Le Framework PHP
Laravel

Pr. Abdelali El Gourari

Plan: Laravel

- *Introduction*
- *MVC*
- *Composer*
- *Installation de Composer*
- *Installation de Laravel*
- *Structure des dossiers*
- *Relier la base de données*
- *Créer les models*
- *Controllers*
- *Routes*
- *Créer la vue*

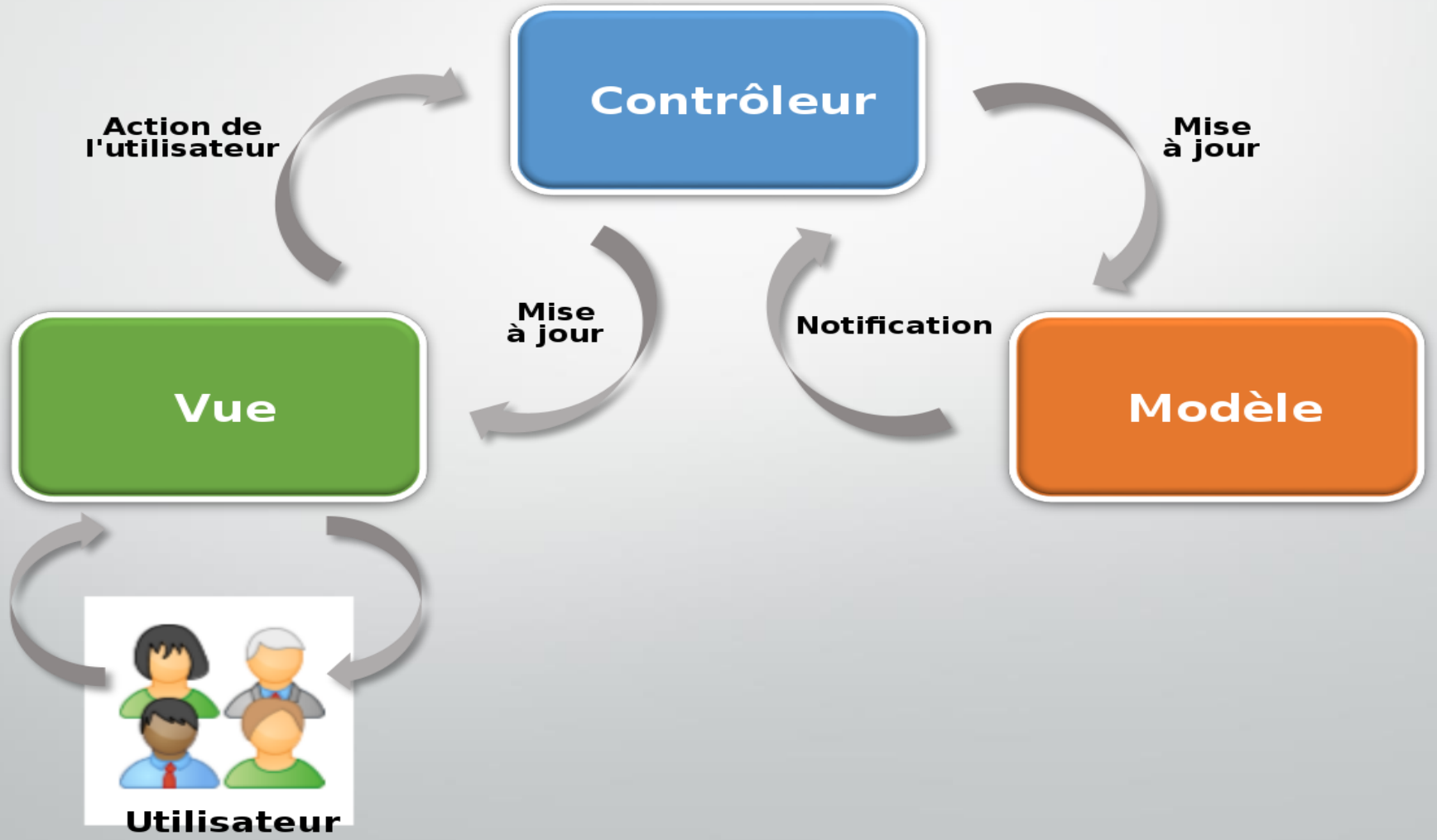
Laravel

Introduction

- *Laravel est un Framework PHP libre de droits qui a fait son apparition en 2011.*
- *L'un des meilleurs frameworks avec Symfony et CodeIgniter.*
- *Laravel utilise la toute dernière version de PHP et a fréquemment des patches de disponibles avec de nouveaux éléments et des mises à jour qui règlent les problèmes, ce qui prouve qu'il est en constante évolution et amélioration.*
- *il se base sur « Composer », le meilleur outil de dépendance qui gère des projets en PHP jusqu'à maintenant.*

MVC

Laravel se base effectivement sur le patron de conception MVC, c'est-à-dire modèle, vue et contrôleur.



Composer

- La première étape serait d'installer le Composer dont Laravel utilise.
- Qu'est ce qu'un composer? Composer trouve les fichiers PHP qu'on a besoin dans un projet.
- Il va les chercher et les installer à la bonne place.
- Comme Laravel est un Framework PHP, il est très pratique de l'utiliser pour bien partir un projet.

Installation de Composer

- On peut le télécharger sur le site getcomposer.org ou directement sur le site de laravel.com.
- Ce Composer a une entente avec Laravel et va chercher tous les fichiers que Laravel utilise pour bien fonctionner.
- En installant le Composer, on pourra installer beaucoup plus facilement et rapidement le Framework Laravel, sans faire d'erreur.
- Une fois le fichier « Composer-Setup.exe » est installé dans l'ordinateur, on peut maintenant installer Laravel.

Installation de Laravel

- Il faut savoir où installer le projet. Je vous conseille de travailler en localhost, avant de travailler sur le serveur lui-même, pour simple efficacité.
- Ouvrez une fenêtre Windows de où vous voulez mettre le dossier de projet et faites click-droit, si vous avez bien installé le Composer, vous devriez voir dans la liste « use Composer here ».
- Après avoir installé PHP et Composer, vous pouvez créer un nouveau projet Laravel via la commande Composer create-project :

composer create-project laravel/laravel your-project-name

Installation de Laravel

- Composer : signifie qu'on utiliser le Composer.
- create-project : signifie qu'on crée un projet.
- laravel/laravel : va chercher les fichiers à installer pour le Framework Laravel.
- your-project-name : on met le nom qu'on veut donner à notre projet.

Vous pouvez également créer de nouveaux projets Laravel en installant globalement le programme d'installation de Laravel via Composer :

- ✓ **composer global require laravel/installer**
- ✓ **laravel new your-project-name**

Installation de 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 :
  - ✓ **cd your-project-name**
  - ✓ **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.

Artisan !

- **Artisan** est une interface de ligne de commande pour **Laravel**
- Commandes qui font économiser **le temps**
- Générer des fichiers avec Artisan est **recommandé**
- L'exécution de php **artisan list** dans la console

```
app
 app:name              Set the application namespace
auth
 auth:clear-resets     Flush expired password reset tokens
cache
 cache:clear           Flush the application cache
 cache:forget          Remove an item from the cache
 cache:table           Create a migration for the cache database table
config
 config:cache          Create a cache file for faster configuration loading
 config:clear          Remove the configuration cache file
db
 db:seed               Seed the database with records
event
 event:generate        Generate the missing events and listeners based on registration
ide-helper
 ide-helper:generate   Generate a new IDE Helper file.
 ide-helper:meta       Generate metadata for PhpStorm
 ide-helper:models     Generate autocompletion for models
key
 key:generate          Set the application key
make
 make:auth             Scaffold basic login and registration views and routes
 make:command          Create a new Artisan command
 make:controller       Create a new controller class
 make:event            Create a new event class
 make:job              Create a new job class
 make:listener         Create a new event listener class
 make:mail             Create a new email class
 make:middleware       Create a new middleware class
 make:migration        Create a new migration file
 make:model            Create a new Eloquent model class
 make:notification     Create a new notification class
 make:policy           Create a new policy class
 make:provider         Create a new service provider class
 make:request          Create a new form request class
 make:seeder           Create a new seeder class
 make:test             Create a new test class
migrate
 migrate:install       Create the migration repository
 migrate:refresh       Reset and re-run all migrations
 migrate:reset         Rollback all database migrations
 migrate:rollback      Rollback the last database migration
 migrate:status        Show the status of each migration
notifications
 notifications:table   Create a migration for the notifications table
queue
 queue:failed          List all of the failed queue jobs
 queue:failed-table    Create a migration for the failed queue jobs database table
 queue:flush           Flush all of the failed queue jobs
```

