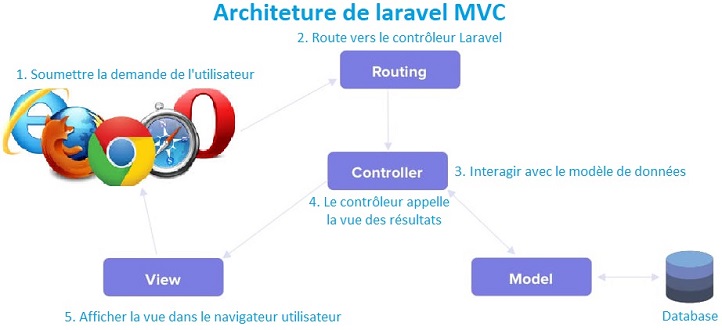
## Laravel :

Est un framework PHP open source pour le développement d'applications web, basé sur le schéma architectural MVC (Model-View-Controller). Sa syntaxe expressive et simple accélère le développement et permet d'obtenir une base de code propre et maintenable ainsi que des applications évolutives.



## Installation d’un projet laravel:

***Via composer:***

*C:\xampp\htdocs****Via laravel Installer:***

*composer global require laravel/installer*

*laravel new myapp*

**Artisan :**

Lorsqu’on construit une application avec Laravel on a de nombreuses tâches à accomplir, comme par exemple créer des classes, vérifier les routes…

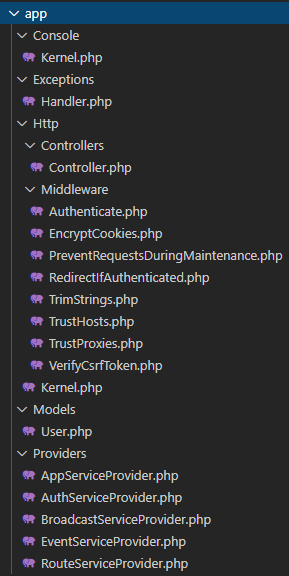
C’est là qu’intervient Artisan, le compagnon indispensable. Il fonctionne en ligne de commande, donc à partir de la console. Il suffit de se positionner dans le dossier racine et d’utiliser la commande : *php artisan*

Laravel est équipé d’un serveur sommaire pour le développement qui se lance avec cette commande : *php artisan serve*

On y accède à cette adresse : [http://localhost:8000](http://localhost:8000/).

## Organisation de Laravel

### **Dossier app**

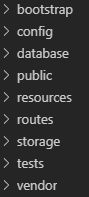


Ce dossier contient les éléments essentiels de l’application

* **Console** : toutes les commandes en mode console,
* **Exceptions** : pour gérer les erreurs d’exécution,
* **Http** : tout ce qui concerne la communication : contrôleurs, middlewares (il y a 7 middlewares de base qui servent à filtrer les requêtes HTTP) et le kernel,
* **Providers** : tous les fournisseurs de services (providers), il y en a déjà 5 au départ. Les providers servent à initialiser les composants.
* **Models**: le dossier des modèles avec déjà un présent qui concerne les utilisateurs.

Évidemment tout cela doit vous paraître assez nébuleux pour le moment mais nous verrons en détail ces éléments au fil du cours. Et on verra d’ailleurs que seront créés bien d’autres dossiers selon nos besoins.

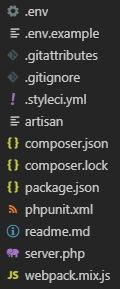
### **Autres dossiers**

[](https://laravel.sillo.org/wp-content/uploads/2019/08/Capture-23.png)

Voici une description du contenu des autres dossiers :

* **bootstrap** : scripts d’initialisation de Laravel pour le chargement automatique des classes, la fixation de l’environnement et des chemins, et pour le démarrage de l’application,
* **public** : tout ce qui doit apparaître dans le dossier public du site : images, CSS, scripts…
* **config** : toutes les configurations : application, authentification, cache, base de données, espaces de noms, emails, systèmes de fichier, session…
* **database** : migrations et populations,
* **resources** : vues, fichiers de langage et assets (par exemple les fichiers Sass),
* **routes** : la gestion des urls d’entrée de l’application,
* **storage** : données temporaires de l’application : vues compilées, caches, clés de session…
* **tests** : fichiers de tests unitaires,
* **vendor** : tous les composants de Laravel et de ses dépendances (créé par composer).

