

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. Création 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

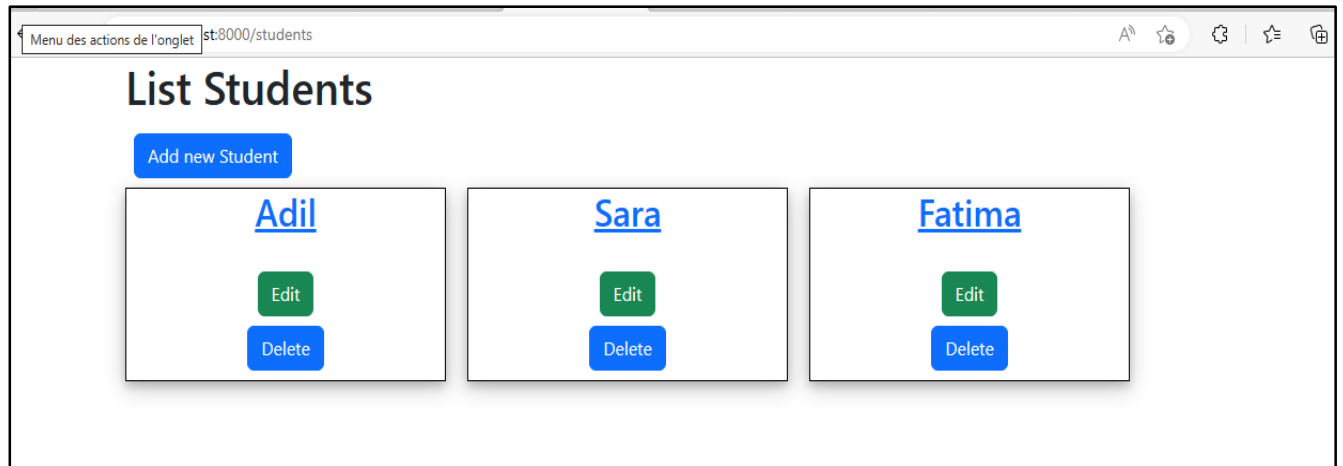
            <button class="btn btn-primary m-2">Delete</button>
        </form>

    </div>
```

```

</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, retournons-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.

Implémentions 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 spécifiée :

