Laravel

Title

- Laravel
- Installation
- Application Directory Structure
- Setting Environment
- Routing
- Controller
- View
- Connecting with Database
- Session
- Custom Template Design
- Login / Logout
- Middleware
- Sample work

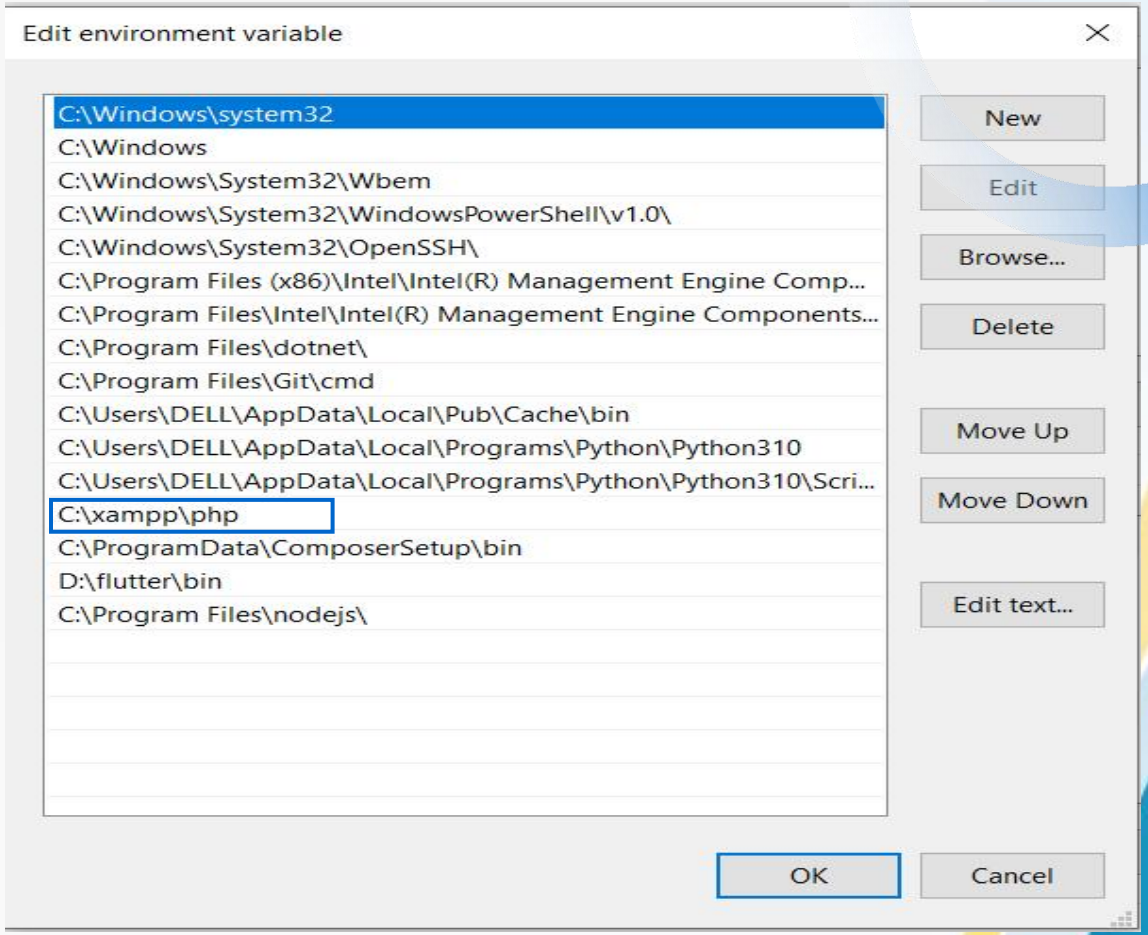
Laravel

- ✓ Laravel is a php web application frame work
- ✓ MVC Architecture Support
- ✓ Open source
- ✓ Blade Templating Engine

Installatoin

Step 1: Install XAMPP

- ✓ Download XAMPP from the official website: https://www.apachefriends.org/
- ✓ Run the installer and follow the on-screen instructions to install XAMPP
- ✓ Set php to environment variable



Step 2 : Install Composer

- ✓ Composer is a dependency manager for PHP that you need to install Laravel.
- ✓ Download and install Composer from the official website: https://getcomposer.org/
- ✓ Follow the installation instructions provided on-screen.
- ✓ To verify composer installation

```
composer --version
```

Step 3 : Install Laravel

✓ Install Laravel gobally using composer

```
composer global require laravel/installer
```

✓ Verify laravel installation

```
laravel --version
```

Step 4 : Set up Project

✓ Create a Project

```
composer create-project laravel/laravel project-name
or
laravel new project-name
```

✓ To start server open command prompt

```
cd project-name
```

✓ Run the project

```
php artisan serve
```

✓ open browser and run below command

```
http://127.0.0.1:8000/
```

Note: The default Laravel page will open.