## Fichiers de la racine

[](https://laravel.sillo.org/wp-content/uploads/2019/08/Capture-24.png)

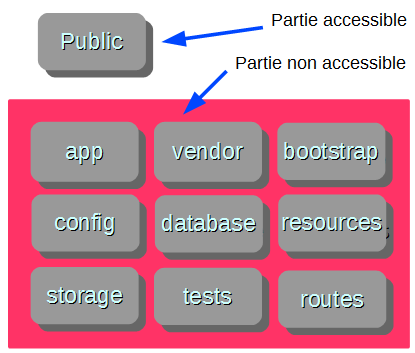
Il y a un certain nombre de fichiers dans la racine dont voici les principaux :

* **artisan** : outil en ligne de Laravel pour des tâches de gestion,
* **composer.json** : fichier de référence de composer,
* **package.json** : fichier de référence de npm pour les assets,
* **phpunit.xml** : fichier de configuration de phpunit (pour les tests unitaires),
* **.env** : fichier pour spécifier l’environnement d’exécution.

Nous verrons tout cela progressivement dans le cours, ne vous inquiétez pas !

## Accessibilité

Pour des raisons de sécurité sur le serveur seul le dossier **public**doit être accessible.



## En résumé

* Pour son installation et sa mise à jour Laravel utilise le gestionnaire de dépendances **composer**.
* La création d’une application Laravel se fait à partir de la console avec une simple ligne de commande.
* Laravel est organisé en plusieurs dossiers.
* Le dossier **public**est le seul qui doit être accessible pour le client.
* L’environnement est fixé à l’aide du fichier **.env**.

**Étape 2: configurer la base de données MySQL**

Nous utiliserons une base de données **MySQL** pour créer une base de données et revenir au projet. utiliser Xampp,Wamp, Mamp ou tout autre server Local pour cette operation.

Laravel fournit le  fichier **.env** pour ajouter des informations d'identification. Ouvrez le fichier et modifiez le code suivant.

DB\_CONNECTION=mysql  
DB\_HOST=127.0.0.1  
DB\_PORT=3306  
DB\_DATABASE=laravel8cars  
DB\_USERNAME=root  
DB\_PASSWORD=root

Le nom d'utilisateur et le mot de passe seront différents pour les vôtres en fonction des informations d'identification de votre base de données.

Laravel est livré avec certaines migrations par défaut comme **users**, **password\_resets** et la table **create\_failed\_jobs**. Accédez maintenant au terminal et tapez la commande suivante pour exécuter vos migrations.

 php artisan migrate

Vous pouvez voir dans votre base de données que ces tables sont créées.

**Étape 3: créer un modèle et une migration personnalisée**

Nous allons créer un projet autour des **marques des voitures**. Ainsi, les utilisateurs peuvent créer des voitures, éditer et les supprimer. Alors, créons un modèle de voiture et une migration.

 php artisan make:model Car -m

Cela créera deux fichiers :

Modèle :  **Car.php**

Migration : **create\_cars\_table.php**

Ajoutez les nouveaux champs dans le  fichier de migration ***create\_cars\_table.php***.

// -> create\_cars\_table.php  
  
public function up()  
  
{  
       Schema::create('cars', function (Blueprint $table) {  
           $table->id();  
           $table->string('marque');  
           $table->integer('prix');  
           $table->timestamps();  
       });  
}

Les champs ***id*** et ***timestamps*** sont créés par défaut par Laravel. La ***marque*** et le ***prix*** sont nos champs personnalisés que l'utilisateur peut ajouter via les formulaires Web.

Vous pouvez exécuter la migration pour créer la table dans la base de données.

 php artisan migrate

**Étape 4: Créez un contrôleur Laravel 8.**

Le routage des ressources Laravel attribue les routes « **CRUD** » typiques à un contrôleur avec une seule ligne de code. Puisque notre application a des opérations crud de base, nous utiliserons le contrôleur de ressources pour ce petit projet.

 php artisan make:controller CarController --resource

Dans une nouvelle installation de Laravel 8, aucun préfixe d'espace de noms n'est appliqué à vos groupes de routes dans lesquels vos routes sont chargées.

