :

```
use App\Http\Controllers\Student;
Route::get('show/{id}', StudentController::class);
```

- Vous pouvez générer un contrôleur invocable en utilisant l'option `--invokable` de la commande Artisan `make:controller` :

```
php artisan make:controller ProvisionServer --invokable
```

2. Contrôleur Middleware

- Un middleware peut être affecté aux routes du contrôleur dans vos fichiers de routes :

```
Route::get('show/{id}', [StudentController::class, 'show'])->middleware('terminate');
```

- Vous trouverez peut-être pratique de spécifier un middleware dans le constructeur de votre contrôleur. En utilisant la méthode middleware dans le constructeur de votre contrôleur, vous pouvez affecter un middleware aux actions du contrôleur :

```
class StudentController extends Controller

{
    public function __construct()
    {
        //Appliquer ce middleware à toutes les actions du contrôleur
        $this->middleware('terminate');

        //Affecter le middleware à toutes les actions sauf l'action 'show'
        $this->middleware('terminate')->except('show');

        //Affecter le middleware seulement à l'action 'create'
        $this->middleware('terminate')->only('create');

    }
}
```

Les contrôleurs vous permettent également d'enregistrer un middleware à l'aide d'une fermeture. Cela fournit un moyen pratique de définir un middleware en ligne pour un seul contrôleur sans définir une classe complète de middleware :

```
$this->middleware(function ($request, $next)
{
    return $next($request);
});
```

3. Contrôleur de ressources

La route de ressources de Laravel permet aux routes classiques "CRUD" pour les contrôleurs d'avoir une seule ligne de code. Ceci peut être créé rapidement en utilisant la commande `make:controller` (commande Artisan) quelque chose comme ceci".

```
php artisan make:controller StudentController --resource
```

Le code ci-dessus produira un contrôleur dans l'emplacement `app/Http/Controllers/` avec le nom de fichier `StudentController.php` qui contiendra une méthode pour toutes les tâches disponibles des ressources.

Ensuite, vous pouvez enregistrer une route de ressources qui pointe vers le contrôleur :

```
use App\Http\Controllers\StudentController;
Route::resource('students', studentController::class);
```

Cette déclaration de route unique crée plusieurs routes pour gérer diverses actions sur la ressource. Le contrôleur généré aura déjà des méthodes pour chacune de ces actions. N'oubliez pas que vous pouvez toujours obtenir un aperçu rapide des routes de votre application en exécutant la commande Artisan

```
php artisan route:list --name=students
```

```
PS C:\xampp\htdocs\tp1> php artisan route:list --name=students

GET|HEAD students ..... students.index > StudentController@index
POST students ..... students.store > StudentController@store
GET|HEAD students/create ..... students.create > StudentController@create
DELETE students/{student} ..... students.destroy > StudentController@destroy
GET|HEAD students/{student} ..... students.show > StudentController@show
PUT students/{student} ..... students.update > StudentController@update
GET|HEAD students/{student}/edit ..... students.edit > StudentController@edit
```

Actions gérées par le contrôleur de ressources

Verb	URI	Action	Route Name
GET	/students	index	students.index
GET	/students/create	create	students.create
POST	/students	store	students.store
GET	/students/{student}	show	students.show
GET	/students/{student}/edit	edit	students.edit
PUT	/students/{student}	update	students.update
DELETE	/students/{student}	destroy	students.destroy

Personnalisation du comportement du modèle manquant

En règle générale, une réponse HTTP 404 est générée si un modèle de ressource implicitement lié n'est pas trouvé. Cependant, vous pouvez personnaliser ce comportement en appelant la méthode `missing` lors de la définition de votre route de ressource. La méthode `missing` accepte une fermeture qui sera invoquée si un modèle lié implicitement ne peut être trouvé pour aucune des routes de la ressource :

```
Route::resource('students', StudentController::class)->missing(function
(Request $request) {
    return redirect()->route('student.index');
});
```

Les développeurs Laravel ont également la liberté d'enregistrer plusieurs contrôleurs de ressources à la fois en passant un tableau à une méthode de ressource, quelque chose comme ceci :

```
Route::resources([
    'password' => 'PasswordController',
    'picture' => 'DpController'
]);
```

Travail à faire :

Refaire le TP de la séance 5 en utilisant un contrôleur ressource. Remplacez les sept routes que nous avons définies manuellement par la route suivante.

```
Route::resource('students', StudentController::class)
```


Ressources imbriquées :

Parfois, vous devrez peut-être définir des itinéraires vers une ressource imbriquée. Par exemple, une ressource `post` peut avoir plusieurs commentaires qui peuvent être joints au `post`. Pour imbriquer les contrôleurs de ressources, vous pouvez utiliser la notation "point" dans votre déclaration de route :

```
use App\Http\Controllers\PostCommentController;  
  
Route::resource('posts.comments', PostCommentController::class);
```

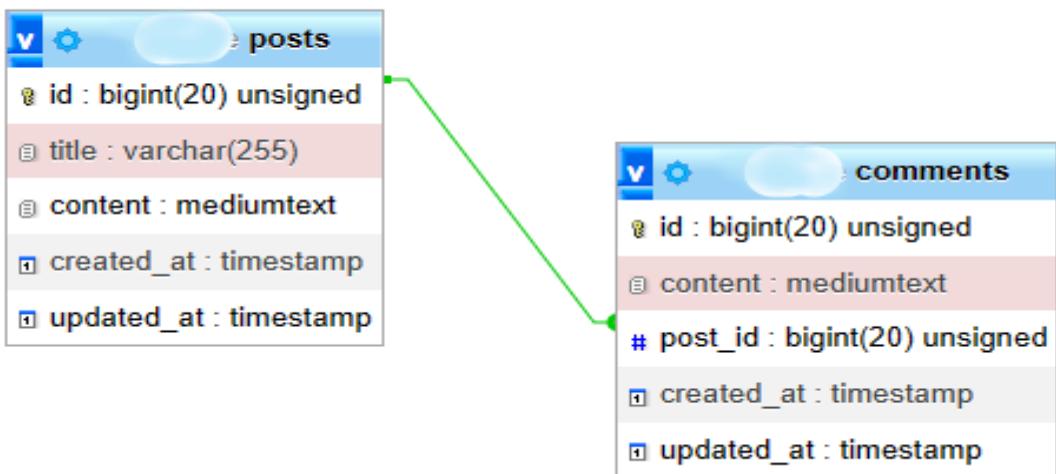
Cette définition de route définira les routes suivantes :

Verb	URI	Action	Route Name
GET	/posts/{post}/comments	index	posts.comments.index
GET	/posts/{post}/comments/create	create	posts.comments.create
POST	/posts/{post}/comments	store	posts.comments.store
GET	/posts/{post}/comments/{comment}	show	posts.comments.show
GET	/posts/{post}/comments/{comment}/edit	edit	posts.comments.edit
PUT	/posts/{post}/comments/{comment}	update	posts.comments.update
DELETE	/posts/{post}/comments/{comment}	destroy	posts.comments.destroy

La méthode `scopeBindings` nous facilitera la vie lorsque nous travaillerons avec des routes qui injectent 2 modèles ayant une relation parent-enfant.

Scénario

Supposons que nous ayons un projet dans lequel les utilisateurs du système peuvent se voir attribuer une ou plusieurs commentaires à un `post` et dont le schéma de données ressemble à ceci :



posts

	id	title	content	created_at	updated_at
<input type="checkbox"/>	1	my first post	learn css an javascript	NULL	NULL
<input type="checkbox"/>	2	my second post	learn python	NULL	NULL

Comments

	id	content	post_id	created_at	updated_at
<input type="checkbox"/>	1	thanks so much for this course	1	NULL	NULL
<input type="checkbox"/>	2	goog job dear teacher	1	NULL	NULL

Et dans notre backend nous aurions quelque chose comme :

[// app/Models/Post.php](#)

```
<?php

namespace App\Models;

use App\Models\Comment;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Factories\HasFactory;

class Post extends Model
{
    use HasFactory;
    public function comments(){
        return $this->hasMany(Comment::class);
    }
}
```

// app/Models/Comment.php

```
<?php

namespace App\Models;

use App\Models\Post;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Factories\HasFactory;

class Comment extends Model
{
    use HasFactory;
    public function post(){

        return $this->belongsTo(Post::class);

    }
}
```

//database/migrations/create_posts_table

```
<?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('posts', function (Blueprint $table) {
            $table->id();
            $table->string('title');
            $table->mediumText('content');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *

```

```
 * @return void
 */
public function down()
{
    Schema::dropIfExists('posts');
}
};
```

//database/migrations/create_comments_table

```
<?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('comments', function (Blueprint $table) {
            $table->id();
            $table->mediumText('content');
            $table->unsignedBigInteger('post_id');
            $table->foreign('post_id')->references('id')->on('posts')-
>onDelete('cascade');
            $table->timestamps();
        });
    }

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

```
// routes/web.php
```

```
Route::resource('posts.comments', PostCommentController::class);
```

```
// app/Http/Controllers/PostCommentController.php
```

```
public function show(Post $post, Comment $comment)
{
    return view('posts.show', compact("comment", "post"));
}

public function edit(Post $post, Comment $comment)
{
    return view('posts.edit', compact('post', 'comment'));
}
```

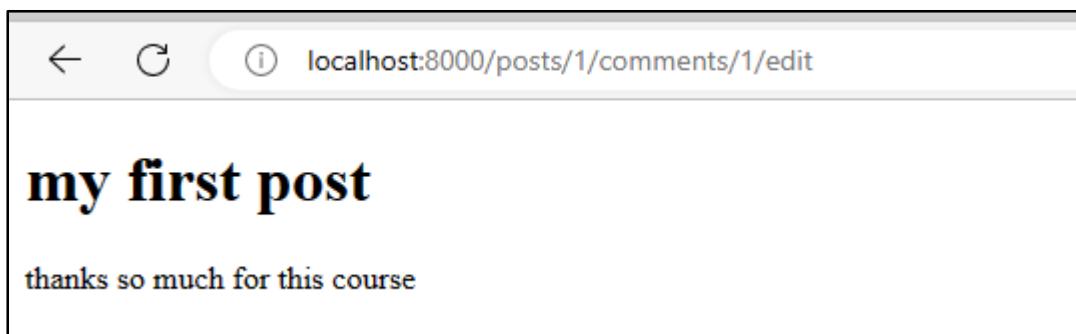
```
// resources/views/posts/edit.blade.php
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
<body>

    <h1>{{$post->title}}</h1>
    <p>{{$comment->content}}</p>

</body>
</html>
```

Supposons maintenant que nous voulons modifier le commentaire avec l'ID 1 du post avec l'ID 1 :



Jusqu'à ce point, tout semble "normal" et fluide, mais si l'URL saisie était :

<localhost:8000/posts/2/comments/2/edit>

C'est ce que nous aurions fait :



Cela constituerait une incohérence grave pour nos informations car nous éditerions le commenataire d'un autre post. Ce problème se produit parce qu'à aucun moment nous ne validons que l'instance du commentaire est nécessairement un enfant de l'instance du post, nous validons seulement implicitement que chacun existe dans la base de données.

Validation manuelle

Pour s'assurer que seuls les commentaires du post correspondant peuvent être modifiés, nous devons effectuer une validation dans notre contrôleur, qui pourrait par exemple être :

```
public function edit(Post $post, Comment $comment)
{
    abort_if($comment->post_id != $post->id, 404);
    return view('posts.edit', compact('post', 'comment'));
}
```

Validation avec scopeBingdings

Avec la fonction **scopeBingdings** dans la définition du chemin, nous pouvons dire à Laravel de définir la portée des liaisons "enfants" même si aucune clé personnalisée n'est fournie :

[// routes/web.php](#)

```
Route::scopeBindings()->group(function () {
    Route::resource('posts.comments', PostCommentController::class);
});
```

L'utilisation de cette fonction est plus pratique, car elle nous permet de garder notre contrôleur libre de validations supplémentaires lorsque nous utilisons des liens modèle-route dans nos projets.

Localisation des URI de ressource

Par défaut, `Route::resource` créera des URI de ressource en utilisant des verbes anglais et des règles de pluriel. Si vous avez besoin de localiser les verbes d'action `create` et `edit`, vous pouvez utiliser la méthode `Route::resourceVerbs`. Cela peut être fait au début de la méthode `boot` dans votre application `App\Providers\RouteServiceProvider`:

```
public function boot() {
    Route::resourceVerbs([
        'create' => 'créer',
        'edit' => 'éditer', ]);
    // ...
}
```

Exemple :

localhost:8000/photos/1/comments/5/éditer

Compléter les contrôleurs de ressources

Si vous devez ajouter des routes supplémentaires à un contrôleur de ressources au-delà de l'ensemble de routes de ressources par défaut, vous devez définir ces routes avant votre appel à la méthode `Route::resource` ; sinon, les routes définies par la méthode `resource` peuvent involontairement prendre le pas sur vos routes supplémentaires:

```
use App\Http\Controller\PhotoController;
Route::get('/student/event', [StudentController::class, 'event']);
Route::resource('student', StudentController::class);
```


Manipulation des requêtes Http :

A. Interaction avec les requêtes

1. Introduction :

La classe de Laravel `Illuminate\Http\Request` fournit un moyen orienté objet d'interagir avec la requête HTTP actuelle gérée par votre application, ainsi que de récupérer l'entrée, les cookies et les fichiers qui ont été soumis avec la requête.

2. Accéder à la demande:

Pour obtenir une instance de la requête HTTP actuelle via l'injection de dépendances, vous devez indiquer la classe `Illuminate\Http\Request` sur votre méthode de fermeture de route ou de contrôleur. L'instance de requête entrante sera automatiquement injectée par le conteneur de service Laravel :

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Store a new user.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        $name = $request->input('name');
        $mail = $request->mail;

        //
    }
}
```

Comme mentionné, vous pouvez également indiquer la classe `Illuminate\Http\Request` sur une fermeture d'itinéraire. Le conteneur de

service injectera automatiquement la requête entrante dans la fermeture lors de son exécution :

```
use Illuminate\Http\Request;

Route::get('/', function (Request $request) {
    $request->isMethod('post') // affiche false
});
```

3. Paramètres d'injection de dépendance et de routage

Si votre méthode de contrôleur attend également une entrée d'un paramètre de route, vous devez lister vos paramètres de route après vos autres dépendances. Par exemple, si votre route est définie comme ceci :

// routes/web.php

```
use App\Http\Controllers\StudentController;

Route::put('/user/{student}', [StudentController::class, 'update']);
```

Vous pouvez toujours taper le `Illuminate\Http\Request` et accéder à votre paramètre `id` route en définissant votre méthode de contrôleur comme suit :

// app/Http/Controllers/StudentController.php

```
public function update(Request $req, Student $student)
{
    //
}
```

4. Récupération du chemin de la requête

La méthode `path` renvoie les informations de chemin de la requête. Ainsi, si la requête entrante est ciblée sur `http://localhost:8000/students/create`, la méthode `path` retournera `students/create`:

```
public function create(Request $request)
{
    dd($request->path());
    return view('students.create');
}
```

//Output

```
localhost:8000/students/create
"students/create" // app\Http\Controllers\StudentController.php:23
```

5. Inspecter le chemin/la route de la demande

- La méthode `is` vous permet de vérifier que le chemin de la demande entrante correspond à un pattern donné. Vous pouvez utiliser le caractère `*` comme caractère générique lors de l'utilisation de cette méthode :

```
if ($request->is('students/*')) {
//
```

- En utilisant la méthode `routeIs`, vous pouvez déterminer si la requête entrante correspond à une route nommée :

```
if ($request->routeIs('students.edit')) {
//
```

6. Récupération de l'URL de la requête

Pour récupérer l'URL complète de la requête entrante, vous pouvez utiliser les méthodes `url` ou `fullUrl`. La méthode `url` renverra l'URL sans la chaîne de requête (`queryString`), tandis que la méthode `fullUrl` inclut la chaîne de requête :

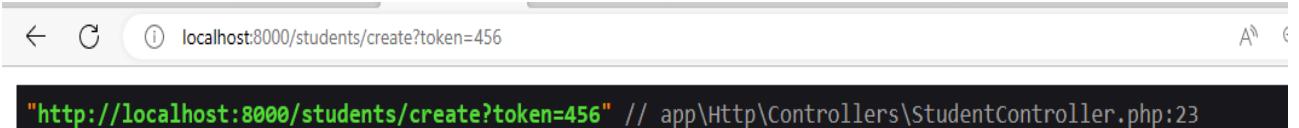
```
dd($request->url());
```

//Output

```
localhost:8000/students/create?token=456
"http://localhost:8000/students/create" // app\Http\Controllers\StudentController.php:23
```

```
dd($request->fullUrl());
```

//Output

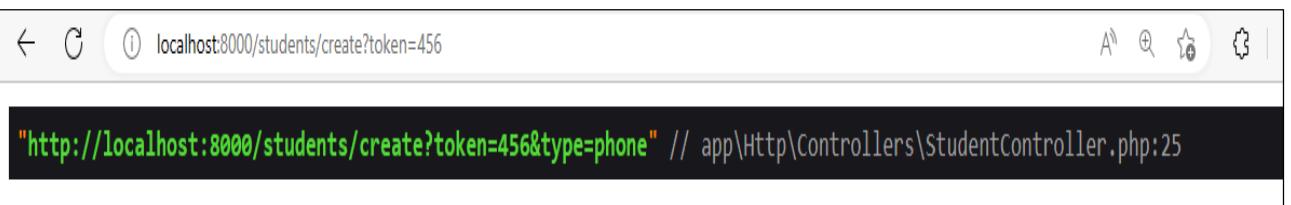


```
"http://localhost:8000/students/create?token=456" // app\Http\Controllers\StudentController.php:23
```

Si vous souhaitez ajouter des données de chaîne de requête à l'URL actuelle, vous pouvez appeler la méthode **fullUrlWithQuery**. Cette méthode fusionne le tableau donné de variables de chaîne de requête avec la chaîne de requête actuelle :

```
public function create(Request $request)
{
    $uri=$request->fullUrlWithQuery(['type' => 'phone']);
    dd($uri);
    return view('students.create');
}
```

//Output



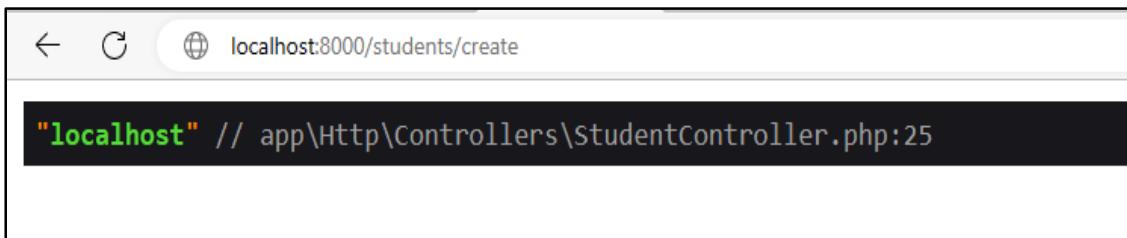
```
"http://localhost:8000/students/create?token=456&type=phone" // app\Http\Controllers\StudentController.php:25
```

7. Récupération de l'hôte de requête

Vous pouvez récupérer "l'hôte" de la requête entrante via la méthode **host** :

```
public function create(Request $request)
{
    $host=$request->host();
    // $request->httpHost();
    // $request->schemeAndHttpHost();
    dd($host);
    return view('students.create');
}
```

//Output



A screenshot of a browser window. The address bar shows 'localhost:8000/students/create'. The main content area of the browser displays the text 'localhost' in green, which is a common output for PHP code indicating the current host.

8. Récupération de la méthode Request :

La méthode `method` renverra le verbe HTTP pour la requête. Vous pouvez utiliser la méthode `isMethod` pour vérifier que le verbe HTTP correspond à une chaîne donnée :

```
$method = $request->method();
if ($request->isMethod('post')) {
//
```

9. En-têtes de demande

Vous pouvez récupérer un en-tête de requête à partir de l'instance `Illuminate\Http\Request` à l'aide de la méthode `header`. Si l'en-tête n'est pas présent sur la requête, `null` sera renvoyé. Cependant, la méthode `header` accepte un deuxième argument facultatif qui sera retourné si l'en-tête n'est pas présent sur la requête :

```
$value = $request->header('X-Header-Name');
//Example : $value = $request->header('cookie');
$value = $request->header('X-Header-Name', 'default');
```

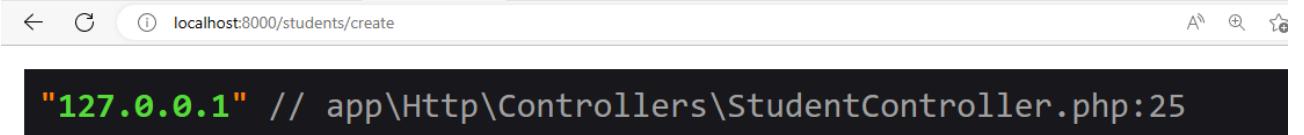
10. Demander l'adresse IP

La méthode `ip` peut être utilisée pour récupérer l'adresse IP du client qui a fait la requête à votre application :

```
public function create(Request $request)
{
    $ipAddress = $request->ip();
```

```
        dd($ipAddress);
        return view('students.create');
    }
```

//Output



A screenshot of a browser window showing the URL "localhost:8000/students/create". The page content is a single line of text: "127.0.0.1" in green, indicating the IP address of the local machine. The browser interface includes a back button, a refresh button, and a status bar with icons.

```
"127.0.0.1" // app\Http\Controllers\StudentController.php:25
```

11. Négociation de contenu

Étant donné que de nombreuses applications ne servent que du HTML ou du JSON, vous pouvez utiliser la méthode `expectsJson` pour déterminer rapidement si la requête entrante attend une réponse JSON :

```
if ($request->expectsJson())
{ // ...
}
```

B. Input

1. Récupération de toutes les données d'entrée

Vous pouvez récupérer toutes les données d'entrée de la demande entrante sous forme d'un array en utilisant la méthode `all`. Cette méthode peut être utilisée que la requête entrante provienne d'un formulaire HTML ou soit une requête XHR :

```
$input = $request->all();
```

2. Récupération d'une valeur d'entrée

À l'aide de quelques méthodes simples, vous pouvez accéder à toutes les entrées utilisateur de votre instance `Illuminate\Http\Request` sans vous soucier du verbe HTTP utilisé pour la requête. Quel que soit le verbe HTTP, la méthode `input` peut être utilisée pour récupérer l'entrée utilisateur :

```
$name = $request->input('name');
```

Vous pouvez passer une valeur par défaut comme second argument de la méthode `input`. Cette valeur sera renvoyée si la valeur d'entrée demandée n'est pas présente dans la requête :

```
$name = $request->input('name', 'Sally');
```

Lorsque vous travaillez avec des formulaires contenant des entrées de tableau, utilisez la notation "point" pour accéder aux tableaux :

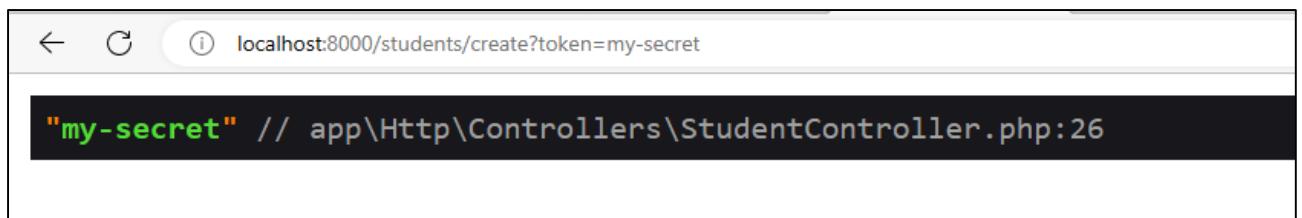
```
<input type="checkbox" name="food[]"/>Apple  
<input type="checkbox" name="food[]"/>Pear
```

```
$name = $request->input('food.0')
```

3. Récupération de l'entrée de la chaîne de requête

Alors que la méthode `input` récupère les valeurs de l'intégralité de la charge utile de la requête (y compris la chaîne de requête), la méthode `query` ne récupère que les valeurs de la chaîne de requête :

```
$name = $request->query('name');
```



4. Récupération de valeurs d'entrée stringables

Au lieu de récupérer les données d'entrée de la requête en tant que primitive string, vous pouvez utiliser la méthode `string` pour récupérer les données de la requête en tant qu'instance de `Illuminate\Support\Stringable`:

```
$name = $request->string('name')->trim();
```

5. Récupération des valeurs d'entrée booléennes

Lorsqu'il s'agit d'éléments HTML tels que des cases à cocher, votre application peut recevoir des valeurs "véridiques" qui sont en fait des chaînes. Par exemple, "vrai". Pour plus de commodité, vous pouvez utiliser la méthode `boolean` pour récupérer ces valeurs sous forme de booléens. La méthode `boolean` renvoie `true` pour 1. Toutes les autres valeurs renverront `false` :

//Form.blade.php

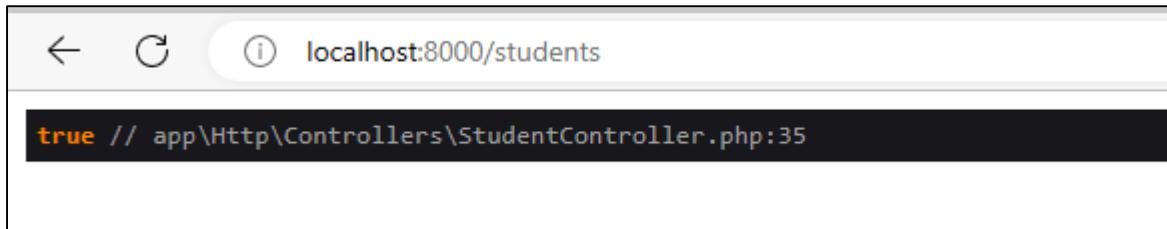
```
<input type="checkbox" name="Fruit"/>Apple
```

//MyController.php

```
$archived = $request->boolean('Fruit');

dd($archived);
```

//Output



```
true // app\Http\Controllers\StudentController.php:35
```

6. Récupération des valeurs d'entrée de date

Le paquetage **Carbon** peut aider à rendre la gestion de la date et de l'heure en PHP beaucoup plus facile et plus sémantique, afin que notre code devienne plus lisible et plus facile à maintenir.

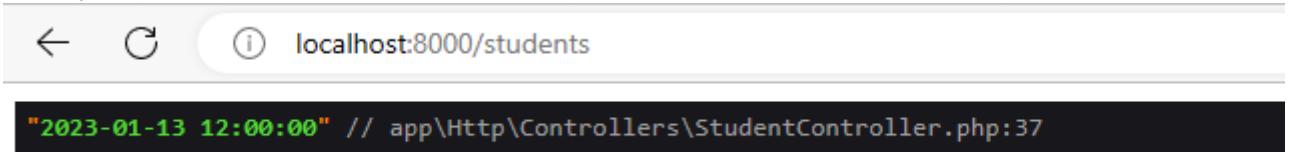
//Form.blade.php

```
<input type="date" name="birthday">
```

//MyController.php

```
$birthday = Carbon::parse($request->birthday)->format('Y-m-d h:i:s');  
dd($birthday);
```

//Output



```
"2023-01-13 12:00:00" // app\Http\Controllers\StudentController.php:37
```

7. Récupération d'une partie des données d'entrée

Si vous avez besoin de récupérer un sous-ensemble des données d'entrée, vous pouvez utiliser les méthodes **only** et **except**. Ces deux méthodes acceptent un seul array ou une liste dynamique d'arguments :

```
$input = $request->only(['name', 'birthday']);  
  
$input = $request->except(['mail']);
```

8. Déterminer si l'entrée est présente

Vous pouvez utiliser la méthode **has** pour déterminer si une valeur est présente sur la demande. La méthode **has** retourne **true** si la valeur est présente sur la requête :

```
if ($request->has('name')) {  
    //  
}
```

Lorsqu'on lui donne un tableau, la méthode **has** déterminera si toutes les valeurs spécifiées sont présentes :

```
if ($request->has(['name', 'mail']))  
{ //  
}
```

La méthode **hasAny** renvoie **true** si l'une des valeurs spécifiées est présente :

```
if ($request->hasAny(['name', 'mail']))  
{  
//  
}
```

Si vous souhaitez déterminer si une valeur est présente sur la requête et n'est pas vide, vous pouvez utiliser la méthode **filled** :

```
if ($request->filled('name')) {  
    //  
}
```

9. Conserver les valeurs des Inputs lors de la session

La méthode **flash** sur la classe **Illuminate\Http\Request** fera clignoter l'entrée actuelle de la session afin qu'elle soit disponible lors de la prochaine requête de l'utilisateur à l'application :

```
$request->flash(); // on conserve tous les inputs
$request->flashOnly(['name', 'mail']); // on sauvegarde seulement les entrées
name et mail
$request->flashExcept('password'); // on conserve toutes les valeurs sauf le
password
```

10. Entrée clignotante puis redirection

Étant donné que vous souhaiterez souvent flasher l'entrée dans la session, puis rediriger vers la page précédente, vous pouvez facilement enchaîner l'entrée clignotante sur une redirection en utilisant la méthode `withInput` :

```
return redirect('form')->withInput();
return redirect()->route('user.create')->withInput();
return redirect('form')->withInput( $request->except('password') );
```

11. Récupération de l'ancienne entrée

Laravel fournit également un helper `old` global. Si vous affichez d'anciennes entrées dans un modèle Blade, il est plus pratique d'utiliser l'assistant `old` pour remplir à nouveau le formulaire. Si aucune ancienne entrée n'existe pour le champ donné, null sera renvoyé :

```
<input name="mail" value="{{old('mail')}}">
```

12. Crédation et récupération d'un cookie

Un cookie peut être créé par le helper global `cookie` de Laravel. Il s'agit d'une instance de `Symfony\Component\HttpFoundation\Cookie`. Le cookie peut être attaché à la réponse à l'aide de la méthode `withCookie()`. Créez une instance de réponse de la classe `Illuminate\Http\Response` pour appeler la méthode `withCookie()`. Les cookies générés par Laravel sont cryptés et signés et ne peuvent pas être modifiés ou lus par le client.