Structure des dossiers

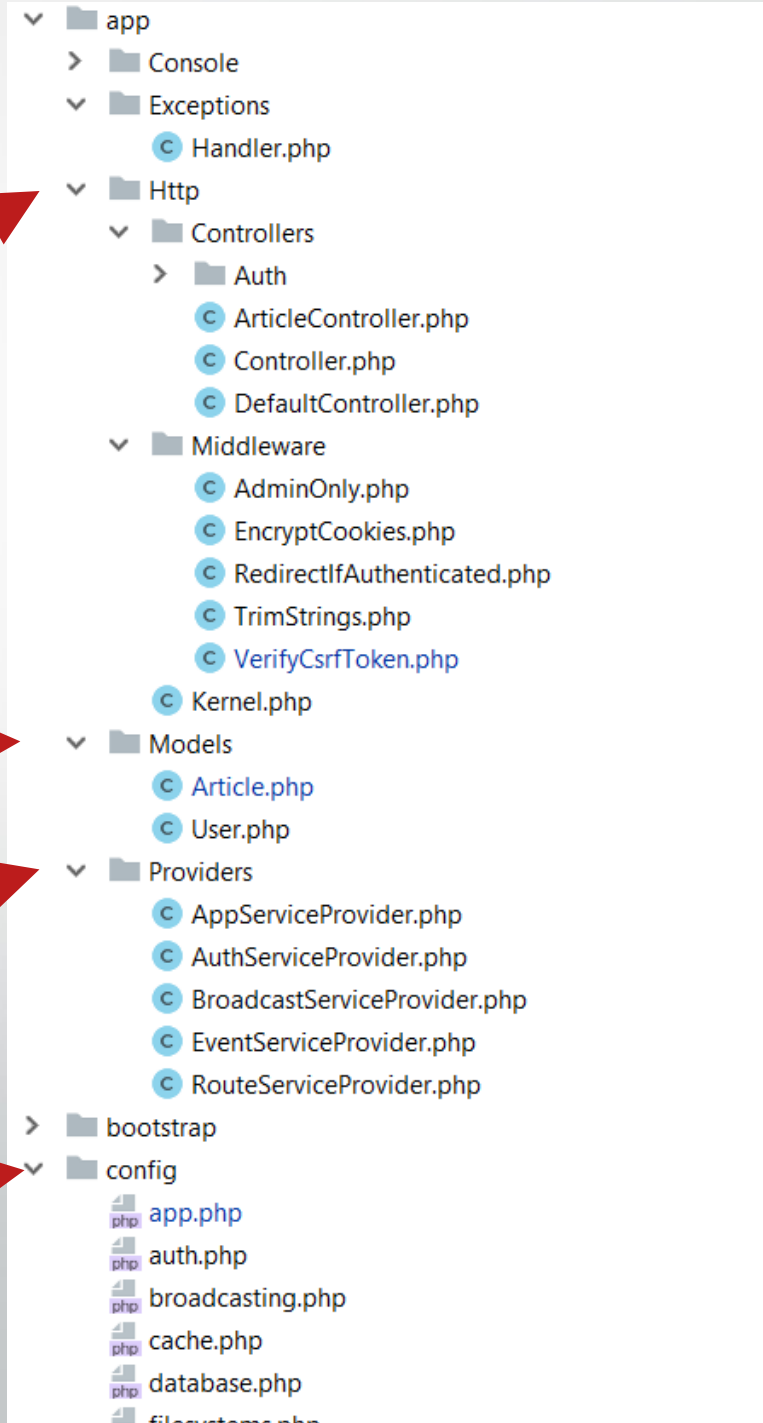
Il est important de d'abord analyser la structure des dossiers pour savoir comment la hiérarchie fonctionne.

Le dossier **app/Http** contient le fichier **Controllers, Middlewares** et **Kernel**.

Tous les modèles doivent être situés dans le dossier **app/Models**.

Les fournisseurs de services qui amorcent les fonctions de notre application sont situés dans le dossier **app/Providers**.

Tous les fichiers de configuration sont situés dans le dossier **app/config**.



Le dossier **Database** contient les **migrations** et **seeds**

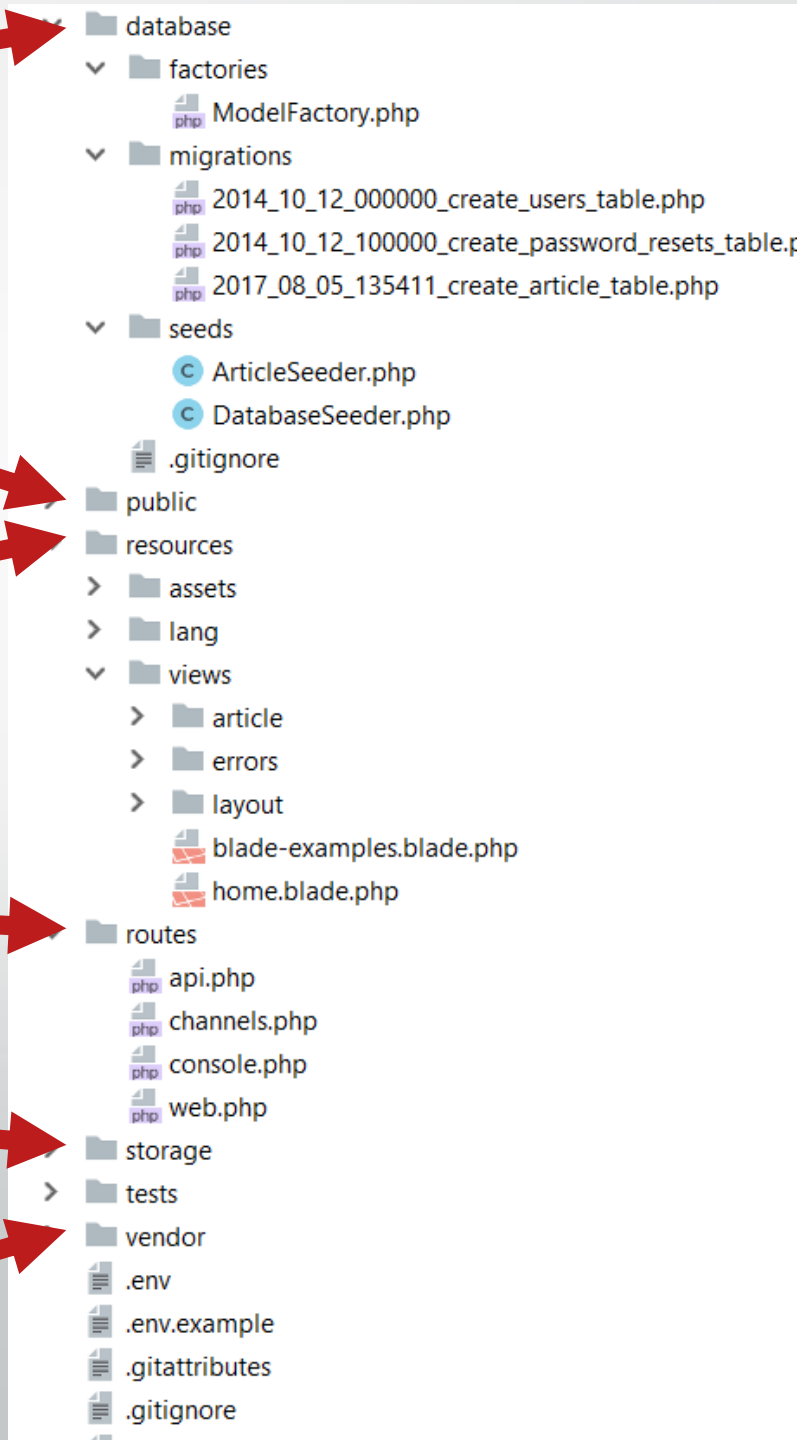
Le dossier public est le dossier réel que vous ouvrez sur le serveur Web.
Tous les fichiers JS / CSS / Images / Uploads sont placés là.

Le dossier des ressources contient toutes les **traductions,** les **vues** et **les actifs** (SASS, LESS, JS) qui sont compilés dans le dossier public.

Le dossier **routes** contient toutes les routes du projet.

Tous **les journaux / fichiers** de cache sont situés dans le dossier de **stockage**

Le dossier **vendor** contient tous les paquets (dépendances) du compositeur.



Relier la base de données

il faut relier la base de données au projet. Dans le dossier **config**, il existe le fichier **database.php**. En l'ouvrant, on peut voir plusieurs array, dont un qui est associé à mysql.

```
'mysql' => array(
    'driver'    => 'mysql',
    'host'      => 'localhost',
    'database'  => '',
    'username'  => '',
    'password'  => '',
    'charset'   => 'utf8',
    'collation' => 'utf8_unicode_ci',
    'prefix'    => '',
),
```

Relier la base de données

Pour 'database' => il faut mettre ensuite le nom de la base de données, soit ’portfolio’ dans mon cas.
Pour 'username' => notre nom d’utilisateur de la connexion à phpmyadmin, qui est ‘root’ pour moi.
Pour 'password' => le mot de passe.
Et pour le charset, il est par défaut en utf8.

Comme il y a plusieurs type de connexion à une base de données disponible, il ne faut pas oublier de mettre par défaut le type de connexion que l’on va utiliser, soit ‘mysql’ comme ci-dessous, toujours dans le même fichier :

Routage

Le routage dans Laravel vous permet d'acheminer toutes les demandes de votre application vers leur contrôleur approprié. Les routes principales et primaires de Laravel reconnaissent et acceptent un URI (Uniform Resource Identifier) accompagné d'une fermeture, étant donné qu'il doit s'agir d'un moyen simple et expressif de routage.

Routing

Le fichier route de l'application est défini dans le fichier **routes/web.php**. Le routage général dans Laravel pour chacune des requêtes possibles ressemble à ceci :**http://localhost/**

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

        return 'Welcome to index';

});
```

Routing

http://localhost/user/dashboard

```
Route:: post('user/dashboard', function () {
    return 'Welcome to dashboard';
});
```

http://localhost/user/add

```
Route:: put('user/add', function () {
    //
});
```

http://localhost/post/example

```
Route:: delete('post/example', function () {
    //
});
```

Le mécanisme de routage dans Laravel

Le mécanisme de routage se déroule en trois étapes différentes :

- Tout d'abord, vous devez créer et exécuter l'URL racine de votre projet.
- L'URL que vous exécutez doit correspondre exactement à la méthode définie dans le fichier **root.php**, et elle exécutera toutes les fonctions connexes.
- La fonction invoque les fichiers de modèle. Elle appelle ensuite la fonction view() avec le nom du fichier situé dans **resources/views/**, et élimine l'extension de fichier **blade.php** au moment de l'appel.

Le mécanisme de routage dans Laravel

Routes/web.php

```
<?php Route:: get ('/', function () { return view('abdelali'); });
```

resources/view/abdelali.blade.php

```
<!DOCTYPE html>
<html>
    <head>
        <title>ESMA_GI4</title>
    </head>
    <body>
        <h2>My name is Abdelali</h2>
        <p>Welcome to Laravel</p>
    </body>
</html>
```

Le mécanisme de routage dans Laravel

Laravel propose deux façons de capturer le paramètre passé :

- Paramètre requis
- Paramètre optionnel

Paramètres requis:

Les paramètres de route sont encapsulés dans des {} (accolades) avec des alphabets à l'intérieur. Prenons un exemple où vous devez capturer l'ID du client.