Ce que vous pouvez faire ici, c'est ouvrir le fichier ***App\Providers\RouteServiceProvider.php*** et ajouter le code suivant dans la méthode boot().

// RouterServiceProvider.php  
  
Route::middleware('web')  
               ->namespace('App\Http\Controllers')  
                ->group(base\_path('routes/web.php'));

Voilà !. Maintenant, il peut trouver le contrôleur. Si vos fichiers de contrôleur sont ailleurs, vous devez affecter le chemin dans l'espace de noms.

Notez ici que j'ai ajouté le flag ***–resource***, qui définira six méthodes à l'intérieur du **CarController**, à savoir:

**Index**: (La méthode index() est utilisée pour afficher une liste de voitures).

**Create**: (La méthode create() affichera le formulaire ou la vue pour créer une voiture).

**Store**: (La méthode store() est utilisée pour insérer une voiture dans la base de données. Remarque: la méthode create submit les données du formulaire à la méthode store()).

**Show**: (La méthode show() affichera une voiture spécifié).

**Edit**: (La méthode edit() affichera le formulaire pour éditer une voiture. Le formulaire sera rempli avec les données de la voiture existante).

**Update**: (La méthode update() est utilisée pour mettre à jour une voiture dans la base de données. Remarque: la méthode edit() submit les données du formulaire à la méthode update()).

**Destroy**: (La méthode destroy() est utilisée pour supprimer la voiture specifique spécifié).

Par défaut, le  fichier **CarController.php**  est créé dans le  dossier des contrôleurs **app > Http >**

<?php  
  
// CarController.php  
  
  
namespace App\Http\Controllers;  
  
use Illuminate\Http\Request;  
  
  
class CarsController extends Controller  
{  
   /\*\*  
    \* Display a listing of the resource.  
    \*  
    \* @return \Illuminate\Http\Response  
     \*/  
  
   public function index()  
   {  
       //  
   }  
  
   /\*\*  
    \* 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)  
   {  
       //  
    }  
  
  
   /\*\*  
    \* 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.  
    \*  
    \* @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)  
   {  
       //  
   }  
}

Vous pouvez voir que le fichier contient des opérations CRUD sous la forme de différentes fonctions. Nous utiliserons ces fonctions, une par une, pour créer des opérations specifiques.

Le flag ***–resource*** appellera la méthode internal *resource()* par Laravel pour générer les routes suivantes.

Vous pouvez consulter la liste des routes à l'aide de la commande suivante.

php artisan route: list

**Étape 5: définir les routes**

Pour définir une route dans Laravel, vous devez ajouter le code de la route dans le  fichier **routes > web.php .**

// web.php  
  
Route::resource('cars', 'CarController');

**Étape 6: Configurer Bootstrap dans Laravel 8**

Laravel fournit le framework **Bootstrap** et **Vue** qui se trouve dans le package **laravel/ui**, qui peut être installé à l'aide de Composer.

composer require laravel/ui

Une fois le  package *laravel/ui* installé, vous pouvez installer Bootstrap à l'aide de la commande Artisan suivante.

php artisan ui bootstrap

Maintenant, exécutez "**npm install**" puis "**npm run dev**" pour compiler votre nouvelle structure.

Si vous n'avez pas installer NPM, suivez ce Tutoriel :[**Installation de NodeJS**](https://www.letecode.com/tutoriel-nodejs-installation-de-nodejs-sur-windows-et-linux)

**Étape 7: créer des vues dans Laravel 8**

**Blade**: est le moteur de template utilisé par Laravel. Son but est de permettre d’utiliser du php sur notre vue mais d'une manière assez particulière. Pour créer un fichier qui utilise le moteur de template Blade il vous faut ajouter l'extension ".blade.php". Comme nous l'avons vu dans la présentation de l'architecture de Laravel, les fichiers de vos vues se situent dans le dossier resources/views.

La première fonctionnalité la plus basique de Blade est l'affichage d'une simple variable comme nous l’avons vu dans le chapitre précédent.

Les vues contiennent le code HTML servi par votre application et séparent la logique de votre contrôleur/application de votre logique de présentation. Les vues sont stockées dans le répertoire *resources/views*.

Dans le répertoire  views, nous devons également créer un fichier de mise en page. Nous allons donc créer le fichier dans le répertoire views appelé  **layout.blade.php**. Ajoutez le code suivant dans le  fichier *layout*.