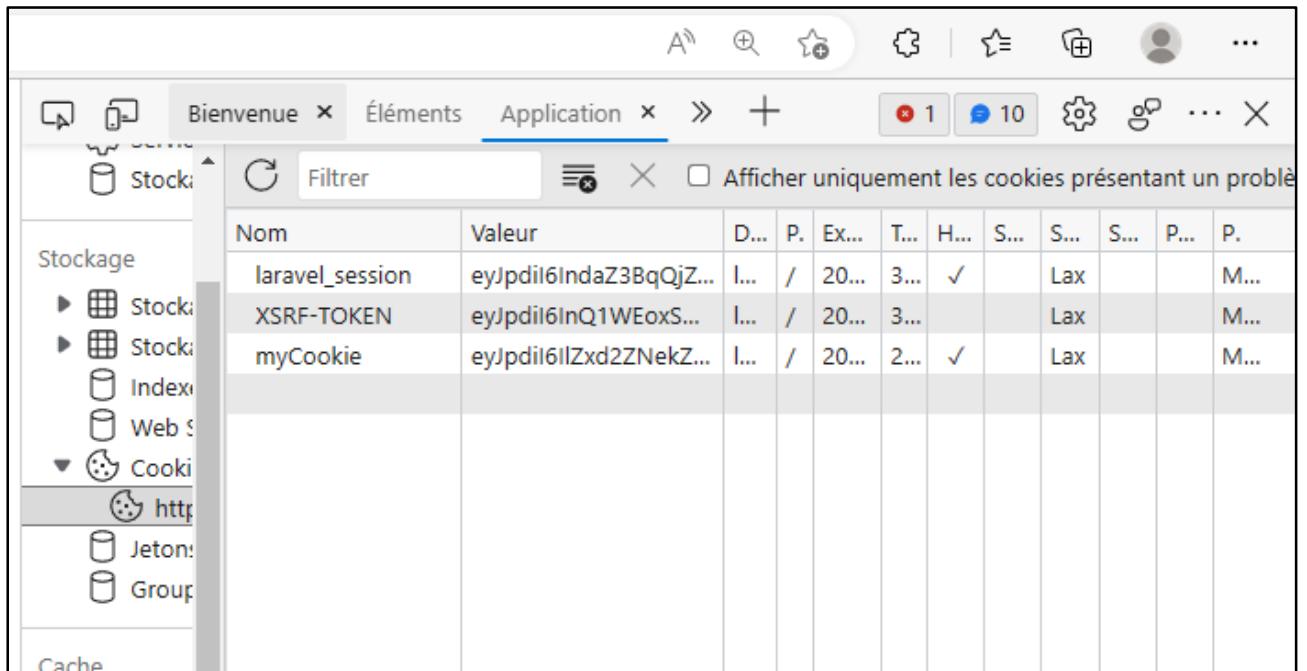
//set cookie

```
return response(view('welcome'))->cookie('myCookie','value',$min);
```

```
//get cookie
```

```
$value = $request->cookie('myCookie');  
dd($value);
```

Vous pouvez vérifier la création de ce cookie dans le navigateur



Nom	Valeur	D...	P...	Ex...	T...	H...	S...	S...	S...	P...	P...
laravel_session	eyJpdi6lndaZ3BqQjZ...	l...	/	20...	3...	✓		Lax			M...
XSRF-TOKEN	eyJpdi6lInQ1WEoxS...	l...	/	20...	3...			Lax			M...
myCookie	eyJpdi6lZxd2ZNekZ...	l...	/	20...	2...	✓		Lax			M...

C. Fichiers

1. Récupération des fichiers téléchargés (uploaded)

Vous pouvez récupérer des fichiers téléchargés à partir d'une instance `Illuminate\Http\Request` à l'aide de la méthode `file` ou à l'aide de propriétés dynamiques. La méthode `file` renvoie une instance de la classe `Illuminate\Http\UploadedFile`, qui étend la classe PHP `SplFileInfo` et fournit une variété de méthodes pour interagir avec le fichier :

```
$file = $request->file('photo');  
$file = $request->photo;
```

Vous pouvez déterminer si un fichier est présent dans la requête en utilisant la méthode `hasFile` :

```
if ($request->hasFile('picture')) {  
// }
```

2. Validation des téléchargements réussis

En plus de vérifier si le fichier est présent, vous pouvez vérifier qu'il n'y a eu aucun problème lors du téléchargement du fichier via la méthode **isValid** :

```
if ($request->file('picture')->isValid()) {  
//  
}
```

3. Chemins de fichiers et extensions

La classe **UploadedFile** contient également des méthodes pour accéder au chemin complet du fichier et à son extension. La méthode **extension** tentera de deviner l'extension du fichier en fonction de son contenu. Cette extension peut être différente de l'extension fournie par le client :

```
$path = $request->picture->path();  
$extension = $request->picture->extension();
```

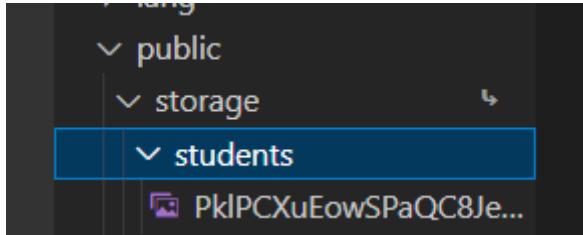
4. Stockage des fichiers téléchargés

- Pour stocker un fichier téléchargé, vous utiliserez généralement l'un de vos systèmes de fichiers configurés. La classe **UploadedFile** a une méthode **store** qui déplacera un fichier téléchargé vers l'un de vos disques, qui peut être un emplacement sur votre système de fichiers local ou un emplacement de stockage en cloud.
- La méthode **store** accepte le chemin où le fichier doit être stocké par rapport au répertoire racine configuré du système de fichiers. Ce chemin ne doit pas contenir de nom de fichier, car un identifiant unique sera automatiquement généré pour servir de nom de fichier.

- La méthode `store` accepte également un deuxième argument facultatif pour le nom du disque qui doit être utilisé pour stocker le fichier. La méthode renverra le chemin du fichier par rapport à la racine du disque :

```
$path_image=$request->picture->store('students','public');
```

Vérifiez que le fichier a été bien stocké dans `public/storage/students`



Manipulation des réponses Http :

A. Cration

1. Chanes et tableaux :

Toutes les routes et tous les contrôleurs doivent renvoyer une réponse à renvoyer au navigateur de l'utilisateur. Laravel propose plusieurs façons différentes de renvoyer des réponses. La réponse la plus basique consiste à renvoyer une chane à partir d'une route ou d'un contrôleur. Le framework convertira automatiquement la chane en une réponse HTTP complete.

En plus de renvoyer des chanes à partir de vos routes et de vos contrôleurs, vous pouvez également renvoyer des tableaux. Le framework convertira automatiquement le tableau en une réponse JSON :

```
Route::get('/url1', function () {
    return 'Hello World';
});
Route::get('/url2', function () {
    return [1, 2, 3];
});
```

2. Objets de réponse

- En rgle genale, vous ne renverrez pas simplement de simples chanes ou des tableaux à partir de vos actions de routage. Au lieu de cela, vous renverrez des instances `Illuminate\Http\Response` ou des vues completes.
- Le renvoi d'une instance complete `Response` vous permet de personnaliser le code d'état HTTP et les en-têtes de la réponse. Une instance `Response` herite de la classe `Symfony\Component\HttpFoundation\Response`, qui fournit diverses methodes pour crer des reponses HTTP :

```
Route::get('/home', function () {
    return response('Hello World', 200) ->header('Content-Type', 'text/plain');
});
```

3. Modèles et collections éloquents

Vous pouvez également renvoyer des modèles et des collections **ORM Eloquent** directement à partir de vos routes et de vos contrôleurs. Lorsque vous le faites, Laravel convertira automatiquement les modèles et collections en réponses JSON tout en respectant les attributs cachés du modèle :

```
use App\Models\User;
Route::get('/user/{user}', function (User $user) {
return $user;
});
```

4. Attacher des cookies aux réponses

Vous pouvez attacher un cookie à une instance `Illuminate\Http\Response` sortante à l'aide de la méthode `cookie`. Vous devez transmettre le nom, la valeur et le nombre de minutes pendant lesquelles le cookie doit être considéré comme valide à cette méthode

```
return response('Hello World')->cookie( 'name', 'value', $minutes );
```

Si vous souhaitez vous assurer qu'un cookie est envoyé avec la réponse sortante mais que vous n'avez pas encore d'instance de cette réponse, vous pouvez utiliser la façade `Cookie` pour "mettre en file d'attente" les cookies à attacher à la réponse lorsqu'elle est envoyée. La méthode `queue` accepte les arguments nécessaires pour créer une instance de cookie. Ces cookies seront joints à la réponse sortante avant qu'elle ne soit envoyée au navigateur :

```
use Illuminate\Support\Facades\Cookie;
Cookie::queue('name', 'value', $minutes);
return response('Hello World') ;
```

Si vous souhaitez générer une instance `Symfony\Component\HttpFoundation\Cookie` pouvant être attachée à une instance de réponse ultérieurement, vous pouvez utiliser l'assistant cookie global. Ce cookie ne sera pas renvoyé au client sauf s'il est attaché à une instance de réponse :

```
$cookie = cookie('name', 'value', $minutes);
return response('Hello World')->cookie($cookie);
```

5. Expiration anticipée des cookies

Si vous n'avez pas encore d'instance de la réponse sortante, vous pouvez utiliser la méthode `expire` de la façade `Cookie` pour faire expirer un cookie :

```
Cookie::expire('name');
```

6. Cookies et cryptage

Par défaut, tous les cookies générés par Laravel sont cryptés et signés afin qu'ils ne puissent pas être modifiés ou lus par le client. Si vous souhaitez désactiver le chiffrement pour un sous-ensemble de cookies générés par votre application, vous pouvez utiliser la propriété `$except` du middleware `App\Http\Middleware\EncryptCookies`, qui se trouve dans le répertoire `app/Http/Middleware` :

```
<?php

namespace App\Http\Middleware;

use Illuminate\Cookie\Middleware\EncryptCookies as Middleware;

class EncryptCookies extends Middleware
{
    /**
     * The names of the cookies that should not be encrypted.
     *
     * @var array<int, string>
     */
    protected $except = [
        'firstCookie'
    ];
}
```

Cookies en cours d'utilisation

Autorisé Bloqué

Les cookies suivants ont été définis lorsque vous avez affiché cette page

 XSRF-TOKEN
 firstCookie
 laravel_session

Nom firstCookie

Contenu my-secret

Domaine localhost

Chemin /

Envoyer... Connexions au même site uniquement

Créé lundi 23 janvier 2023 à 12:11:59

Expire lundi 23 janvier 2023 à 12:13:13

[Bloquer](#) [Supprimer](#) [Terminé](#)

B. Redirection

```
Route::get('/dashboard', function () {  
    return redirect('home/dashboard');  
});
```

1. Redirection vers des routes nommées

Lorsque vous appelez l'assistant `redirect` sans paramètre, une instance de `Illuminate\Routing\Redirector` est renvoyée, vous permettant d'appeler n'importe quelle méthode sur l'instance `Redirector`. Par exemple, pour

générer un **RedirectResponse** vers une route nommée, vous pouvez utiliser la méthode `route` :

```
return redirect()->route('login');
```

Si votre route a des paramètres, vous pouvez les passer comme deuxième argument à la méthode `route` :

```
// pour une route avec l'URI suivant:  
/profile/{id}  
return redirect()->route('profile', ['id' => 1]);
```

2. Remplir les paramètres via des modèles éloquents

Si vous redirigez vers une route avec un paramètre "ID" qui est renseigné à partir d'un modèle Eloquent, vous pouvez transmettre le modèle lui-même. L'ID sera extrait automatiquement :

```
// pour une route avec l'URI suivant : /profile/{id}  
return redirect()->route('profile',$student);  
or  
<a href="{{route('profile',$student)}}"><h1>{{$student->name}}</h1> </a>
```

Si vous souhaitez personnaliser la valeur placée dans le paramètre `route`, vous pouvez spécifier la colonne dans la définition du paramètre `Route`.

```
Route::get('/students/{student:name}',[StudentController::class,'show'])-  
>name('students.show');
```

3. Redirection vers des domaines externes

Parfois, vous devrez peut-être rediriger vers un domaine en dehors de votre application. Vous pouvez le faire en appelant la méthode `away`, qui crée un **RedirectResponse** sans aucun codage, validation ou vérification d'URL supplémentaire :

```
return redirect()->away('https://www.google.com');
```

4. Redirection avec des données de session flashées

La redirection vers une nouvelle URL et le flashage des données vers la session sont généralement effectués en même temps. En règle générale, cela se fait après avoir effectué une action avec succès lorsque vous envoyez un message de réussite à la session. Pour plus de commodité, vous pouvez créer une instance `RedirectResponse` et envoyer des données flash à la session dans une seule chaîne de méthodes fluide :

```
Route::post('/user/profile', function () {
// ...
return redirect('dashboard')->with('success', 'Profile updated!');
});
```

Une fois l'utilisateur redirigé, vous pouvez afficher le message flashé de la session. Par exemple, en utilisant la syntaxe `Blade` :

```
@if (Session::has('success'))
{{Session::get('success')}}
@endif
```

C. Types de réponse

1. Afficher les vues (views)

Vous pouvez également renvoyer une vue comme contenu de la réponse, vous devez utiliser la méthode `view` :

```
return view('students.index');
```

2. Réponses JSON

La méthode `json` va automatiquement définir l'en-tête Content-Type à `application/json`, et convertir le tableau donné en JSON:

```
$data = [
[
```

```

        "id" => 7,
        "name" => "na like this",
        "description" => "",
    ],
    [
        "id" => 5,
        "name" => "write a book",
        "description" => "hohoho",
    ]
);
return response()->json($data);

```

3. Téléchargements de fichiers

La méthode **download** peut être utilisée pour générer une réponse qui force le navigateur de l'utilisateur à télécharger le fichier au chemin donné. La méthode **download** accepte un nom de fichier comme deuxième argument de la méthode, qui déterminera le nom de fichier qui est vu par l'utilisateur téléchargeant le fichier.

```
return response()->download("storage/students/photo1.png");
```

4. Téléchargements en streaming

Parfois, vous souhaiterez peut-être transformer la réponse de chaîne d'une opération donnée en une réponse téléchargeable sans avoir à écrire le contenu de l'opération sur le disque. Vous pouvez utiliser la méthode **streamDownload** dans ce scénario. Cette méthode accepte une fonction de rappel, un nom de fichier et un tableau facultatif d'en-têtes comme arguments :

```

return response()->streamDownload(function () {
    echo 'CSV Contents...';
},
'downloadme.csv',
[
    'Content-Type' => 'text/csv',
]
);

```

5. Fichier de réponses

La méthode `file` peut être utilisée pour afficher un fichier, tel qu'une image ou un PDF, directement dans le navigateur de l'utilisateur au lieu de lancer un téléchargement. Cette méthode accepte le chemin d'accès au fichier comme premier argument et un tableau d'en-têtes comme deuxième argument :

```
return response()->file('storage/laravel-fr.pdf');
```

6. Réponse Marco

Si vous souhaitez définir une réponse personnalisée que vous pouvez réutiliser dans une variété de vos routes et contrôleurs, vous pouvez utiliser la méthode `macro` sur la façade `Response`. En règle générale, vous devez appeler cette méthode à partir de la méthode `boot` de l'un des fournisseurs de services de votre application, tel que le fournisseur de services : `App\Providers\AppServiceProvider`

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Illuminate\Support\Facades\Response;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
```

```
}

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Response::macro('caps', function ($value) {
        return Response::make(strtoupper($value));
    });

}

}
```

La fonction `macro` accepte un nom comme premier argument et une fermeture comme deuxième argument. La fermeture de la macro sera exécutée lors de l'appel du nom de la macro depuis une implémentation :

```
Route::get('/', function () {

    return response()->caps('morocco');
});
```


Manipulation des vues:

A. Crédit

1. Introduction :

Bien sûr, il n'est pas pratique de renvoyer des chaînes de documents HTML entières directement à partir de vos routes et de vos contrôleurs. Heureusement, les vues offrent un moyen pratique de placer tout notre code HTML dans des fichiers séparés. Les vues séparent la logique de votre contrôleur/application de votre logique de présentation et sont stockées dans le répertoire `resources/views`. Une vue simple pourrait ressembler à ceci :

```
<!--vue stockée dans resources/views/home.blade.php-->
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
<body>
    <h1> Home page , {{$name}} </h1>
</body>
</html>
```

Nous pouvons la renvoyer en utilisant le helper global `view` comme ceci :

```
Route::get('/', function () {
    return view('home', ['name' => 'James']);
});
```

L'extension `.blade.php` informe le `framework` que le fichier contient un modèle `Blade`. Les modèles de Blade contiennent du HTML ainsi que des directives Blade qui vous permettent de faire facilement écho des valeurs, de créer des instructions "if", d'itérer sur les données, etc.

2. Répertoires de vue imbriqués :

Les vues peuvent également être imbriquées dans des sous-répertoires du répertoire `resources/views`. La notation "Point" peut être utilisée pour référencer des vues imbriquées. Par exemple, si votre vue est stockée dans `resources/views/admin/profile.blade.php`, vous pouvez la renvoyer depuis l'une des routes/contrôleurs de votre application comme suit :

```
return view('admin.profile', $data);
```

3. Déterminer si une vue existe

Si vous avez besoin de déterminer si une vue existe, vous pouvez utiliser la façade `View`. La méthode `exists` retournera `true` si la vue existe :

```
use Illuminate\Support\Facades\View;  
if (View::exists('emails.customer')) {  
//  
}
```

B. Transmission des données

1. Introduction :

Comme vous l'avez vu dans les exemples précédents, vous pouvez passer un tableau de données aux vues pour rendre ces données disponibles à la vue :

```
return view('home', ['name' => 'Victoria']);
```

Lors de la transmission d'informations de cette manière, les données doivent être un tableau avec des paires **clé/valeur**. Après avoir fourni des données à une vue, vous pouvez ensuite accéder à chaque valeur de votre vue à l'aide des clés de données, telles que `{{$name}}`

Au lieu de transmettre un tableau complet de données à la fonction helper `view`, vous pouvez utiliser la méthode `with` pour ajouter des éléments de

données individuels à la vue. La méthode `with` renvoie une instance de l'objet `view` afin que vous puissiez continuer à enchaîner les méthodes avant de renvoyer la vue :

```
return view('home')
    ->with('name', 'Victoria')
    ->with('occupation', 'Astronaut');
```

2. Partage de données avec toutes les vues

Parfois, vous devrez peut-être partager des données avec toutes les vues rendues par votre application. Vous pouvez le faire en utilisant la méthode `share` de la façade `View`. En règle générale, vous devez placer des appels à la méthode `share` dans la méthode d'un fournisseur de services boot. Vous êtes libre de les ajouter à la classe `App\Providers\AppServiceProvider` ou de générer un fournisseur de services distinct pour les héberger :

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\View;
use Illuminate\Support\ServiceProvider;
use Illuminate\Support\Facades\Response;

class AppServiceProvider extends ServiceProvider
{
    public function register()
    {
        //
    }

    public function boot()
    {
        View::share('key', 'value');
    }
}
```

C. Composeurs des vues

1. Introduction

Les composeurs de vue sont des rappels ou des méthodes de classe qui sont appelées lorsqu'une vue est rendue. Si vous avez des données que vous souhaitez lier à une vue chaque fois que cette vue est rendue, un composeur de vue peut vous aider à organiser cette logique en un seul emplacement. Les composeurs de vues peuvent s'avérer particulièrement utiles si la même vue est renvoyée par plusieurs routes ou contrôleurs au sein de votre application et a toujours besoin d'un élément de données particulier.

- En règle générale, les composeurs de vues seront enregistrés auprès de l'un des fournisseurs de services de votre application.

Nous utiliserons la méthode `composer` de la façade `View` pour enregistrer le composeur de vue :

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\View;
use Illuminate\Support\ServiceProvider;
use Illuminate\Support\Facades\Response;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
```

```
View::composer('pages.home', function ($view) {
    $view->with('title', 'how to be a great programmer');
});

}
```

Maintenant que nous avons enregistré le composeur, le contenu de la fonction de rappel sera exécutée à chaque ‘**home**’ de la vue est rendu.

Vous pouvez attacher un composeur de vues à plusieurs vues à la fois en passant un tableau de vues comme premier argument à la méthode **composer** :

```
public function boot()
{
    View::composer(['pages.home', 'pages.welcome'], function ($view) {
        $view->with('mytitle', 'how to be a great programmer');
    });

}
```

La méthode **composer** accepte également le caractère ***** comme caractère générique, vous permettant d'attacher un composeur à toutes les vues :

```
View::composer('*', function ($view) {
// 
});
```


Création des template Blade:

1. Introduction :

Le **Blade** est un puissant moteur de templating dans un framework Laravel. Le blade permet d'utiliser le moteur de templating facilement, et il rend l'écriture de la syntaxe très simple. Le moteur de templating blade fournit sa propre structure comme les instructions conditionnelles et les boucles. Pour créer un modèle de blade, il suffit de créer un fichier de vue et de l'enregistrer avec une extension **.blade.php** au lieu de **.php**. Les modèles de blade sont stockés dans le répertoire **/resources/view**. Le principal avantage de l'utilisation du modèle de blade est que nous pouvons créer le modèle principal, qui peut être étendu par d'autres fichiers.

2. Affichage des données :

Si vous souhaitez imprimer la valeur d'une variable, il vous suffit de la placer entre des accolades.

Syntaxe

```
 {{$variable}} ;
```

Dans le modèle de blade, nous n'avons pas besoin d'écrire le code php
`<?php echo $variable ; ?>`

3. Opérateur ternaire :

Dans le modèle de blade, la syntaxe de l'opérateur ternaire peut s'écrire comme suit :

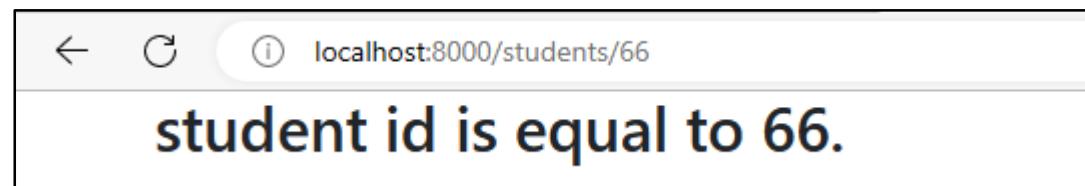
```
 {{ isset($mytitle) ? $mytitle : "empty variable"}}
```

4. Instructions de contrôle

Le moteur de templating Blade fournit également les instructions de contrôle dans Laravel ainsi que des raccourcis pour les instructions de contrôle.

```
@if ($student->id==66)
    student id is equal to 66.
@else
    student id is not equal to 66.
@endif
```

//output



En plus des directives conditionnelles déjà abordées, les directives **@isset** et **@empty** peuvent être utilisées comme raccourcis pratiques pour leurs fonctions PHP respectives :

```
@isset($mytitle)
    is defined and is not null...
@endisset

@empty($mytitlep)
    $mytitlep is empty
@endempty
```

5. Boucles Blade

Le moteur de modélisation blade fournit des boucles telles que les directives **@for**, **@endfor**, **@foreach**, **@endforeach**, **@while** et **@endwhile**.

test.blade.php

```
@for ($i = 0; $i < 10; $i++)
    The current value is {{ $i }}
@endfor

@foreach ($users as $user)
    <p>This is user {{ $user->id }}</p>
@endforeach

@forelse ($users as $user)
    <li>{{ $user->name }}</li>
@empty
    <p>No users</p>
@endforelse

@while(($i)<5)
    javapoint
    {{$i++}}
@endwhile
```

//output



Lorsque vous utilisez des boucles, vous pouvez également sauter l'itération en cours ou terminer la boucle à l'aide des directives **@continue** et **@break** :

```

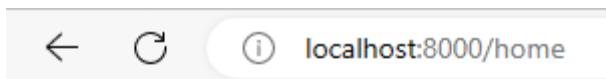
@foreach ($users as $user)
@if ($user->id == 1)
@continue
@endif

<li>{{ $user->name }}</li>

@if ($user->id == 5)
@break
@endif
@endforeach

```

//output



Home page

- 2
- 3
- 4
- 5

La variable Loop

Lors de l'itération d'une boucle **foreach**, une variable **\$loop** sera disponible à l'intérieur de votre boucle. Cette variable permet d'accéder à certaines informations utiles, comme l'index actuel de la boucle et le fait qu'il s'agisse de la première ou de la dernière itération de la boucle :

```

@foreach ($users as $user)
@if ($loop->first)
    This is the first iteration.
@endif

```

```

@if ($loop->last)
    This is the last iteration.

@endif

<p>This is user {{ $user->id }}</p>
@endforeach

```

6. Classes conditionnelles

La directive `@class` compile de manière conditionnelle une chaîne de classes CSS. La directive accepte un tableau de classes où la clé du tableau contient la ou les classes que vous souhaitez ajouter, tandis que la valeur est une expression booléenne. Si l'élément du tableau a une clé numérique, il sera toujours inclus dans la liste des classes rendues :

Si nous voulons montrer l'utilisateur actif en vert et l'utilisateur inactif en rouge, nous pouvons utiliser la directive lame `@if @endif`.

```

@php
$active = true;
@endphp
@if ($active)
<span class="p-2 text-success">user</span>
@else
<span class="p-2 text-danger">user</span>
@endif

```

Afficher l'utilisateur actif en vert et l'utilisateur inactif en rouge
Directive blade `@class`.

```

<span
@class([
    'p-2',
    'p-2 text-success' => $active,
    'p-2 text-danger' => !$active,
])>user
</span>

```

7. PHP brut

Dans certaines situations, il est utile d'intégrer du code PHP dans vos vues. Vous pouvez utiliser la directive `@php` de Blade pour exécuter un bloc de code PHP dans votre modèle :

```
@php
    $counter = 1 ;
@endphp
```

Si vous n'avez besoin d'écrire qu'une seule instruction PHP, vous pouvez inclure l'instruction dans la directive `@php` :

```
@php($counter = 1)
```

8. L'héritage

On a vu qu'une vue peut en étendre une autre, c'est un héritage. Ainsi pour les vues de l'application, on a un template de base :

Ce Template comporte la structure globale des pages et est déclaré comme parent par les autres vues :

```
@extends('template')
```

Dans le template on prévoit un emplacement (`@yield`) pour que les vues enfants puissent placer leur code :

```
<main class="section">
    <div class="container">
        @yield('content')
    </div>
</main>
```

Ainsi dans la vue `index.blade.php` on utilise cet emplacement :

```
@section('content')
// Code de la vue
```

```
@endsection
```

Activité

- **Étape 1:** Créez un nouveau dossier «layouts» dans le répertoire `/resources/views/` .
- **Étape 2:** Créez un nouveau fichier «`master.blade.php`» dans le répertoire `/resources/views/layouts/`.
- **Étape 3:** Copiez le code suivant dans le fichier “`master.blade.php`” que nous avons créé.