```
Route :: get (‘emp/{id}', function ($id) {
echo 'Emp '.$id;
});
```

Le mécanisme de routage dans Laravel

Paramètre optionnel

De nombreux paramètres ne restent pas présents dans l'URL, mais les développeurs ont dû les utiliser.
Ces paramètres sont donc indiqués par un " ?" (point d'interrogation) après le nom du paramètre.

Example:

```
Route :: get ('emp/{desig?}', function ($desig = null) {
echo $desig; });
 Route :: get ('emp/{name?}', function ($name = 'Guest') {
echo $name; });
```

Les contrôleurs

Les contrôleurs sont une autre fonctionnalité essentielle fournie par Laravel. Au lieu de définir la logique de traitement des demandes sous la forme de fermetures dans les fichiers de route, il est possible d'organiser ce processus à l'aide de classes de contrôleurs. Que font les contrôleurs ? Les contrôleurs sont destinés à regrouper la logique de traitement des demandes associées dans une seule classe. Dans votre projet Laravel, ils sont stockés dans le répertoire **app/Http/Controllers**. La forme complète de MVC est Model View Controller, qui dirige le trafic entre les vues et les modèles.

Création de contrôleurs

Ouvrez votre cmd ou terminal et tapez la commande :

php artisan make:controller Controller-Name> --plain

Remplacez ce <nom-du-contrôleur> dans la syntaxe ci-dessus par votre contrôleur. Cela va éventuellement faire un constructeur simple puisque vous passez l'argument --plain.

Création de contrôleurs

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

Route::get('base URI', 'controller@method') ;

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

Exemple

```php
<?php

namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Http\Requests;
use App\Http\Controllers\Controller;

class AdminController extends Controller
{
//
}
```

contrôleurs middlewares

Vous pouvez assigner des contrôleurs aux middlewares à router dans les fichiers de route de votre projet en utilisant la commande ci-dessous :

```
Route::get('profile', 'AdminController@show')->middleware('auth');
```

contrôleurs middlewares

Les méthodes middleware du contrôleur permettent d'affecter facilement un middleware à l'action et à l'activité du contrôleur. Des restrictions sur l'implémentation de certaines méthodes peuvent également être fournies aux middlewares sur la classe du contrôleur.

```
class AdminController extends Controller
{
    public function __construct()
    {
        // function body
    }
}
```

contrôleurs middlewares

En utilisant les closures, les contrôleurs peuvent permettre aux développeurs de Laravel d'enregistrer des middleware.

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

contrôleurs 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 PasswordController --resource

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

contrôleurs de ressources

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

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

Actions gérées par les contrôleurs de ressources

Verb	URI	Action	Route Name
GET	/users	Users list	users.index
POST	/users/add	Add a new user	users.add
GET	/users/{user}	Get user	users.show
GET	/users/{user}/edit	Edit user	users.edit
PUT	/users/{user}	Update user	users.update
DELETE	/users/{user}	Delete user	users.destroy

Contrôleurs implicites

Ces types de contrôleurs permettent aux développeurs de définir une seule route pour traiter plusieurs actions au sein du contrôleur. La syntaxe d'utilisation est la suivante :

Route::controller('base URI','<class-name-of-the-controller>');

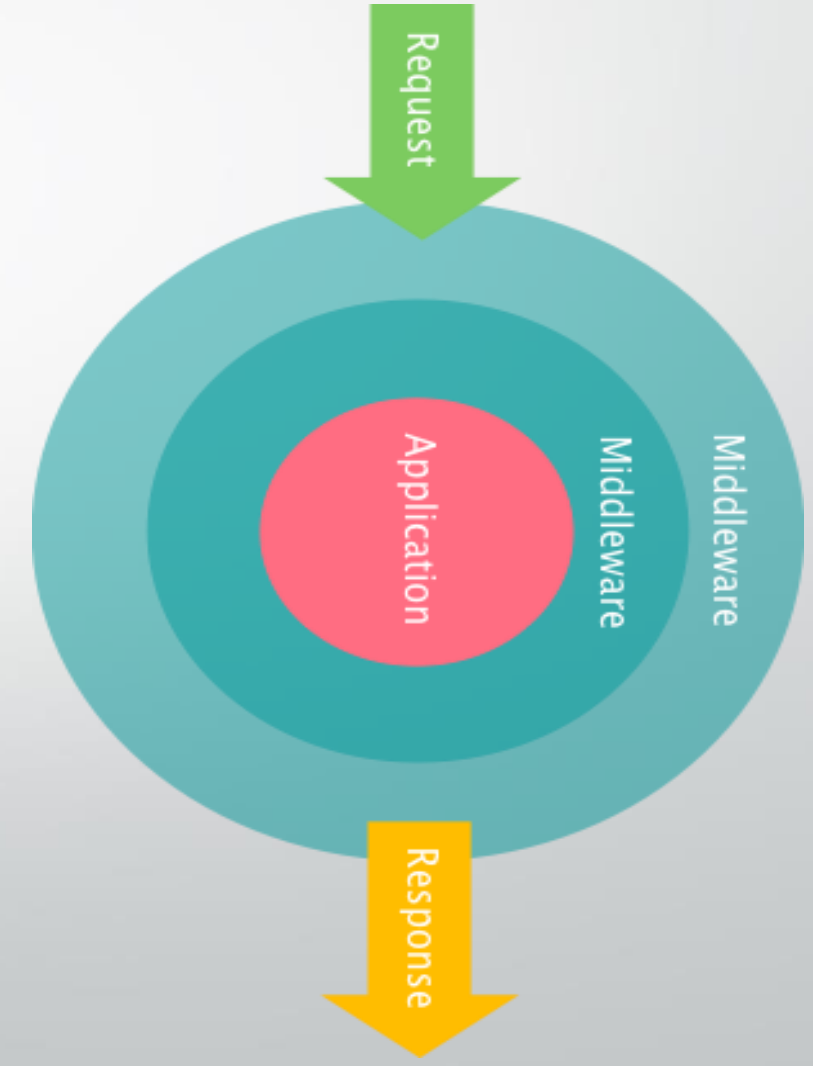
C'est là que sera stocké votre fichier de contrôleur implicite : **app/Http/Controllers/ImplicitController.php ;** et il ressemblera à un script comme celui-ci :

Exemple

```php
<?php

namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Http\Requests;
use App\Http\Controllers\Controller;
class ImplicitController extends Controller {
    public function getIndex(){
        echo 'starting method';
    }
    public function getVal($id){
        echo 'show value';
    }
    public function getAdminData(){
        echo 'admin data method';
    }
    public function adminPassword(){
        echo 'password method';
    }
}
```

Middleware

- Le **Middleware** est un mécanisme permettant de filtrer les requêtes HTTP.
- Laravel comprend plusieurs middlewares - Authentification, protection CSRF.
- Le middleware d'authentification vérifie si l'utilisateur qui visite la page est authentifié par un cookie de session.
- Le middleware de protection CSRF protège votre application contre les attaques de type "cross-site request forgery" en ajoutant une clé d'authentification pour chaque formulaire généré.



Middleware

- Un Middleware peut être défini comme un intermédiaire ou une interface agissant en coordination entre une requête et une réponse. Comme le mentionne le scénario de test ci-dessus, votre projet peut rediriger l'utilisateur de la page **login.php** vers la page **index.php** si l'utilisateur n'est pas authentifié.

- Vous pouvez créer votre Middleware en exécutant la syntaxe :

php artisan make:middleware Middleware_Name

Middleware

Ici, vous devez remplacer le <nom_du_middleware> par votre middleware. Vous pouvez voir cet emplacement de chemin **app/Http/Middleware**, le middleware que vous allez créer pour votre projet.

php artisan make:middleware CheckUser

L'enregistrement d'un middleware

Avant d'utiliser un Middleware, vous devez l'enregistrer.

Laravel fournit deux types Middleware. Ce sont :

- Middleware global
- Middleware de route

Les middlewares globaux sont ceux qui seront exécutés lors de chaque requête HTTP de votre application. Dans la propriété **$middleware** de votre classe **app/Http/Kernel.php**, vous pouvez lister tous les middlewares globaux de votre projet.

L'enregistrement d'un middleware

Lorsque vous voulez un middleware pour des routes spécifiques, vous devez ajouter le middleware avec une clé pour votre fichier **app/Http/Kernel.php**, qui est appelé route middleware. **$routeMiddleware**, par défaut, contient des entrées pour les middleware qui sont déjà incorporés dans Laravel. Pour ajouter vos middleware personnalisés, vous devez les ajouter à la liste et ajouter une clé de votre choix.

Exemple

```
rotected $routeMiddleware = [
        'auth' => \Illuminate\Auth\Middleware\Authenticate::class,
        'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
        'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
        'userAuth' => \Illuminate\Routing\Middleware\UserAuthRequests::class,
 ];
```

Paramètres du middleware

Les paramètres peuvent également être transmis à la middleware. Diverses situations paramétrées peuvent être lorsque votre projet a des attributs comme un client, un employé, un administrateur, un propriétaire, etc. Vous voulez exécuter différents modules en fonction des rôles de l'utilisateur ; pour ces situations, les paramètres des middlewares deviennent utiles.

Paramètres du middleware

Le middleware nouvellement créé peut être manipulé à l'aide du code : **app/Http/Middleware/ProfileMiddleware.php**