Note:

- In some cases, Laravel misses some files in the latest version. Consider **downgrading** to Laravel **10** or **9**.

```
composer create-project laravel/laravel:^10 project-name
```

Application Directory Structure

laravel_project_root/

├── app/	
├── bootstrap/	
├── **config/**	**-Contains configuration files.**
├── database/	
├── public/	
├── resources/	
├── routes/	
├── storage/	
├── tests/	
├── **vendor/**	**-Contains Composer dependencies.**
├── **.env**	**-Environment configuration file.**
├── **.env.example**	**-Example environment configuration file.**
├── **.gitignore**	**-Specifies files and directories to be ignored by Git.**
├── **artisan**	**-The Artisan command-line tool.**
├── **composer.json**	**-Composer configuration file.**
├── **composer.lock**	**-Locked versions of Composer dependencies.**
├── **package.json**	**-Node.js dependencies.**
├── **phpunit.xml**	**-PHPUnit configuration file.**
└── **README.md**	**-The README file for the application.**

app/: Contains the core code of the application.

```
laravel_project_root/
├── app/
│   ├── Console/
│   ├── Exceptions/
│   ├── Http/
│   │   ├── Controllers/
│   │   ├── Middleware/
│   └── Models/
```

- **Console/**: Contains custom Artisan commands.
- **Exceptions/**: Contains the application's exception handling files.
- **Http/**: Contains controllers, middleware, and requests.
- **Models/**: Contains Eloquent models

bootstrap/: Contains files for bootstrapping the framework.

```
laravel_project_root/
├── app/
├── bootstrap/
│   ├── cache/
```

- **cache/:** Contains framework-generated cache files.

database/: Contains database-related files.

```
laravel_project_root/
├── app/
├── bootstrap/
├── config/
├── database/
│   ├── factories/
│   ├── migrations/
│   └── seeders/
```

- **factories/** : Contains model factory files.
- **migrations/**: Contains database migration files.
- **seeders/:** Contains database seeder files.

public/: Contains publicly accessible files.

```
laravel_project_root/
├── app/
├── bootstrap/
├── config/
├── database/
├── public/
│   ├── css/
│   ├── js/
│   └── index.php
```

- **css/**: Contains CSS files.
- **js/**: Contains JavaScript files.
- **index.php:** The entry point for all requests to the application.

resources/: Contains view files and other resources.

```
laravel_project_root/
├── app/
├── bootstrap/
├── config/
├── database/
├── public/
├── resources/
│   ├── js/
│   ├── lang/
│   ├── sass/
│   └── views/
```

- **js/**: Contains JavaScript files.
- **lang/**: Contains language files.
- **sass/**: Contains Sass files.
- **views/**: Contains Blade template files.

routes/: Contains route definition files.

```
laravel_project_root/
├── app/
├── bootstrap/
├── config/
├── database/
├── public/
├── resources/
├── routes/
│   ├── api.php
│   ├── channels.php
│   ├── console.php
│   └── web.php
```

- **api.php**: Routes for API.
- **channels.php**: Event broadcasting channels.
- **console.php**: Console-based routes.
- **web.php**: Web routes.

storage/: Contains compiled Blade templates, file-based sessions, file caches, and other files generated by the framework.

```
laravel_project_root/
├── app/
├── bootstrap/
├── config/
├── database/
├── public/
├── resources/
├── routes/
├── storage/
│   ├── app/
│   ├── framework/
│   │   ├── cache/
│   │   ├── sessions/
│   │   ├── views/
│   └── logs/
```

- **app/**: Contains application-generated files.
- **framework/**: Contains framework-generated files and caches.
- **logs/**: Contains application log files.

tests/: Contains automated tests.

```
laravel_project_root/
├── app/
├── bootstrap/
├── config/
├── database/
├── public/
├── resources/
├── routes/
├── storage/
├── tests/
│   ├── Feature/
│   └── Unit/
```

- **Feature/**: Contains feature tests.
- **Unit/**: Contains unit tests.