```
<html>
  <head>
    <title>App Name - @yield('title')</title>
  </head>
  <body>
    @section('sidebar')
      This is the master sidebar.
    @show
    <div class="container">
      @yield('content')
    </div>
  </body>
</html>
```

- Ici, dans le modèle principal ci-dessus –
 - `@yield` (‘title’) est utilisé pour afficher la valeur du titre
 - `@section` (‘sidebar’) est utilisée pour définir une section nommée **sidebar**
 - `@show` est utilisé pour afficher le contenu d’une section
 - `@yield` (‘content’) est utilisé pour afficher le contenu du contenu
- **Extension de la mise en page principale**
 - Nous allons maintenant vous montrer comment étendre la mise en page principale que nous venons de créer.
 - **Étape 1:** Créez un nouveau fichier de vue `page.blade.php` dans `/resources/views/`
 - **Étape 2:** Copiez le code suivant dans le fichier `page.blade.php`

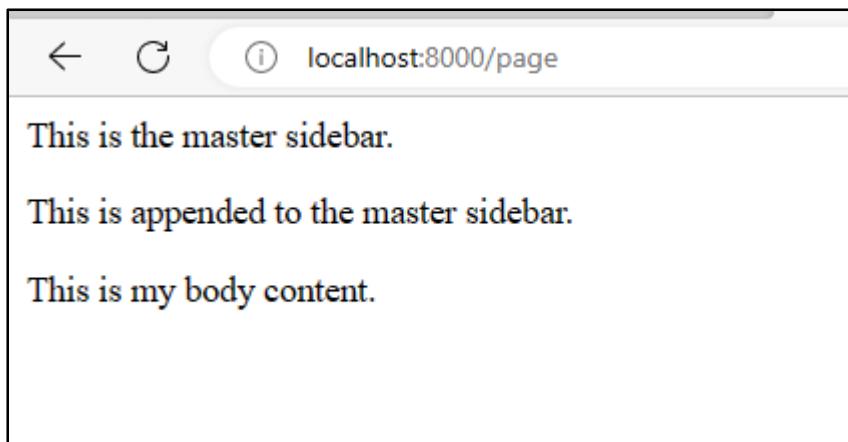
```
<!-- path: resources/views/page.blade.php -->
@extends('layouts.master')
@section('title', 'Page Title')
@section('sidebar')
  @parent
  <p>This is appended to the master sidebar.</p>
@endsection
@section('content')
  <p>This is my body content.</p>
```

```
@endsection
```

- Ici, dans la page ci-dessus
 - `@extends` ('layouts.master') étend la mise en page principale
 - `@section` ('title', 'Page Title') définit la valeur de la section de titre.
 - `@section` ('sidebar') définit une section de barre latérale dans la page enfant de la disposition principale
 - `@parent` affiche le contenu de la section de barre latérale, définie dans la disposition principale.
 - Ceci est ajouté à la barre latérale principale.
 - `@endsection` termine la section de la barre latérale
 - `@section` ('content') définit la section de contenu
 - `@endsection` termine la section de contenu
-
- **Étape 3:** Ouvrez `routes/web.php` et configuez l'itinéraire comme ci-dessous:

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

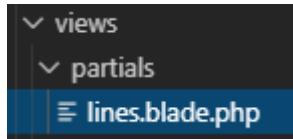
- **Étape 4:** Ouvrez maintenant l'URL suivante dans le navigateur pour voir la sortie.
- <http://localhost:8000/page>



9. L'inclusion

On peut faire beaucoup de choses avec l'héritage, mais il est souvent utile de pouvoir inclure une vue dans une autre, classiquement on parle de vue partielle (partial).

On peut mettre le code qui génère ces lignes dans une vue partielle :



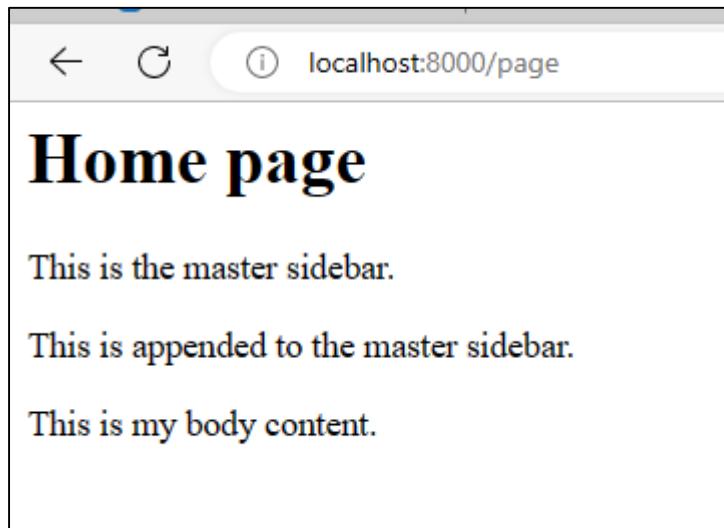
Copiez le code suivant dans le fichier `lines.blade.php`

```
<h1>Home page</h1>
```

Dans le code de la vue `page`, vous rajoutez la ligne suivante :

```
@include('partials.lines')
```

`//output`



10. Erreurs de validation

La directive `@error` peut être utilisée pour vérifier rapidement si des messages d'erreur de validation existent pour un attribut donné. Dans une directive `@error`, vous pouvez faire écho à la variable `$message` pour afficher le message d'erreur :

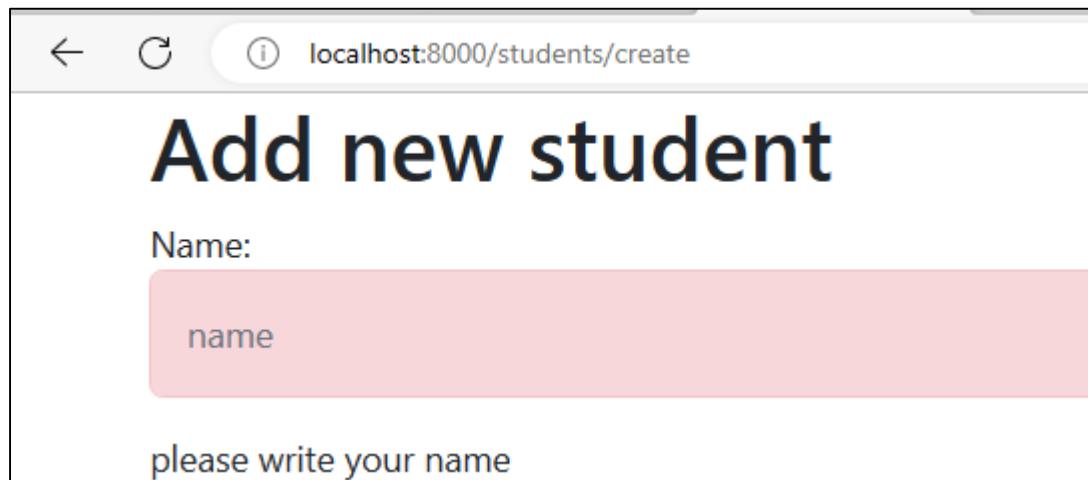
```
<!-- /resources/views/post/create.blade.php -->

<label for="title">Post Title</label>

<input id="title"
      type="text"
```

```
class=" @error('title') is-invalid @else is-valid @enderror">>

@error('title')
    <div class="alert alert-danger">{{ $message }}</div>
@enderror
```



localhost:8000/students/create

Add new student

Name:

please write your name

Génération d'URL:

1. Introduction :

Laravel fournit plusieurs aides pour vous aider à générer des URL pour votre application. Ces aides sont principalement utiles pour créer des liens dans vos modèles et vos réponses API, ou pour générer des réponses de redirection vers une autre partie de votre application.

2. Les bases

a. Génération d'URLs

Le helper `url` peut être utilisé pour générer des URL arbitraires pour votre application. L'URL générée utilisera automatiquement le schéma (HTTP ou HTTPS) et l'hôte de la requête en cours de traitement par l'application :

```
@php
    $student=App\Models\Student::find(66);
@endphp
<a href="{{url('/students/$student->id')}}">show</a>
```

b. Accéder à l'URL courante

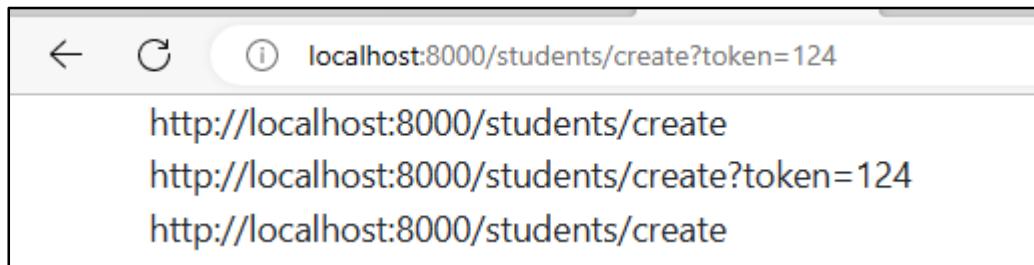
Si aucun chemin d'accès n'est fourni à l'aide `url`, une instance `Illuminate\Routing\UrlGenerator` est renvoyée, ce qui vous permet d'accéder aux informations sur l'URL actuelle :

```
{{-- // Get the current URL without the query string... --}}
{{url()->current()}}

{{-- // Get the current URL including the query string... --}}
{{url()->full()}}

{{-- // Get the full URL for the previous request... --}}
{{url()->previous()}}
```

```
//output
```



3. URLs pour les routes nommées

Le helper `route` peut être utilisé pour générer des URL vers des routes nommées. Les routes nommées vous permettent de générer des URL sans être couplé à l'URL réelle définie sur la route. Par conséquent, si l'URL de la route change, aucune modification ne doit être apportée à vos appels à la fonction `route`. Par exemple, imaginez que votre application contient une route définie comme suit :

```
Route::get('/students/{student}', [StudentController::class, 'show'])  
->name('students.show');
```

Pour générer une URL vers cette route, vous pouvez utiliser le helper `route` comme suit :

```
<a href="{{route('students.show', $student)}}><h1>{{$student->name}}</h1> </a>
```

a. URLs signées

Laravel vous permet de créer facilement des URL "signées" vers des routes nommées. Ces URL ont un hachage de "signature" ajouté à la chaîne de requête, ce qui permet à Laravel de vérifier que l'URL n'a pas été modifiée depuis sa création. Les URL signées sont particulièrement utiles pour les routes qui sont accessibles au public mais qui nécessitent une couche de protection contre la manipulation des URL.

Par exemple, vous pouvez utiliser des URL signées pour mettre en place un lien public de "profile" qui est envoyé par courriel à vos clients. Pour créer une URL signée vers un itinéraire nommé, utilisez la méthode `signedRoute` de la façade URL :

```
<a href="{{URL::signedRoute('students.show',$student)}}><h1>{{$student->name}}</h1> </a>
```

Le code ci-dessus retournera une chaîne de caractères comme ceci :

localhost:8000/students/66?signature=bc6c9281b12d4ca90acb0a6d428391a5ed8c36612258d42aa23c419739ae2ad1

http://localhost:8000/students/66?signature=bc6c9281b12d4ca90acb0a6d428391a5ed8c36612258d42aa23c419739ae2ad1

Si vous souhaitez générer une URL de route signée temporaire qui expire après un laps de temps déterminé, vous pouvez utiliser la méthode `temporarySignedRoute`. Lorsque Laravel valide une URL de route signée temporaire, il s'assure que l'horodatage d'expiration encodé dans l'URL signée n'est pas écoulé :

```
<a href="{{URL::temporarySignedRoute('students.show', now()->addMinutes(1),$student)}}>
<h1>{{$student->name}}</h1>
</a>
```

Maintenant que nous avons l'URL signée prête à être utilisée, nous devons valider les demandes entrantes et pour ce faire, nous avons deux options :

Avec un middleware

Laravel propose un middleware signé que nous pouvons utiliser. Tout d'abord, nous devons nous assurer que nous incluons le middleware dans `App\Http\Kernel`

```
protected $routeMiddleware = [
    ...
    'signed' => \App\Http\Middleware\ValidateSignature::class,
];
```

Maintenant nous pouvons l'utiliser dans notre fichier routes comme ceci :

```
Route::get('/students/{student}', [StudentController::class, 'show'])->name('students.show')->middleware('signed');
```

Utilisation de la méthode `hasValidSignature()` sur l'objet de requête

Ou nous pouvons vérifier que la demande entrante a une signature valide à l'intérieur de l'action du contrôleur:

```
public function show(Request $request, Student $student)
{
    if (! $request->hasValidSignature()) {
        abort(401);
    }
    return view('students.profile', ['student' => $student]);
}
```

4. URLs pour les actions du contrôleur

La fonction `action` génère une URL pour l'action de contrôleur donnée :

```
<a href="{{ action([App\Http\Controllers\StudentController::class, 'index'])}}>
    Return to students list
</a>
```

Si la méthode du contrôleur accepte des paramètres de route, vous pouvez transmettre un tableau associatif de paramètres de route comme deuxième argument de la fonction :

```
<a href="{{action([App\Http\Controllers\StudentController::class, 'show'], [
    'student'=>$student])}}><h1>{{$student->name}}</h1> </a>
```


Session:

1. Introduction :

Comme les applications HTTP sont sans état, les sessions permettent de stocker des informations sur l'utilisateur à travers plusieurs requêtes. Ces informations sur l'utilisateur sont généralement placées dans un **store / backend** persistant auquel il est possible d'accéder lors de requêtes ultérieures.

Dans laravel, le fichier de configuration de session est stocké dans ‘`app/config/session.php`’.

2. Interagir avec la session

2.1. Récupération des données

Il y a deux façons principales de travailler avec les données de session dans Laravel : avec le helper `session` global et via une instance de `Request`. Tout d'abord, examinons l'accès à la session via une instance de requête, qui peut être indiquée par un type sur une fermeture de route ou une méthode de contrôleur. N'oubliez pas que les dépendances des méthodes de contrôleur sont automatiquement injectées par le conteneur de services Laravel :

```
public function show(Request $request, Student $student)
{
    $value = $request->session()->get('_token', 'default');
    dd($value);
    return view('students.profile', ['student' => $student]);
}
```

//output

```
"c3mc8An5ySbbbIrVxWsHdy0Lh8InycGrx2bTkwe" // app\Http\Controllers\StudentController.php:72
```

Lorsque vous récupérez un élément dans la session, vous pouvez également passer une valeur par défaut comme deuxième argument de la méthode `get`. Cette valeur par défaut sera renvoyée si la clé spécifiée n'existe pas

dans la session. Si vous passez une fermeture comme valeur par défaut à la méthode `get` et que la clé demandée n'existe pas, la fermeture sera exécutée et son résultat sera renvoyé :

```
$value = $request->session()->get('key', 'default');

$value = $request->session()->get('key', function () {
    return 'default';
});
```

- Le helper session

Vous pouvez également utiliser la fonction PHP de session globale pour récupérer et stocker des données dans la session. Lorsque la fonction de session est appelée avec un argument unique de type chaîne, elle renvoie la valeur de la clé de session. Lorsque le helper est appelé avec un tableau de paires `clé/valeur`, ces valeurs seront stockées dans la session :

```
Route::get('/home', function () {
    // Retrieve a piece of data from the session...
    $value = session('key');

    // Specifying a default value...
    $value = session('key', 'default');

    // Store a piece of data in the session...
    session(['key' => 'value']);
});
```

- Récupération de toutes les données de la session

Si vous souhaitez récupérer toutes les données de la session, vous pouvez utiliser la méthode `all` :

```
$data = $request->session()->all() ;  
//output
```

```
array:4 [▼ // app\Http\Controllers\StudentController.php:80  
  "_token" => "7z1rYDhqqtA9uMe40l5XE2C2NsD2NhxDuV7tHrJv"  
  "key" => "value"  
  "_previous" => array:1 [▶]  
  "_flash" => array:2 [▶]  
]
```

- Déterminer si un élément existe dans la session

Pour déterminer si un élément est présent dans la session, vous pouvez utiliser la méthode `has`. La méthode `has` renvoie `true` si l'élément est présent et n'est pas nul :

```
if ($request->session()->has('users')) {  
    //  
}
```

Pour déterminer si un élément est présent dans la session, même si sa valeur est nulle, vous pouvez utiliser la méthode `exists` :

```
if ($request->session()->exists('users')) {  
    //  
}
```

2.2. Stockage des données

Pour stocker des données dans la session, vous utiliserez généralement la méthode `put` de l'instance `Request` ou via la fonction `session` :

```
// Via a request instance...  
$request->session()->put('key', 'value');  
// Via the global "session" helper...  
session(['key' => 'value']);
```

- Insérer des valeurs de session de type tableau

La méthode `push` peut être utilisée pour insérer une nouvelle valeur dans une valeur de session qui est un tableau. Par exemple, si la clé `user.teams` contient un tableau de noms d'équipes, vous pouvez insérer une nouvelle valeur dans le tableau de la manière suivante :

```
$request->session()->put('teams', ['programmers']);
$request->session()->push('teams', 'developers');
dd(session('teams'));
```

//output

```
array:2 [▼ // app\Http\Controllers\StudentController.php:82
  0 => "programmers"
  1 => "developers"
]
```

- Récupérer un élément et l'oublier

La méthode `pull` permet de récupérer et de supprimer un élément de la session en une seule instruction :

```
$value = Session::pull('key', 'default');
$value = $request->session()->pull('key', 'default');
```

- Incrémation et décrémation des valeurs de session

Si vos données de session contiennent un nombre entier que vous souhaitez incrémenter ou décrémenter, vous pouvez utiliser les méthodes d'incrémation et de décrémation :

```
$request->session()->increment('count');

$request->session()->increment('count', $incrementBy = 2);

$request->session()->decrement('count');

$request->session()->decrement('count', $decrementBy = 2);
```

```
echo Session::get('count'); // display value of count 2 4 6 8 ...
```

- Données flash

Il peut arriver que vous souhaitiez stocker des éléments dans la session pour la prochaine demande. Vous pouvez le faire en utilisant la méthode **flash**. Les données stockées dans la session à l'aide de cette méthode seront disponibles immédiatement et pendant la demande HTTP suivante. Après la demande HTTP suivante, les données flashées seront supprimées. Les données flash sont principalement utiles pour les messages d'état de courte durée :

```
$request->session()->flash('status', 'Task was successful!');
```

- Régénérer l'ID de session

Si vous souhaitez régénérer tous les **ID** de la session, vous pouvez utiliser la méthode **regenerate()**.

```
Session::regenerate();  
$request->session()->regenerate();
```

- Suppression d'un élément de la session

La méthode **forget** supprimera l'élément spécifié de la session.

Si vous souhaitez d'abord obtenir la valeur de l'élément, puis supprimer cet élément de la session, vous pouvez utiliser la méthode **pull**.

La différence entre la méthode **forget()** et la méthode **pull()** est que:

- La méthode **forget()** ne retournera pas la valeur de la session.
- La méthode **pull()** la retournera et supprimera cette valeur de la session.

```
// Forget a single key...  
$request->session()->forget('name');  
  
// Forget multiple keys...  
$request->session()->forget(['name', 'status']);
```

La méthode **forget** permet de supprimer un élément de données de la session. Si vous souhaitez supprimer toutes les données de la session, vous pouvez utiliser la méthode **flush** :

```
$request->session()->flush();  
Session::flush();
```

1. Activité

Étape 1 – Créez un contrôleur appelé **SessionController**.

Étape 2 – Copiez le code suivant dans un fichier à **app/Http/Controllers/SessionController.php**.

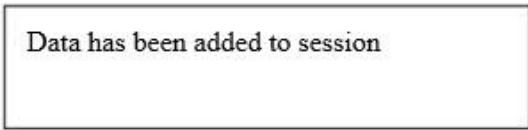
```
<?php  
  
namespace App\Http\Controllers;  
  
use Illuminate\Http\Request;  
use App\Http\Requests;  
use App\Http\Controllers\Controller;  
  
class SessionController extends Controller {  
    public function accessSessionData(Request $request) {  
        if($request->session()->has('my_name'))  
            echo $request->session()->get('my_name');  
        else  
            echo 'No data in the session';  
    }  
    public function storeSessionData(Request $request) {  
        $request->session()->put('my_name', 'Virat Gandhi');  
        echo "Data has been added to session";  
    }  
    public function deleteSessionData(Request $request) {  
        $request->session()->forget('my_name');  
        echo "Data has been removed from session.";  
    }  
}
```

Étape 3 – Ajoutez les lignes suivantes dans le fichier app/Http/routes.php .

```
Route::get('session/get',[SessionController::class, 'accessSessionData'] );
Route::get('session/set',[SessionController::class, 'storeSessionData']);
Route::get('session/remove',[SessionController::class, 'deleteSessionData']);
```

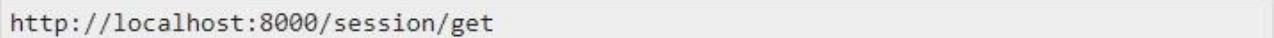
- **Étape 4** – Visitez l’URL suivante pour définir les données en session :
<http://localhost:8000/session/set> .

Étape 5 - La sortie apparaîtra comme indiqué dans l’image suivante.



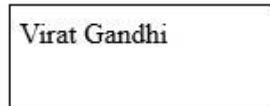
Data has been added to session

Étape 6 - Visitez l’URL suivante pour **obtenir les données de la session** ,



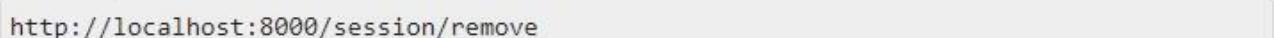
<http://localhost:8000/session/get>

Étape 7 - La sortie apparaîtra comme indiqué dans l’image suivante.



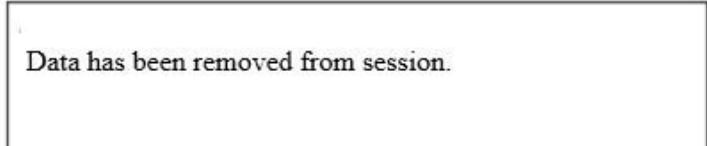
Virat Gandhi

Étape 8 - Visitez l’URL suivante pour **supprimer les données de session** .



<http://localhost:8000/session/remove>

Étape 9 - Vous verrez un message comme indiqué dans l’image suivante.



Data has been removed from session.

Validation des données d'entrée:

1. Introduction :

Nous avons vu dans le chapitre précédent un scénario mettant en œuvre un formulaire. Nous n'avons imposé aucune contrainte sur les valeurs transmises. Dans une application réelle, il est toujours nécessaire de vérifier que ces valeurs correspondent à ce qu'on attend. Par exemple un nom doit comporter uniquement des caractères alphabétiques et avoir une longueur maximale ou minimale, une adresse email doit correspondre à un certain format...

Il faut donc mettre en place des règles de validation. En général on procède à une première validation côté client pour éviter de faire des allers retours avec le serveur. Mais quelle que soit la pertinence de cette validation côté client elle n'exonère pas d'une validation côté serveur.

On ne doit jamais faire confiance à des données qui arrivent sur le serveur !

Dans l'exemple de ce chapitre je ne prévoirai pas de validation côté client, d'une part ce n'est pas mon propos, d'autre part elle masquerait la validation côté serveur pour les tests.

2. Validation manuelle

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Validator;
class ContactController extends Controller
{
    public function create()
    {
        return view('contact');
    }
    public function store(Request $request)
    {
```

```

$rules=[  

    'name' => ['required', 'between:5,20'],  

    'mail' => ['required', 'email'],  

    'section' => ['required', 'max:250'],  

    'picture'=>['required','max:1024','image'],  

]  

$messages=[  

    'name.required'=>'please write your name',  

    'picture.max'=>'this picture is greater than 1024 kb'  

];  

$validator = Validator::make($request->all(),$rules , $messages  

);  

if ($validator->fails()) {  

    return back()->withErrors($validator)->withInput();  

}  

//save data  

    $path_image=$request->picture->store('students','public');  

    $student= new Student();  

    $student->name=$request->name;  

    $student->mail=$request->mail;  

    $student->section=$request->section;  

    $student->picture=$path_image;  

    $student->save();  

    session()->flash('success', 'student saved successfully');  

    return redirect()->route('students.create');  

}  

}
}

```

On utilise la façade **Validator** en précisant toutes les entrée (`$request->all()`) et les règles de validation. Ensuite si la validation échoue (`fails`) on renvoie (`back`) le formulaire avec les erreurs (`withErrors`) et les valeurs entrées (`withInput`) pour pouvoir les afficher dans le formulaire.

3. Arrêt au premier échec de validation

Il peut arriver que vous souhaitez arrêter l'exécution des règles de validation sur un attribut après le premier échec de validation. Pour ce faire, affectez la règle de validation à l'attribut :

```

$request->validate([  

    'title' => 'bail|required|unique:posts|max:255',  

    'body' => 'required',  

]
)

```

```
]);
```

Dans cet exemple, si la règle **unique** sur l'attribut **title** échoue, la règle **max** ne sera pas vérifiée. Les règles seront validées dans l'ordre où elles sont attribuées.

4. Affichage des erreurs de validation

En cas de réception du formulaire suite à des erreurs on reçoit une variable **\$errors** qui contient un tableau avec comme clés les noms des contrôles et comme valeurs les textes identifiant les erreurs.

*La variable **\$errors** est générée systématiquement pour toutes les vues.*

Laravel met aussi à notre disposition la directive Blade **@error("nom_champ")** pour vérifier s'il existe une erreur de validation pour le champ **nom_champ** :

```
@error('name')
    <div class="alert alert-danger">{{ $message }}</div>
@enderror
```

Dans la directive **@error**, nous avons accès au message d'erreur à travers la variable **\$message**.

Enfin en cas d'erreur de validation les anciennes valeurs saisies sont retournées au formulaire et récupérées avec l'helper **old** :

```
value="{{ old('nom') }}"
```

5. Personnalisation des messages d'erreur

Si nécessaire, vous pouvez fournir des messages d'erreur personnalisés qu'une instance de validateur doit utiliser à la place des messages d'erreur par défaut fournis par Laravel. Il existe plusieurs façons de spécifier des messages personnalisés. Premièrement, vous pouvez passer les messages personnalisés comme troisième argument à la méthode **Validator::make** :

```
$validator = Validator::make($input, $rules, $messages = [
    'email.required' => 'We need to know your email address!',
```

```
]);
```

6. Afficher une erreur Laravel au formulaire Bootstrap

Pour afficher les erreurs d'un formulaire avec Bootstrap, nous avons :

1. La classe CSS **.is-invalid** qu'il faut ajouter à la classe CSS `.form-control` d'un champ `<input>`, `<textarea>`, ... pour y indiquer la présence d'erreur
2. La classe CSS **.invalid-feedback** qu'il faut ajouter sur un autre élément HTML (`<div>`, ``, ...) pour afficher le message d'erreur

Ainsi, nous pouvons vérifier la présence d'erreurs sur un input en utilisant la directive `@error` ou les méthodes de l'objet `$errors` pour ajouter/retirer la classe CSS **.is-invalid** qui va déclencher l'affichage/masquage de l'élément de classe CSS **.invalid-feedback**.

Le code initial de la vue `resources/views/contact.blade.php` devient :

```
<input type="email" class="form-control @error('email') is-invalid @enderror" id="email" name="email" >

    <!-- Le message d'erreur -->
    @error('email')
        <div class="invalid-feedback">{{ $message }}</div>
    @enderror
```

7. Liste des règles de validation disponibles

Voici une liste de toutes les règles de validation disponibles dans Laravel que vous pouvez utiliser.

Règle	Exemple	Description
accepted	'field' => 'accepted'	Useful for checkbox validation it must be yes, on, 1 or true

active_url	'field' => 'active_url'	The field under validation must have a valid A or AAAA record according to the dns_get_record PHP function
after:date	'field' => 'date after:tomorrow'	The field under validation must be a value after a given date
after_or_equal:date	'field' => 'date after_or_equal:other_field'	The field under validation must be a value after or equal to the given date.
alpha	'field' => 'required alpha'	The field under validation must be entirely alphabetic characters.
alpha_dash	'field' => 'required alpha_dash'	The field under validation may have alphanumeric characters, as well as dashes and underscores.
alpha_num	'field' => 'required alpha_num'	The field under validation must be entirely alphanumeric characters.
array	'field' => 'array'	The field under validation must be a PHP array.
bail	'field' => 'bail required'	Stop running validation rules after the first validation failure.
before:date	'field' => 'date before:tomorrow'	The field under validation must be a value preceding the given date.
before_or_equal:date	'field' => 'date before_or_equal:other_field'	The field under validation must be a value preceding or equal to the given date.
between:min,max	'field' => 'required between:1,10'	The field under validation must have a size between the given <i>min</i> and <i>max</i> .
boolean	'field' => 'boolean'	The field under validation must be able to be cast as a boolean. Allowed: true, false, 1, 0, "1", and "0".
confirmed	'field' => 'confirmed'	If the field under validation is password, a matching password_confirmation field must be present.
date	'field' => 'date'	The field under validation must be a valid, non-relative date according to the strtotime PHP function.
date_equals:date	'field' => 'date date_equals:compare_field'	The field under validation must be equal to the given date.
date_format:format	'field' => 'date date_format:YYYY:mm:dd'	The field under validation must match the given <i>format</i> .
different:field	'field' => 'different:compare_field'	The field under validation must have a different value than <i>field</i> .
digits:value	'field' => 'digits:10'	The field under validation must be <i>numeric</i> and must have an exact length of <i>value</i> .
digits_between:min,max	'field' => 'digits_between:0,10'	The field under validation must have a length between the given <i>min</i> and <i>max</i> .
dimensions	'avatar' => 'dimensions:min_width=100,min_height=200'	The file under validation must be an image meeting the dimension constraints as specified by the rule's parameters: Available constraints are: <i>min_width</i> , <i>max_width</i> , <i>min_height</i> , <i>max_height</i> , <i>width</i> , <i>height</i> , <i>ratio</i> .
distinct	'foo.*.id' => 'distinct'	When working with arrays, the field under validation must not have any duplicate values.
email	'field' => 'email'	The field under validation must be formatted as an e-mail address.
ends_with:word1,word2	'field' => 'ends_with:foo,bar'	The field under validation must end with one of the given values.
exists:table,column	'country' => 'exists:countries'	The field under validation must exist on a given database table.

file	'field' => 'file'	The field under validation must be a successfully uploaded file.
filled	'field' => 'filled'	The field under validation must not be empty when it is present.
gt:field	'field' => 'gt:100'	The field under validation must be greater than the given <i>field</i> .
gte:field	'field' => 'gte:100'	The field under validation must be greater than or equal to the given <i>field</i> .
image	'field' => 'image'	The file under validation must be an image (jpeg, png, bmp, gif, svg, or webp)
in:foo,bar	'field' => 'in:foo,bar'	The field under validation must be included in the given list of values.
integer	'field' => 'integer'	The field under validation must be an integer.
ip	'field' => 'ip'	The field under validation must be an IP address.
json	'field' => 'json'	The field under validation must be a valid JSON string.
lt:field	'field' => 'lt:100'	The field under validation must be less than the given <i>field</i> .
lte:field	'field' => 'lte:100'	The field under validation must be less than or equal to the given <i>field</i> .
max:value	'field' => 'max:100'	The field under validation must be less than or equal to a maximum <i>value</i> .
mimetypes:text/plain	'photo' => 'mimes:jpeg,bmp,png'	The file under validation must match one of the given MIME types
min:value	'field' => 'min:100'	The field under validation must have a minimum <i>value</i> .
not_in:foo,bar	'field' => 'not_in:foo,bar'	The field under validation must not be included in the given list of values.
not_regex:pattern	'email' => 'not_regex:/^.+\$/i'	The field under validation must not match the given regular expression.
nullable	'field' => 'nullable'	The field under validation may be null.
numeric	'field' => 'numeric'	The field under validation must be numeric.
present	'field' => 'present'	The field under validation must be present in the input data but can be empty.
regex:pattern	'email' => 'regex:/^.+\$/i'	The field under validation must match the given regular expression.
required	'field' => 'required'	The field under validation must be present in the input data and not empty.
string	'field' => 'string'	The field under validation must be a string.
unique:table,column,except,id	'email' => 'unique:users,email_address'	The field under validation must not exist within the given database table.
url	'field' => 'url'	The field under validation must be a valid URL.
uuid	'field' => 'uuid'	The field under validation must be a valid RFC 4122 (version 1, 3, 4, or 5) UUID.

