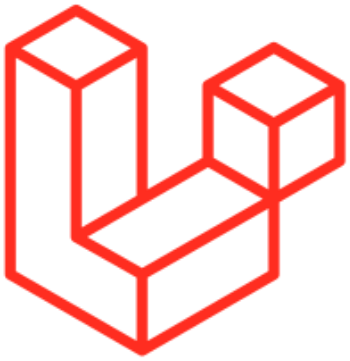


Laravel Tutorial

Provided by

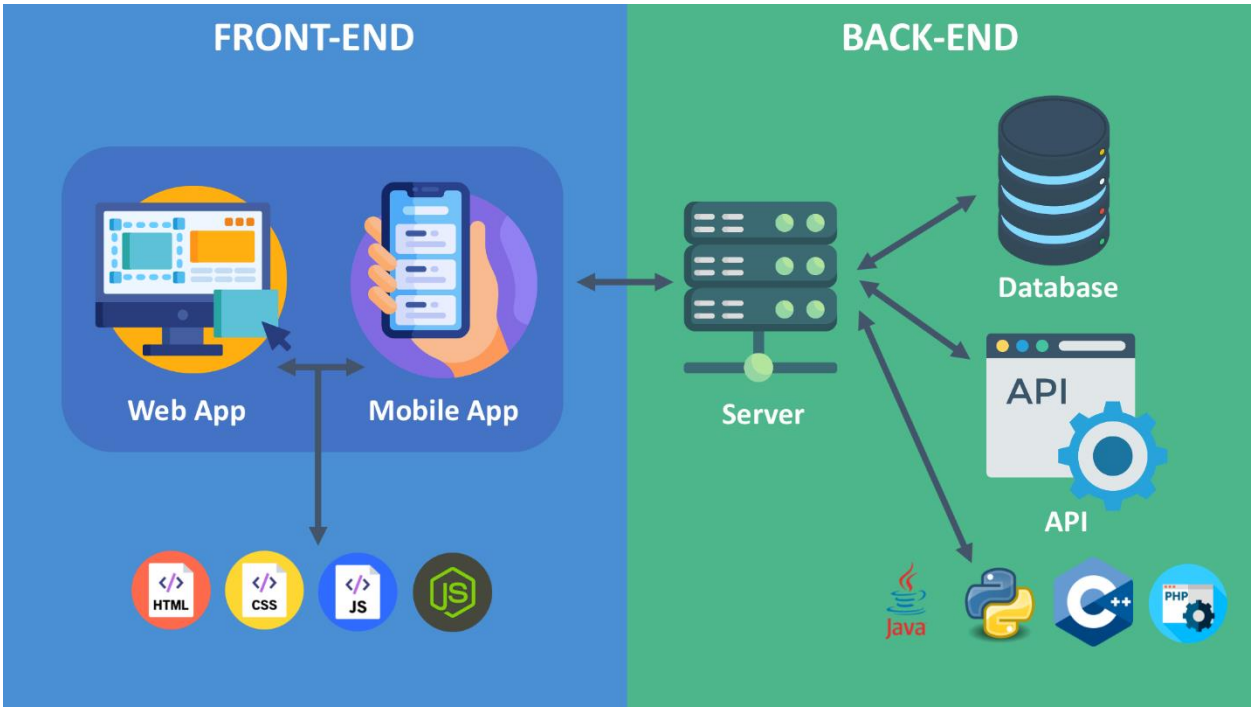
Sir Muhammad Farhan

Contact: 03162859445

Email: farhanteacher990@gmail.com

GitHub: https://github.com/muhammadfarhandevelper?tab=repositories

Front-End & Backend:



Introduction of Framework:

A framework in programming is a tool that provides ready-made components or solutions that are customized in order to speed up development. A framework may include a library, but is defined by the principle of inversion of control (IoC). With traditional programming, the custom code calls into the library to access reusable code. With IoC, the framework calls on custom pieces of code when necessary.

What is Laravel:

Laravel is a powerful MVC PHP framework, designed for developers who need a simple and elegant toolkit to create full-featured web applications. Laravel was created by Taylor Otwell. This is a brief tutorial that explains the basics of Laravel framework.