Setting Enviorment

Required

- IDE Visual Studio
- XAMP

Routing

- ✓ Routing in Laravel is a crucial aspect of the framework that defines how an application's endpoints respond to client requests.

- ✓ In Laravel, routes are defined in the **routes/web.php** file for **web** routes and **routes/api.php** for **API** routes.

routes/web.php

```php
<?php

use Illuminate\Support\Facades\Route;

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

    return view('welcome');

});
```

- use Illuminate\Support\Facades\Route;
  - ✓Purpose: This line imports the Route facade from the Illuminate\Support\Facades namespace.
  - ✓Explanation: In Laravel, facades provide a "static" interface to classes that are available in the application's service container. The Route facade allows you to define routes using a convenient syntax.

- Route::get('/', ...);

- ✓ HTTP Method: Route::get specifies that this route should respond to GET requests.
- ✓ URI: '/' is the URI (Uniform Resource Identifier) for this route. It represents the root URL of the application, i.e., http://your-app-domain/.

- function () {.....}

✓ Anonymous Function (Closure): function () { ... } defines an anonymous function, also known as a closure, which will be executed when this route is accessed.

✓ Inside the Closure: The code inside the closure is what will be executed when the route is hit. In this case, it returns a view named welcome.

- return view('welcome'):

✓ The view function returns the content of the welcome view. By default, Laravel looks for view files in the resources/views directory, so this will render the welcome.blade.php file located at resources/views/welcome.blade.php.

Basic Laravel Route Example: Returning a Plain Text Response

```php
<?php

use Illuminate\Support\Facades\Route;

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

    return “Welcome to Laravel”;

});
```

```php
<?php

use Illuminate\Support\Facades\Route;

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

    echo“Welcome to Laravel”;

});
```

Note: To run laravel project use: **”php artisan serve”.** Open your browser and type **“http://127.0.0.1:8000/”**

Welcome to Laravel

Echo and return

- **return**

- ❑ When you need to send a final response back to the client in a routes / controller.
- ❑ Use this in function , methods or closure where a response need to be captured & processed.
- ❑ Essential for sending structured responses like views, JSON,redirects in laravel application.

- **echo**

- ☐ When you need to immediate output to the browser.
- ☐ For simple debugging / quick output in script
- ☐ Not recommended for use in laravel routes or controller for production code

Passing value through url

```php
<?php

use Illuminate\Support\Facades\Route;

Route::get('/retrive/{id}', function ($id) {

    return “The value from browser is”.$id;

});
```

Note:

To run laravel project use:

”php artisan serve”

Open your browser and type

“**http://127.0.0.1:8000/**retrive/**5**”

The value from browser is 5

Passing multiple value through url

```
<?php

use Illuminate\Support\Facades\Route;

Route::get('/retrive/{v1}/{v2}',

function ($v1,$v2) {

    return “The value from browser

is”.$v1.”,”.$v2;

});
```

Note:

To run laravel project use:

”php artisan serve”

Open your browser and type

“**http://127.0.0.1:8000/**retrive/**5/6**”

The value from browser is 5 , 6

Loading html page/ templates

Step 1: Create an HTML Page (Blade Template)

- ✓ **Navigate to the resources/views Directory:**Go to the resources/views directory in your Laravel project.

- ✓ **Create a New Blade Template File**:Create a new file named **filename.blade.php**.

- ✓ **Add HTML Content to the Blade Template**: Open **filename.blade.php** and add your HTML content. Here's an example:

resources/views/filename.blade.php

```
<!DOCTYPE html>
<html>
<head>
    <title>Welcome to Laravel</title>
</head>
<body>
    <h1>Welcome to Laravel</h1>
    <p>This is a sample HTML page rendered using Laravel Blade.</p>
</body>
</html>
```

Step 2: Create a Route in routes/web.php

- ✓ **Open the routes/web.php File in your Laravel project.**
- ✓ **Define a Route to Load the Blade Template:**Add a route definition that uses the **view()** function to load the **filename.blade.php** template. Here's an example:

```
<?php

use Illuminate\Support\Facades\Route;

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

    return view('filename');

});
```

Note: In view() eliminates the file extension "blade.php" at the time of calling

Step 3: Access the Route

- ✓ Start the Laravel Development Server

```
php artisan serve
```

- ✓ Access the Route in a Web Browser:Open a web browser and navigate to **http://localhost:8000/.** You should see the HTML content defined in filename.blade.php.