Gérer les exceptions dans Laravel 9:

1. Introduction :

Lorsque vous démarrez un nouveau projet Laravel, la gestion des erreurs et des exceptions est déjà configurée pour vous. La classe **App\Exceptions\Handler** est l'endroit où toutes les exceptions lancées par votre application sont enregistrées et ensuite présentées à l'utilisateur.

2. Configuration

L'option de débogage de votre fichier de configuration **config/app.php** détermine la quantité d'informations relatives à une erreur qui sont effectivement affichées à l'utilisateur. Par défaut, cette option est définie pour respecter la valeur de la variable d'environnement **APP_DEBUG**, qui est stockée dans votre fichier **.env**.

Pendant le développement local, vous devez définir la variable d'environnement **APP_DEBUG** sur **true**. Dans votre environnement de production, cette valeur doit toujours être **false**. Si la valeur est définie sur **true** en production, vous risquez d'exposer des valeurs de configuration sensibles aux utilisateurs finaux de votre application.

3. Le gestionnaire d'exceptions

Il vous est possible de créer vos propres exceptions dans Laravel grâce à la commande suivante :

```
php artisan make :exception StudentNotFoundException
```

3.1 Exceptions à signaler et à rendre

Vous pouvez définir des méthodes **report** et **render** directement sur vos exceptions personnalisées. Lorsque ces méthodes existent, elles sont automatiquement appelées par le framework :

```
<?php

namespace App\Exceptions;

use Exception;

class StudentNotFoundException extends Exception
{
    public function report(){
        return false;
    }

    public function render(){
        return view('errors.studentnotfound');

    }

    public function context()
    {
        return ['order_id' =>45];
    }
}
```

Cette class hérite d'une autre Class Exception. On peut également remarquer qu'elle contient 2 méthodes :

report() : permet de faire des logs de l'exception ainsi que de les envoyer vers des services extérieurs (par exemple par mail).

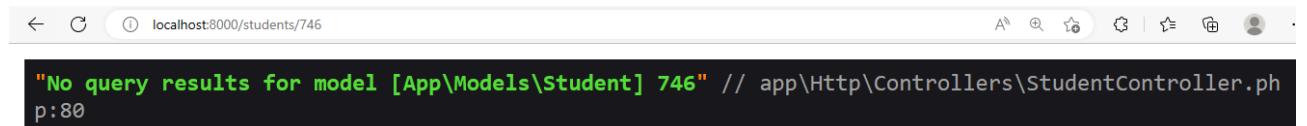
render() : permet de faire un rendu de l'exception en réponse HTTP.

Dans notre code nous allons maintenant faire appel à cette Exception :

```
public function show(Request $request,$student)
{
    try {
        $student=Student::findOrFail($student);
    } catch (ModelNotFoundException $e) {
        // dd($e->getMessage());

        throw new studentNotFoundException();
    }
    return view('students.profile', ['student' => $student]);
}
```

//Output



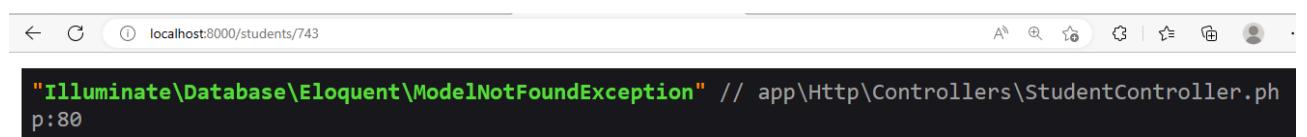
"No query results for model [App\Models\Student] 746" // app\Http\Controllers\StudentController.php:80

Après avoir mis en place le bloc **try** et **catch** il nous faut maintenant lancer l'exception avec **throw** qui nous permettra de créer une instance d'Exception. On y passe en paramètre le message que l'on souhaite retourner.

Exemple : Afficher la classe de l'exception déclenchée

```
public function show(Request $request,$student)
{
    try {
        $student=Student::findOrFail($student);
    } catch (ModelNotFoundException $e) {
        dd(get_class($e));
    }

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



"Illuminate\Database\Eloquent\ModelNotFoundException" // app\Http\Controllers\StudentController.php:80

3.2 Contexte global de journalisation

S'il est disponible, Laravel ajoute automatiquement l'ID de l'utilisateur actuel au message de journal de chaque exception en tant que donnée contextuelle. Vous pouvez définir vos propres données contextuelles globales en surchargeant la méthode contextuelle de la classe **App\Exceptions\Handler** de votre application. Ces informations seront incluses dans chaque message de journal d'exception écrit par votre application :

[App\Exceptions\Handler.php](#)

```
protected function context()
{
    return ['foo' => 'bar'];
}
```

3.3 Contexte du journal des exceptions

Si l'ajout d'un contexte à chaque message de journal peut être utile, il arrive qu'une exception particulière présente un contexte unique que vous souhaitez inclure dans vos journaux. En définissant une méthode de contexte sur l'une des exceptions personnalisées de votre application, vous pouvez spécifier toute donnée pertinente pour cette exception qui doit être ajoutée à l'entrée du journal de l'exception :

[App\Exceptions\StudentNotFoundException.php](#)

```
<?php

namespace App\Exceptions;

use Exception;

class StudentNotFoundException extends Exception
{

    public function context()
    {
```

```

        return ['order_id' => 45];
    }
}

```

Voici le journal des exceptions (`\storage\logs\laravel.log`)

```

"'}
[2023-02-11 09:05:00] local.ERROR: {"order_id":45,"foo":"bar","exception":"[object] (App\Exceptions\studentNotFoundException(code: 0): at [stacktrace]
#0 C:\xampp\htdocs\tp1\vendor\laravel\framework\src\Illuminate\Routing\Controller.php(54): App\Http\Controllers\StudentController->index()
#1 C:\xampp\htdocs\tp1\vendor\laravel\framework\src\Illuminate\Routing\ControllerDispatcher.php(43): Illuminate\Routing\ControllerDispatcher->dispatch()
#2 C:\xampp\htdocs\tp1\vendor\laravel\framework\src\Illuminate\Routing\Route.php(260): Illuminate\Routing\ControllerDispatcher->dispatch()
#3 C:\xampp\htdocs\tp1\vendor\laravel\framework\src\Illuminate\Routing\Route.php(205): Illuminate\Routing\Route->runController()
#4 C:\xampp\htdocs\tp1\vendor\laravel\framework\src\Illuminate\Routing\Router.php(798): Illuminate\Routing\Route->run()
#5 C:\xampp\htdocs\tp1\vendor\laravel\framework\src\Illuminate\Pipeline\Pipeline.php(141): Illuminate\Routing\Router->Illuminate\Pipeline\{closure}()
#6 C:\xampp\htdocs\tp1\vendor\laravel\framework\src\Illuminate\Routing\Middleware\SubstituteBindings.php(50): Illuminate\Pipeline\{closure}()
#7 C:\xampp\htdocs\tp1\vendor\laravel\framework\src\Illuminate\Pipeline\Pipeline.php(180): Illuminate\Routing\Middleware\SubstituteBindings->handle()
#8 C:\xampp\htdocs\tp1\vendor\laravel\framework\src\Illuminate\Foundation\Http\Middleware\VerifyCsrfToken.php(78): Illuminate\Pipeline\{closure}()
#9 C:\xampp\htdocs\tp1\vendor\laravel\framework\src\Illuminate\Pipeline\Pipeline.php(180): Illuminate\Foundation\Http\Middleware\VerifyCsrfToken->handle()
#10 C:\xampp\htdocs\tp1\vendor\laravel\framework\src\Illuminate\View\Middleware\ShareErrorsFromSession.php(49): Illuminate\Pipeline\{closure}()
#11 C:\xampp\htdocs\tp1\vendor\laravel\framework\src\Illuminate\Pipeline\Pipeline.php(180): Illuminate\View\Middleware\ShareErrorsFromSession->handle()
#12 C:\xampp\htdocs\tp1\vendor\laravel\framework\src\Illuminate\Session\Middleware\StartSession.php(121): Illuminate\Pipeline\{closure}()

```

4. Pages d'erreur HTTP personnalisées :

Vous voulez personnaliser la page d'erreur pour le code d'état HTTP 404, alors créez un fichier `404.blade.php` dans le répertoire **resources/views/errors**. Laravel exécutera cette page sur toutes les erreurs 404 générées par votre application. Vous pouvez également créer une vue nommée le code d'état HTTP. Il passera la fonction d'interruption à la vue comme une variable d'exception.

```

{{ $exception->getMessage() }}

```


Journalisation (logging) dans Laravel:

1. Introduction :

Le journal de Laravel 9 fait partie intégrante de l'application Laravel. Fondamentalement, il est utilisé comme une application pour surveiller l'activité de l'application. Laravel dispose de services de log robustes pour convertir les messages de log dans un fichier. Aujourd'hui, nous allons faire la démonstration de la journalisation de Laravel. Ce cours vous aidera à comprendre l'état de votre application. Ce qui se passe avec votre application. S'il y avait une erreur dans votre logiciel, vous verriez le message d'erreur du système dans votre fichier journal.

Comment cela fonctionne-t-il ?

Laravel 9 permet aux développeurs d'enregistrer des messages sur le comportement du système de n'importe quelle application dans des fichiers, comme le système de journaux d'erreurs, et de les envoyer pour informer les équipes de développement.

Il est basé sur les canaux de Laravel 9. Les canaux sont une façon spécifique d'écrire le message de journal d'un système. Chaque canal représente une destination différente, et nous pouvons envoyer des messages à différents canaux simultanément.

2. Écriture de messages de journal

Vous pouvez écrire des informations dans le journal à l'aide de la façade **Log**. Comme mentionné précédemment, le journal fournit les huit niveaux de

journalisation définis dans la spécification RFC 5424 : emergency, alert, critical, error, warning, notice, info et debug :

```
use Illuminate\Support\Facades\Log;

Log::emergency($message);
Log::alert($message);
Log::critical($message);
Log::error($message);
Log::warning($message);
Log::notice($message);
Log::info($message);
Log::debug($message);
```

Vous pouvez appeler n'importe laquelle de ces méthodes pour enregistrer un message pour le niveau correspondant. Par défaut, le message sera écrit dans le canal de journalisation par défaut tel que configuré par votre fichier de configuration de journalisation :

```
public function show(Request $request,$student)
{
    try {
        $student=Student::findOrFail($student);
        Log::info('User see his profile :'.$student->id);

    } catch (ModelNotFoundException $e) {

        throw new studentNotFoundException(100);
    }catch (\Exception $e) {

        dd(get_class($e));
    }
    return view('students.profile', ['student' => $student]);
}
```

//Output (laravel.log)

```
[2023-02-11 20:53:44] local.INFO: User see his profile :74
```

3. Informations contextuelles

Un tableau de données contextuelles peut être transmis aux méthodes de journalisation. Ces données contextuelles seront formatées et affichées avec le message du journal :

```
use Illuminate\Support\Facades\Log;  
  
Log::info('User see his profile.', ['id' => $student->id, 'ip' => $request->ip()]);
```

\storage\logs\laravel.log

```
storage > logs > laravel.log  
1 [2023-03-20 14:57:11] local.INFO: User see his profile :adil1  
2 [2023-03-20 14:58:38] local.WARNING: User see his profile :adil1  
3 [2023-03-20 14:59:38] local.WARNING: User see his profile :Fatima  
4 [2023-03-20 15:04:23] local.INFO: User see his profile. {"id":92,"ip":"127.0.0.1"}  
5 [2023-03-20 15:21:40] local.CRITICAL: User see his profile. {"id":74,"ip":"127.0.0.1"}  
6
```

Exercice :

Le Log Viewer d'OPcodes est un compagnon parfait pour votre application Laravel. Vous n'aurez plus besoin de lire les fichiers logs bruts de Laravel en essayant de trouver ce que vous cherchez.

Log Viewer vous aide à voir rapidement et clairement les entrées de log individuelles, à rechercher, filtrer et donner un sens à vos logs Laravel rapidement. Il est gratuit et facile à installer.

Installation :

Installer le package via composer:

```
composer require opcodesio/log-viewer
```

Une fois l'installation terminée, vous pourrez accéder à Log Viewer directement dans votre navigateur.

Par défaut, l'application est disponible à l'adresse suivante :
{APP_URL}/log-viewer.

Better Log Viewer [?](#)

Back to Laravel

Debug: 333,872 | Info: 91,756 | Warning: 60,888 | Error: 672 | Search... RegEx welcome!

Level	Time	Env	Description	Count									
Error	2022-07-09 23:57:02	production	SQLSTATE[HY093]: Invalid parameter number (SQL: ANALYZE TABLE 'Action toplist')	487,156									
Debug	2022-07-09 23:48:48	production	Handling facebook login.	487,155									
Error	2022-07-09 23:47:48	production	Maximum execution time of 100 seconds exceeded	487,154									
Debug	2022-07-09 23:47:46	production	Initiating facebook login.	487,153									
Debug	2022-07-09 23:46:11	production	[test@example.com] credentials_not_correct	487,152									
[test@example.com] credentials_not_correct				Copy link to this log entry									
HTTP request													
User ID	guest												
Request	POST http://127.0.0.1/login												
Agent	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/103.0.0.0 Safari/537.36												
Warning	2022-07-09 23:46:03	production	Exception: User not found	487,151									
Warning	2022-07-09 23:41:58	production	Exception: User not found	487,150									
Warning	2022-07-09 23:41:33	production	Exception: User not found	487,149									
Debug	2022-07-09 23:37:54	production	Two-Factor Auth request.	487,148									
Warning	2022-07-09 23:37:06	production	Exception: User not found	487,147									
Warning	2022-07-09 23:36:57	production	Exception: User not found	487,146									
Debug	2022-07-09 23:32:46	production	Two-Factor Auth request.	487,145									
Debug	2022-07-09 23:32:43	production	Two-Factor Auth request.	487,144									
Debug	2022-07-09 23:31:22	production	[UID 50314] - Got 0 clients after filters have been applied	487,143									
Debug	2022-07-09 23:31:22	production	[UID 50314] - Got 0 clients who should be deleted today	487,142									
Debug	2022-07-09 23:31:22	production	[UID 50314] Starting deletion of marked clients.	487,141									
Debug	2022-07-09 23:31:22	production	[UID 50038] - Got 0 clients after filters have been applied	487,140									
Debug	2022-07-09 23:31:22	production	[UID 50038] - Got 0 clients who should be deleted today	487,139									
Debug	2022-07-09 23:31:22	production	[UID 50038] - Got 0 clients after filters have been applied	487,138									
Log	2022-07-09 23:31:22	production											
1	2	3	4	5	6	7	8	9	10	...	7,908	7,909	→
Memory: 105.52 MB, Duration: 209ms													

Laravel : Quelques commandes artisan:

1. Introduction

Laravel Artisan est une Interface en Ligne de Commande (CLI) qui va vous permettre de gérer votre application en lançant des commandes via le terminal. Cette commande vous permettra d'effacer le cache de l'application, gérer des modèles, des contrôleurs, des routes... Ces commandes vont permettre de vous faire gagner du temps lors du développement de votre application Web.

Pour afficher une liste de toutes les commandes Artisan disponibles, vous pouvez utiliser la commande **list** :

```
php artisan list
```

Toutes les routes définies :

```
php artisan route:list
```

a. Laravel Tinker (Repl)

Laravel Tinker vous permet d'interagir avec une base de données sans créer les routes. Laravel **tinker** est utilisé avec un artisan php pour créer les objets ou modifier les données. Le php artisan est une interface de ligne de commande qui est disponible avec un Laravel. Un tinker joue autour de la base de données, c'est-à-dire qu'il vous permet de créer les objets, d'insérer les données, etc.

Installation

Toutes les applications Laravel incluent Tinker par défaut. Cependant, vous pouvez installer Tinker en utilisant Composer si vous l'avez préalablement supprimé de votre application :

```
composer require laravel/tinker
```

Pour entrer dans l'environnement Tinker, exécutez la commande donnée ci-dessous :

```
php artisan tinker
```

```
PS C:\xampp\htdocs\tp1> php artisan tinker
Psy Shell v0.11.9 (PHP 8.1.12 - cli) by Justin Hileman
> Student::find(4)
[!] Aliasing 'Student' to 'App\Models\Student' for this Tinker session.
= null

> Student::find(74)
= App\Models\Student {#4736
    id: 74,
    name: "Sara123",
    picture: "students/38Es1HLrkzCtox9oYRKYwtiab1DPLiBSE9voypf4.png",
    section: "info",
    mail: "sara@gmail.com",
    created_at: "2023-01-31 21:29:27",
    updated_at: "2023-02-01 09:01:38",
}
```

Nous pouvons créer les enregistrements dans les tables de la base de données en utilisant l'outil de ligne de commande. Nous utilisons l'instruction suivante dans l'outil de ligne de commande qui insère les données directement dans la table de la base de données :

```
Post::create(["title"=>"new post", "content"=>"comment for this post"])
```

```
PS C:\xampp\htdocs\tp1> php artisan tinker
Psy Shell v0.11.9 (PHP 8.1.12 - cli) by Justin Hileman
> Post::create(["title"=>"new post", "content"=>"comment for this post"])
[!] Aliasing 'Post' to 'App\Models\Post' for this Tinker session.
= App\Models\Post {#4730
    title: "new post",
    content: "comment for this post",
    updated_at: "2023-02-12 17:44:07",
    created_at: "2023-02-12 17:44:07",
    id: 3,
}
```

Nous pouvons récupérer les enregistrements de la base de données de trois façons :

La première façon est d'utiliser la méthode **find()**.

```
Student::find(74)
```

```
Post::where('id',1)->first() //Récupérer un seul enregistrement
```

```
Post::where('id', '>', 1)->get()
```

Dans l'écran ci-dessus, nous récupérons les enregistrements dont l'identifiant est supérieur à 1. Dans ce cas, plus d'un enregistrement est récupéré, nous utilisons donc la méthode **get()**. Comme la méthode **get()** est utilisée lorsqu'un tableau d'enregistrements est récupéré.

2. Comment créer une commande artisan personnalisée dans Laravel

Laravel est un framework complet qui offre de nombreuses commandes **artisan** pour automatiser diverses actions, comme la création d'un contrôleur, la population de la base de données et le démarrage du serveur. Cependant, lorsque vous construisez des solutions personnalisées, vous avez vos propres besoins spécifiques, qui peuvent inclure une nouvelle commande. Laravel ne vous limite pas à ses seules commandes ; vous pouvez créer les vôtres en quelques étapes.

Voici les étapes à suivre pour créer une nouvelle commande artisanale.

Étape 1 : Créer une commande

Utilisez la commande **make:command** pour créer une nouvelle commande. Il suffit d'entrer le nom de la commande, comme ceci :

```
php artisan make:command CreatePostCommand
```

Dans cet exemple, nous allons créer une commande appelée **CreatePostCommand**.

La commande crée un fichier **CreatePostCommand.php**, nommé d'après le nom de la commande, dans un répertoire **Commands** nouvellement créé dans le dossier Console.

Le fichier généré contient les configurations de la commande nouvellement créée qui sont faciles à comprendre et à modifier.

Étape 2 : Personnaliser la commande

Tout d'abord, définissez la signature de la commande. C'est ce qui sera mis après `php artisan` pour exécuter la commande. Dans cet exemple, nous utiliserons `check:posts`, donc la commande sera accessible en exécutant :

```
php artisan create:post
```

Pour ce faire, mettez à jour la propriété **`$signature`** de la commande, comme ceci :

```
protected $signature = 'create:post' ;
```

Ensuite, configurez une description appropriée qui s'afficherait lorsque la liste `php artisan` affiche la commande avec d'autres commandes.

Pour ce faire, mettez à jour la propriété **`$description`** pour qu'elle corresponde à ceci :

```
protected $description = 'Creates new Post' ;
```

Enfin, dans la méthode **`handle()`**, effectuez l'action que vous souhaitez qu'elle effectue. Dans cet exemple, le nombre de post sur la plateforme est renvoyé.

```
public function handle()
{
    // add a random post to database
}
```

Étape 3 : Test de la commande

Dans le terminal, exécutez la commande pour intégrer un nouvel enregistrement dans votre base de données.

```
php artisan create:post
```

Passage d'arguments à la commande

Vous pouvez avoir une commande qui nécessite un argument pour la fonction. Par exemple, une commande pour effacer tous les commentaires d'un post spécifique de la base de données nécessiterait l'identifiant du post.

Pour ajouter un argument, mettez à jour la chaîne `$signature` et ajoutez l'argument entre accolades.

```
protected $signature = 'remove:comments {postId}' ;
```

Cette commande serait alors appelée lorsque l'id d'un post est 5 :

```
php artisan remove:comments 5
```

À d'autres moments, vous voulez pouvoir passer un argument, mais pas toujours, alors vous pouvez rendre l'argument facultatif en ajoutant un point d'interrogation à la fin, comme suit :

```
protected $signature = 'remove:comments {postId?}' ;
```

Vous pouvez également définir une valeur par défaut pour un argument, comme suit :

```
protected $signature = 'remove:comments {postId=6}' ;
```

Vous pouvez également passer plusieurs arguments et les rendre facultatifs ou avec des valeurs par défaut comme vous le souhaitez.

```
protected $signature = 'remove:comments {postId} {$numbers_comments?}'
```

On peut accéder à ces arguments en utilisant :

```
$this->arguments()
```

Cela renvoie un tableau associatif avec les arguments comme clé et leurs valeurs comme valeurs. Ainsi, pour accéder à l'argument **postId**, vous pouvez l'obtenir comme ceci :

```
$post_id = $this->arguments()['postId'] ;
```

Cependant, il existe une autre façon d'obtenir un seul argument :

```
$post_id = $this->argument('postId') ;
```

```
<?php

namespace App\Console\Commands;

use App\Models\Post;
use Illuminate\Console\Command;

class CountPosts extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'remove:comments {postId}';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'remove comments for given posId';

    /**
     * Execute the console command.
     *
     * @return int
     */
}
```

```
public function handle()
{
    $postId=$this->argument('postId');
    $post=Post::find($postId);
    $post->comments()->delete();
}
}
```

Dans le terminal, exécutez la commande pour supprimer tous les commentaires du post ayant id=10.

```
php artisan remove:comments 10
```

Passer des options à la commande

Les options, comme les arguments, sont une autre forme de saisie de l'utilisateur. Les options sont préfixées par deux traits d'union (--) lorsqu'elles sont fournies par la ligne de commande. Il existe deux types d'options : celles qui reçoivent une valeur et celles qui n'en reçoivent pas. Les options qui ne reçoivent pas de valeur servent de "commutateur" booléen. Voyons un exemple de ce type d'option :

Par exemple, pour obtenir le nombre d'utilisateurs dont l'adresse électronique est vérifiée, vous pouvez passer une option **--verified** à la commande. Pour créer une option, passez-la dans la propriété **\$signature** comme l'argument, mais préfixez-la par **--**.

```
protected $signature = 'check:users {--verified}' ;
```

Maintenant, la commande peut être utilisée avec une option comme ceci :

```
php artisan check:users --verified
```

Dans cet exemple, l'option **--verified** peut être spécifiée lors de l'appel de la commande Artisan. Si le booléen **--verified** est passé, la valeur de l'option sera **true**. Sinon, la valeur sera **false**.

Vous pouvez définir une valeur par défaut pour une option ou la définir pour exiger une valeur.

```
protected $signature = 'check:users {--verified} {--add=} {--delete=5}' ;
```

Dans cet exemple, **add** nécessite une valeur pour être utilisé et **delete** a une valeur par défaut de 5. **verified** se voit attribuer une valeur booléenne selon qu'elle est passée ou non.

On peut accéder facilement à la valeur de ces options en utilisant **\$this->option('verified')** pour les uniques et **\$this->options()** pour obtenir toutes les options sous forme de tableau associatif.

Décrire les paramètres d'entrée

Jusqu'à présent, nous avons appris à accepter des arguments et même des options, mais lorsque la commande est utilisée avec l'aide de php artisan, ces entrées n'ont pas de description. Pour les définir, il suffit d'ajouter un deux-points, **:**, après le nom de l'argument ou de l'option.

```
Protégé $signature = '
    check:users
    {userId : Id de l'utilisateur à récupérer}
    {--verified : Obtient le nombre des utilisateurs vérifiés}
    ' ;
```

Maintenant, lorsque **php artisan --help check:users** est exécuté, vous devriez voir quelque chose comme ceci :

```
21260@DESKTOP-BKOTU4T MINGW64 /c/xampp/htdocs/tp1
$ php artisan --help check:users
Description:
  count numbers of verifies users

Usage:
  check:users [options]

Options:
  --verified
  -h, --help           Display help for the given command. when no command is given display help for the list command
  -q, --quiet          Do not output any message
  -v, --version         Display this application version
  --ansi|--no-ansi    Force (or disable --no-ansi) ANSI output
  -n, --no-interaction Do not ask any interactive question
  --env[=ENV]           The environment the command should run under
  -v|vv|vvv, --verbose Increase the verbosity of messages: 1 for normal output, 2 for more verbose output and 3 for debu
```

Exercice :

- 1- Créer une commande **CreatePostCommand** qui permet d'ajouter un nouvel post.
- 2- Personnalisez les propriétés **\$description** et **\$signature**.
- 3- Afficher la liste des commandes avec **php artisan list**.