Prerequisites:

- HTML
- CSS
- JS
- PHP

- PHP CRUD
- OOP

Feature of Laravel:

- **Modularity**: Laravel is built with a modular packaging system with a dedicated dependency manager, allowing easy integration with third-party libraries.
- **Eloquent ORM**: Laravel includes a powerful ActiveRecord implementation called Eloquent ORM, which simplifies interactions with databases and reduces the amount of SQL code you need to write.
- **MVC Architecture**: Laravel follows the MVC architectural pattern, ensuring clarity between logic and presentation. This improves performance and allows for better documentation.
- **Blade Templating Engine**: Blade, Laravel's templating engine, provides convenient shortcuts and structures for writing views, making the process more intuitive and efficient.
- **Routing**: Laravel offers a simple and expressive syntax for defining routes, making it easy to handle complex routing requirements effectively.
- **Built-in Authentication and Authorization**: Laravel provides a simple way to organize authorization logic and control access to resources. It also includes built-in authentication services, making implementation quick and secure.
- **Artisan CLI**: Laravel's command-line interface, Artisan, automates repetitive tasks like database migrations, seeding, and testing setup, boosting developer productivity.
- **Security**: Laravel takes security seriously, with features like CSRF (Cross-Site Request Forgery) protection, SQL injection prevention, and secure authentication.

Composer:

- Composer is a tool which includes all the dependencies and libraries. It allows a user to create a project with respect to the mentioned framework (for example, those used in Laravel installation). Third party libraries can be installed easily with help of composer.
- All the dependencies are noted in **composer.json** file which is placed in the source folder.

Artisan

- Command line interface used in Laravel is called **Artisan**. It includes a set of commands which assists in building a web application.

Downloading and Installing Laravel:z

- Wamp Server / Xampp Server / Mamp Server
- Composer (Dependency Manager for PHP)
- Visual Studio Code

Laravel Install using Composer:

composer global require laravel/installer

For Creating Laravel Project:

laravel new projectname

Another Command

composer create-project --prefer-dist laravel/laravel project-name

Basic Structure of Laravel Project:

- **app/**: This directory houses the core application code
- **bootstrap/**: Contains the application bootstrapping scripts and configurations
- **config/**: Configuration files for various services and components used in the application.
- **database/**: Contains database-related files, including migrations and seeders
- **public/**: The entry point for all requests. Contains the front controller (index.php) and assets like images, CSS, and JavaScript files.
- **resources/**: Contains views, language files, and raw assets like SCSS or JavaScript files
- **routes/**: Defines the application's route declarations
- **storage/**: Contains files generated by the framework, such as logs, cached views, and compiled files.
- **vendor/**: Composer dependencies.
- **.env**: Environment configuration file.

Routing in Laravel

Routing:

Routing is the process of selecting a path for traffic in a network or between or across multiple networks. Broadly, routing is performed in many types of networks, including circuit-switched networks, such as the public switched telephone network, and computer networks, such as the Internet.

you will begin by defining routes in your `routes/web.php` file. The routes defined in `routes/web.php` may be accessed by entering the defined route's URL in your browser. For example, you may access the following route by navigating to `http://example.com/user` in your browser:

Example: (Routing without Controller):

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

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

    return view('about');
});
```

Here: '/about' is routing. View('about' **)** is called about page or view with .blade.php.

When defining multiple routes that share the same URI, routes using the `get`, `post`, `put`, `patch`, `delete`, and `options` methods should be defined before routes using the `any`, `match`, and `redirect` methods. This ensures the incoming request is matched with the correct route.

Routing with return Views:

View folder must have the views.

Data Sharing with routing:

Example:

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

Another Example: (with function)

Syntax:

with(key , value);

```
Route::get('/', function () {
   return view('welcome')->with('myinfo','Muhammad Farhan');
});
```

Another Example (compact function):

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

      $name = "Muhammad Farhan";
      $age = "20";

      return view('welcome',compact('name','age'));
   });
```

Routing Parameters:

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

Here {id} is parameter and return this id.

```
Route::get('/posts/{name}/{age}', function ($name, $age) {

   return $name . $age;

   });
```

Here two parameters: name, age and this is like the get method this data show in url.

Blade Template

What is Blade Template?

- Blade is a templating engine in a Laravel framework.
- Blade templating engine provides its own structure such as conditional statements and loops.
- To create a view file and save it with **.blade.php** extension instead of **.php** extension.

Syntax (Mostly Syntax):

```
{{ $name }}
```

Also Syntax:

```
{!! $name !!}
```

Blade Directives:

In addition to template inheritance and displaying data, Blade also provides convenient shortcuts for common PHP control structures, such as conditional statements and loops. These shortcuts provide a very clean, terse way of working with PHP control structures while also remaining familiar to their PHP counterparts.