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

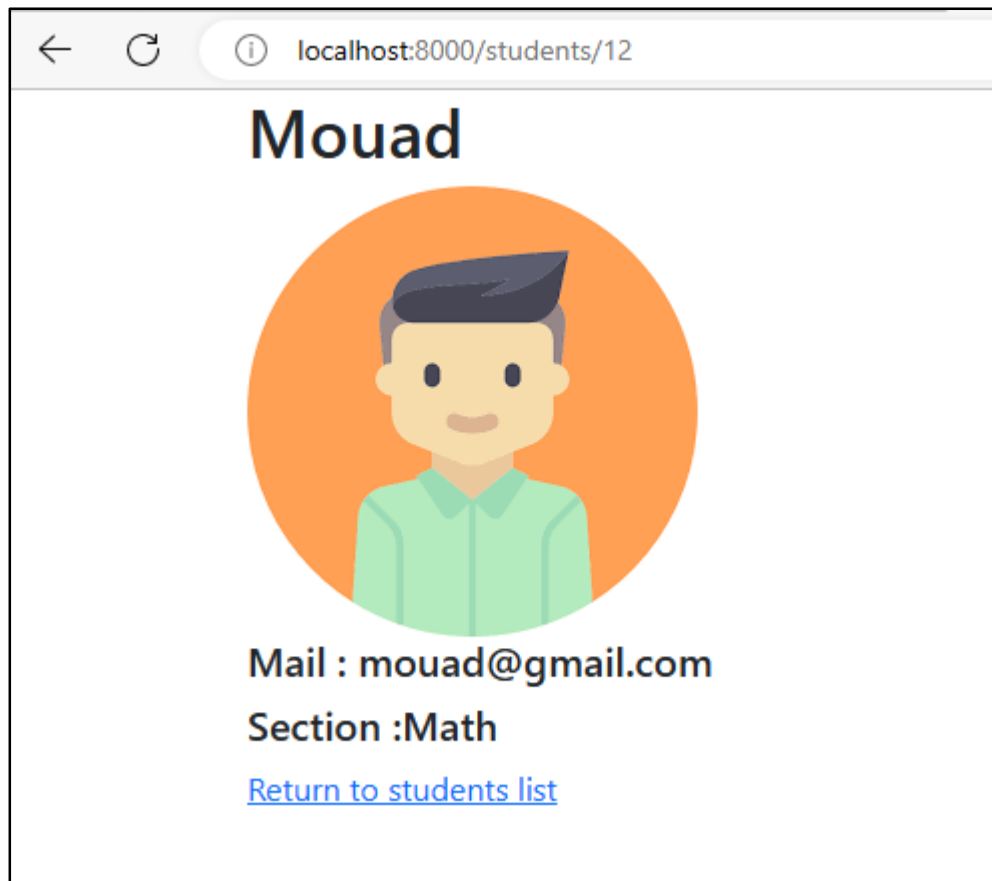
Pour afficher un Student enregistré, nous le récupérons à partir de son identifiant puis le transmettons à la vue **/resources/views/students/show.blade.php** :

```
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:

Sara

E-mail:

sara@gmail.com

Picture:



Choisir un fichier

Aucun fichier n'a été sélectionné

Section:

Physique

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

Implémentions 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")) {

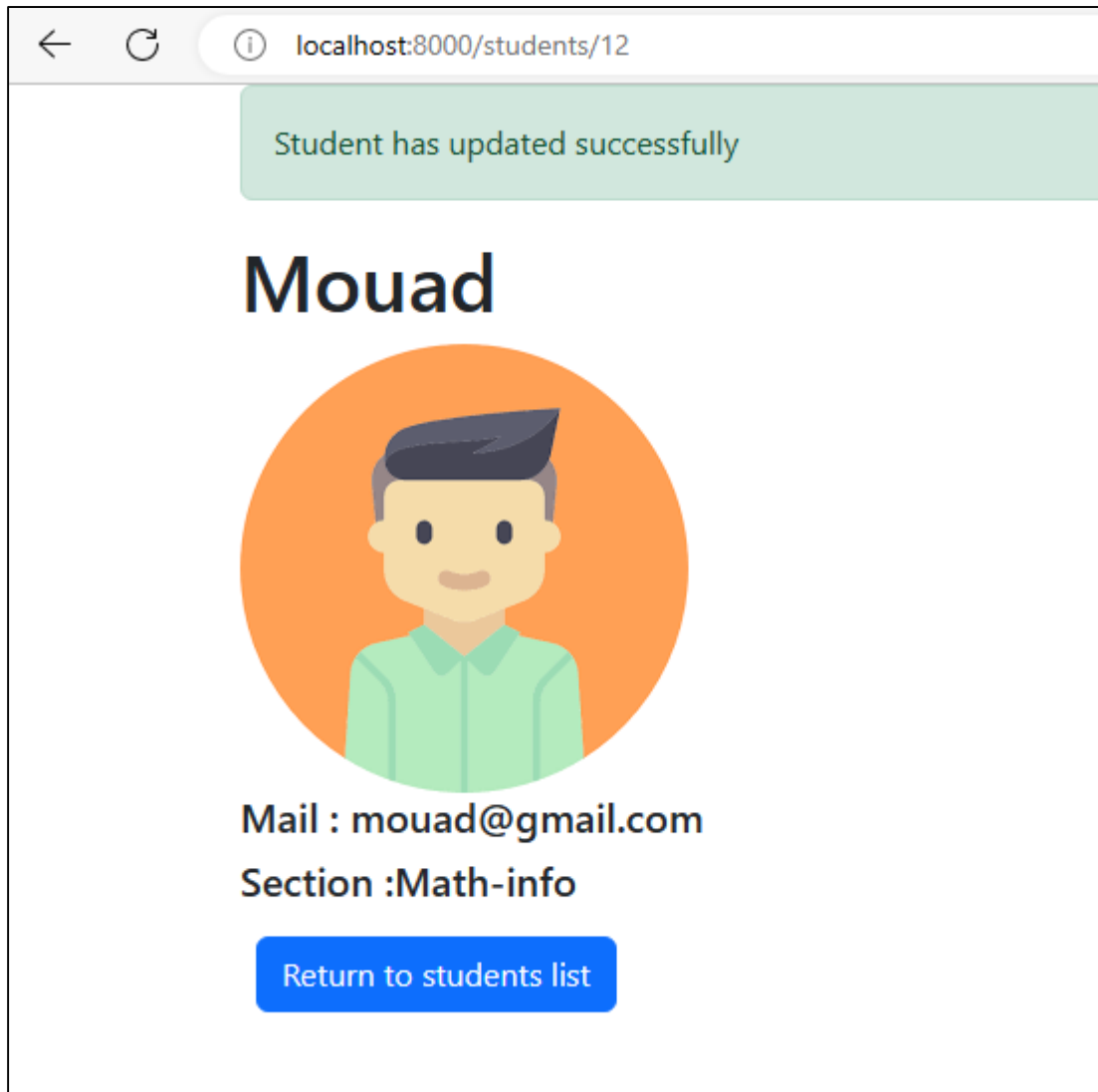
        //On supprime l'ancienne image
        Storage::delete($student->picture);

        $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"]);
}