```
config
  config:cache          Create a cache file for faster configuration loading
  config:clear          Remove the configuration cache file
create
  create:post           Creates new post
db
  db:monitor           Monitor the number of connections on the specified database
  db:seed               Seed the database with records
  db:show               Display information about the given database
```

- 4- Tester la commande.
- 5- Ajouter un post via les arguments et les options

Laravel : Broadcasting partie 1 (public channel)

1. Introduction

Il y a de plus en plus d'applications web qui utilisent le **WebSocket** permettant d'afficher les informations aux utilisateurs en temps réel. Etant l'une des frameworks PHP les plus en vogue actuellement, Laravel offre avec "Laravel Echo", un système de "**broadcasting** d'évènement" qui donne la possibilité, coté client, de facilement se souscrire à des évènements qui sont diffusés en Back-End via des **canaux** ou **channels** par le serveur. Le coté client réagit vis-à-vis de ces évènements sans que celui-ci ait besoin de rafraîchir la page. Ce TP va vous guider pas à pas à utiliser Laravel Echo et Pusher.

2. CONFIGURATION

On va avoir besoin du service de **broadcasting** de Laravel, il faut donc activer le service provider correspondant dans `/config/app.php` en décommentant cette ligne:

```
<?php  
App\Providers\BroadcastServiceProvider::class
```

Cette classe va ajouter les routes nécessaires au broadcasting d'évènements et ensuite charger les routes qui se trouvent dans le fichier `routes/channels.php`. Dans le fichier `config/broadcasting.php`, on a les configurations du broadcasting de Laravel. Laravel support par défaut les drivers suivants: **pusher, redis, log, null**

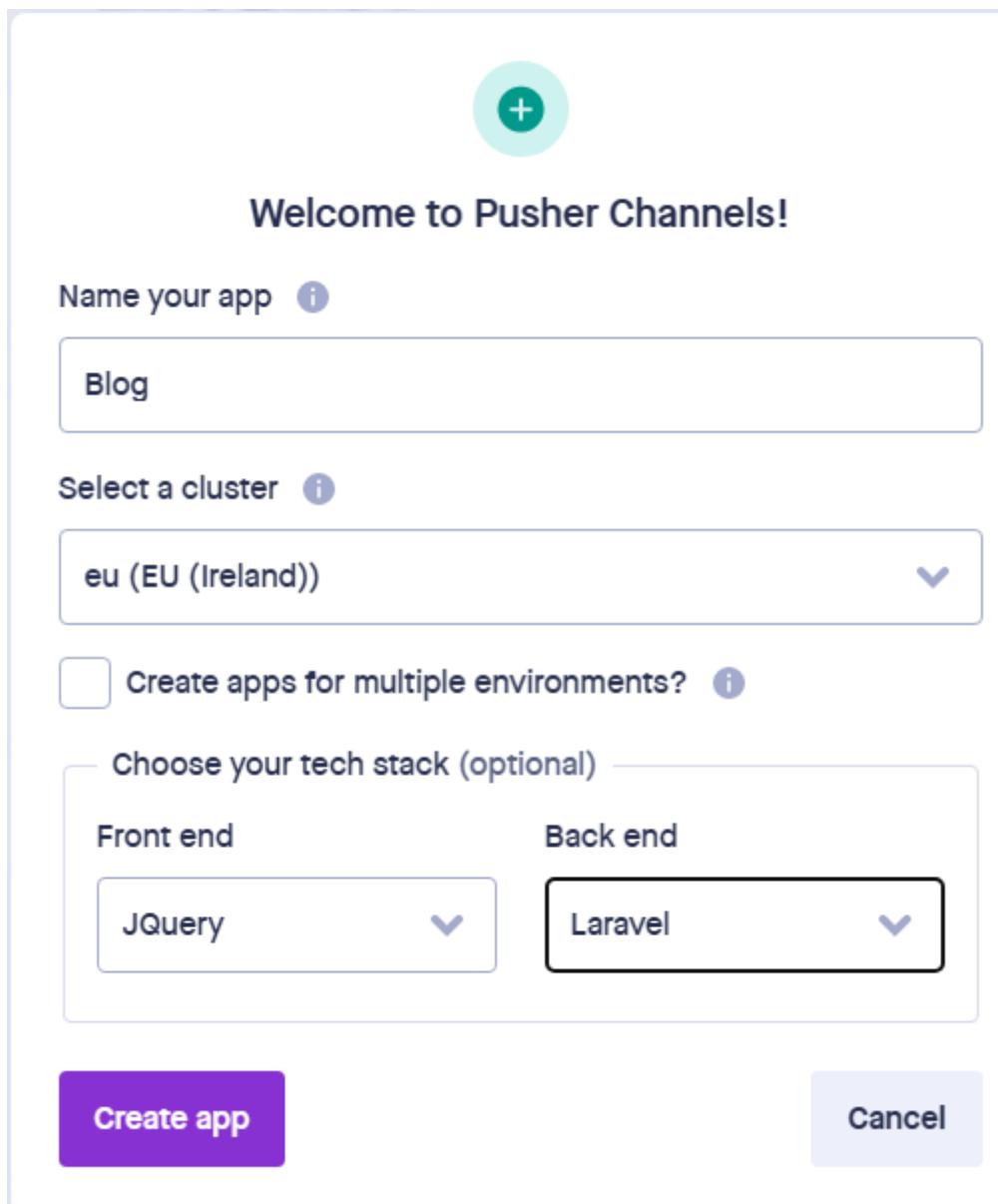
Le driver log nous permet de diffuser un évènement dans le fichier log de Laravel, utilisé surtout à des fins de test et de débogage.

Mais nous allons utiliser pusher parce que pusher propose un service gratuit et c'est surtout le plus facile à utiliser.

Dans le fichier `.env`, modifier le `BROADCAST_DRIVER=log` en `BROADCAST_DRIVER=pusher`.

Dans la configuration de connexion de pusher `config/broadcasting.php`, n'oublier pas de mettre `encrypted` à `false` pour que Pusher accepte aussi les requêtes http et pas seulement des https. (A noter que ceci est seulement pour les environnements de développement et n'est surtout pas recommandé en production).

On doit créer un compte sur dashboard.pusher.com et créer notre première application.



The image shows the Pusher Channels app creation interface. At the top is a teal circular button with a white plus sign. Below it, the text "Welcome to Pusher Channels!" is displayed. The first step is to "Name your app" with an information icon, and the user has entered "Blog". The next step is to "Select a cluster" with an information icon, and the user has chosen "eu (EU (Ireland))". There is an optional checkbox for "Create apps for multiple environments?" with an information icon, which is unchecked. Below this, there is a section for "Choose your tech stack (optional)" with two dropdown menus: "Front end" set to "JQuery" and "Back end" set to "Laravel". At the bottom are two buttons: a purple "Create app" button on the left and a grey "Cancel" button on the right.

Welcome to Pusher Channels!

Name your app i

Blog

Select a cluster i

eu (EU (Ireland))

Create apps for multiple environments? i

Choose your tech stack (optional)

Front end

JQuery

Back end

Laravel

Create app

Cancel

Après la création du compte, on arrive à l'écran de génération d'application pusher, il suffit de nommer notre app et cliquer sur "Create my app"

Ensuite, on est redirigé sur le dashboard de notre app. Sur ce dashboard on a un onglet *App Keys* qui contient la configuration de notre app.

[Overview](#)[Getting Started](#)[App Keys](#)[Stats](#)[Debug Console](#)[Error Logs](#)[Webhooks](#)[Collaborators](#)[App Settings](#)

App keys

If a token is compromised, you can create a new key/secret pair. You should delete the old token once you've updated your app.

Created 2 minutes ago

[Copy](#)

```
app_id = "1556712"  
key = "d060438970a2a9259582"  
secret = "1ec10896fec0ed46f0bf"  
cluster = "eu"
```

[Create new key and secret](#)

Dans notre projet Laravel, on doit installer les dépendances de pusher

```
composer require pusher/pusher-php-server
```

Et y renseigner les configurations suivantes (dans le .env):

```
PUSHER_APP_ID= 1556712
```

```
PUSHER_APP_KEY= d060438970a2a9259582
```

```
PUSHER_APP_SECRET= 1ec10896fec0ed46f0bf
```

```
PUSHER_APP_CLUSTER=eu
```

Pour terminer la configuration, il faut installer les diffuseurs javascript côté client, notamment **laravel-echo** et **pusher-js**.

```
npm install laravel-echo pusher-js
```

3. UTILISATION

3.1 DIFFUSION DU MESSAGE DU COTÉ SERVEUR / PUSHER

On aura besoin d'un évènement Laravel à diffuser. Créons un évènement *ProfileUpdated* qui va, soit disant, s'exécuter lorsque le profil d'un student est modifié.

```
php artisan make:event ProfileUpdated
```

Par défaut, l'évènement *ProfileUpdated* est un évènement *standard* coté serveur avec lequel on peut brancher plusieurs *listeners*.

Pour en faire un évènement diffusable coté client, il faut que celui-ci implémente l'interface "**ShouldBroadcast**" déjà importer avec:

```
<?php
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
```

La déclaration de la classe *ProfileUpdated* deviendra donc :

App\Events:

```
<?php
class ProfileUpdated implements ShouldBroadcast
```

Par défaut, on a la fonction *broadcastOn* qui retourne un *PrivateChannel* nommé "channel-name".

Un "PrivateChannel" (`Illuminate\Broadcasting\PrivateChannel`) est un channel qui ne peut être écouté que par des utilisateurs authentifiés et autorisés.

Pour commencer, nous allons utiliser le public channel. Dans la fonction *broadcastOn*, on aura:

```

<?php

public function broadcastOn()
{
    return new Channel('profile');
}

```

On va supposer que l'action *update* fait la mise à jour d'un student et exécute l'évènement *ProfileUpdated*.

```

public function update(Request $request, Student $student)
{
    //validation
    $rules=[
        'name'=>['required','max:100'],
        'mail'=>['required','max:100','email'],
        'picture'=>$request->hasFile('picture') ?
['required','max:1024','image'] : "",
        'section'=>['required','max:100'],
    ] ;
    $messages=[
        'name.required'=>'please write your name',
    ];
    $validateForm=Validator::make($request->all(),$rules,$messages);
    if($validateForm->fails()){
        return redirect()->back()->withErrors($validateForm);
    }
    //
    if($request->hasFile('picture')){
        //delete image from server
        unlink('storage/'.$student->picture);
        $student->picture=$request->picture->store('students','public');

    }
    $student=$student->update([
        'name'=>$request->name,
        'mail'=>$request->mail,
        'section'=>$request->section,
        'picture'=>$student->picture
    ]);
ProfileUpdated::dispatch($student);
    return redirect()->route('students.index')->withSuccess('student
updated successfully');
}

```

```
}
```

L'évènement *ProfileUpdated* doit donc recevoir un objet Student:

```
<?php

namespace App\Events;

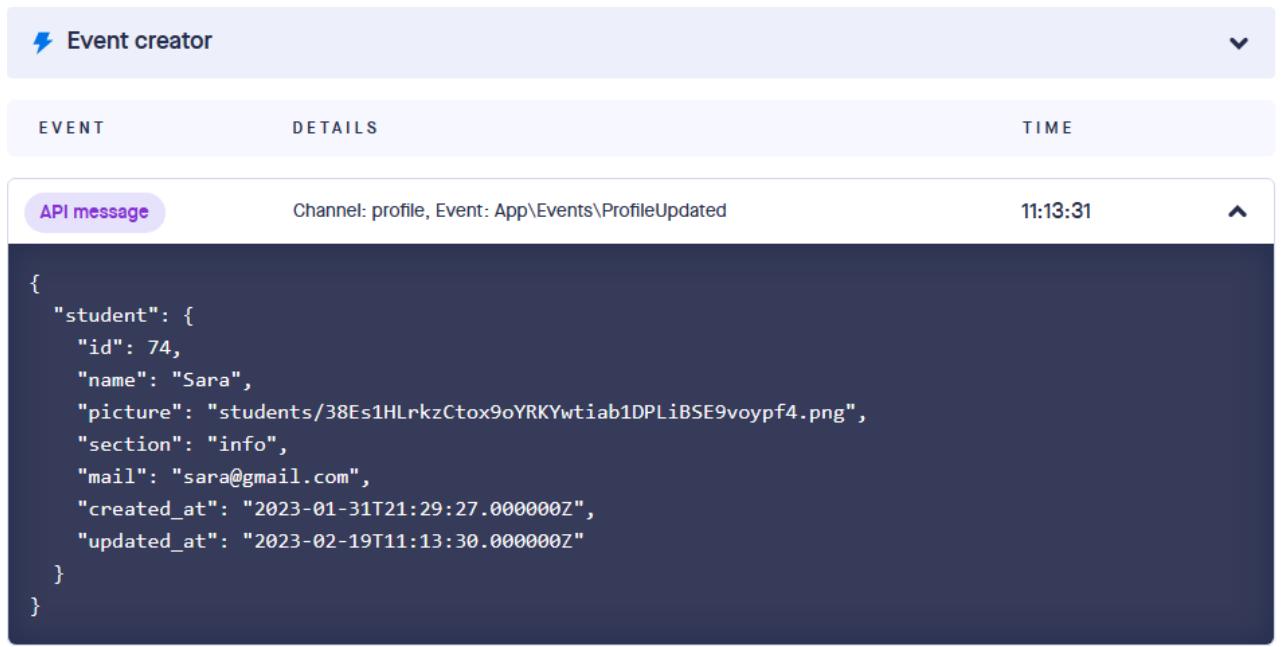
use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Queue\SerializesModels;

class ProfileUpdated implements ShouldBroadcast
{
    use Dispatchable, InteractsWithSockets, SerializesModels;
    public $student;
    public function __construct(Student $student)
    {
        $this->student=$student;
    }

    public function broadcastOn()
    {
        return new Channel('profile');
    }
}
```

Pour vérifier si tout fonctionne bien, lancer le serveur php (php artisan serve), naviguer vers la route *students.update* et vérifier sur dashboard.pusher.com si l'évènement a été capturé.

Sur dashboard.pusher.com, aller dans l'onglet "Debug console" pour voir la liste des évènements capturés par pusher. Vous devriez voir le student avec l'id 74:



The screenshot shows the Pusher Event creator interface. At the top, there is a header with a lightning bolt icon and the text 'Event creator'. Below the header, there is a table with three columns: 'EVENT', 'DETAILS', and 'TIME'. A single row is visible in the table, representing an 'API message'. The 'EVENT' column shows 'Channel: profile, Event: App\Events\ProfileUpdated'. The 'TIME' column shows '11:13:31'. The 'DETAILS' column contains a JSON object representing a student profile:

```
{  
  "student": {  
    "id": 74,  
    "name": "Sara",  
    "picture": "students/38Es1HLrkzCtox9oYRKYwtiab1DPLiBSE9voypf4.png",  
    "section": "info",  
    "mail": "sara@gmail.com",  
    "created_at": "2023-01-31T21:29:27.000000Z",  
    "updated_at": "2023-02-19T11:13:30.000000Z"  
  }  
}
```

3.2 ECOUTE DU MESSAGE DU COTÉ CLIENT

Maintenant qu'on a compris comment diffuser le message sur Pusher, nous allons voir comment écouter sur le channel donné avec Javascript, Laravel Echo, et pusher-js afin de réagir vis-à-vis du message.

Vu qu'on va modifier des fichiers javascript, il est nécessaire d'exécuter la commande ce compilation de npm.

```
npm run dev
```

Dans `resources/js/bootstrap.js`, décommenter les lignes correspondantes à Laravel-echo et comme dans `config/broadcasting.php`, mettre `encrypted` à `false`.

```
// resources/js/bootstrap.js
```

```
import Echo from 'laravel-echo';  
  
import Pusher from 'pusher-js';  
window.Pusher = Pusher;  
  
window.Echo = new Echo({  
  broadcaster: 'pusher',
```

```

key: import.meta.env.VITE_PUSHER_APP_KEY,
wsHost: import.meta.env.VITE_PUSHER_HOST ? import.meta.env.VITE_PUSHER_HOST :
`ws-${import.meta.env.VITE_PUSHER_APP_CLUSTER}.pusher.com`,
wsPort: import.meta.env.VITE_PUSHER_PORT ?? 80,
wssPort: import.meta.env.VITE_PUSHER_PORT ?? 443,
forceTLS: (import.meta.env.VITE_PUSHER_SCHEME ?? 'https') === 'https',
enabledTransports: ['ws', 'wss'],
cluster:import.meta.env.VITE_PUSHER_APP_CLUSTER, //add this line
});

```

Ensute, on va modifier notre template `welcome.blade.php` et y ajouter la directive `@vite` dans le tag head:

```
@vite('resources/js/app.js')
```

```

<body>
  <script>
    window.onload=function(){
      Echo.channel('profile')
        .listen('ProfileUpdated', e => {
          alert('profile updated :'+JSON.stringify(e.student.name));
        });
    }
  </script>
</body>

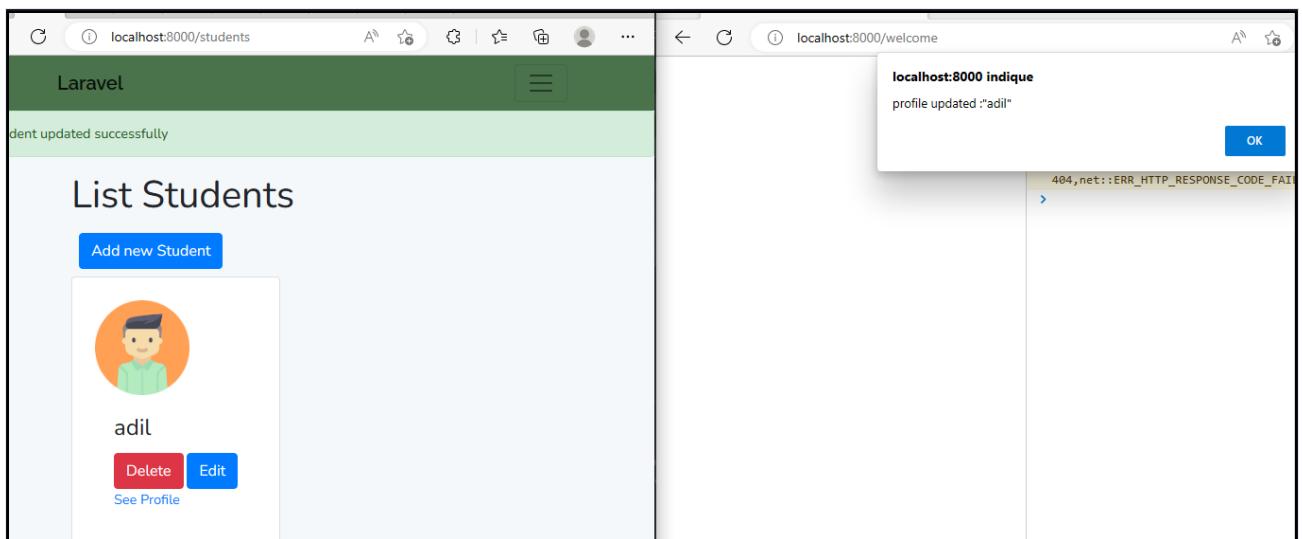
```

- la méthode **channel** correspond au channel publice “Channel”. Noter que cette méthode varie selon le type de channel utilisé (ex: `private` pour les `PrivateChannel`).
- “profile” est le nom de notre channel dans **app/Events/ProfileUpdated.php**
- “**ProfileUpdated**” est le nom de l’évènement qu’on va écouter, par convention Echo utilise le nom complet de la classe mais on n’a pas besoin de le spécifier parce que Echo va assumer que la classe donnée se trouve dans le namespace **App\Event**

- `e` est l'évènement qu'on va recevoir. A la réception du message, on va juste afficher un message dans le console ou bien alerte.

Pour voir le résultat, il nous faut 2 navigateurs différents:

- Ouvrez deux navigateurs différents (navigateur A et B) cote à cote
- Dans le navigateur A:
 - naviguer sur la page welcome.
 - vider le contenu de la console.
- Dans le navigateur B:
 - naviguer sur la route update
- Dans la console du navigateur A, vous devriez apercevoir le message comme dans la figure suivante:



Gestion des fichiers

1. Introduction :

Laravel utilise [Flysystem](#) comme composant de gestion de fichiers. Il en propose une API bien pensée et facile à utiliser. On peut ainsi manipuler fichiers et dossiers de la même manière en local ou à distance, que ce soit en FTP ou sur le cloud.

La configuration du système se trouve dans le fichier `config/filesystem.php`. Par défaut on est en local :

```
'default' => env('FILESYSTEM_DISK', 'local'),
```

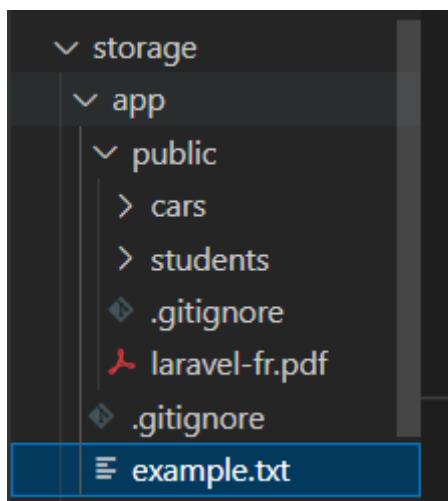
Mais on peut aussi choisir FTP, SFTP ou [s3](#).

On peut définir des « disques » (**disks**) qui sont des cibles combinant un driver, un dossier racine et différents éléments de configuration :

2. Le disque local

Lorsque vous utilisez le pilote local, toutes les opérations sur les fichiers sont relatives au répertoire racine défini dans votre fichier de configuration des systèmes de fichiers. Par défaut, cette valeur est fixée au répertoire `storage/app`. Par conséquent, la méthode suivante écrirait dans le répertoire `storage/app/example.txt` :

```
use Illuminate\Support\Facades\Storage;  
  
Storage::disk('local')->put('example.txt', 'Contents');
```



3. Le disque public

Le disque public inclus dans le fichier de configuration du système de fichiers de votre application est destiné aux fichiers qui seront accessibles au public. Par défaut, le disque public utilise le pilote local et stocke ses fichiers dans `storage/app/public`.

Pour que ces fichiers soient accessibles depuis le web, vous devez créer un lien symbolique entre `public/storage` et `storage/app/public`. L'utilisation de cette convention de dossier permet de conserver vos fichiers accessibles au public dans un seul répertoire qui peut être facilement partagé entre les déploiements lors de l'utilisation de systèmes de déploiement sans interruption.

Pour créer le lien symbolique, vous pouvez utiliser la commande Artisan :

```
php artisan storage:link
```

Une fois qu'un fichier a été stocké et que le lien symbolique a été créé, vous pouvez créer une URL vers les fichiers à l'aide de l'assistant d'actifs :

```
<div id="app">{{asset('/storage/test.txt')}}</div>
```

4. Obtention d'instances de disque

La façade `Stockage` peut être utilisée pour interagir avec n'importe lequel des disques configurés. Par exemple, vous pouvez utiliser la méthode `put` de la façade pour stocker un avatar sur le disque par défaut. Si vous appelez des méthodes de la façade `Storage` sans appeler au préalable la méthode `disk`, la méthode sera automatiquement transmise au disque par défaut :

```
Storage::disk('public')->put('avatars/1', $content);
```

5. Récupération du contenu d'un fichier

La méthode `get` peut être utilisée pour récupérer le contenu d'un fichier. La méthode renvoie la chaîne de caractères brute du fichier. N'oubliez pas que tous les chemins d'accès aux fichiers doivent être spécifiés par rapport à l'emplacement "racine" du disque :

```
if (Storage::disk('local')->exists('/public/avatars/1')) {  
    $content= Storage::get('/public/avatars/1');  
    echo $content;  
}
```

5.1 Téléchargement de fichiers

La méthode **download** peut être utilisée pour générer une réponse qui force le navigateur de l'utilisateur à télécharger le fichier au chemin donné. La méthode **download** accepte un nom de fichier comme deuxième argument de la méthode, qui déterminera le nom de fichier qui sera vu par l'utilisateur téléchargeant le fichier. Enfin, vous pouvez passer un tableau d'en-têtes HTTP comme troisième argument de la méthode :

```
return Storage::download('public/laravel-fr.pdf', $file_name);
```

6. Stockage des fichiers

La méthode **put** peut être utilisée pour stocker le contenu d'un fichier sur un disque. Vous pouvez également passer une ressource PHP à la méthode **put**, qui utilisera le support de flux sous-jacent de Flysystem. N'oubliez pas que tous les chemins d'accès aux fichiers doivent être spécifiés par rapport à l'emplacement "root" configuré pour le disque :

```
use Illuminate\Support\Facades\Storage;

Storage::put('file.jpg', $contents);
Storage::disk('public')->put('example.txt', "contents");
```

6.1 Chargements de fichiers (upload)

Dans les applications web, l'un des cas d'utilisation les plus courants pour le stockage de fichiers est le stockage de fichiers téléchargés par l'utilisateur, tels que des photos et des documents. Laravel facilite le stockage des fichiers téléchargés en utilisant la méthode **store** sur une instance de fichier téléchargé. Appelez la méthode **store** avec le chemin où vous souhaitez stocker le fichier téléchargé :

```
public function update(Request $request): string
{
    $path = $request->file('avatar')->store('avatars');
    $path = $request->file('avatar')->store('avatars', 'public');

    return $path;
}
```

Il y a quelques points importants à noter dans cet exemple. Nous n'avons spécifié qu'un nom de répertoire, et non un nom de fichier. Par défaut, la méthode **store** génère un identifiant unique qui sert de nom de fichier. L'extension du fichier sera

déterminée en examinant le type MIME du fichier. Le chemin d'accès au fichier sera renvoyé par la méthode store afin que vous puissiez stocker le chemin d'accès, y compris le nom de fichier généré, dans votre base de données.

7. Supprimer un fichier

```
use Illuminate\Support\Facades\Storage;  
Storage::delete('file.jpg');  
Storage::delete(['file1.jpg', 'file2.jpg']);
```

Si nécessaire, vous pouvez spécifier le disque à partir duquel le fichier doit être supprimé :

```
use Illuminate\Support\Facades\Storage;  
  
Storage::disk('public')->delete('path/file.jpg');
```


Compilation des assets (vite)

First React.js "Hello World" with Laravel & Vite

1. Introduction :

Vite est un outil moderne de construction d'applications frontales qui fournit un environnement de développement extrêmement rapide et qui regroupe votre code pour la production. Lorsque vous créez des applications avec Laravel, vous utilisez généralement Vite pour regrouper les fichiers CSS et JavaScript de votre application dans des ressources prêtes pour la production.

Laravel s'intègre parfaitement à Vite en fournissant un plugin officiel et une directive Blade pour charger vos ressources pour le développement et la production.

2. Installation de Node

Vous devez vous assurer que Node.js (16+) et NPM sont installés avant de lancer Vite et le plugin Laravel :

```
node -v
npm -v
```

3. Installation de Vite et du plugin Laravel

Nous installons React lui-même :

```
npm install react@latest react-dom@latest
```

De plus, pour que React fonctionne avec Vite, nous devons installer un plugin spécifique :

```
npm install --save-dev @vitejs/plugin-react
```

Vous pouvez ensuite inclure le plugin dans votre fichier de configuration **vite.config.js** :

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';
import react from '@vitejs/plugin-react';
```

```
export default defineConfig({
  plugins: [
    laravel([
      'resources/css/app.css',
      'resources/js/app.jsx',
    ]),
    react(),
  ],
});
```

Nous ajoutons import react et react() sans paramètres. Remarquez également que j'ai renommé resources/js/app.js en resources/js/app.jsx. C'est exactement le fichier dont nous allons nous occuper ensuite.

1. Afficher Hello World dans Blade

Installons un nouveau projet Laravel et ajoutons du texte statique à la page d'accueil. Mais ce texte statique proviendrait du composant React.js, et non de Blade?

Allons à **resources/views/welcome.blade.php** et ajoutons du HTML statique comme la balise <h1> :

```
<h1>Hello world from blade</h1>
```

2. Afficher le composant React.js dans Blade

Donc, à ce stade :

- Nous avons un composant React.
- Nous avons un emplacement id="app" pour le charger.
- Nous avons installé React et configuré Vite.

Enfin, nous lions le tout en créant une application React dans le fichier **resources/js/app.jsx** - en renommant l'ancien fichier par défaut de Laravel **resources/js/app.js**. Au bas de la ligne **import './bootstrap'**, nous ajoutons ceci :

```
// resources/js/app.jsx
import './bootstrap';
import '../css/bootstrap.min.css';
import '../css/app.css';

import ReactDOM from 'react-dom/client';
import HelloWorld from './components/HelloWorld';

ReactDOM.createRoot(document.getElementById('app')).render( <HelloWorld/> );
```

Nous importons le composant et le montons (render) dans la <div id="app"> du fichier Blade.

Maintenant, nous devons démarrer le serveur Vite, en exécutant :

```
npm run dev
```

```
VITE v4.1.2  ready in 21719 ms

→ Local:  http://127.0.0.1:5173/
→ Network: use --host to expose
→ press h to show help

LARAVEL v9.45.0  plugin v0.7.4

→ APP_URL: http://localhost
```

Enfin, dans **resources/views/welcome.blade.php**, ne laissez également que du JS, aucun CSS n'est nécessaire ici :

```
// resources/views/welcome.blade.php
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  @viteReactRefresh
  @vite(['resources/js/app.jsx'])

  <title>Document</title>

</head>
<body>
```

```
<div id="app"></div>
<h1>Hello world from blade</h1>
</body>
</html>
```

Vous devrez également inclure la directive supplémentaire **@viteReactRefresh** Blade dans votre directive **@vite** existante.

```
@viteReactRefresh
@vite('resources/js/app.jsx')
```

La directive **@viteReactRefresh** doit être appelée avant la directive **@vite**.

Manipulation de Mailer

Introduction :

L'envoi d'emails ne doit pas être compliqué. Laravel fournit une API d'envoi d'emails simple et propre, alimentée par le composant populaire [Symfony Mailer](#). Laravel et Symfony Mailer fournissent des pilotes pour l'envoi d'emails via SMTP, Mailgun, Postmark, Amazon SES, et [sendmail](#), vous permettant de commencer rapidement à envoyer des emails via un service local ou basé sur le cloud de votre choix.

Étape 1 – Créer un compte Mailtrap pour envoyer des emails en phase développement

Lorsque vous développez une application qui envoie des courriels, vous ne voulez probablement pas envoyer des courriels à des adresses électroniques réelles. Laravel propose plusieurs façons de "désactiver" l'envoi d'emails pendant le développement local.

Vous pouvez également utiliser un service comme [Mailtrap](#) et le pilote `smtp` pour envoyer vos messages électroniques à une boîte aux lettres "factice" où vous pourrez les consulter avec un véritable client de messagerie. Cette approche présente l'avantage de vous permettre d'inspecter les messages finaux dans le visualisateur de messages de Mailtrap.

[Integrations](#) ②

Laravel 7+ ▾

Laravel provides a clean, simple API over the popular SwiftMailer library.