```php
<?php

namespace App\Http\Middleware;
use Closure;

class ProfileMiddleware {
   public function handle($request, Closure $next, $Profile) {
      echo "Role: ".$Profile;
      return $next($request);
   }
}
```

Middleware Terminable

Ces types particuliers de middleware commencent à fonctionner dès qu'une réponse est envoyée au navigateur. La méthode de terminaison est utilisée à cet effet. Lorsqu'une méthode de terminaison est utilisée dans le middleware de votre projet, elle est appelée automatiquement après l'envoi de la réponse du navigateur.

Exemple

```php
<?php

namespace Illuminate\Session\Middleware;
use Closure;
class SessionBegin
{
    public function handle($request, Closure $next)
    {
        return $next($request);
    }
    public function terminate($request, $response)
    {
        // tasks assigned within terminate method
    }
}
```

Middleware Terminable

Ces types particuliers de middleware commencent à fonctionner dès qu'une réponse est envoyée au navigateur. La méthode de terminaison est utilisée à cet effet. Lorsqu'une méthode de terminaison est utilisée dans le middleware de votre projet, elle est appelée automatiquement après l'envoi de la réponse du navigateur.

Vues

- Qu'est-ce qu'une vue de Laravel ?
- Créer une vue
- Rendre une vue
- Utiliser les layouts

Vues

Une vue est un fichier contenant un mélange de **code PHP**, de balises **HTML** et de modèles **Blade**.

Ces modèles contiennent des espaces réservés pour le contenu dynamique et sont utilisés pour définir la structure et la disposition d'une page web.

Lorsqu'un utilisateur demande l'application, la vue est rendue et renvoyée au navigateur de l'utilisateur.

Les vues sont stockées dans le répertoire **resources/views** du projet Laravel. Par défaut, Laravel est livré avec un ensemble de vues prédéfinies, telles que **welcome.blade.php** et **errors/404.blade.php**.

Création d'une vue

Pour créer une vue dans Laravel, suivez les étapes suivantes :

1. Naviguez jusqu'au répertoire resources/views de votre projet Laravel.

2. Créez un nouveau fichier avec une extension .blade.php. Cette extension indique à Laravel d'utiliser le moteur de template Blade pour analyser la vue.

3. Dans le fichier de vue, ajoutez les modèles **HTML**, **PHP** et/ou **Blade** qui définissent la structure et la disposition de la page. Vous pouvez utiliser des espaces réservés pour le contenu dynamique, tels que ou .

4. Enregistrez le fichier de vue.

Création d'une vue

Voici un exemple de vue simple qui affiche un message d'accueil :

```
<!-- resources/views/greeting.blade.php -->

<h1>Hello, {{$name}}!</h1>
```

Dans l'exemple ci-dessus, **$name** est un espace réservé pour un contenu dynamique. La valeur de **$name** sera affichée dans la vue lors de son rendu.

Rendre une vue

Les fonctions d'aide à l'affichage peuvent être utilisées pour afficher une vue dans une application Laravel. Cette fonction prend le nom de la vue comme premier argument et un tableau de données comme second argument.

Voici un exemple de rendu de la vue greeting :

```
// in a Laravel controller

return view('greeting', ['name' => 'Abdelali']);
```

Cela affichera la vue d'accueil avec la variable **$name** fixée à "Abdelali".

Rendre une vue

Vous pouvez également rendre une vue à partir d'un **route**.

```
// in routes/greeting.php

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

Le code ci-dessus affichera la vue d'accueil lorsque l'utilisateur visitera la route **/greeting**.

Utiliser les layouts

Dans Laravel, une mise en page est un modèle de lame qui définit la structure de base d'une page web.

Il se compose généralement de balises **HTML head** et **body** et peut inclure des liens vers des fichiers **CSS** et **JavaScript** et d'autres éléments communs tels que des en-têtes, des pieds de page et des menus de navigation.

Elle peut également contenir des espaces réservés pour le contenu dynamique, comme le titre de la page et le contenu principal.

Pour créer une présentation, créez un nouveau fichier de vue dans le répertoire **resources/views** et définissez la structure **HTML** et les espaces réservés.

Utiliser les layouts

Voici un exemple de modèle de mise en page utilisant la lame :

```
<!-- Stored in resources/views/layouts/app.blade.php -->

<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>
```

Cette mise en page définit une section de barre latérale (sidebar) et une section de contenu (content).

Utiliser les layouts

Pour utiliser cette mise en page dans une **vue**, la mise en page peut être étendue et les sections peuvent être définies comme suit :

```
<!-- Stored in resources/views/home.blade.php -->

@extends('layouts.app')

@section('title', 'Home')

@section('sidebar')
    @parent

    <p>This is appended to the master sidebar.</p>
@endsection

@section('content')
    <p>This is my body content.</p>
@endsection
```

Utiliser les layouts

La directive **@extends** indique à Laravel d'utiliser la présentation de l'application comme modèle parent.

Les directives **@section** définissent les sections qui seront rendues dans la présentation.

La directive **@parent** dans la section **sidebar** indique à Laravel d'ajouter le contenu à la section **sidebar** définie dans le modèle plutôt que de la remplacer.

Enfin, pour rendre cette vue, la fonction **view** helper du contrôleur peut être utilisée comme suit :

```
return view('home');
```

https://codepen.io/khadkamhn/pen/ZGvPLo

Utiliser les layouts

Le code ci-dessus rendra la vue d'home et injectera le contenu dans la mise en page, ce qui aura pour effet d'envoyer le code **HTML** suivant au navigateur :

```
<html>
    <head>
        <title>App Name - Home</title>
    </head>
    <body>
        This is the main sidebar.

        <p>This is appended to the main  sidebar.</p>

        <div class="container">
            <p>This is my body content.</p>
        </div>
    </body>
</html>
```

Créez une application **CRUD** avec **Laravel**, **Bootstrap** et **MySQL**

Tout au long de ce chapitre, vous apprendrez à utiliser ***Laravel 8*** la dernière version de l'un des frameworks PHP les plus populaires pour créer une application Web ***CRUD*** avec une base de données ***MySQL*** et des styles ***Bootstrap 5.23*** à partir de zéro et étape par étape en commençant par l'installation de ***Composer*** (gestionnaire de packages PHP) pour implémenter et servir votre application.

Qu'est-ce que **Bootstrap** ?

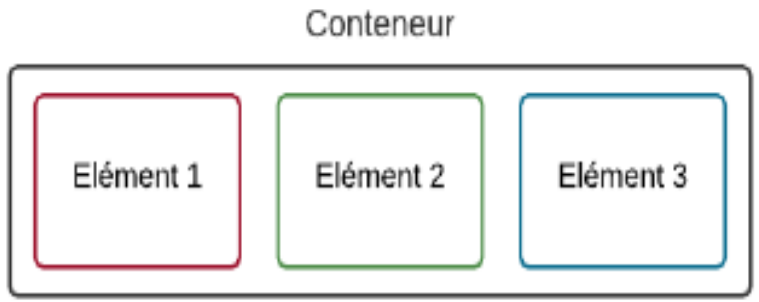
Bootstrap est une version de Twitter Bootstrap qui est un framework **CSS** qui permet aux développeurs web de styliser professionnellement leurs interfaces web sans être experts en CSS.

Les version de bootstrap est basé sur Flexbox et vous permet de créer des mises en page réactives avec des classes et des utilitaires simples.

Le principe de la mise en page avec Flexbox est simple : vous définissez un conteneur, et à l'intérieur vous placez plusieurs éléments. Imaginez un carton dans lequel vous rangez plusieurs objets : c'est le principe !
Sur une même page web, vous pouvez sans problème avoir plusieurs conteneur. Ce sera à vous d'en créer autant que nécessaire pour obtenir la mise en page que vous voulez.

Commençons par étudier le fonctionnement d'un conteneur.

Qu'est-ce que Bootstrap ?

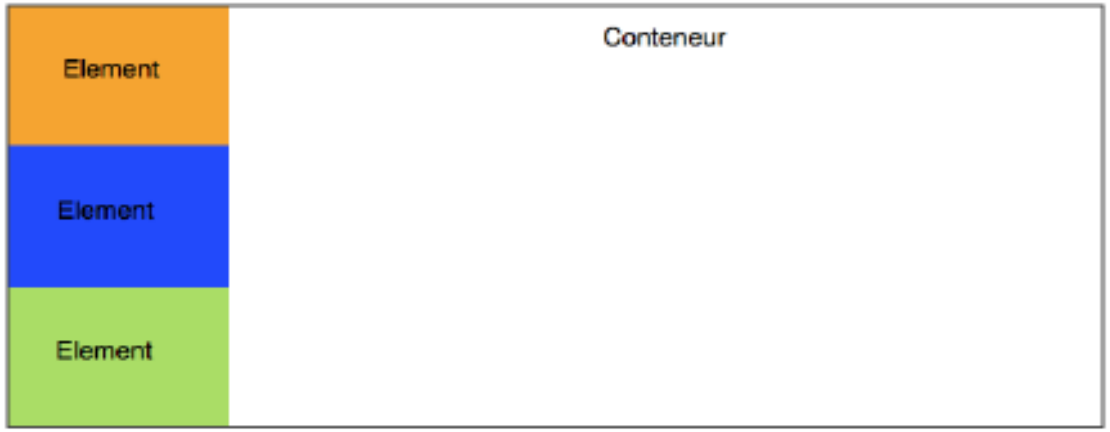


Le conteneur est une balise HTML, et les éléments sont d'autres balises HTML à l'intérieur :

```
<div id="conteneur">
   <div class="element 1">Element </div>
   <div class="element 1">Element </div>
   <div class="element 3">Element </div>
</div>
```

Par défaut, mes éléments vont se mettre les uns en-dessous des autres non ? Ce sont des blocs après tout. Si on mets une bordure au conteneur, une taille et une couleur de fond aux éléments, on va vite voir comment ils s'organisent :

Qu'est-ce que Bootstrap ?