```



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	/students/{student}	destroy	students.destroy

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 contrôleurs 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);
```

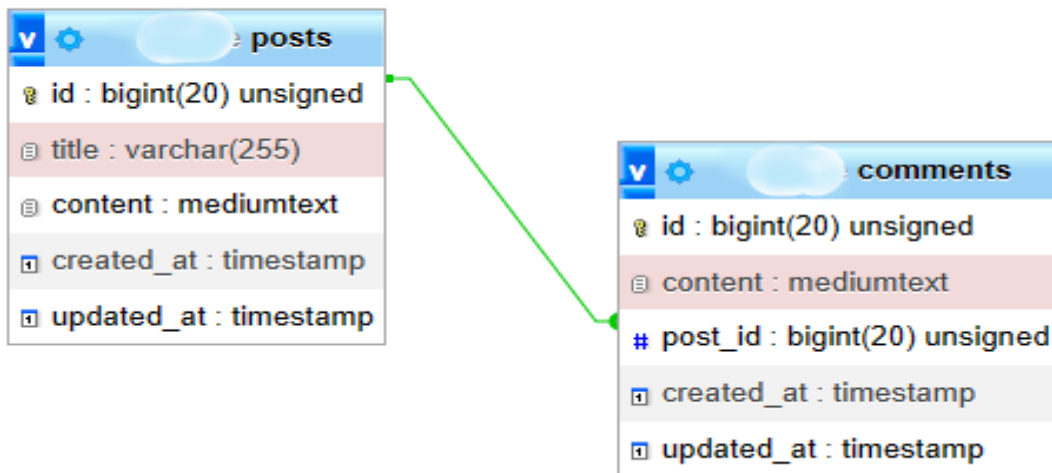
Cette définition de route définira les routes suivantes :

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"/>		Éditer	 Copier	 Supprimer	1	my first post	learn css an javascript	NULL	NULL
<input type="checkbox"/>		Éditer	 Copier	 Supprimer	2	my second post	learn python	NULL	NULL

Comments

						id	content	post_id	created_at	updated_at	
<input type="checkbox"/>		Éditer		Copier		Supprimer	1	thanks so much for this course	1	NULL	NULL
<input type="checkbox"/>		Éditer		Copier		Supprimer	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'));
}
```

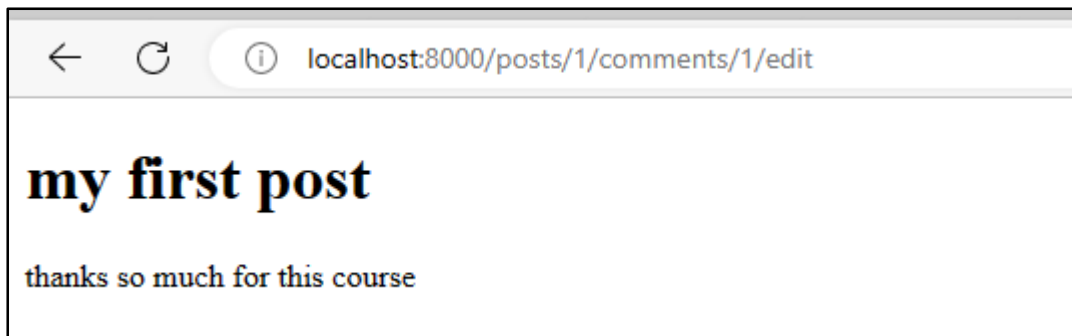
// resources/views/posts/edit.blade.php

```
<!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 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 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);
```