With the default Laravel setup you can configure your mailing configuration by setting these values in the `.env` file in the root directory of your project.

```
MAIL_MAILER=smtp
MAIL_HOST=sandbox.smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=b84fba37705c05
MAIL_PASSWORD=21d400d37c5d23
MAIL_ENCRYPTION=tls
```

[Copy](#)

Il est ainsi facile de renseigner le fichier **.env** :

```
MAIL_MAILER=smtp
MAIL_HOST=smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=b84fba37705c05
MAIL_PASSWORD=21d400d37c5d23
MAIL_ENCRYPTION=tls
```

Étape 2 - Après avoir modifié le fichier **.env**, exécutez les deux commandes ci-dessous pour vider le cache et redémarrer le serveur Laravel.

```
php artisan config:cache
```

```
//config/mail.php
```

```
'from' => [
    'address' => env('MAIL_FROM_ADDRESS', 'aminefste@gmail.com'),
    'name' => env('MAIL_FROM_NAME', 'Mail for you'),
],
```

Étape 2 - Créer la classe Mailable

Dans cette étape, nous allons créer la classe **Mailable**, qui sera utilisée pour envoyer des courriels. La classe Mailable est responsable de l'envoi des emails en utilisant un mailer qui est configuré dans le fichier **config/mail.php**. En fait, Laravel fournit déjà une commande artisan qui nous permet de créer un modèle de base.

```
php artisan make:mail SendEmail
```

```
<?php

namespace App\Mail;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Mail\Mailable;
use Illuminate\Mail\Mailables\Content;
use Illuminate\Mail\Mailables\Envelope;
use Illuminate\Queue\SerializesModels;
```

```
class SendMail extends Mailable
{
    use Queueable, SerializesModels;

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

    /**
     * Get the message envelope.
     *
     * @return \Illuminate\Mail\Mailables\Envelope
     */
    public function envelope()
    {
        return new Envelope(
            subject: 'notification',
        );
    }

    /**
     * Get the message content definition.
     *
     * @return \Illuminate\Mail\Mailables\Content
     */
    public function content()
    {
        return new Content(
            view: 'mail.mailView',
        );
    }

    /**
     * Get the attachments for the message.
     *
     * @return array
     */
    public function attachments()
    {
        return [];
    }
}
```

```
}
```

Étape 3 -Pièces jointes

Pour ajouter des pièces jointes à un courrier électronique, vous devez les ajouter au tableau renvoyé par la méthode **attachments** du message. Tout d'abord, vous pouvez ajouter une pièce jointe en fournissant un chemin d'accès au fichier à la méthode `fromPath` fournie par la classe **Attachment** :

```
use Illuminate\Mail\Mailables\Attachment;

public function attachments()
{
    return [
        Attachment::fromStorageDisk('local','example.txt'),
    ];
}
```

Étape 3 – Créer la vue de l'email

Dans la vue **resources\views\mail\mailView.blade.php**

Copiez le code suivant :

```
<h1>Hi, {{ $name }}</h1>
<p>Sending Mail from Laravel.</p>
```

Étape 4 – Ajouter des routes

Ajoutez les lignes suivantes dans **routes/web.php**. Ici on veut tester l'email.

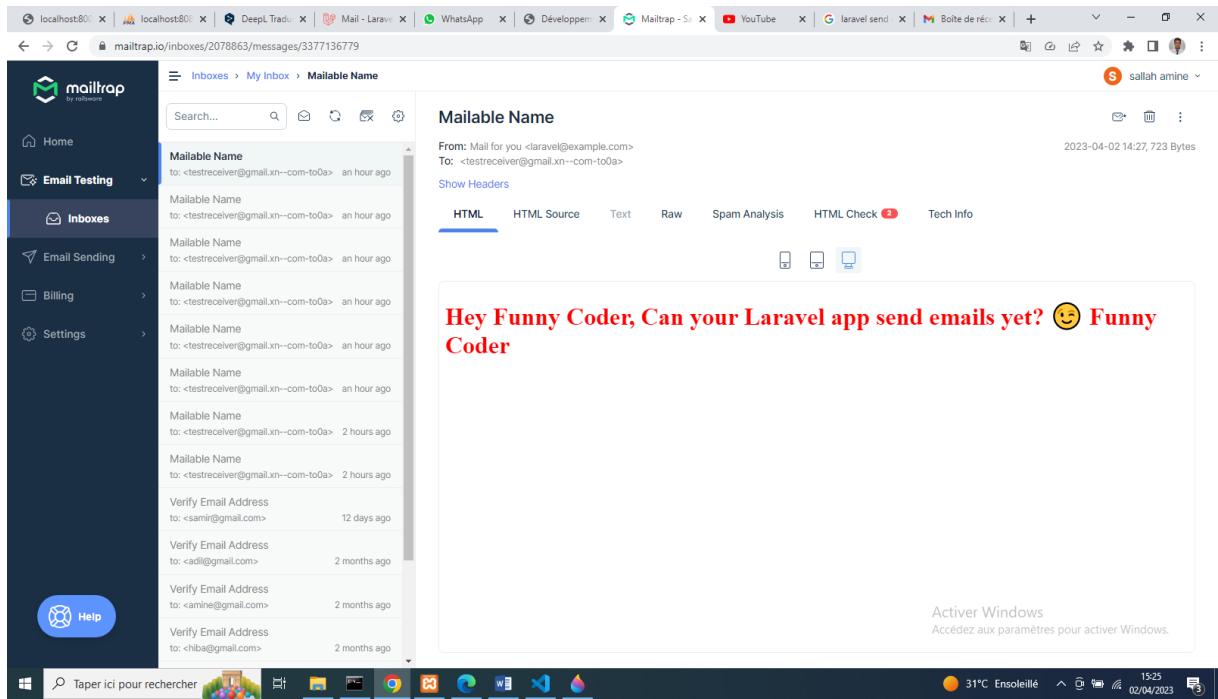
```
Route::get('/sendemail', function() {
    $name = "Funny Coder";

    //The email sending is done using the to method on the Mail facade
    Mail::to('testreceiver@gmail.com')->send(new SendEmail($name));
});
```

Étape 5 - Visitez l'URL suivante pour tester le courrier électronique de base.

```
http://localhost:8000/sendemail
```

Étape 6 - Vous pouvez voir le résultat suivant dans le serveur Mailtrap.



Exercice : Envoyer des emails dans Laravel en utilisant le serveur SMTP de Gmail

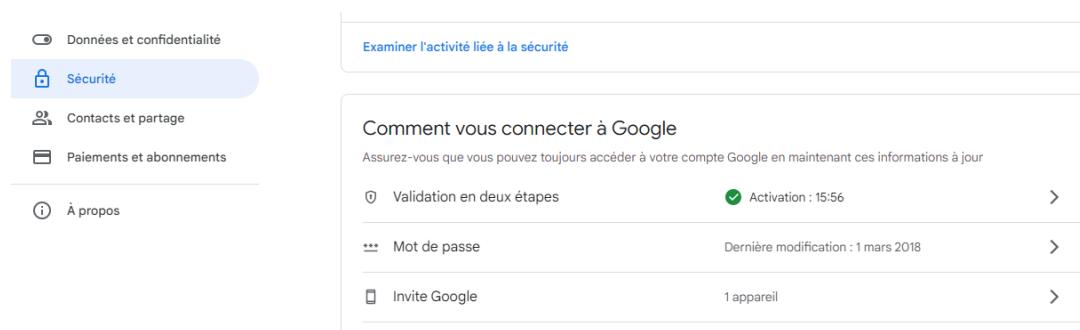
La plupart des gens préfèrent utiliser leur compte Gmail par défaut comme serveur SMTP pour leur application Laravel car il est gratuit. Dans cette partie, je vais vous montrer comment utiliser Gmail comme serveur SMTP pour envoyer vos emails dans Laravel.

Il est bon de noter que Gmail ne vous donne droit qu'à 500 emails par jour pour les comptes gratuits et 2000 emails pour les comptes payants Google Workspace, vous pouvez donc envisager de l'utiliser uniquement comme outil de test et non comme service de production.

Configuration du compte Google

Vous devez d'abord configurer certains paramètres de sécurité dans votre compte Gmail. Pour ce faire, connectez-vous à votre compte Gmail et sélectionnez "Gérer votre compte Google" dans votre profil.

Vous devez ensuite activer **la vérification en deux étapes**



The screenshot shows the 'Sécurité' (Security) section of the Google Account settings. On the left, a sidebar lists 'Données et confidentialité', 'Sécurité' (selected), 'Contacts et partage', 'Paiements et abonnements', and 'À propos'. The main content area is titled 'Comment vous connecter à Google' (How to connect to Google) with the sub-instruction 'Assurez-vous que vous pouvez toujours accéder à votre compte Google en maintenant ces informations à jour' (Make sure you can always access your Google account by keeping these info up-to-date). It lists three items: 'Validation en deux étapes' (Two-step verification) with an 'Activation : 15:56' status and a green checkmark; 'Mot de passe' (Password) with a 'Dernière modification : 1 mars 2018' (Last modified: March 1, 2018) timestamp; and 'Invite Google' (Invite Google) with a '1 appareil' (1 device) status.

Une fois la vérification en deux étapes activée, vous aurez accès à la section du mot de passe de l'application.

Cliquez sur la section **App passwords** et créez un nouveau mot de passe pour votre application Laravel Mail.

Choisissez Messagerie et autre(Nom personnalisé)

The screenshot shows a web application for generating application passwords. At the top, a table lists a single entry: 'laravel' created at '16:07' and last used at '16:09'. To the right of the last use timestamp is a small trash can icon. Below the table is a note: 'Sélectionnez l'application et l'appareil pour lesquels vous souhaitez générer le mot de passe d'application.' A dropdown menu is open, showing the following options: 'Messagerie' (selected), 'Sélectionnez un appareil', 'iPhone', 'iPad', 'BlackBerry', 'Mac', 'Windows Phone', 'Ordinateur Windows', and 'Autre (Nom personnalisé)'. To the right of the dropdown is a 'GÉNÉRER' button. In the bottom right corner of the interface, there are links: 'Activer W' and 'Accédez aux'.

Mise à jour des configurations de messagerie

Une fois que vous avez généré le mot de passe de votre application, vous pouvez mettre à jour vos configurations de messagerie dans le fichier `.env` pour utiliser les configurations SMTP de Gmail.

```
MAIL_MAILER=smtp
MAIL_HOST=smtp.gmail.com
MAIL_PORT=587
MAIL_USERNAME=youremail@gmail.com
MAIL_PASSWORD=azertyuiop
MAIL_ENCRYPTION= tls
```

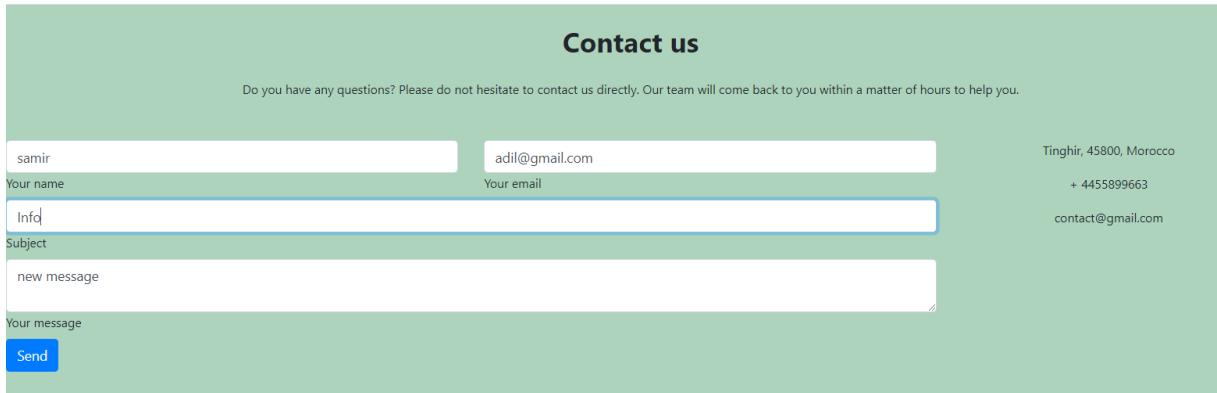
Une fois que vous avez défini les configurations, vous êtes prêt à envoyer des courriels en utilisant Gmail gratuitement. N'oubliez pas que Gmail ne permet d'envoyer que 500 courriels par jour. Si votre application doit envoyer beaucoup de courriels, vous devriez envisager d'autres services.

Nb : N'oubliez pas d'exécuter les commandes pour vider le cache :

```
php artisan config:cache  
php artisan config:clear
```

Créer un formulaire de contact

Le formulaire de contact est un système qui permet aux utilisateurs d'un site web d'envoyer des mails aux administrateurs du site et vice versa.



The screenshot shows a contact form with a green header and a light green body. The header contains the title 'Contact us'. Below the header, a message says: 'Do you have any questions? Please do not hesitate to contact us directly. Our team will come back to you within a matter of hours to help you.' The form fields are as follows:

- Your name:** samir
- Your email:** adil@gmail.com
- Subject:** Info
- Your message:** new message
- Address:** Tinghir, 45800, Morocco
- Phone:** + 4455899663
- Email:** contact@gmail.com

At the bottom left is a blue 'Send' button.

La vue **contact-form.blade.php**

Observons le code complet du formulaire HTML **resources\views\mail\contact-form.blade.php** avant d'expliquer ses éléments clés. Les classes CSS « .mb-3 », « .form-label », « .form-control », « .is-invalid », ... sont du framework CSS Bootstrap 5 :

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
    <meta charset="UTF-8">  
    <meta name="viewport" content="width=device-width, initial-scale=1.0">  
    <meta http-equiv="X-UA-Compatible" content="ie=edge">  
    <title>Document</title>  
    @vite('resources/js/app.jsx')  
</head>  
<body>  
    <!--Section: Contact v.2-->  
    <section class="mb-4 ">  
        <!--Section heading-->
```

```
<h2 class="h1-responsive font-weight-bold text-center my-4">Contact us</h2>
  <!--Section description-->
  <p class="text-center w-responsive mx-auto mb-5">Do you have any questions? Please do not hesitate to contact us directly. Our team will come back to you within a matter of hours to help you.</p>

<div class="row">

  <!--Grid column-->
  <div class="col-md-9 mb-md-0 mb-5 ">
    <form id="contact-form" name="contact-form" action="{{route('sendMail')}}" method="POST">
      @csrf
      <!--Grid row-->
      <div class="row">

        <!--Grid column-->
        <div class="col-md-6">
          <div class="md-form mb-0">
            <input type="text" id="name" name="name" class="form-control">
            <label for="name" class="">Your name</label>
          </div>
        </div>
        <!--Grid column-->

        <!--Grid column-->
        <div class="col-md-6">
          <div class="md-form mb-0">
            <input type="text" id="email" name="email" class="form-control">
            <label for="email" class="">Your email</label>
          </div>
        </div>
        <!--Grid column-->

      </div>
      <!--Grid row-->

      <!--Grid row-->
      <div class="row">
        <div class="col-md-12">
          <div class="md-form mb-0">
            <input type="text" id="subject" name="subject" class="form-control">
            <label for="subject" class="">Subject</label>
          </div>
        </div>
      </div>
    </form>
  </div>
</div>
```

```
        </div>
    </div>
<!--Grid row-->

<!--Grid row-->
<div class="row">

    <!--Grid column-->
    <div class="col-md-12">

        <div class="md-form">
            <textarea type="text" id="msg" name="msg" rows="2" class="form-control md-textarea"></textarea>
            <label for="msg">Your message</label>
        </div>

        </div>
    </div>
    <!--Grid row-->
    <button type="submit" class="btn btn-primary">Send</button>
</form>

</div>
<!--Grid column-->

<!--Grid column-->
<div class="col-md-3 text-center">
    <ul class="list-unstyled mb-0">
        <li><i class="fas fa-map-marker-alt fa-2x"></i>
            <p>Tinghir, 45800, Morocco</p>
        </li>

        <li><i class="fas fa-phone mt-4 fa-2x"></i>
            <p>+ 4455899663</p>
        </li>

        <li><i class="fas fa-envelope mt-4 fa-2x"></i>
            <p>contact@gmail.com</p>
        </li>
    </ul>
</div>
<!--Grid column-->

</div>

</section>
<!--Section: Contact v.2-->
</body>
</html>
```

La classe Mailable et la vue du mail

Pour envoyer le mail via le formulaire de contact, nous allons passer par la classe [Mailable](#) « **ContactMail** » que nous pouvons créer en exécutant la commande *artisan* suivante :

```
php artisan make:mail ContactMail
```

Cette commande crée le fichier **app\Mail>ContactMail.php**. Créons aussi la vue **resources\views\mail\contact-mail.blade.php** pour la présentation du message.

Dans la classe **ContactMail.php**, on déclare les propriétés *public* `$name`, `$email` et `$msg` qu'on passe via le constructeur, on renvoie ensuite le sujet du message `subject("...")` et la vue **contact-mail.blade.php** via la méthode `content()` et `envelope()` :

```
<?php

namespace App\Mail;

use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Mail\Mailables\Address;
use Illuminate\Mail\Mailables\Content;
use Illuminate\Queue\SerializesModels;
use Illuminate\Mail\Mailables\Envelope;
use Illuminate\Mail\Mailables\Attachment;
use Illuminate\Contracts\Queue\ShouldQueue;

class ContactMail extends Mailable
{
    use Queueable, SerializesModels;

    public $name, $email, $subject, $msg;
```

```

// On hydrate $name, $email et $msg
public function __construct($name, $email,$subject , $msg)
{
    $this->name = $name;
    $this->email = $email;
    $this->subject = $subject;
    $this->msg = $msg;
}

public function envelope()
{
    return new Envelope(
        subject: $this->subject,
    );
}

/**
 * Get the message content definition.
 *
 * @return \Illuminate\Mail\Mailables\Content
 */
public function content()
{
    return new Content(
        view: 'mail.contact-mail'
    );
}
}

```

Nous pouvons présenter les attributs \$name, \$email et \$msg dans la vue **resources\views\mail\contact-mail.blade.php** comme ceci :

```

<div>
    <p><strong>Name :</strong> : {{ $name }}</p>
    <p><strong>Email :</strong> : {{ $email }}</p>
    <p><strong>Message :</strong> :<br>{{ $msg }}</p>

```

```
</div>
```

La publication [envoyer un mail via le serveur SMTP Google](#) explique comment configurer le driver SMTP pour envoyer des mails dans une application Laravel.

Envoyer le mail

Dans le contrôleur **MailController** `App\Http\Controllers\MailController.php` :

```
<?php

namespace App\Http\Controllers;

use App\Mail\MailableName;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Mail;

class MailController extends Controller
{
    public function sendMail(Request $request){

        //The email sending is done using the to method on the Mail facade
        Mail::to('aminefste@gmail.com')->send(new ContactMail($request-
>name,$request->email, $request->subject, $request->msg));
    }

    public function contact(){
        return view('mail.contact-form');
    }
}
```

Le code ci-dessus s'explique comme suit :

Si la validation réussit, on envoie le mail à une adresse e-mail :

- l'adresse de destinataire (to),
- l'envoi avec la méthode send,
- la création d'une instance de la classe ContactMail avec la transmission des données saisies.

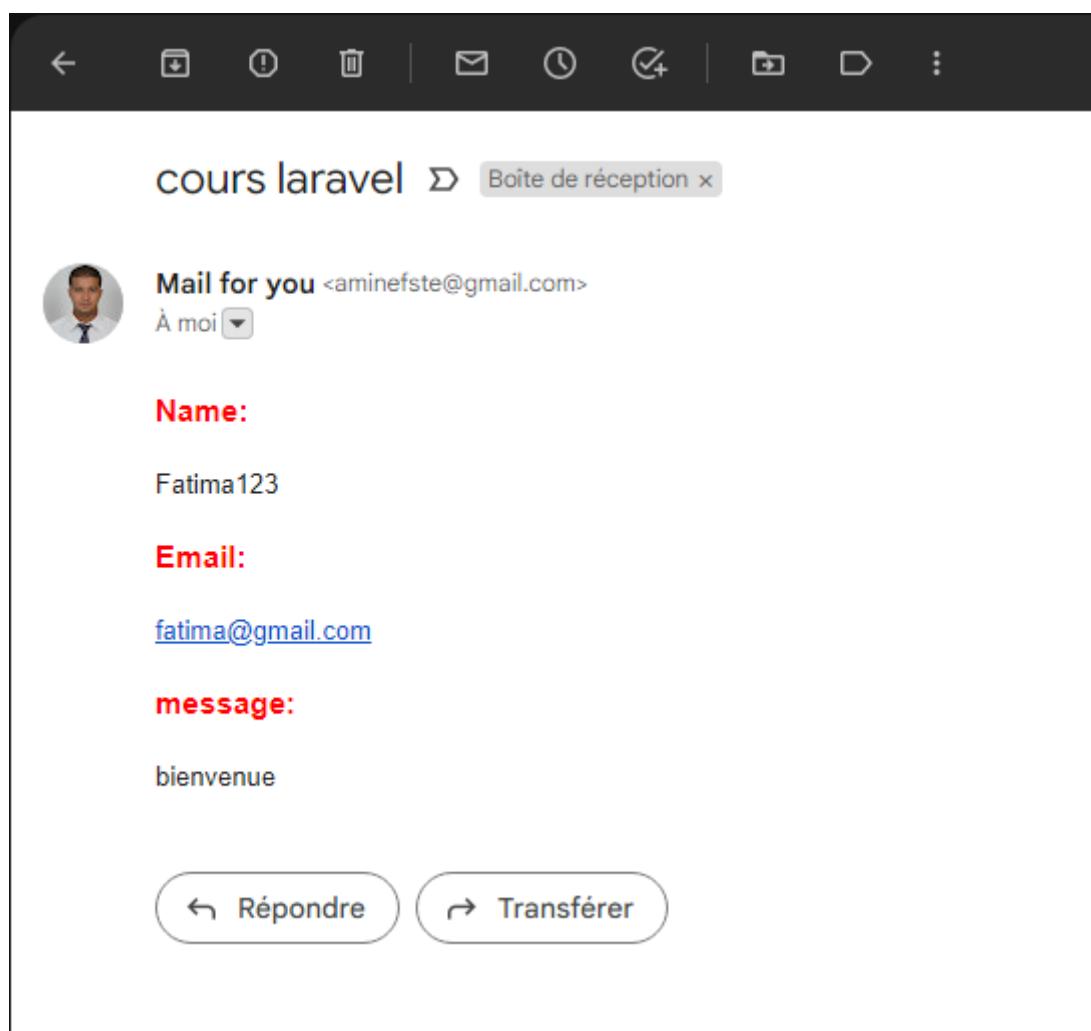
Et si tout se passe bien le message doit arriver jusqu'à le destinataire :

Test de l'email

Quand on crée la vue pour l'email il est intéressant de voir l'aspect final avant de faire un envoi réel. Pour le faire il suffit de créer une simple route :

```
Route::get('/contact',[MailController::class,'contact']);  
Route::post('/contact',[MailController::class,'sendMail'])  
->name('sendMail');
```

Maintenant en utilisant l'url **localhost:8000/contact** on obtient l'aperçu directement dans le navigateur !



Créer un site web multilingue avec Laravel Localization en utilisant un middleware

1. Introduction :

Les fonctionnalités de localisation de Laravel offrent un moyen pratique de récupérer des chaînes dans différentes langues, ce qui vous permet de prendre facilement en charge plusieurs langues dans votre application.

Etape 1 : configuration

Par défaut, la langue locale de notre application est l'anglais ou **en**.

Vous pouvez la trouver dans le fichier **config >> app.php**.

```
|  
| The application locale determines the default locale that will be used  
| by the translation service provider. You are free to set this value  
| to any of the locales which will be supported by the application.  
|  
*/  
  
'locale' => 'fr',  
  
/*  
|-----  
| Application Fallback Locale  
|-----  
  
| The fallback locale determines the locale to use when the current one  
| is not available. You may change the value to correspond to any of  
| the language folders that are provided through your application.  
|  
*/  
  
'fallback_locale' => 'en',
```

Ici, la locale est définie sur **fr**.

Maintenant, si vous vérifiez le dossier **lang**, il y a un dossier appelé **en**, dans lequel il y a quelques fichiers comme **auth.php** ,**messages.php** , **pagination.php**, **passwords.php**, **validation.php**.

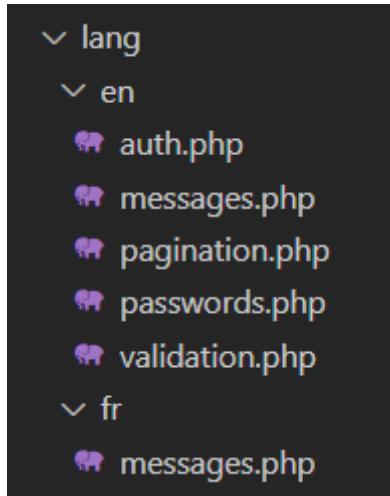
Ces fichiers sont donc des fichiers de traduction. Mais, tout d'abord, vérifions le fichier **lang/fr/messages.php**.

```
<?php
return [
    /*
    |--------------------------------------------------------------------------
    |
    | Pagination Language Lines
    |
    |
    | The following language lines are used by the paginator library to build
    | the simple pagination links. You are free to change them to anything
    | you want to customize your views to better match your application.
    |
    */
    'name required' => 'nom est obligatoire',
    'phone integer' => 'téléphone non valide',
    'welcome' => 'Bienvenue dans notre application',
    'home' => 'Accueil',
    'about' => 'A propos de nous',
    'title' => 'Comment créer un site web multilingue dans Laravel',
    'Select Language'=>'Sélectionner une langue',
];

```

Ce fichier renvoie donc un tableau contenant une paire clé-valeur. Vous pouvez donc définir n'importe quelle clé, mais les valeurs dépendent de votre langue.

Le nom du dossier racine d'un fichier **app.php** est en, donc la traduction des instructions de ce fichier est en anglais ou en, si le dossier racine est **fr**, ce qui signifie une traduction en français, le fichier **app.php** pourrait ressembler à ce qui suit.



Étape 2 : Création de fichiers de traduction

D'accord, nous allons maintenant passer une langue. L'anglais est déjà là.

1- French

Créons donc un dossier **fr** dans le dossier **lang**.

Maintenant, créez un fichier appelé **messages.php** et ajoutez le code suivant dans les deux dossiers.

Pour le fichier **fr>> messages.php**

```
<?php
// messages.php
return [
    'welcome' => 'Bienvenue dans notre application',
    'home' => 'Accueil',
    'about' => 'A propos de nous',
    'title' => 'Comment créer un site web multilangage dans Laravel 9',
    'Select Language'=>'Sélectionner une langue',
];
```

Pour le fichier **en>> messages.php**

```
<?php
// messages.php
return [
```

```

'welcome' => 'Welcome to our application!',
'home' => 'Home',
'about' => 'About',
'title' => 'How To Create Multi Language Website In Laravel 9',
'Select Language'=>'Select Language',
];

```

Étape 3 : Configurer la vue.

Tout d'abord, créez un dossier "layouts" dans le dossier "resources >> views", et dans ce dossier "layouts", créez un fichier "master.blade.php" et ajoutez le code suivant.