Découvrons maintenant Flexbox. Si je mets une (une seule !) propriété CSS, tout change. Cette propriété, c'estflex, et je l'applique au conteneur :

```
#conteneur
{
   display: flex;
}
```

Alors les blocs se placent par défaut côte à côte :

Qu'est-ce que ***CRUD*** ?

CRUD signifie *Créer*, *Lire*, *mettre à jour* et *supprimer*, qui sont des opérations nécessaires dans la plupart des applications pilotées par les données qui accèdent aux données d'une base de données et les utilisent.

✓ Installation et création d'un projet Laravel 8

Dans cette section, nous allons présenter Laravel, puis procéder à l'installation et à la création d'un projet Laravel 8.
La génération d'un projet Laravel 8 est simple et directe. Dans votre terminal, on se place sur document root de notre serveur (xampp\htdocs) ; puis exécutez la commande suivante :

$ laravel new laravel-first-crud-app

On se place après sur le dossier du projet qu'on vient de créer :

$ cd laravel-first-crud-app

Qu'est-ce que ***CRUD*** ?

✓ Installation des dépendances frontales

Dans votre projet généré, vous pouvez voir qu'un fichier ***package.json*** est généré qui comprend de nombreuses bibliothèques frontales pouvant être utilisées par votre projet :

axios,
amorcer,
cross-env,
jquery,
mélange de laravel,
lodash,
popper.js,
resolver-url-loader,
toupet,
sass-loader,
vue

Qu'est-ce que ***CRUD*** ?

✓ Remarque :

Vous devez avoir installé ***Node.js*** et ***npm*** sur votre système avant de pouvoir installer les dépendances frontales.

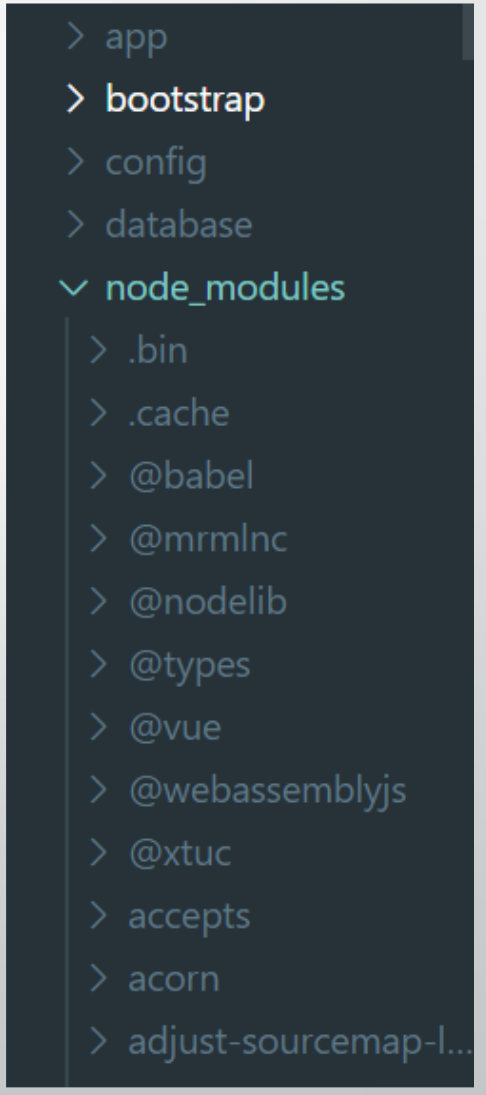
La dernière version de ***nodejs*** est disponible sur le lien suivant :

https://nodejs.org/en/download/current/ .

Vous devez ensuite utiliser npm pour installer les dépendances frontales :

$ npm install

Après avoir exécuté cette commande, un dossier node_modules sera créé et les dépendances y seront installées.



Création d'une base de données MySQL

Créons maintenant une base de données MySQL que nous utiliserons pour conserver les données dans notre application Laravel. Dans votre terminal, exécutez la commande suivante pour exécuter le client mysql :

$ mysql -u root -p

Lorsque vous y êtes invité, entrez le mot de passe de votre serveur MySQL lorsque vous l'avez installé.

Ensuite, exécutez l'instruction SQL suivante pour créer une base de données db :

mysql > create database db ;

ou utilisez ***PhpMyAdmin*** pour créer votre base de données.

Création d'une base de données MySQL

Ouvrez le fichier ***.env*** et mettez à jour les informations d'identification pour accéder à votre base de données **MySQL** :

DB_CONNECTION = mysql
DB_HOST = 127.0.0.1
DB_PORT = 3306
DB_DATABASE = db
DB_USERNAME = root
DB_PASSWORD =

Vous devez saisir le nom de la base de données, le nom d'utilisateur et le mot de passe.

À ce stade, vous pouvez exécuter la commande migrate, en utilisant la CLI artisan, pour créer la base de données ***db*** un tas de tables SQL nécessaires à Laravel :

$ php artisan migrate

Création d'une base de données MySQL

✓ Remarque :

Vous pouvez exécuter la commande ***migrate*** à tout autre moment de votre développement pour ajouter d'autres tables SQL dans votre base de données ou ultérieurement votre base de données si vous devez ajouter des modifications ultérieurement. En cas de besoin d'annuler la dernière migration, utilisez la ligne de commande :

$ php artisan migrate :rollback

Création de votre premier modèle Laravel

Laravel utilise le modèle architectural ***MVC*** pour organiser votre application en trois parties découplées :

- ***Le modèle*** qui encapsule la couche d'accès aux données,
- ***La vue*** qui encapsule la couche de représentation,
- ***Contrôleur*** qui encapsule le code pour contrôler l'application et communique avec les couches modèle et vue.

Création de votre premier modèle Laravel

✓ MVC :

Model – view – controller est un modèle architectural couramment utilisé pour développer des interfaces utilisateur qui divise une application en trois parties interconnectées. Cela permet de séparer les représentations internes des informations de la manière dont les informations sont présentées et acceptées par l'utilisateur.

Maintenant, créons notre premier modèle Laravel. Dans votre terminal, exécutez la commande suivante :

$ php artisan make:model Contact --migration

Ou simplement :

$ php artisan make:model Contact -m

Cela créera un modèle de contact et un fichier de migration. Dans le terminal, nous obtenons une sortie similaire à

Création de votre premier modèle Laravel

```
PS D:\xampp\htdocs\CRUD> php artisan make:model Contact -m
Model created successfully.
Created Migration: 2020_02_21_112214_create_contacts_table
PS D:\xampp\htdocs\CRUD>
```

Ouvrez le fichier de migration de la database/migrations/ !!!!!!!!!!_create_contacts_table et mettez-le à jour en conséquence :

Création de votre premier modèle Laravel

```php
public function up()
{
    Schema::create('contacts', function (Blueprint $table) {
        $table->bigIncrements('id');
        $table ->string ('first_name');
        $table ->string ('last_name');
        $table ->string ('email');
        $table ->string ('job_title');
        $table ->string ('city');
        $table ->string ('country') ;
        $table->timestamps();
    });
```

Nous avons ajouté les last_name , last_name , email , job_title , city et country dans le tableau des contacts.

Vous pouvez maintenant créer la table des contacts dans la base de données à l'aide de la commande suivante :

$ php artisan migrate

Création de votre premier modèle Laravel

```
PS D:\xampp\htdocs\CRUD> php artisan migrate
Migration table created successfully.
Migrating: 2014_10_12_000000_create_users_table
Migrated:  2014_10_12_000000_create_users_table (0.22 seconds)
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated:  2014_10_12_100000_create_password_resets_table (0.14 seconds)
Migrating: 2019_08_19_000000_create_failed_jobs_table
Migrated:  2019_08_19_000000_create_failed_jobs_table (0.14 seconds)
Migrating: 2020_02_21_112214_create_contacts_table
Migrated:  2020_02_21_112214_create_contacts_table (0.1 seconds)
PS D:\xampp\htdocs\CRUD>
```

Maintenant, regardons notre modèle de Contact , qui sera utilisé pour interagir avec la table de base de données de contacts . Ouvrez l' app/Contact.php et mettez-la à jour :

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Contact extends Model
{
    protected $fillable = ['first_name','last_name','email','city','country','job_title'];
}
```

Création du contrôleur et des itinéraires

Après avoir créé le modèle et migré notre base de données. Créons maintenant le contrôleur et les routes pour travailler avec le modèle Contact . Dans votre terminal, exécutez la commande suivante :

$ php artisan make:controller ContactController --resource

```
PS D:\xampp\htdocs\CRUD> php artisan make:controller ContactController --resource
Controller created successfully.
PS D:\xampp\htdocs\CRUD>
```

Le routage des ressources Laravel attribue les routes "CRUD" typiques à un contrôleur avec une seule ligne de code. Le contrôleur contiendra une méthode pour chacune des opérations de ressources disponibles.

Ouvrez le fichier app/Http/Controllers/ContactController.php . Voici le contenu initial :

Création du contrôleur et des itinéraires

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class ContactController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
```

Création du contrôleur et des itinéraires

```
        //
    }

    /**
     * Show the form for creating a new resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function create()
    {
        //
    }

    /**
     * Store a newly created resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        //
    }
```

Création du contrôleur et des itinéraires

```php
/**
 * Display the specified resource.
 *
 * @param  int  $id
 * @return \Illuminate\Http\Response
 */
public function show($id)
{
    //
}

/**
 * Show the form for editing the specified resource.
 *
 * @param  int  $id
 * @return \Illuminate\Http\Response
 */
public function edit($id)
{
    //
}

/**
 * Update the specified resource in storage.
 *
```