All directive start with **@.**

- php
- If
- For loop
- While loop
- Switch
- Extends
- Section
- Isset
- Old
- Include

- Require

Example:

```
@php

    $array = ['farhan','ali','afan','ramzan','shahid'];

@endphp


@for ($i =0 ; $i < 10 ; $i++)

<p>This is Counring {{$i }}</p>

@endfor

    @while ($i< 10)

     <p>This is counting {{ $i }}</p>

     {{$i++}}
    @endwhile

    @foreach ($array as $item)

    <p>{{$item}}</p>

    @endforeach



    @if (2 < 1)

    <p> 2 is less than 1 </p>

    @elseif ( 1 < 2 )

    <p> 1 is less than 2 </p>

    @else
```

```
    <p>All items are equals</p>

    @endif


    @isset($name)

    <p>name is exist</p>

    @endisset
```

How to Use Layout in Laravel:

@yield

@extends

@section

What is Controller?

- Controller are class based php files.
- Controller can group related request handling logic into a single class.

Types of Controllers:

1. Basic Controller
2. Single Action Controllers
3. Resource Controllers

Command for make Controller.

```
php artisan make:controller HomeController
```

After Creating Controller.

Web.php file

```
Route::get('/student',[StudentController::class,'Index']);
```

Controller File:

```
public function Index(){

    return view(student.index');
}
```

Basic Controller:

Let's take a look at an example of a basic controller. It is not give the build in any code and functions.

```
class UserController extends Controller
{
    /**
     * Show the profile for a given user.
     *
     * @param  int  $id
     * @return \Illuminate\View\View
     */
    public function show()
    {
        return view('user.profile');
    }
}
```

Single Action Controller:

If a controller action is particularly complex, you might find it convenient to dedicate an entire controller class to that single action. To accomplish this, you may define a single `__invoke` method within the controller:

Example:

```
class LoginController extends Controller
{
    /**
     * Handle the incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
```

```
    */
    public function __invoke(Request $request)
    {
        echo "this is invokable method for the login cotnroller";
    }
}
```

Web.PHP:

```
Route::get('/login',LoginController::class);
```

Resource Controller:

If you think of each Eloquent model in your application as a "resource", it is typical to perform the same sets of actions against each resource in your application. For example, imagine your application contains a `Photo` model and a `Movie` model. It is likely that users can create, read, update, or delete these resources.

Because of this common use case, Laravel resource routing assigns the typical create, read, update, and delete ("CRUD") routes to a controller with a single line of code. To get started, we can use the `make:controller` Artisan command's `--resource` option to quickly create a controller to handle these actions

Command:

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

Use in web.php:

```
Route::resource('/employee',EmployeeController::class);
```

Submitting Form:

Form:

```
<div class="container mt-5">
   <div class="row">
      <div class="offset-md-3 col-md-6">

      <form action="{{ url('/store')}}" method="POST">

      @csrf

  <div class="mb-3">
   <label class="form-label">Name </label>
   <input type="text" class="form-control"  name="name" value="{{ old('name')}}">
   <p class="text-danger">
      @error('name')
         {{ $message }}
      @enderror
   </p>
</div>

  <div class="mb-3">
   <label class="form-label">Age </label>
   <input type="text" class="form-control"  name="age" value="{{ old('age')}}">
   <p class="text-danger">
      @error('age')
         {{ $message }}
      @enderror
   </p>
  </div>
  <div class="mb-3">
   <label class="form-label">Course</label>
   <input type="text" class="form-control"  name="course" value="{{ old('course')}}">
   <p class="text-danger">
      @error('course')
         {{ $message }}
      @enderror
   </p>
  </div>
  <div class="mb-3">
   <label class="form-label">Address </label>
   <input type="text" class="form-control"  name="address" value="{{ old('address')}}">
   <p class="text-danger">
      @error('address')
         {{ $message }}
```

```
    @enderror
  </p>
 </div>
<div class="mb-3">
  <label class="form-label">Date </label>
  <input type="date" class="form-control"  name="dob" value="{{ old('dob')}}">
  <p class="text-danger">
     @error('dob')
        {{ $message }}
     @enderror
  </p>
 </div>
 <div class="mb-3">
  <label class="form-label">Amount </label>
  <input type="number" class="form-control"  name="amount" value="{{ old('amount')}}">
  <p class="text-danger">
     @error('amount')
        {{ $message }}
     @enderror
  </p>
 </div>
<div class="mb-3">
  <label class="form-label">Email </label>
  <input type="email" class="form-control"  name="email" value="{{ old('email')}}">
  <p class="text-danger">
     @error('email')
        {{ $message }}
     @enderror
  </p>
 </div>
<div class="mb-3">
  <label class="form-label">Password </label>
  <input type="password" class="form-control"  name="pass" >
  <p class="text-danger">
     @error('pass')
        {{ $message }}
     @enderror
  </p>
 </div>
<div class="mb-3">
  <label class="form-label">Confirm Password </label>
  <input type="password" class="form-control"  name="cpass">
  <p class="text-danger">
     @error('cpass')
        {{ $message }}
```

```
      @enderror
    </p>
  </div>

  <button type="submit" class="btn btn-primary">Submit</button>
</form>

      </div>
    </div>
</div>
```