```

<html>
<head>
<title>@yield('title')</title>
@vite('resources/js/app.jsx')
</head>
<body>

<div class="container">
    <div class="row" style="text-align: center; margin-top: 40px;">
        <h2>{{__('messages.title')}}</h2><br>
        <div class="col-md-2 col-md-offset-3 text-right">
            <strong>{{__('messages.Select Language')}}: </strong>
        </div>
        <div class="col-md-2">

            <a
            href="{{route(\Illuminate\Support\Facades\Route::currentRouteName(), ['locale' => 'en'])}}>  English</a>
            <a
            href="{{route(\Illuminate\Support\Facades\Route::currentRouteName(), ['locale' => 'fr'])}}>  French</a>

        </div>
        <nav>
            <a href="{{route('home')}}>{{__('messages.home')}}</a>
            <a href="{{route('about')}}>{{__('messages.about')}}</a>
        </nav>
    </div>
</div>

<div class="container">
@yield('content')

```

```
</div>
</body>
</html>
```

Ouvrez maintenant votre fichier **home.blade.php**, qui se trouve dans le dossier resources >> views >> pages, et placez le code ci-dessous :

```
@extends('layouts.master')
@section('title')
    Home page
@endsection
@section('content')

<h1>{{__('messages.home')}}</h1>
{{__('messages.welcome')}}
```

```
@endsection
```

Ouvrez maintenant votre fichier **about.blade.php**, qui se trouve dans le dossier resources >> views >> pages, et placez le code ci-dessous :

```
@extends('layouts.master')
@section('title')
    About page
@endsection
@section('content')

<h1>{{__('messages.about')}}</h1>
{{__('messages.welcome')}}
```

```
@endsection
```

Étape 4 : Préfixer les routes avec la locale

Ensuite, nous allons ajouter **Route::prefix()** pour toutes les URLs possibles - ainsi toutes les pages du projet auront un préfixe de **/[locale]/[any_page]**. Voici ce que cela donne dans le fichier **routes/web.php** :

```
Route::prefix('{locale}')
    ->where(['locale' => '[a-zA-Z]{2}'])
    ->middleware('locale')
```

```

->group(function () {
    Route::get('/home',[LocalizationController::class,'index'])->name('home');
    Route::get('/about',[LocalizationController::class,'about'])->name('about');
});

Route::get('/', function () {
    return redirect()->route('home',app()->getLocale());
});

```

Toutes les routes ci-dessus ont été pré-générées, nous les avons simplement placées dans notre groupe **Route::prefix()**.

Ce groupe couvrira des URLs telles que /en/home ou /en/about.

Maintenant, à des fins de validation, définissons le **{locale}** pour qu'il ne contienne que deux lettres, comme **en** ou **fr** ou **de**. Nous ajoutons une règle basée sur des expressions régulières à l'intérieur du groupe :

Étape 5 : Définir la localisation de l'application à l'aide d'un middleware

Les gens pensent souvent que les classes Middleware servent à restreindre certains accès. Mais il est possible d'effectuer d'autres actions à l'intérieur de ces classes. Dans notre exemple, nous allons définir **app()->setLocale()**.

php artisan make:middleware SetLocale

Cette commande générera `app/Http/Middleware/SetLocale.php`, où nous devons ajouter seulement deux lignes de code :

```

class SetLocale
{
    public function handle(Request $request, Closure $next)
    {
        app()->setLocale($request->segment(1));

        URL::defaults(['locale' => $request->segment(1)]);

        return $next($request);
    }
}

```

```
}
```

La variable `$request->segment(1)` contiendra notre partie `{locale}` de l'URL, de sorte que nous définissons la locale de l'application sur toutes les requêtes. Et `URL::defaults()` définira que chaque route aura la locale par défaut.

Bien sûr, nous devons enregistrer cette classe dans `app/Http/Kernel.php` dans le tableau `$routeMiddleware` :

```
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    //
    'locale' => \App\Http\Middleware\SetLocale::class,
```

Enfin, nous appliquerons ce middleware à l'ensemble du groupe de locales que nous avons créé ci-dessus :

```
Route::prefix('{locale}')
    ->where(['locale' => '[a-zA-Z]{2}'])
    ->middleware('locale')
    ->group(function () {
        // ...
```

Étape 6 : Redirection automatisée de la page d'accueil

Nous devons ajouter une autre ligne à `routes/web.php` - pour rediriger l'utilisateur de la page d'accueil non localisée vers la page d'accueil `/fr/`. Cette route sera en dehors de la `Route::prefix()` que nous avons créée ci-dessus :

```
Route::get('/', function () {
    return redirect()->route('home', app()->getLocale());
});
```

Ceci redirigera l'utilisateur de `votredomaine.com` vers `votredomaine.com/en/home`, qui affichera la page d'accueil par défaut.

← → ⌛ ⓘ localhost:8000/en/about

How To Create Multi Language Website In Laravel 9

Select Language: English French

[Home](#) [About](#)

About

Welcome to our application!

Autorisation- Laravel 9:

1. Introduction

Laravel fournit un moyen simple d'autoriser les actions des utilisateurs sur des ressources spécifiques. Avec l'autorisation, vous pouvez autoriser de manière sélective les utilisateurs à accéder à certaines ressources tout en refusant l'accès aux autres. Laravel fournit une API simple pour gérer les autorisations utilisateur à l'aide de **Gates** et de **Policies**. **Gates** **AuthServiceProvider** approche simple de l'autorisation basée sur la fermeture à l'aide de **AuthServiceProvider** tandis que les **Policies** vous permettent d'organiser la logique d'autorisation autour des modèles utilisant des classes.

2. Création d'une politique (Policy)

Les **policies** (politiques) sont des classes qui vous aident à organiser la logique d'autorisation autour d'une ressource de modèle. En utilisant notre exemple précédent, nous pourrions avoir un **CarPolicy** qui gère l'accès des utilisateurs au modèle de **Car**. Pour rendre **CarPolicy**, laravel fournit une commande artisan. Il suffit d'exécuter la commande :

```
php artisan make:policy CarPolicy
```

Cela fera une classe de politique vide et placera dans le dossier **app/Policies**. Si le dossier n'existe pas, Laravel le créera et placera la classe à l'intérieur. Une fois créée, les stratégies doivent être enregistrées pour aider Laravel à savoir quelles politiques utiliser lors de l'autorisation d'actions sur des modèles. **AuthServiceProvider** de Laravel, fourni avec toutes les nouvelles installations Laravel, possède une propriété **\$policies** qui lie vos modèles Eloquent à leurs règles d'autorisation. Tout ce que vous devez faire ajoute le mappage au tableau.

```
protected $policies = [ Car::class => CarPolicy::class ];
```

3. Ecriture de la politique :

L'écriture de **Policies** la forme comme celle-ci:

```
function view(User $user, Car $car) {  
    return $user->id==$car->user_id;  
}
```

La politique peut contenir plus de méthodes que nécessaire pour prendre en charge tous les cas d'autorisation d'un modèle.

4. Autoriser des actions en utilisant une politique (Policy):

a. Via le contrôleur

Laravel fournit une méthode **authorize** utile à tous vos contrôleurs qui étendent la classe de base **App\Http\Controllers\Controller**.

Comme la méthode **can**, cette méthode accepte le nom de l'action que vous souhaitez autoriser et le modèle correspondant. Si l'action n'est pas autorisée, la méthode **authorize** lèvera une exception **Illuminate\Auth\Access\AuthorizationException** que le gestionnaire d'exceptions de Laravel convertira automatiquement en une réponse HTTP avec un code d'état 403 :

```
public function show(Car $car) {  
    $this->authorize('view', $car);  
    /* user can view content */  
}
```

- **Actions qui ne nécessitent pas de modèle**

Comme nous l'avons vu précédemment, certaines méthodes de politique, comme **create**, ne nécessitent pas d'instance de modèle. Dans ces situations, vous devez transmettre un nom de classe à la méthode **authorize**. Le nom de la classe sera utilisé pour déterminer la politique à utiliser lors de l'autorisation de l'action :

```
public function create(Request $request)  
{  
    $this->authorize('create', Car::class);  
}
```

```
// The current user can create content...
}
```

b. Via le modèle Blade

Lorsque vous rédigez des modèles Blade, vous pouvez souhaiter afficher une partie de la page uniquement si l'utilisateur est autorisé à effectuer une action donnée. Par exemple, vous pouvez souhaiter afficher un formulaire de mise à jour pour un article de blog uniquement si l'utilisateur peut effectivement mettre à jour l'article. Dans ce cas, vous pouvez utiliser les directives **@can** et **@cannot** :

```
@can('update', $post)
    <!-- The current user can update the post... -->
@elsecan('create', App\Models\Post::class)
    <!-- The current user can create new posts... -->
@else
    <!-- ... -->
@endcan

@cannot('update', $post)
    <!-- The current user cannot update the post... -->
@elsecannot('create', App\Models\Post::class)
    <!-- The current user cannot create new posts... -->
@endcannot
```

- **Actions qui ne nécessitent pas de modèle**

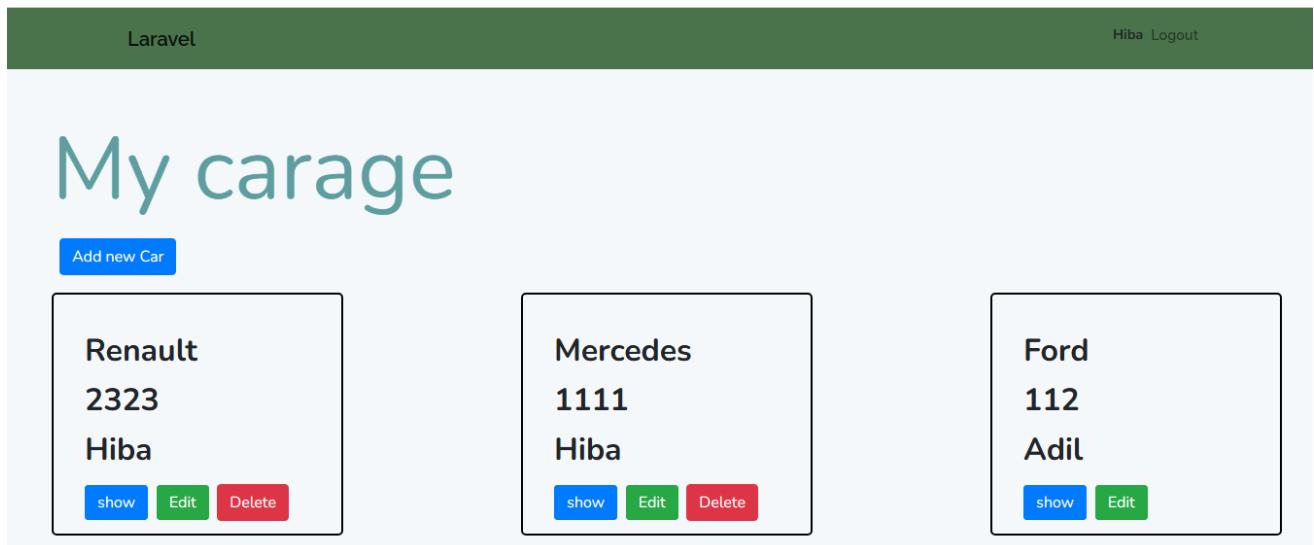
Comme la plupart des autres méthodes d'autorisation, vous pouvez passer un nom de classe aux directives **@can** et **@cannot** si l'action ne nécessite pas d'instance de modèle :

```
@can('create', App\Models\Content::class)
    <!-- The current user can create contents... -->
@endcan

@cannot('create', App\Models\Content::class)
    <!-- The current user can't create contents... -->
@endcannot
```

Travail à faire:

Dans cette section, nous allons créer une politique pour le modèle Car qui sera utilisée pour autoriser toutes les actions CRUD.



Etape 1 : Créer les modèles

La commande artisan de Laravel est votre meilleur ami lorsqu'il s'agit de créer du code. Vous pouvez utiliser les commandes artisan suivante pour créer un modèle Car.

```
php artisan make:model Car -mcr
```

app/Models/Car

```
<?php

namespace App\Models;

use App\Models\User;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Factories\HasFactory;

class Car extends Model
{
    use HasFactory;
    public function user(){
```

```
        return $this->belongsTo(User::class);

    }

    protected $fillable = [
        'brand',
        'price',
        'picture',
        'origin'
    ];
}
```

app/Models/User

```
<?php

namespace App\Models;

use App\Models\Car;
use Illuminate\Notifications\Notifiable;
use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use HasFactory, Notifiable;
    public function cars(){
        return $this->hasMany(Car::class);
    }

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

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

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

```
}
```

Etape 2 : Ajouter une colonne à la table des utilisateurs (users)

Pour que les choses restent simples et agréables, commençons par attribuer à un utilisateur le statut d'administrateur. Les administrateurs seront en mesure de voir du contenu supplémentaire et d'effectuer des tâches additionnelles dans l'application. Pour mettre en place cette fonctionnalité, nous allons ajouter une colonne booléenne à la table des utilisateurs, `isAdmin`. Ajoutez ce qui suit à votre fichier de migration des utilisateurs :

```
php artisan make:migration add_column_isAdmin_table_users --table users
```

`app/database/migrations/2023_XXXX_add_column_isAdmin_table_users.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::table('users', function (Blueprint $table) {
            $table->boolean('isAdmin')->default(0);
        });
    }

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

```

        });
    }
};
```

Exécutez maintenant la migration :

```
php artisan migrate
```

id	name	email	email_verified_at	password	remember_token	created_at	updated_at	isAdmin
24	Hiba	hiba@gmail.com	2023-01-29 15:27:36	\$2y\$10\$D3XxhH4tcjCenfGq228gwOFMRY2IJ1sS41mvh7y6FLJ... NULL		2023-01-29 15:27:16	2023-01-29 15:27:36	1
25	Adil	adil@gmail.com	2023-01-31 09:54:45	\$2y\$10\$R.EjUdpMnGwPkLz3Fgx3O7Eg9z4tuldz9tQ31rDWpD... NULL		2023-01-31 09:54:01	2023-01-31 09:54:45	0

id	brand	price	origin	picture	user_id	created_at	updated_at
1	Renault	2323	French	cars/DpJp7nrLamZtGIOBEONYc4SyKbzDJntwZyOldkW.png	24	2023-01-31 09:46:14	2023-02-01 19:49:52
2	Mercedes	1111	Germany	cars/5z3D3KlsOm67fbGBwU7u1yTtwD0u4AaCbBPiFLWk.jpg	24	2023-01-31 09:51:15	2023-02-01 19:44:43
3	Ford	112	USA	cars/DQUxHeuX8eCOiVaWBIRa55eOpbtLAatEx6u2lfie.png	25	2023-01-31 09:57:15	2023-02-01 20:00:00

Etape 3 : Création de *Policy*

```
php artisan make:policy CarPolicy --model=Car
```

Comme vous pouvez le voir, nous avons fourni l'argument **--model=Car** pour qu'il crée toutes les méthodes CRUD. En l'absence de cela, il créera une classe de politique vierge. Vous pouvez localiser la classe de politique nouvellement créée à :

app/Policies/CarPolicy.php.

```
<?php

namespace App\Policies;

use App\Models\Car;
```

```
use App\Models\User;
use Illuminate\Auth\Access\HandlesAuthorization;

class CarPolicy
{
    use HandlesAuthorization;

    public function before(User $user, $ability){
        if($user->isAdmin and $ability!='delete')
            return true;
    }

    public function viewAny(User $user)
    {
        //
    }

    public function view(User $user, Car $car)
    {
        return $user->id==$car->user_id;
    }

    public function create(User $user)
    {
        return true;
    }

    public function update(User $user, Car $car)
    {
        return $user->id==$car->user_id;
    }

    public function delete(User $user, Car $car)
    {
        return $user->id==$car->user_id;
    }

    public function restore(User $user, Car $car)
    {
        //
    }

    public function forceDelete(User $user, Car $car)
    {
        //
    }
}
```

```
}
```

Pour pouvoir utiliser notre classe de politique, nous devons l'enregistrer en utilisant le fournisseur de services de Laravel, comme le montre l'extrait suivant.

```
<?php

namespace App\Providers;

// use Illuminate\Support\Facades\Gate;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * The model to policy mappings for the application.
     *
     * @var array<class-string, class-string>
     */
    protected $policies = [
        'App\Models\Car' => 'App\Policies\CarPolicy',
    ];

    /**
     * Register any authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        //
    }
}
```

Nous avons ajouté le **mapping** de notre politique dans la propriété **\$policies**. Elle indique à Laravel d'appeler la méthode de politique correspondante pour autoriser l'action CRUD.

Etape 4 : Utiliser l'autorisation dans le contrôleur :

On va donc protéger les méthodes **edit**, **destroy** et **show** :

```
<?php

namespace App\Http\Controllers;

use App\Models\Car;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;
use Illuminate\Support\Facades\Validator;

class CarController extends Controller
{
    public function index()
    {
        if(Auth::user()->isAdmin)
            $cars=Car::all();
        else
            $cars=Auth::user()->cars;

        return view('cars.index',[

            "cars"=>$cars
        ]);
    }

    public function create()
    { $this->authorize('create',Car::class);
        return view('cars.create') ;
    }

    public function store(Request $request)
    {
        $rules=[

            "brand"=>'required',
            "price"=>['required','integer'],
            "origin"=>'required',
            "picture"=>['required','max:2048','image']

        ];
        $messages=[

            'picture.image'=>'please import an image'
        ];
        $validateData=Validator::make($request->all(),$rules,$messages);
        if($validateData->fails())
        {
```

```

        return redirect()->back()->withErrors($validateData)->withInput();
    }

    //save data
    $path_image=$request->picture->store('cars','public');
    $car=new Car();
    //strip_tags pour empêcher les scripts malveillants de s'exécuter
    $car['brand']=$request->input('brand');
    $car['origin']=$request->input('origin');
    $car['price']=$request->input('price');
    $car['picture']=$path_image;
    $car['user_id']=Auth::user()->id;
    $car->save();
    return redirect()->route('cars.index')->withSuccess('Car added
successfully');
}

public function show(Car $car)
{
    $this->authorize('view',$car);
    return view('cars.show',[

        "car"=> $car
    ]);
}

public function edit(Car $car)//
{
    $this->authorize('update',$car);
    return view('cars.edit',[

        "car"=>$car
    ]);
}

public function update(Request $request, $id)
{
    $request->validate([
        "brand"=>'required',
        "price"=>['required','integer'],
        "origin"=>'required'
    ]);

    $car=Car::findOrFail($id);
    //strip_tags pour empêcher les scripts malveillants de s'exécuter
    $car['brand']=strip_tags($request->input('brand'));
    $car['origin']=strip_tags($request->input('origin'));
    $car['price']=strip_tags($request->input('price'));
}

```

```

        $car->save();
        return redirect()->route('cars.show', $car->id);
    }

    public function destroy(Car $car)
    {
        $this->authorize('delete', $car);
        Car::delete($id);

        return Route::redirect('car.index');
    }
}

```

Et les routes correspondantes qu'on protège avec les middlewares **auth** et **verified**:

```
Route::resource('cars', CarController::class)->middleware(['auth', 'verified']);
```

Etape 5: Créer les vues en Blade

Ici, nous devons créer des fichiers de blade pour `layout`, `login`, `register` and `index` page. Donc créons un par un les fichiers :

resources/views/cars/index.blade.php

```

@extends('layouts.layout')
@section('title', "cars")
@section('content')



<h1>My garage</h1>

    <a href="{{route('cars.create')}}" class="btn btn-primary m-2">Add new Car</a>
    <div class="cars">
        @forelse ($cars as $car)

            <div class="car">
                <span> {{$car['brand'] }}</span>
                <span>{{$car['price'] }}</span>
                {{-- <span>{{$item['origin'] }}</span> --}}
                <span>{{$car['user']->name}}</span>
                <div class='groupbtn'>

                    <form action="{{route('cars.destroy', $car)}}" method="POST">
                        @method('DELETE')


```

```
@csrf
  <a href="{{route('cars.show',$car)}}" class="btn btn-primary"> show</a>
  <a href="{{route('cars.edit',$car)}}" class="btn btn-success">Edit</a>
  @can('delete', $car)
    <button class="btn btn-danger">Delete</button>
  @endcan

</form>
</div>

</div>

@empty
  <h3>there is no cars</h3>
@endforelse

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


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.

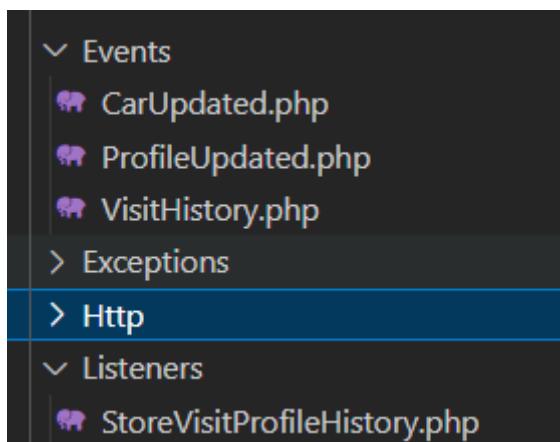
Etape 2 : Générer des événements et des écouteurs

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

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

Database/migrations/xxx_create_visit_history_table.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.php
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)

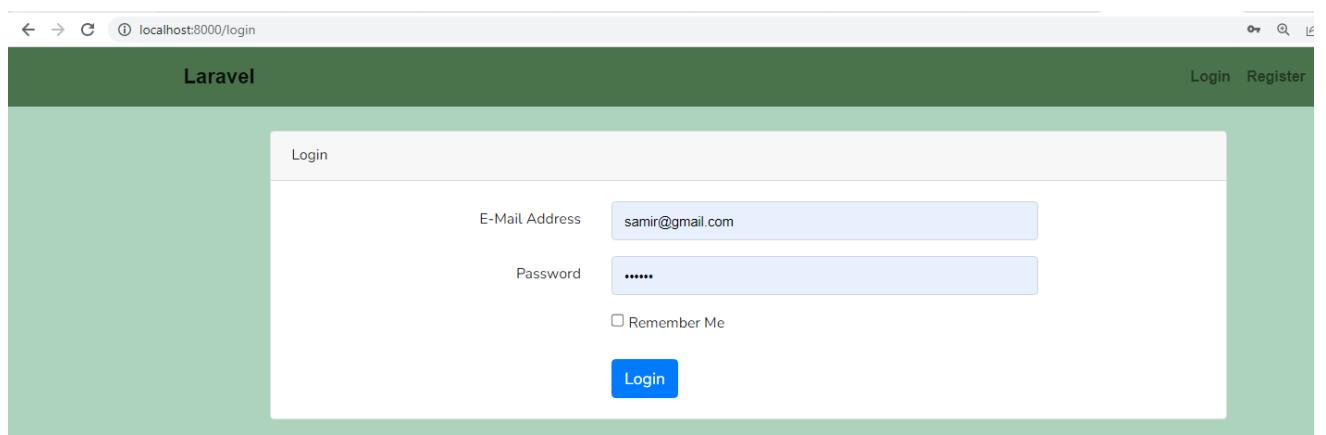
Options supplémentaires

	id	name	email	created_at	updated_at	
<input type="checkbox"/>	1	samir123	sara@gmail.com	2023-03-25 13:50:07	2023-03-25 13:50:07	<input type="checkbox"/> Éditer <input type="checkbox"/> Copier <input type="checkbox"/> Supprimer
<input type="checkbox"/>	2	Fatima123	email1@gmail.com	2023-03-25 13:53:34	2023-03-25 13:53:34	<input type="checkbox"/> Éditer <input type="checkbox"/> Copier <input type="checkbox"/> Supprimer

Authentification et inscription personnalisées dans Laravel:

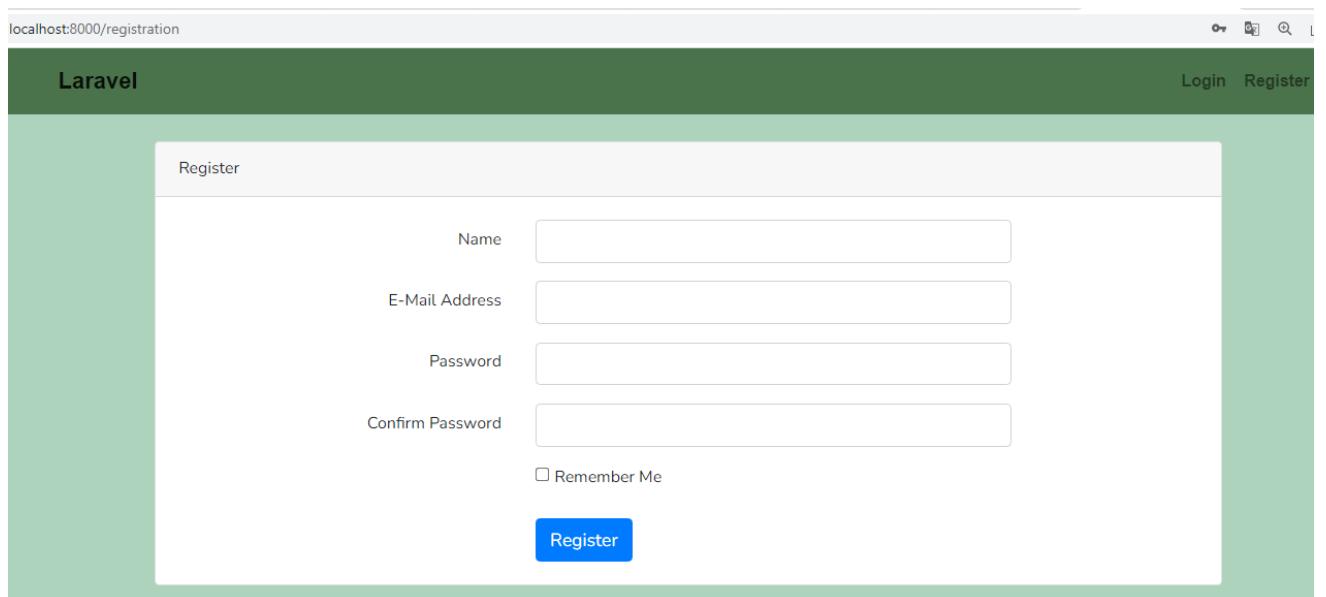
Laravel fournit l'authentification en utilisant `jetstream` et le package `UI`. Mais parfois nous avons besoin de créer notre propre `login`, enregistrement, tableau de bord, et `logout` alors on va créer étape par étape un login personnalisé et une page d'enregistrement dans l'application Laravel.

Login Page:



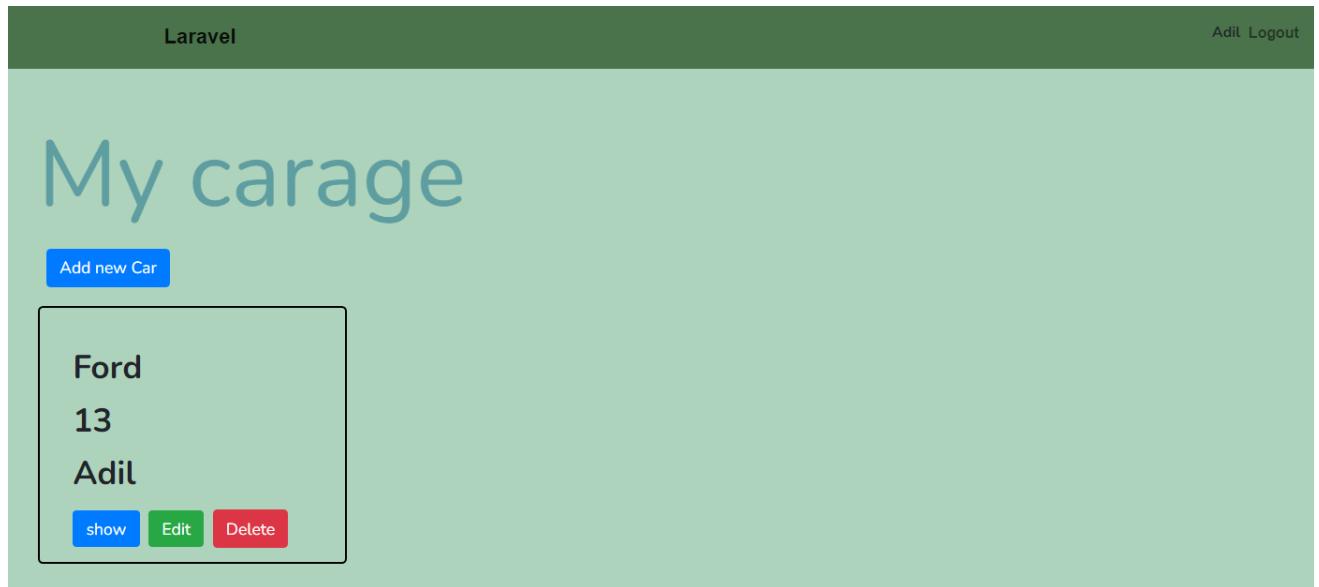
A screenshot of a web browser showing the Laravel login page. The URL in the address bar is `localhost:8000/login`. The page has a dark green header with the word "Laravel" on the left and "Login" and "Register" on the right. The main content area is titled "Login". It contains fields for "E-Mail Address" (with the value `samir@gmail.com`) and "Password" (with the value `*****`). There is a "Remember Me" checkbox and a blue "Login" button.

Register Page:



A screenshot of a web browser showing the Laravel register page. The URL in the address bar is `localhost:8000/registration`. The page has a dark green header with the word "Laravel" on the left and "Login" and "Register" on the right. The main content area is titled "Register". It contains fields for "Name", "E-Mail Address", "Password", and "Confirm Password". There is a "Remember Me" checkbox and a blue "Register" button.

Index Page:



Etape 1: Configuration de la base de données

Tout d'abord, nous devons obtenir une nouvelle version de l'application Laravel en utilisant la commande ci-dessous. Ouvrez votre terminal ou votre invite de commande et exécutez la commande ci-dessous :

```
composer create-project laravel/laravel example-app
```

Ensuite, nous devons ajouter la configuration de la base de données dans le fichier `.env`. Ajoutons donc les détails suivants et exécutons ensuite la commande de migration :

.env

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=myDB
DB_USERNAME=root
```

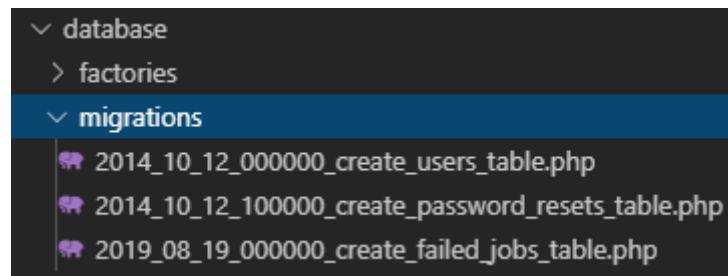
```
DB_PASSWORD=
```

Ensuite, exécutez la commande de migration pour créer la table `users` .

```
php artisan migrate
```

Par défaut la partie persistance de l'authentification (c'est à dire la manière de retrouver les renseignements des utilisateurs) avec Laravel se fait en base de données avec Eloquent et part du principe qu'il y a un modèle `App\Models\User` (dans le dossier `app`).

Lors de l'installation on a vu dans les chapitres précédents qu'il existe déjà des migrations :



```
✓ database
  > factories
  ✓ migrations
    2014_10_12_000000_create_users_table.php
    2014_10_12_100000_create_password_resets_table.php
    2019_08_19_000000_create_failed_jobs_table.php
```

Repartez d'une installation vierge et faites la migration avec Artisan :

```
λ php artisan migrate
Migration table created successfully.
Migrating: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_000000_create_users_table (375.60ms)
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated: 2014_10_12_100000_create_password_resets_table (347.34ms)
Migrating: 2019_08_19_000000_create_failed_jobs_table
Migrated: 2019_08_19_000000_create_failed_jobs_table (359.37ms)
```

Vous devriez normalement obtenir ces tables :

- table **users** : par défaut Laravel considère que cette table existe et il s'en sert comme référence pour l'authentification.
- table **password_resets** : cette table va nous servir pour la réinitialisation des mots de passe.