Création du contrôleur et des itinéraires

```
     * @param  \Illuminate\Http\Request  $request
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, $id)
    {
        //
    }

    /**
     * Remove the specified resource from storage.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function destroy($id)
    {
        //
    }
}
```

Création du contrôleur et des itinéraires

La classe ***ContactController*** étend la classe Controller disponible auprès de Laravel et définit un ensemble de méthodes qui seront utilisées pour effectuer les opérations CRUD sur le modèle Contact.
Vous pouvez lire le rôle de la méthode dans le commentaire au-dessus.
Nous devons maintenant fournir des implémentations pour ces méthodes.
Mais avant cela, ajoutons le routage. Ouvrez le fichier routes/web.php et mettez-le à jour en conséquence : <?php

Création du contrôleur et des itinéraires

```php
<?php

/*
|--------------------------------------------------------------------------
| Web Routes
|--------------------------------------------------------------------------
|
| Here is where you can register web routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| contains the "web" middleware group. Now create something great!
|
*/

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

Route::resource('contacts',ContactController::class);

Création du contrôleur et des itinéraires

À l'aide de la méthode statique ***resource()*** de Route, vous pouvez créer plusieurs itinéraires pour exposer plusieurs actions sur la ressource.
Ces itinéraires sont mappés à diverses méthodes ***ContactController*** qui devront être implémentées dans la section suivante :

$ php artisan route :list

```
PS D:\xampp\htdocs\CRUD> php artisan route:list
+--------+----------+----------+------+---------+--------------+
| Domain | Method   | URI      | Name | Action  | Middleware   |
+--------+----------+----------+------+---------+--------------+
|        | GET|HEAD | /        |      | Closure | web          |
|        | GET|HEAD | api/user |      | Closure | api,auth:api |
+--------+----------+----------+------+---------+--------------+
PS D:\xampp\htdocs\CRUD>
```

Après l'ajout de la route ::ressource en obtient la liste des routes suivantes :

```
PS D:\xampp\htdocs\CRUD> php artisan route:list
+--------+-----------+-------------------------+-----------------+-----------------------------------------------------------------+--------------+
| Domain | Method    | URI                     | Name            | Action                                                          | Middleware   |
+--------+-----------+-------------------------+-----------------+-----------------------------------------------------------------+--------------+
|        | GET|HEAD  | /                       |                 | Closure                                                         | web          |
|        | GET|HEAD  | api/user                |                 | Closure                                                         | api,auth:api |
|        | GET|HEAD  | contacts                | contacts.index  | App\Http\Controllers\ContactController@index                    | web          |
|        | POST      | contacts                | contacts.store  | App\Http\Controllers\ContactController@store                    | web          |
|        | GET|HEAD  | contacts/create         | contacts.create | App\Http\Controllers\ContactController@create                   | web          |
|        | GET|HEAD  | contacts/{contact}      | contacts.show   | App\Http\Controllers\ContactController@show                     | web          |
|        | PUT|PATCH | contacts/{contact}      | contacts.update | App\Http\Controllers\ContactController@update                   | web          |
|        | DELETE    | contacts/{contact}      | contacts.destroy| App\Http\Controllers\ContactController@destroy                  | web          |
|        | GET|HEAD  | contacts/{contact}/edit | contacts.edit   | App\Http\Controllers\ContactController@edit                     | web          |
+--------+-----------+-------------------------+-----------------+-----------------------------------------------------------------+--------------+
PS D:\xampp\htdocs\CRUD>
```

Création du contrôleur et des itinéraires

GET /contacts , mappé à la méthode index() ,
GET /contacts/create , mappé à la méthode create() ,
POST /contacts , mappé à la méthode store() ,
GET /contacts/{contact} , mappé à la méthode show() ,
GET /contacts/{contact}/edit , mappé à la méthode edit() ,
PUT / PATCH /contacts/{contact} , mappé à la méthode update() ,
DELETE /contacts/{contact} , mappé à la méthode destroy() .

Ces routes sont utilisées pour servir des modèles HTML et également comme points de terminaison API pour travailler avec le modèle Contact.
Remarque : Si vous souhaitez créer un contrôleur qui exposera uniquement une API RESTful, vous pouvez utiliser la méthode apiResource pour exclure les routes utilisées pour servir les modèles HTML :

$Route::apiResource('contacts', 'ContactController');

Mise en œuvre des opérations **CRUD**

✓ C: Implémentation de l'opération de création et ajout d'un formulaire

Implémentons maintenant les méthodes du contrôleur aux côtés des vues.

C: Implémentation de l'opération de création et ajout d'un formulaire

ContactController inclut la méthode store() qui mappe au point de terminaison de l'API POST /contacts qui sera utilisé pour créer un contact dans la base de données et le create() qui mappe à la route GET /contacts/create qui sera utilisé pour servir le Formulaire HTML utilisé pour soumettre le contact au point de terminaison API POST /contacts.

Implémentons ces deux méthodes.

Ouvrez de nouveau le fichier app/Http/Controllers/ContactController.php et commencez par importer le modèle Contact :

```
use App\Contact;
```

Ensuite, localisez la méthode store() et mettez-la à jour en conséquence :

```
    /**
     * Store a newly created resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
```

Mise en œuvre des opérations **CRUD**

```
    */
    public function store(Request $request)
    {
        $request->validate([
            'first_name'=>'required',
            'last_name'=>'required',
            'email'=>'required'
        ]);

        $contact = new Contact([
            'first_name' => $request->get('first_name'),
            'last_name' => $request->get('last_name'),
            'email' => $request->get('email'),
            'job_title' => $request->get('job_title'),
            'city' => $request->get('city'),
            'country' => $request->get('country')
        ]);
        $contact->save();
        return redirect('/contacts')->with('success', 'Contact saved!');
    }
```

Mise en œuvre des opérations **CRUD**

Ensuite, localisez la méthode create() et mettez-la à jour :

```php
/**
 * Show the form for creating a new resource.
 *
 * @return \Illuminate\Http\Response
 */
public function create()
{
    return view('contacts.create');
}
```

La fonction create() utilise la méthode view() pour renvoyer le modèle *create.blade.php* qui doit être présent dans le dossier *resources/views*.

Avant de créer le modèle *create.blade.php* , nous devons créer un modèle de base qui sera étendu par le modèle create et tous les autres modèles qui seront créés plus tard dans ce didacticiel.

Mise en œuvre des opérations **CRUD**

Dans le dossier resources/views , créez un fichier *base.blade.php* et ajoutez le modèle de layout suivant :

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Laravel [redacted] & MySQL CRUD Tutorial</title>
  <link href="{{ asset('css/app.css') }}" rel="stylesheet" type="text/css" />
</head>
<body>
  <div class="container">
    @yield('main')
  </div>
  <script src="{{ asset('js/app.js') }}" type="text/js"></script>
```

```
</body>
</html>
```

Maintenant, créons le modèle *create.blade.php*. Créez d'abord un dossier de contacts dans le dossier des vues :

Ouvrez le fichier resources/views/contacts/create.blade.php et ajoutez le code suivant :

Mise en œuvre des opérations **CRUD**

```
@extends('base')

@section('main')
<div class="row">
 <div class="col-sm-8 offset-sm-2">
    <h1 class="display-3">Add a contact</h1>
  <div>
    @if ($errors->any())
      <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
              <li>{{ $error }}</li>
            @endforeach
        </ul>
      </div><br />
    @endif
      <form method="post" action="{{ route('contacts.store') }}">
          @csrf
          <div class="form-group">
              <label for="first_name">First Name:</label>
              <input type="text" class="form-control" name="first_name"/>
          </div>
```

Mise en œuvre des opérations **CRUD**

```
<div class="form-group">
    <label for="last_name">Last Name:</label>
    <input type="text" class="form-control" name="last_name"/>
</div>

<div class="form-group">
    <label for="email">Email:</label>
    <input type="text" class="form-control" name="email"/>
</div>
<div class="form-group">
    <label for="city">City:</label>
    <input type="text" class="form-control" name="city"/>
</div>
<div class="form-group">
    <label for="country">Country:</label>
    <input type="text" class="form-control" name="country"/>
</div>
<div class="form-group">
    <label for="job_title">Job Title:</label>
    <input type="text" class="form-control" name="job_title"/>
</div>
```

Mise en œuvre des opérations **CRUD**

```
          <button type="submit" class="btn btn-primary-
outline">Add contact</button>
      </form>
  </div>
</div>
</div>
@endsection
```

Ceci est une capture d'écran de notre formulaire de création!

Mise en œuvre des opérations **CRUD**

Configurer Bootstrap

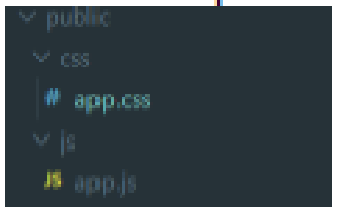
Avant on doit importer la structure des css et js dans le dossier public de votre projet. Pour cela nous allons exécuter la commande suivante :

$ npm run dev

```
PS D:\xampp\htdocs\CRUD> npm run dev

> @ dev D:\xampp\htdocs\CRUD
> npm run development


> @ development D:\xampp\htdocs\CRUD
> cross-env NODE_ENV=development node_modules/webpack/bin/webpack.js --progress --hide-modules --config=node_modules/laravel-mix/setup/webpack.config.js

        Additional dependencies must be installed. This will only take a moment.

        Running: npm install vue-template-compiler --save-dev --production=false

npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.11 (node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.11: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch"
:"x64"})

        Okay, done. The following packages have been installed and saved to your package.json dependencies list:

        - vue-template-compiler

98% after emitting SizeLimitsPlugin

 DONE  Compiled successfully in 2546ms                                                                                                    22:09:31

      Asset       Size   Chunks             Chunk Names
/css/app.css  0 bytes  /js/app  [emitted]  /js/app
  /js/app.js  592 KiB  /js/app  [emitted]  /js/app
PS D:\xampp\htdocs\CRUD>
```

Configurer Bootstrap

On remarque la création de deux dossiers css et js :



En ce moment, il y a des problèmes, on ne voit aucun code dans le fichier public >> css >> app.css qu'est toujours vide.