Action work call this function from Controller:

```
public function store(Request $req)
  {

     $req->validate([

         'name' => 'required',
         'age' => 'required | numeric',
         'course' => 'required',
         'address' => 'required',
         'dob' => 'required',
         'email' => 'required | email',
         'pass' => ['required' , 'regex: /^(?=.*\d)(?=.*[a-z])(?=.*[A-Z])(?!.*\s).*$/'],
         'cpass' => 'required | same:pass'
     ]);
```

Database Configuration:

1. Create database in mysql.
2. Set database name in .env file in DB_DATABASE variable.

Migration:

Migrations are like version control for your database, allowing your team to define and share the application's database schema definition. If you have ever had to tell a teammate to manually add a column to their local database schema after pulling in your changes from source control, you've faced the problem that database migrations solve.

Create Migration of Customer Table:

```
php artisan make:migration create_employees_table
```

Migration structure or Customer Table Schema:

```
public function up()
{
    Schema::create('employees', function (Blueprint $table) {
        $table->id('empid');
        $table->string('empname');
        $table->string('mail')->unique();
        $table->integer('salary');

        $table->timestamps();
    });
}
```

Running Migration:

To run all of your outstanding migrations, execute the migrate Artisan command:

```
php artisan migrate
```

Rolling Back Migration:

```
php artisan migrate:rollback
```

You may roll back a limited number of migrations by providing the `step` option to the `rollback` command. For example, the following command will roll back the last five migrations:

```
php artisan migrate:rollback --step=5
```

The `migrate:reset` command will roll back all of your application's migrations:

```
php artisan migrate:reset
```

You may roll back and re-migrate a limited number of migrations by providing the step option to the refresh command. For example, the following command will roll back and re-migrate the last five migrations:

```
php artisan migrate:refresh --step=5
```

Drop All Table and Migrate:

```
php artisan migrate:fresh
```

Add Column After Migrate Table:

When we migrated a table then you remember to add a column in existing table so this CLI run.

Example:

```
php artisan make:migrate add_column_to_tblname
```

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('users', function (Blueprint $table) {
    $table->string('email')->after('name');
});
```

Available Column Types:

- Id()
- unSignedBigInteger
- Integer
- Float
- Boolean

- String
- Enum
- Foreign
- bigIncreament
- double
- decimal
- tinyText
- text

What is Model:

A model class is typically used to "model" the data in your application. For example you could write a Model class that connect database table. You could use objects of these classes as vessels to send / receive data.

Example Generate Model Class:

```
php artisan make:model Employee
```

If you would like to generate a database migration when you generate the model, you may use the --migration or -m option:

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

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Employee extends Model
{

    protected $table = "employees";
    protected $primayKey = "empid";

}

?>
```

Laravel Upload File:

HTML:

```
<form method="post" action="{{url('/upload')}}"  enctype="multipart/form-data">

   @csrf
   <input type="file" name="image" required />
   <button type="submit">Upload</button>

</form>
```

Controller:

```
    $img = $req[image'];
    $name = $img->getClientOriginalName();
    echo $name;

    $filename = "uploads/".$name;
    $img->move("uploads",$filename);


    return view('upload',compact('filename'));
```

With Model

```
  public function Upload(Request $req){

    $img = $req->file('image');
    $name = $img->getClientOriginalName();
    echo $name;

    $filename = "uploads/".$name;
    $img->move("uploads",$filename);

    $obj = new Student();
    $obj->img= $filename;
    $obj->save();

  }
```