Etape 2: Configuration du modèle User et Création du modèle Car

//App\Models\Car.php

```
class Car extends Model
{
    use HasFactory;
    public function user(){

        return $this->belongsTo(User::class);

    }
    protected $fillable = [
        'brand',
        'price',
        'picture',
        'origin'
    ];
}
```

//App\Models\User.php

```
<?php

namespace App\Models;

use App\Models\Car;
use Illuminate\Notifications\Notifiable;
use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable implements MustVerifyEmail
{
    use HasFactory, Notifiable;
    public function cars(){
        return $this->hasMany(Car::class);
    }
    public function userverify(){

    }
}
```

```

        return $this->hasOne(UserVerify::class);
    }

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

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

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

```

database\migrations\xxxx_create_cars_table.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('cars', function (Blueprint $table) {
            $table->id();
            $table->string('brand');
            $table->integer('price');
            $table->string('origin');
        });
    }

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

```

```

        $table->string('picture');
        $table->unsignedBigInteger('user_id');
        $table->foreign('user_id')->references('id')->on('users')-
>onDelete('cascade');
        $table->timestamps();
    });
}

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

```

Etape 3: Créer le contrôleur d'Authentification

Dans cette étape, nous devons créer **AuthController** et ajouter le code suivant dans ce fichier :

app/Http/Controllers/Auth/AuthController.php

```

<?php

namespace App\Http\Controllers;

use App\Models\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;
use Illuminate\Support\Facades\Hash;
use Illuminate\Auth\Events\Registered;
use Illuminate\Support\Facades\Session;
use Illuminate\Support\Facades\Validator;

class AuthController extends Controller
{
    public function index()
    {
        return view('auth.login');
    }
}

```

```

public function registration()
{
    return view('auth.registration');
}

public function postLogin(Request $request)
{
    $rules = [
        'email' => ['required', 'email'],
        'password' => ['required','min:6'],
    ];
    $messages=[
        'email.email'=>'email not valid'
    ];

    $validateForm=Validator::make($request->all(),$rules,$messages);
    if($validateForm->fails())
        return redirect()->back()->withErrors($validateForm)->onlyInput('email');

    if (Auth::attempt(array('email'=>$request->email, 'password'=>$request->password),$request->has('remember')))) {
        $request->session()->regenerate();

        return redirect()->intended('cars');
    }

    return redirect("login")->withSuccess('Oppes! You have entered invalid credentials')->onlyInput('email');
}

public function postRegistration(Request $request)
{
    $rules=[

        'name' => 'required',
        'email' => 'required|email|unique:users',
        'password' => 'required|min:6',
        'confirmPassword' => 'required|same:password',
    ];

    $validateData=Validator::make($request->all(),$rules);
    if($validateData->fails())
        return redirect()->back()->withErrors($validateData)->withInput();
    $data=[

        "name"=>$request->name,
        "password"=>$request->password,
        "email"=>$request->email,
    ];
}

```

```

    $user=$this->create($data);
    Auth::attempt(array('email'=>$data['email'], 'password'=>$data['password']));
    return redirect("/cars")->withSuccess('Great! You have Successfully
loggedin');
}

public function create(array $data)
{
    return User::create([
        'name' => $data['name'],
        'email' => $data['email'],
        'password' => Hash::make($data['password'])
    ]);
}

public function logout() {
    Session::flush();
    Auth::logout();

    return Redirect('login');
}
}

```

La méthode **attempt** renvoie **true** si l'authentification a réussi. Dans le cas contraire, **false** sera retourné.

La méthode **intended** fournie par le redirecteur de Laravel redirigera l'utilisateur vers l'URL à laquelle il tentait d'accéder avant d'être intercepté par le middleware d'authentification. Une URI de repli peut être donnée à cette méthode au cas où la destination prévue ne serait pas disponible.

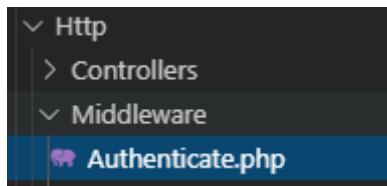
Etape 4 : Les middlewares

Je vous ai déjà parlé des middlewares qui servent à effectuer un traitement à l'arrivée (ou au départ) des requêtes HTTP. On a vu le middleware de groupe **web** qui intervient pour toutes les requêtes qui arrivent. Dans ce TP on va voir deux autres middlewares qui vont nous permettre de savoir si un utilisateur est authentifié ou pas pour agir en conséquence.

Middleware auth

Le middleware **auth** permet de n'autoriser l'accès qu'aux utilisateurs authentifiés. Ce middleware est déjà présent et déclaré dans **app\Http\Kernel.php** :

```
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    ...
];
```



On peut utiliser ce middleware directement sur une route :

```
Route::get('comptes', function() {
    // Réservé aux utilisateurs authentifiés
})->middleware('auth');
```

Ou un groupe de routes :

```
Route::middleware('auth')->group(function() {
    Route::get('/', function() {
        // Réservé aux utilisateurs authentifiés
    });
    Route::get('comptes', function() {
        // Réservé aux utilisateurs authentifiés
    });
});
```

Middleware guest

Ce middleware est exactement l'inverse du précédent : il permet de n'autoriser l'accès qu'aux utilisateurs non authentifiés. Ce middleware est aussi déjà présent et déclaré dans `app\Http\Kernel.php` :

```
protected $routeMiddleware = [
    ...
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    ...
];
```

De la même manière que `auth`, le middleware `guest`, comme d'ailleurs tous les middlewares, peut s'utiliser sur une route, un groupe de routes ou dans le constructeur d'un contrôleur, avec la même syntaxe.

Etape 5: la protection des routes

Dans cette étape, nous devons créer une route personnalisée pour la connexion, l'enregistrement, l'accueil et la déconnexion. Ouvrez donc votre fichier routes `routes/web.php` et ajoutez les routes suivantes.

`routes/web.php`

```
//Car Route
Route::middleware(['auth'])->group(function () {
    Route::resource('cars', CarController::class);
    Route::get('logout', [AuthController::class, 'logout'])->name('logout');
});

//Auth Routes

Route::group(['middleware'=>'guest'],function(){
Route::get('/',function(){
    return redirect()->route('login');
});
Route::get('login', [AuthController::class, 'index'])->name('login');
Route::post('post-login', [AuthController::class, 'postLogin'])->name('login.post');
Route::get('registration', [AuthController::class, 'registration'])->name('register');
```

```
Route::post('post-registration', [AuthController::class, 'postRegistration'])->name('register.post');

});
```

Comme vous pouvez le voir ci-dessus dans cette ligne **"Route::group(['middleware' => ['auth']], function() {"** nous avons protégé la route de déconnexion qui ne peut être accessible que si l'utilisateur est authentifié. Si vous avez d'autres routes à protéger, ajoutez simplement ce groupe de routes.

Etape 6: Créer les vues en Blade

Ici, nous devons créer des fichiers de blade pour layout, login, register et index page. Donc créons un par un les fichiers :

resources/views/auth/login.blade.php

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

@section('content')
<main class="login-form">
<div class="cotainer">
    <div class="row justify-content-center">
        <div class="col-md-8">
            <div class="card">
                <div class="card-header">Login</div>
                <div class="card-body">

                    <form action="{{ route('login.post') }}" method="POST">
                        @csrf
                        <div class="form-group row">
                            <label for="email_address" class="col-md-4 col-form-label text-md-right">E-Mail Address</label>
                            <div class="col-md-6">
                                <input type="text" id="email_address" class="form-control" value="{{ old('email') }}>
                                @error('email')
                                    <span class="text-danger">{{ $message }}</span>
                                @enderror
                            </div>
                        </div>
                    </div>
                <div class="form-group row">
```

```

        <label for="password" class="col-md-4 col-form-label
text-md-right">Password</label>
        <div class="col-md-6">
            <input type="password" id="password" class="form-
control @error('password') border border-danger @enderror" name="password" >
                @error('password')
                    <span class="text-danger">{{ $message }}</span>
                @enderror
            </div>
        </div>

        <div class="form-group row">
            <div class="col-md-6 offset-md-4">
                <div class="checkbox">
                    <label>
                        <input type="checkbox" name="remember">
                    Remember Me
                </label>
            </div>
        </div>
    </div>

    <div class="col-md-6 offset-md-4">
        <button type="submit" class="btn btn-primary">
            Login
        </button>
    </div>
    @if(Session::has('success'))
        <div class="alert alert-danger">
            {{Session::get('success')}}
        </div>
    @endif

    </form>

    </div>
</div>
</div>
</div>
</main>
@endsection

```

resources/views/auth/registration.blade.php

```

@extends('layouts.layout')

@section('content')

```

```
<main class="login-form">
  <div class="cotainer">
    <div class="row justify-content-center">
      <div class="col-md-8">
        <div class="card">
          <div class="card-header">Register</div>
          <div class="card-body">

            <form action="{{ route('register.post') }}" method="POST">
              @csrf
              <div class="form-group row">
                <label for="name" class="col-md-4 col-form-label text-md-right">Name</label>
                <div class="col-md-6">
                  <input type="text" id="name" class="form-control" @error('name') border border-danger @enderror name="name" value="{{old('name')}}" >
                  @error('name')
                    <span class="text-danger">{{ $message }}</span>
                  @enderror
                </div>
              </div>

              <div class="form-group row">
                <label for="email_address" class="col-md-4 col-form-label text-md-right">E-Mail Address</label>
                <div class="col-md-6">
                  <input type="text" id="email_address" class="form-control" @error('email') border border-danger @enderror name="email" value="{{old('email')}}" >
                  @error('email')
                    <span class="text-danger">{{ $message }}</span>
                  @enderror
                </div>
              </div>

              <div class="form-group row">
                <label for="password" class="col-md-4 col-form-label text-md-right">Password</label>
                <div class="col-md-6">
                  <input type="password" id="password" class="form-control" @error('password') border border-danger @enderror name="password" value="{{old('password')}}" >
                  @error('password')
                    <span class="text-danger">{{ $message }}</span>
                  @enderror
                </div>
              </div>
            <div class="form-group row">
```

```

        <label for="confirmPassword" class="col-md-4 col-form-label text-md-right">Confirm Password</label>
        <div class="col-md-6">
            <input type="password" id="confirmPassword" class="form-control" @error('confirmPassword') border border-danger @enderror" name="confirmPassword">
                @error('confirmPassword')
                    <span class="text-danger">{{ $message }}</span>
                @enderror
            </div>
        </div>

        <div class="form-group row">
            <div class="col-md-6 offset-md-4">
                <div class="checkbox">
                    <label>
                        <input type="checkbox" name="remember">
                    </label>
                </div>
            </div>
        </div>

        <div class="col-md-6 offset-md-4">
            <button type="submit" class="btn btn-primary">
                Register
            </button>
        </div>
    </form>

    </div>
</div>
</div>
</div>
</div>
</main>
@endsection

```

resources/views/layouts/layout.blade.php

```

<!DOCTYPE html>
<html>
<head>
    <title>Laravel</title>
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css">
    <style type="text/css">
        @import url(https://fonts.googleapis.com/css?family=Raleway:300,400,600);

```



```

        </button>

        <div class="collapse navbar-collapse" id="navbarSupportedContent">
            <ul class="navbar-nav ml-auto ">
                @guest
                    <li class="nav-item ">
                        <a class="nav-link" href="{{ route('login') }}">Login</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="{{ route('register') }}>Register</a>
                    </li>
                @else
                    <li class="nav-item mt-2">
                        <p>{{Auth::user()->name}}</p>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link" href="{{ route('logout') }}">Logout</a>
                    </li>
                @endguest
            </ul>
        </div>
    </div>
</nav>

@yield('content')

</body>
</html>

```

resources/views/cars/index.blade.php

```

@extends('layouts.layout')
@section('title', "cars")
@section('content')



@if (session('success'))


{{Session::get('success')}}


@endif


# My carage

Add new Car

@forelse ($cars as $car)


```

```
<div class="car">
  <span> {{$car['brand'] }}</span>
  <span>{{$car['price']}}</span>
  {{-- <span>{{$item['origin'] }}</span> --}}
  <span>{{$car['user']->name}}</span>
  <div class='groupbtn'>

<form action="{{route('cars.destroy',$car)}}" method="POST">
  @method('DELETE')
  @csrf
  <a href="{{route('cars.show',$car)}}" class="btn btn-primary"> show</a>
  <a href="{{route('cars.edit',$car)}}" class="btn btn-success">Edit</a>
  @can('delete', $car)
    <button class="btn btn-danger">Delete</button>
  @endcan

</form>
</div>

</div>

@empty
  <h3>there is no cars</h3>
@endforelse

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

My carage

[Add new Car](#)

Renault

23236

Hiba

[show](#)

[Edit](#)

[Delete](#)

Mercedes

12

Hiba

[show](#)

[Edit](#)

[Delete](#)

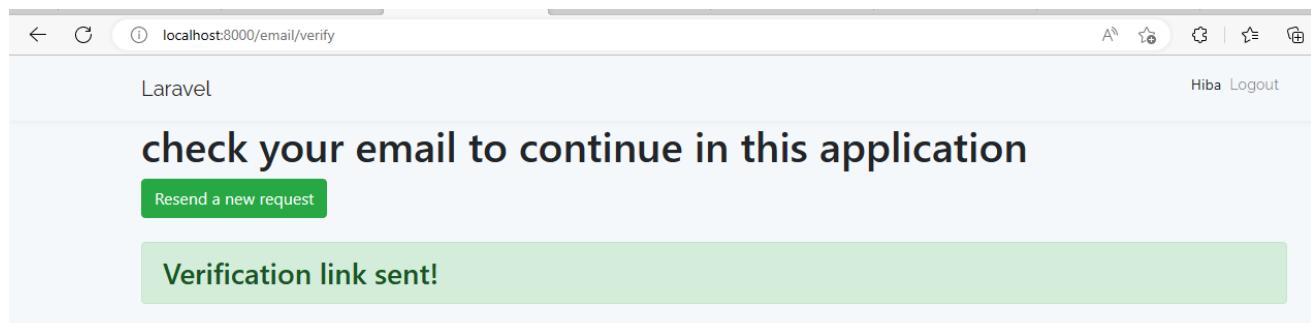
Laravel : Vérification personnalisée par courriel avec code d'activation (Email verification)

Laravel fournit un système de vérification d'email par défaut dans son système d'authentification. Nous pouvons l'utiliser pour créer un système de vérification d'email avant la confirmation d'enregistrement.

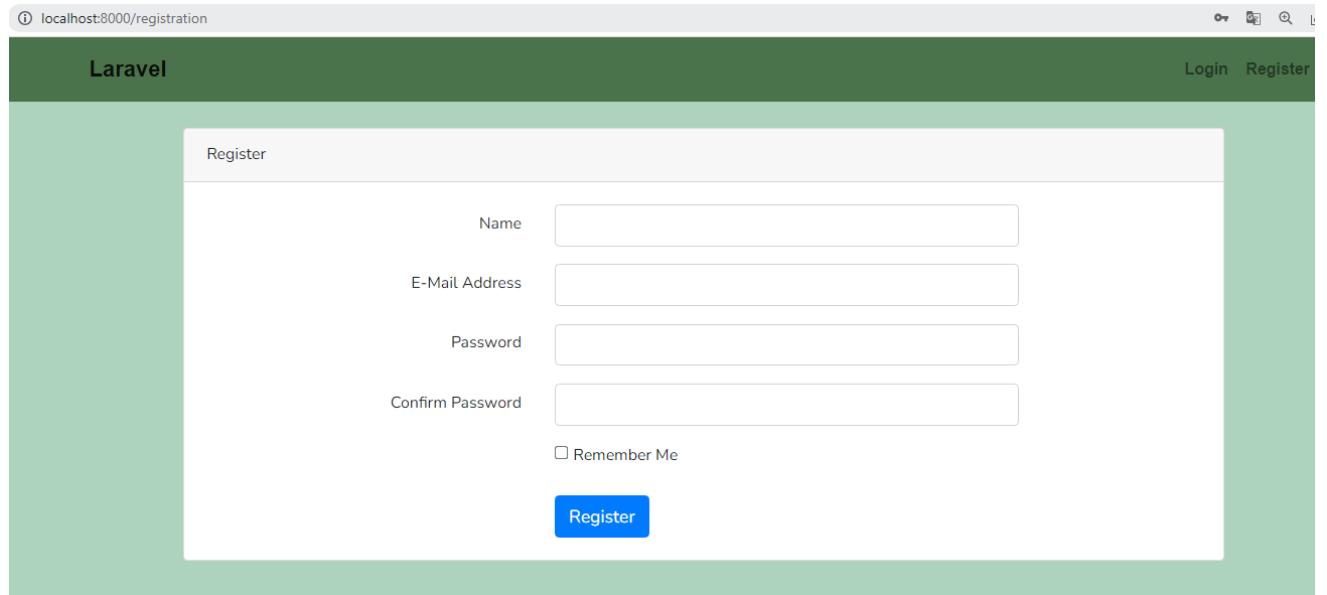
Mais dans cet exemple, je vais créer un modèle de vérification d'email personnalisé laravel. Ainsi, vous apprendrez la vérification d'email personnalisée de Laravel à partir de ce TP avec des conseils étape par étape.

Voir l'aperçu des images ci-dessous :

Email verification notify page:



Register Page:



A screenshot of a web browser showing a Laravel registration form. The URL in the address bar is `localhost:8000/registration`. The page has a dark green header with the word "Laravel" on the left and "Login" and "Register" on the right. The main content area has a light green background and a white registration form. The form is titled "Register" and contains four input fields: "Name", "E-Mail Address", "Password", and "Confirm Password". Below these fields is a "Remember Me" checkbox and a blue "Register" button.

Modèle d'email avec code d'activation:

Verify your address email

From: Mail for you <aminefste@gmail.com>
To: <user10@gmail.com>

[Show Headers](#)

[HTML](#) [HTML Source](#) [Text](#) [Raw](#) [Spam Analysis](#) [HTML Check 2](#) [Tech Info](#)



Click in this link below to Verify your address email :

[link](#)

Etape 1: S'inscrire à mailtrap

Enregistrez-vous sur **mailtrap** pour envoyer des emails de test. Et obtenez les informations d'identification dans la boîte de réception de démonstration.

My Inbox

SMTP
Settings

Email
Address

Auto
Forwar

Integrations ?

Laravel 7+



Laravel provides a clean, simple API over the popular Sw

With the default Laravel setup you can configure your m
.env file in the root directory of your project.

```
MAIL_MAILER=smtp
MAIL_HOST=smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=b84fba37705c05
MAIL_PASSWORD=21d400d37c5d23
MAIL_ENCRYPTION=tls
```

Ensute, nous devons ajouter la configuration de **Mailer** dans le fichier `.env` .

`.env`

```
MAIL_MAILER=smtp
MAIL_HOST=smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=b84fba37705c05
MAIL_PASSWORD=21d400d37c5d23
MAIL_ENCRYPTION=tls
```

Exécutons ensuite les commandes suivantes :

- Arrétez le serveur `ctrl+c`
- `php artisan config:cache`
- `php artisan serve`

Etape 2 : Créer une table de migration et le modèle UserVerify

Exécutez la commande :

```
Php artisan make :model UserVerify -m
```

Nous allons créer une table **users_verify**:

```
// App\Models\UserVerify.php
namespace App\Models;

use App\Models\User;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Factories\HasFactory;

class UserVerify extends Model
{
    use HasFactory;
    protected $table = "users_verify";
    protected $fillable = [
        'user_id',
        'token',
    ];
    public function user()
    {
        return $this->belongsTo(User::class);
    }
}
```

```
// database|migrations\xxxx_xx_xx_xxx_create_user_verifies_table.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('users_verify', function (Blueprint $table) {
            $table->id();
            $table->unsignedBigInteger('user_id');
            $table->foreign('user_id')->references('id')->on('users')-
>onDelete('cascade');
            $table->string('token');
            $table->timestamps();
        });
    }

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

```

Etape 3 : Configurer le modèle User

```

// App\Models\User.php
<?php

namespace App\Models;

use App\Models\Car;
use App\Models\UserVerify;
use Illuminate\Notifications\Notifiable;
use Illuminate\Contracts\Auth\MustVerifyEmail;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use HasFactory, Notifiable;
    public function cars(){
        return $this->hasMany(Car::class);
    }
    public function userverify(){
        return $this->hasOne(UserVerify::class);
    }
}

```

```

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

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

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

```

Etape 3 : Créer des routes

Dans cette étape, nous devons créer une route personnalisée pour la vérification de l'email. Ouvrez donc votre fichier **routes/web.php** et ajoutez la route suivante.

//routes/web.php

```

//Routes Auth

Route::group(['middleware'=>'guest'],function(){
Route::get('/',function(){
    return redirect()->route('login');
});
Route::get('login', [AuthController::class, 'index'])->name('login');
Route::post('post-login', [AuthController::class, 'postLogin'])->name('login.post');
Route::get('registration', [AuthController::class, 'registration'])-
>name('register');
Route::post('post-registration', [AuthController::class, 'postRegistration'])-
>name('register.post');

});
Route::get('logout', [AuthController::class, 'logout'])->name('logout')-
>middleware('auth');

```

```

//Routes for email verification
Route::get('account/verify/{token}', [AuthController::class, 'verifyAccount'])->middleware('auth')->name('user.verify');
Route::get('/email/verify', function () {
    return view('auth.verify-email');
})->middleware('auth')->name('verification.notice');
Route::post('/email/verification-notification',[AuthController::class, 'sendAnotherVerification'])
    ->middleware('auth')->name('sendAnotherVerification');
//Route Car
Route::middleware(['auth','verifiedEmail'])->group(function () {
    Route::resource('cars', CarController::class);
});

});

```

Etape 3: Ajouter deux actions au contrôleur AuthController

Dans cette étape, nous devons mettre à jour le code des méthodes `postRegistration()` et ajoutons les actions `verifyAccount($token)` et `sendAnotherVerification()`. Copions-les comme suit :

app/Http/Controllers/Auth/AuthController.php

```

<?php

namespace App\Http\Controllers;

use Carbon\Carbon;
use App\Models\User;
use App\Models\UserVerify;
use Illuminate\Support\Str;
use Illuminate\Http\Request;
use App\Mail\SendVerificationMail;
use Illuminate\Support\Facades\Auth;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Facades\Mail;
use Illuminate\Auth\Events\Registered;
use Illuminate\Support\Facades\Session;
use Illuminate\Support\Facades\Validator;

class AuthController extends Controller
{
    public function index()
    {

```

```

        return view('auth.login');
    }

    public function registration()
    {
        return view('auth.registration');
    }

    public function postLogin(Request $request)
    {
        $rules = [
            'email' => ['required', 'email'],
            'password' => ['required','min:6'],
        ];
        $messages=[
            'email.email'=>'email not valid'
        ];
        // dd($request->has('remember')); return false if unchecked
        $validateForm=Validator::make($request->all(),$rules,$messages);
        if($validateForm->fails())
            return redirect()->back()->withErrors($validateForm)->onlyInput('email');

        if (Auth::attempt(array('email'=>$request->email,'password'=>$request->password),$request->has('remember'))) {
            $request->session()->regenerate();

            return redirect()->intended('cars');
        }

        return redirect("login")->withSuccess('Oppes! You have entered invalid
credentials')->onlyInput('email');
    }

    public function postRegistration(Request $request)
    {
        $rules=[
            'name' => 'required',
            'email' => 'required|email|unique:users',
            'password' => 'required|min:6',
            'confirmPassword' => 'required|same:password',
        ];
        $validateData=Validator::make($request->all(),$rules);
        if($validateData->fails())
            return redirect()->back()->withErrors($validateData)->withInput();
        $data=[
```

```

        "name"=>$request->name,
        "password"=>$request->password,
        "email"=>$request->email,
    ];
    $user=$this->create($data);
    // dd($user);
    // event(new Registered($user));
    $token=Str::random(50);
    UserVerify::create([
        'user_id'=>$user->id,
        'token'=>$token
    ]);
    Mail::to($request->email)->queue(new SendVerificationMail($token));
    Auth::attempt(array('email'=>$data['email'], 'password'=>$data['password']));
    return redirect("/cars")->withSuccess('Great! You have Successfully
loggedin');
}

public function create(array $data)
{
    return User::create([
        'name' => $data['name'],
        'email' => $data['email'],
        'password' => Hash::make($data['password'])
    ]);
}

public function logout() {
    Session::flush();
    Auth::logout();
    return Redirect('login');
}
public function verifyAccount($token){
    $userVerify=Auth::user()->userverify()->first();
    if($userVerify->token==$token)
    {
        $user=User::find(Auth::user()->id);
        $user->email_verified_at=Carbon::now();
        $user->save();
        return redirect()->route('cars.index');
    }
}

```

```

        else
            return redirect()->route('verification.notice');

    }

    public function sendAnotherVerification(){

        $userVerify=Auth::user()->userverify()->first();
        Mail::to(Auth::user()->email)->queue(new SendVerificationMail($userVerify->token));
        return back()->with('message', 'Verification link sent!');
    }

}

```

Etape 4: Créer les vues en Blade

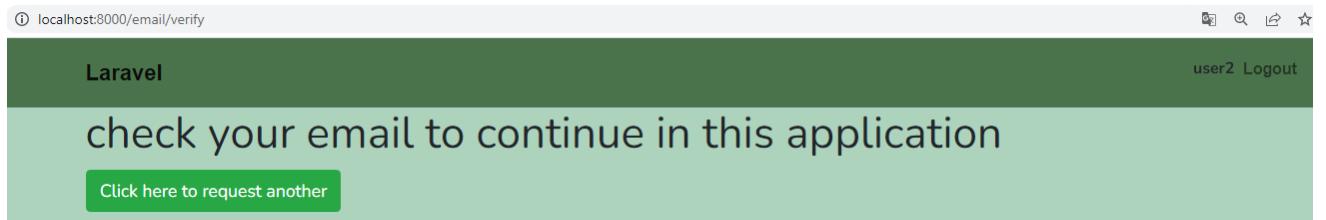
Ici, nous devons créer une vue de blade pour `verify-email`. Une vue demandant à l'utilisateur de cliquer sur le lien de vérification de l'adresse électronique qui lui a été envoyé par Laravel après son inscription. Cette vue sera affichée aux utilisateurs lorsqu'ils essaieront d'accéder à d'autres parties de l'application sans avoir vérifié leur adresse électronique au préalable :

`resources/views/auth/verify-email.blade.php`

```

@extends('layouts.layout')
@section('content')
    <div class="container">
        <h1>check your email to continue in this application</h1>
        <form action="{{ route('sendAnotherVerification') }}" method="POST">
            @csrf
            @method('POST')
            <button class="btn btn-success" type="submit">Click here to request
another</button>
        </form> <br>
        @if (Session::has('message'))
            <h3 class="alert alert-success"> {{ Session::get('message') }} </h3>
        @endif
    </div>
@endsection

```



A screenshot of a web browser showing a Laravel application. The URL is `localhost:8000/email/verify`. The page has a green header with the text "Laravel" and "user2 Logout". The main content area has a green background and contains the text "check your email to continue in this application" and a green button labeled "Click here to request another".

Ici, nous devons créer des fichiers de blade pour l'email uniquement (la vue qui s'affiche dans la boite email).

resources/views/mail/verifymailview.blade.php

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
<body>
    <h3>Click in this link below to Verify your address email :</h3>
    <a href="{{URL::temporarySignedRoute('user.verify', now() ->addMinutes(30), ['token' => $token])}}>link
    </a>

</body>
</html>
```

Verify your address email

From: Mail for you <aminefste@gmail.com>
To: <user1@gmail.com>

[Show Headers](#)

[HTML](#) [HTML Source](#) [Text](#) [Raw](#) [Spam Analysis](#) [HTML Check 2](#) [Tech Info](#)



Click in this link below to Verify your address email :

[link](#)

Etape 5 : Création d'un Middleware pour vérifier les emails

Ici, nous devons créer un middleware pour vérifier si l'email de l'utilisateur est vérifié. Nous allons donc le créer comme ci-dessous :

```
php artisan make:middleware VerifiedEmailMiddleware
```

app/Http/Middleware/ VerifiedEmailMiddleware.php

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

class VerifiedEmailMiddleware
{
    public function handle(Request $request, Closure $next)
    {
        if(Auth::user()->email_verified_at)
            return $next($request);
        else
            return redirect()->route('verification.notice');
    }
}
```

Puis enregistrer le middleware :

//app/Http/Kernel.php

```
protected $routeMiddleware = [
    ....
    'verifiedEmail' =>\App\Http\Middleware\VerifiedEmailMiddleware::class,
];
```

Etape 6 : Création de la classe Mailable

Dans cette étape, nous allons créer la classe **Mailable**, qui sera utilisée pour envoyer des courriels. La classe **Mailable** est responsable de l'envoi des emails en utilisant un mailer qui est configuré dans le fichier **config/mail.php**. En fait, Laravel fournit déjà une commande artisan qui nous permet de créer un modèle de base.

```
php artisan make:mail SendVerificationMail
```

//app/Mail/SendVerificationMail.php

```
<?php

namespace App\Mail;

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

class SendVerificationMail extends Mailable
{
    use Queueable, SerializesModels;

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

    /**
     * Get the message envelope.
     *
     * @return \Illuminate\Mail\Mailables\Envelope
     */
    public function envelope()
    {
        return new Envelope(
            subject: 'Verify your adress email',
        );
    }
}
```

```
        );
    }

    /**
     * Get the message content definition.
     *
     * @return \Illuminate\Mail\Mailables\Content
     */
    public function content()
    {
        return new Content(
            view: 'mail.verifymailview',
        );
    }

    /**
     * Get the attachments for the message.
     *
     * @return array
     */
    public function attachments()
    {
        return [];
    }
}
```