L'échafaudage Bootstrap et Vue fourni par Laravel est situé dans le laravel / ui Composer, que vous pouvez installer à l'aide de Composer via la commande suivante :

```
// Generate basic scaffolding...
php artisan ui vue
php artisan ui react
// Generate login / registration scaffolding...
php artisan ui vue --auth
php artisan ui react --auth
```

La commande :

$ npm install

permet d'installer les dépendances frontales et d'importer les module nodejs dans votre projet :

Configurer Bootstrap

```
PS D:\xampp\htdocs\CRUD> npm install
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.11 (node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.11: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch"
:"x64"})

added 1856 packages from 488 contributors and audited 17272 packages in 36.148s

32 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

PS D:\xampp\htdocs\CRUD>
```

Maintenant nous allons exécuter la commande suivante :

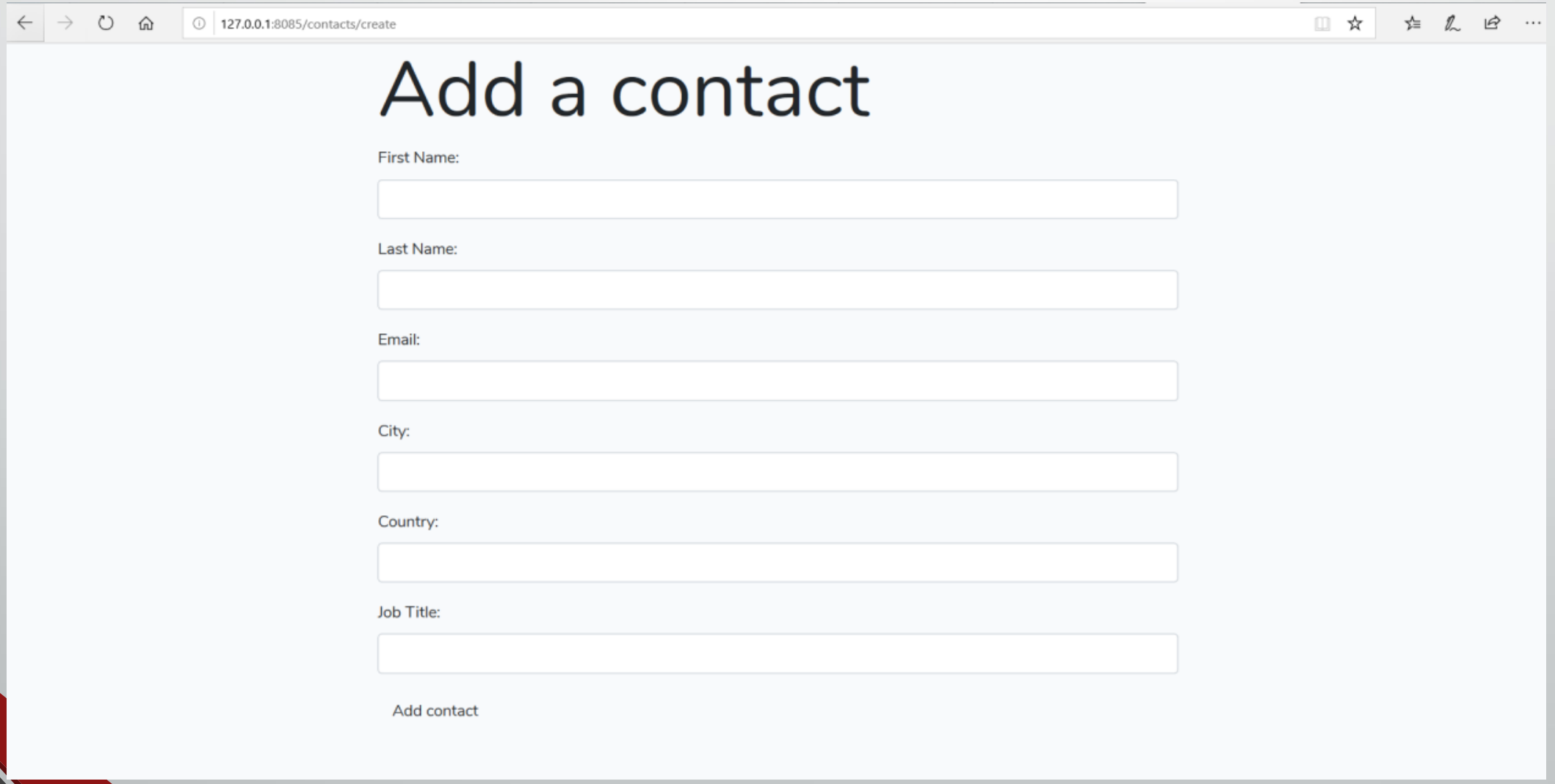
$ composer laravel/ui --dev

Configurer Bootstrap

```
PS D:\xampp\htdocs\CRUD> composer require laravel/ui --dev
Using version ^1.2 for laravel/ui
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 1 install, 0 updates, 0 removals
  - Installing laravel/ui (v1.2.0): Loading from cache
Writing lock file
Generating optimized autoload files
> Illuminate\Foundation\ComposerScripts::postAutoloadDump
> @php artisan package:discover --ansi
Discovered Package: facade/ignition
Discovered Package: fideloper/proxy
Discovered Package: laravel/tinker
Discovered Package: laravel/ui
Discovered Package: nesbot/carbon
Discovered Package: nunomaduro/collision
Package manifest generated successfully.
PS D:\xampp\htdocs\CRUD> npm run dev
```

Configurer Bootstrap

On obtient l'affichage suivant :



Configurer Bootstrap

Remplissez le formulaire et cliquez sur le bouton Ajouter un contact pour créer un contact dans la base de données. Vous devez être redirigé vers la route /contacts qui n'a pas encore de vue associée.

Mise en œuvre des opérations **CRUD**

✓ R: Implémentation de l'opération de lecture et obtention des données

Ensuite, implémentons l'opération de lecture pour obtenir et afficher les données de contacts de notre base de données MySQL.
Accédez au fichier app/Http/Controllers/ContactController.php , localisez la méthode index() et mettez-la à jour :

```php
/**
 * Display a listing of the resource.
 *
 * @return \Illuminate\Http\Response
 */
public function index()
{
    $contacts = Contact::all();

    return view('contacts.index', compact('contacts'));
}
```

Mise en œuvre des opérations **CRUD**

- ✓ R: Implémentation de l'opération de lecture et obtention des données

Ensuite, vous devez créer le modèle d'index.
Créez un fichier resources/views/contacts/index.blade.php :

```
@extends('base')


@section('main')
<div class="row">
<div class="col-sm-12">
    <h1 class="display-3">Contacts</h1>
  <table class="table table-striped">
    <thead>
        <tr>
          <td>ID</td>
          <td>Name</td>
          <td>Email</td>
          <td>Job Title</td>
```

Mise en œuvre des opérations **CRUD**

- ✓ R: Implémentation de l'opération de lecture et obtention des données

Ensuite, vous devez créer le modèle d'index.
Créez un fichier resources/views/contacts/index.blade.php :

```
@extends('base')


@section('main')
<div class="row">
<div class="col-sm-12">
    <h1 class="display-3">Contacts</h1>
  <table class="table table-striped">
    <thead>
        <tr>
          <td>ID</td>
          <td>Name</td>
          <td>Email</td>
          <td>Job Title</td>
```

Mise en œuvre des opérations **CRUD**

- ✓ R: Implémentation de l'opération de lecture et obtention des données

```
            <td>City</td>
            <td>Country</td>
            <td colspan = 2>Actions</td>
        </tr>
    </thead>
    <tbody>
        @foreach($contacts as $contact)
        <tr>
            <td>{{$contact->id}}</td>
            <td>{{$contact->first_name}} {{$contact->last_name}}</td>
            <td>{{$contact->email}}</td>
            <td>{{$contact->job_title}}</td>
            <td>{{$contact->city}}</td>
            <td>{{$contact->country}}</td>
            <td>
                <a href="{{ route('contacts.edit',$contact->id)}}" class="btn btn-primary">Edit</a>
            </td>
            <td>
```