<!-- layout.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>Tutoriel Laravel 8 CRUD</title>  
 <link href="{{ asset('css/app.css') }}" rel="stylesheet" type="text/css" />  
</head>  
  
<body>  
 <div class="container">  
   @yield('content')  
 </div>  
 <script src="{{ asset('js/app.js') }}" type="text/js"></script>  
</body>  
</html>

Dans le dossier ***resources/views***, créez les trois fichiers suivants.

index.blade.php

create.blade.php

edit.blade.php

Dans le  fichier **create.blade.php**, écrivez le code suivant.

<!-- create.blade.php -->  
  
@extends('layout')  
  
@section('content')  
<style>  
 .uper {  
   margin-top: 40px;  
 }  
</style>  
  
<div class="card uper">  
 <div class="card-header">  
   Ajouter une Voiture  
  </div>  
  
 <div class="card-body">  
   @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('cars.store') }}">  
. @csrf  
         <div class="form-group">  
             <label for="marque">Marque de Voiture:</label>  
             <input type="text" class="form-control" name="marque"/>  
          </div>  
  
         <div class="form-group">  
             <label for="prix">Prix :</label>  
             <input type="text" class="form-control" name="prix"/>  
         </div>  
         <button type="submit" class="btn btn-primary">Ajouter</button>  
     </form>  
 </div>  
</div>  
@endsection

Dans ce code, nous avons défini l'action qui appellera la méthode **store()** du controller **CarController**. N'oubliez pas que nous avons utilisé le contrôleur de ressources.

Maintenant, nous devons retourner cette vue de création à partir de la méthode **create()** de CarController. Écrivez donc le code suivant dans la méthode create().

// CarController.php  
  
  
public function create()  
{  
   return view('create')  
}

Accédez à **https://laravel8cars.test/cars/create** ou **http://localhost:8000/cars/create**

**Étape 8: ajoutez des règles de validation et stockez les données.**

Dans cette étape, nous ajouterons une validation de formulaire Laravel.

Maintenant, importer dans le fichier **CarController.php** l'espace de noms du modèle Car.php.

<?php  
  
// CarController.php  
  
  
namespace App\Http\Controllers;  
  
  
use Illuminate\Http\Request;  
use App\Models\Car;

Maintenant, écrivez le code suivant à l'intérieur du CarController.php dans la fonction **store().**

// CarController.php  
  
  
public function store(Request $request)  
{  
  
   $validatedData = $request->validate([  
       'marque' => 'required|max:255',  
       'prix' => 'required',  
   ]);  
  
    $car = Car::create($validatedData);  
  
   return redirect('/cars')->with('success', 'Voiture créer avec succèss');  
}

Nous utilisons la méthode **$request->validate()** pour la validation, qui reçoit un tableau de règles de validation.

Les règles de validation [] sont un tableau associatif. La clé sera le **field\_name** et la valeur étant les règles de validation. Le deuxième paramètre est un tableau facultatif pour les messages de validation personnalisés.

Les règles sont séparées par un signe "| ". Dans cet exemple, nous utilisons les règles de validation les plus élémentaires. Si la validation échoue, elle sera redirigée vers la page de formulaire avec des messages d'erreur. Si la validation réussit, il créera la nouvelle voiture et la sauvegardera dans la base de données.

En cas d'erreur, nous devons parcourir ces messages d'erreur dans le fichier **create.blade.php**, ce que nous avons déjà fait.

Si vous laissez tous les champs du formulaire vides, vous trouverez des messages d'erreur

Comme nous pouvons voir que nous avons obtenu les erreurs, mais si nous remplissons toutes les données correctes, vous ne pourrez toujours pas enregistrer les données dans la base de données en raison d'une exception d'affectation en masse.

Pour éviter l'exception d'affectation en masse , nous devons ajouter un tableau **$fillable** dans le modèle **Car.php**.

<?php  
  
// Car.php  
  
namespace App\Models;  
use Illuminate\Database\Eloquent\Factories\HasFactory;  
use Illuminate\Database\Eloquent\Model;  
  
class Car extends Model  
{  
   use HasFactory;  
  
   protected $fillable = ['marque', 'prix'];  
}

Maintenant, si vous remplissez les champs de formulaire corrects, cela crée une nouvelle ligne dans la base de données.