Mise en œuvre des opérations **CRUD**

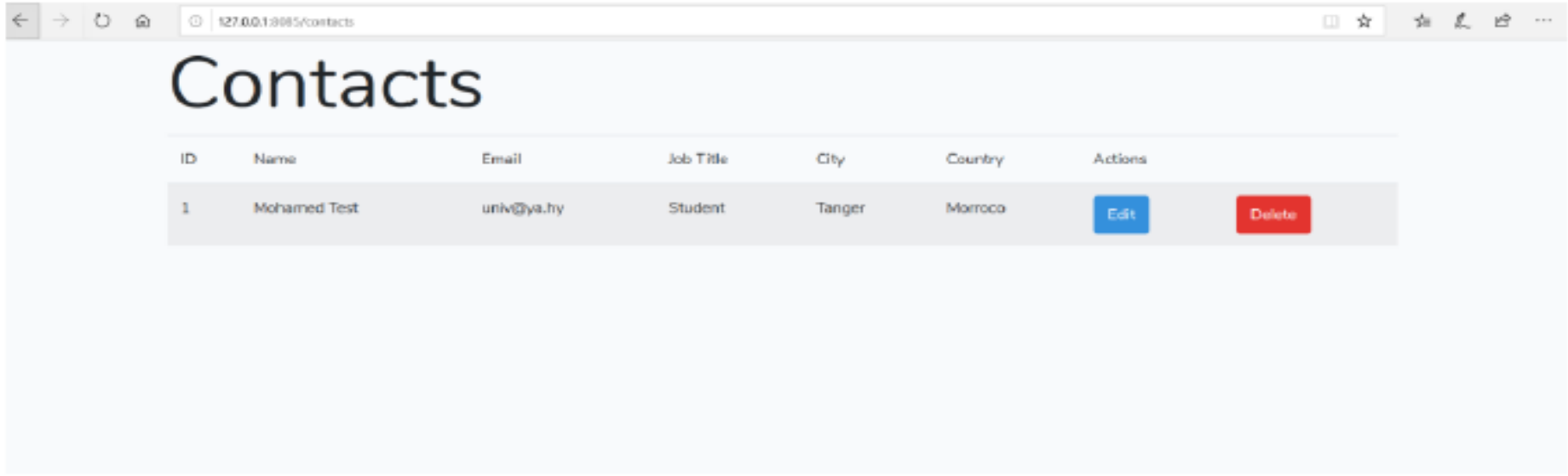
- ✓ R: Implémentation de l'opération de lecture et obtention des données

```
                <form action="{{ route('contacts.destroy', $contact->id)}}" method="post">
                  @csrf
                  @method('DELETE')
                  <button class="btn btn-danger" type="submit">Delete</button>
                </form>
            </td>
        </tr>
        @endforeach
    </tbody>
  </table>
<div>
</div>
@endsection
```

Mise en œuvre des opérations **CRUD**

✓ R: Implémentation de l'opération de lecture et obtention des données

Le résultat obtenu est comme suit :



U: implémentation de l'opération de mise à jour

Ensuite, nous devons implémenter l'opération de mise à jour. Accédez au fichier app/Http/Controllers/ContactController.php , localisez la méthode edit($id) et mettez-la à jour :

```
    /**
     * Show the form for editing the specified resource.
```

Mise en œuvre des opérations **CRUD**

- ✓ R: Implémentation de l'opération de lecture et obtention des données

```
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function edit($id)
    {
        $contact = Contact::find($id);
        return view('contacts.edit', compact('contact'));
    }
```

Ensuite, vous devez implémenter la méthode update() :

Mise en œuvre des opérations **CRUD**

- ✓ R: Implémentation de l'opération de lecture et obtention des données

```php
/**
 * Update the specified resource in storage.
 *
 * @param  \Illuminate\Http\Request  $request
 * @param  int  $id
 * @return \Illuminate\Http\Response
 */
public function update(Request $request, $id)
{
    $request->validate([
        'first_name'=>'required',
        'last_name'=>'required',
        'email'=>'required'
    ]);

    $contact = Contact::find($id);
    $contact->first_name =  $request->get('first_name');
    $contact->last_name = $request->get('last_name');
    $contact->email = $request->get('email');
    $contact->job_title = $request->get('job_title');
    $contact->city = $request->get('city');
    $contact->country = $request->get('country');
    $contact->save();

    return redirect('/contacts')->with('success', 'Contact updated!');

}
```

Mise en œuvre des opérations **CRUD**

- ✓ R: Implémentation de l'opération de lecture et obtention des données

Maintenant, vous devez ajouter le modèle d'édition. Dans les resources/views/contacts/ , créez un fichier edit.blade.php :

```
@extends('base')

@section('main')

<div class="row">

    <div class="col-sm-8 offset-sm-2">

        <h1 class="display-3">Update a contact</h1>


        @if ($errors->any())

        <div class="alert alert-danger">

            <ul>

                @foreach ($errors->all() as $error)
```

Mise en œuvre des opérations **CRUD**

- ✓ R: Implémentation de l'opération de lecture et obtention des données

```
                <li>{{ $error }}</li>
                @endforeach
            </ul>
        </div>
        <br />
        @endif
        <form method="post" action="{{ route('contacts.update', $contact->id) }}">
            @method('PATCH')
            @csrf
            <div class="form-group">

                <label for="first_name">First Name:</label>
                <input type="text" class="form-control" name="first_name" value={{ $contact-
>first_name }} />
            </div>

            <div class="form-group">
                <label for="last_name">Last Name:</label>
                <input type="text" class="form-control" name="last_name" value={{ $contact-
>last_name }} />
            </div>
```

Mise en œuvre des opérations **CRUD**

- ✓ R: Implémentation de l'opération de lecture et obtention des données

```
            <div class="form-group">
                <label for="email">Email:</label>
                <input type="text" class="form-control" name="email" value={{ $contact->email }} />
            </div>
            <div class="form-group">
                <label for="city">City:</label>
                <input type="text" class="form-control" name="city" value={{ $contact->city }} />
            </div>
            <div class="form-group">
                <label for="country">Country:</label>
                <input type="text" class="form-control" name="country" value={{ $contact->country }} />
            </div>
            <div class="form-group">
                <label for="job_title">Job Title:</label>
                <input type="text" class="form-control" name="job_title" value={{ $contact-
>job_title }} />
            </div>
            <button type="submit" class="btn btn-primary">Update</button>
        </form>
    </div>
</div>
@endsection
```

Mise en œuvre des opérations **CRUD**

✓ U: implémentation de l'opération de suppression

Enfin, nous allons procéder à l'implémentation de l'opération de suppression. Accédez au fichier app/Http/Controllers/ContactController.php , localisez la méthode destroy() et mettez-la à jour en conséquence :

```
    * Remove the specified resource from storage.
    *
    * @param  int  $id
    * @return \Illuminate\Http\Response
    */
   public function destroy($id)
   {
       $contact = Contact::find($id);
       $contact->delete();

       return redirect('/contacts')->with('success', 'Contact deleted!');
   }
```

Vous pouvez remarquer que lorsque nous redirigeons vers la route /contacts dans nos méthodes API CRUD, nous transmettons également un message de réussite, mais il n'apparaît pas dans notre modèle d' index . Changeons ça!

Mise en œuvre des opérations **CRUD**

✓ U: implémentation de l'opération de suppression

Accédez au fichier resources/views/contacts/index.blade.php et ajoutez le code suivant :

```
<div class="col-sm-12">
@if(session()->get('success'))
    <div class="alert alert-success">
      {{ session()->get('success') }}
    </div>
@endif
```

Nous devons également ajouter un bouton pour nous amener au formulaire de création. Ajoutez ce code sous l'en-tête :

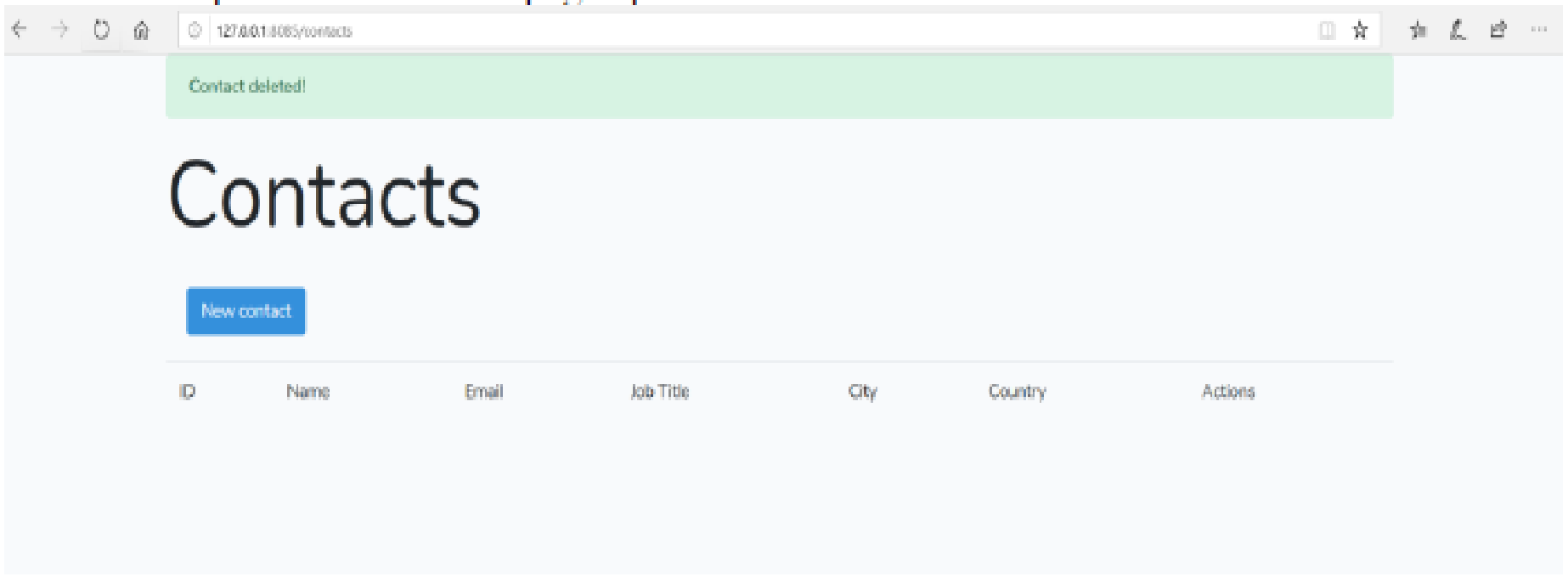
```
    <div>
        <a style="margin: 19px;" href="{{ route('contacts.create')}}" class="b
tn btn-primary">New contact</a>
    </div>
```

Voici une capture d'écran de la page après avoir créé un contact :

Mise en œuvre des opérations **CRUD**

- ✓ U: implémentation de l'opération de suppression

Voici une capture d'écran de la page après avoir créé un contact :



Conclusion

Nous avons atteint la fin de ce chapitre. Nous avons créé une application **CRUD** avec Laravel 8, PHP 8 et MySQL.