**Étape 9: Affichez les voitures**

Pour afficher la liste des voitures, nous devons écrire le code HTML dans le  fichier **index.blade.php**. Mais avant cela, écrivons la fonction *index()* du fichier CarController.php pour obtenir le tableau de données de la base de données.

// CarController.php  
  
public function index()  
{  
    $voitures = Car::all();  
  
   return view('index', compact('voitures'));  
}

Maintenant, écrivez le code suivant dans le  fichier **index.blade.php**.

<!-- index.blade.php -->  
  
@extends('layout')  
  
@section('content')  
  
<style>  
 .uper {  
   margin-top: 40px;  
 }  
</style>  
  
<div class="uper">  
  
 @if(session()->get('success'))  
   <div class="alert alert-success">  
     {{ session()->get('success') }}    
   </div><br />  
  @endif  
  
  <table class="table table-striped">  
  
   <thead>  
       <tr>  
         <td>ID</td>  
         <td>Marque</td>  
         <td>Prix</td>  
         <td colspan="2">Action</td>  
       </tr>  
    </thead>  
  
   <tbody>  
       @foreach($voitures as $voiture)  
       <tr>  
           <td>{{$voiture->id}}</td>  
           <td>{{$voiture->marque}}</td>  
           <td>{{$voiture->prix}}</td>  
           <td><a href="{{ route('cars.edit', $voiture->id)}}" class="btn btn-primary">Modifier</a></td>  
           <td>  
               <form action="{{ route('cars.destroy', $voiture->id)}}" method="post">  
                 @csrf  
                 @method('DELETE')  
                 <button class="btn btn-danger" type="submit">Supprimer</button>  
               </form>  
           </td>  
       </tr>  
       @endforeach  
   </tbody>  
 </table>  
<div>  
@endsection

Nous avons ajouté deux boutons nommés modifier  et  supprimer  pour effectuer les opérations respectives.

**Étape 10: Terminez la modification et la mise à jour**

Pour pouvoir modifier les données, nous avons besoin des données de la base de données. Ajoutez le code suivant dans la  fonction d'édition du fichier CarController.php.

// CarController.php  
  
public function edit($id)  
  
{  
    $car = Car::findOrFail($id);  
  
   return view('edit', compact('car'));  
}

Maintenant, créez le nouveau fichier dans le dossier views appelé  **edit.blade.php**  et ajoutez le code suivant.

@extends('layout')  
  
@section('content')  
  
<style>  
 .uper {  
   margin-top: 40px;  
 }  
</style>  
  
<div class="card uper">  
 <div class="card-header">  
   Modifier la voiture  
  </div>  
  
  <div class="card-body">  
  
   <@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('cars.update', $car->id ) }}">  
         <div class="form-group">  
             @csrf  
             @method('PATCH')  
             <label for="marque">Marque :</label>  
             <input type="text" class="form-control" name="marque" value="{{ $car->marque }}"/>  
          </div>  
  
         <div class="form-group">  
             <label for="cases">Prix :</label>  
             <input type="text" class="form-control" name="prix" value="{{ $car->prix }}"/>  
         </div>  
         <button type="submit" class="btn btn-primary">Modifier</button>  
     </form>  
 </div>  
</div>  
@endsection

Maintenant, allez à la page d'index, puis allez à la page d'édition d'une voiture spécifique, et vous verrez le formulaire avec les valeurs remplies.

Maintenant, ajoutez le code suivant dans la fonction ***update()*** du CarController.

// CarController.php  
  
public function update(Request $request, $id)  
{  
   $validatedData = $request->validate([  
       'marque' => 'required|max:255',  
       'prix' => 'required'  
    ]);  
  
   Car::whereId($id)->update($validatedData);  
  
   return redirect('/cars')->with('success', 'Voiture mise à jour avec succèss');  
}

Vous pouvez maintenant mettre à jour toutes les données dans la base de données.

**Étape 11: Créer une fonctionnalité de suppression**

Pour supprimer des données de la base de données, nous utiliserons la fonction **destroy()** de CarController.

// CarController.php  
  
public function destroy($id)  
{  
   $car = Car::findOrFail($id);  
   $car->delete();  
  
   return redirect('/cars')->with('success', 'Voiture supprimer avec succèss');  
}

La fonction **delete ()** est fournie par Laravel pour supprimer les données de la base de données.