Passing value to template

views/welcome.blade.php

```
<body>
ID      : {{  $id  }} <br>
Name : {{ $name }} <br>
</body>
```

routes/web.php

```
<?php
use Illuminate\Support\Facades\Route;
Route::get('/retrive', function () {
    return view('welcome',['id'=>1,'name'
=>'Anvi']);
});
```

Controller

- ✓ In Laravel, controllers are used to group related route handling logic into a single class.
- ✓ It makes your code more organized and easier to manage.

Create a Controller

❖ step 1 : Create a controller

```
php artisan make:controller controller_name
```

Eg : php artisan make:controller EmpController

Note: This command will create a new file named "EmpController.php" in "app/Http/Controllers" directory

❖ step2 : Define methods in controller

- Open “app/Http/Controllers/EmpController”

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class EmpController extends Controller
{
    public function function1(){

        return "1st function called";

    }
}
```

Note: You can add more functions in the class.

❖ step 3 : Define route to use the controller

- routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

use App\Http\Controllers\EmpController;

Route::get('/function1',[EmpController::class,'function1']);
```

import the file path of a controller

class name of your controller

function name in your controller

Note : Run same as the normal procedure.

Passing Values through URL in Controller

app/Http/Controllers/EmpController

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class EmpController extends Controller
{
    ......
    public function function2($value){

        return $value;

    }
}
```

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

use App\Http\Controllers\EmpController;

......

Route::get('/function2/{id}',

[EmpController::class,'function2']);
```

Loading HTML in the Controller

app/Http/Controllers/EmpController

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class EmpController extends Controller
{
    ......
    public function function3(){

        return view( 'welcome' );

    }
}
```

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

use App\Http\Controllers\EmpController;

......

Route::get('/function3',

[EmpController::class,'function3']);
```

views/welcome.blade.php

```
<!DOCTYPE html>
<html>
<head>
    <title>Welcome to Laravel</title>
</head>
<body>
    <h1>Welcome to Laravel Controller </h1>
    <p>This is a sample HTML page rendered using Laravel Controller.</p>
</body>
</html>
```

Passing values from the controller through HTML

app/Http/Controllers/EmpController

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class EmpController extends Controller
{
    ......
    public function function3{

 return view( ‘welcome’ ,[‘id’=>1,’name’
=>’Anvi’] );
    }
}
```

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

use App\Http\Controllers\EmpController;

......

Route::get('/function3',

[EmpController::class,’function3’]);
```

views/welcome.blade.php

```
<!DOCTYPE html>
<html>
<head>
    <title>Welcome to Laravel</title>
</head>
<body>
    <h1>Welcome to Laravel Controller </h1>
    <p>Employee ID : {{ id }}</p>
   <p>Employee NAme : {{ name }}</p>
</body>
</html>
```

Controller with middleware

- ✓Ensure that only authenticated users can access certains routes or controller methods.

Note: We will cover this section in more detail later.

Views

- ✓ In Laravel, views are used to render your application's HTML user interfaces.
- ✓ Views are typically stored in the **resources/views** directory and are rendered using the Blade templating engine, which provides convenient syntax for working with PHP and displaying data.

Creating a view

- ✓ **Navigate to the resources/views Directory:**Go to the resources/views directory in your Laravel project.
- ✓ **Create a New Blade Template File**:Create a new file named **filename.blade.php**.
- ✓ **Add HTML Content to the Blade Template**: Open **filename.blade.php** and add your HTML content. Here's an example:

resources/views/filename.blade.php

```
<!DOCTYPE html>
<html>
<head>
    <title>Welcome to Laravel</title>
</head>
<body>
    <h1>Welcome to Laravel Views</h1>
    <p>This is a sample HTML page rendered using Laravel Blade.</p>
</body>
</html>
```

Rendering Views

- ✓ **There are mainly two methods for rendering views**
  1. **Using the view() helper function in routes.**
  2. **Using the view() helper function in controllers.**

1. Using the view() helper function in **routes**.

- ✓ **Open the routes/web.php File**
- ✓ **Define a Route to Load the Blade Template:** Add a route definition that uses the **view()** function to load the **filename.blade.php** template. Here's an example:

```php
<?php

use Illuminate\Support\Facades\Route;

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

    return view('filename');

});
```

Note: In view() eliminates the file extension “blade.php” at the time of calling

2.Using the view() helper function in controllers.

- Create a Controller:

php artisan make:controller WelcomeController

- Define a Method in the Controller:
    1. open app/Http/Controllers/WelcomeController.php
    2. Define methods

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;

class WelcomeController extends Controller
{
    public function show()
    {
        return view('filename');
    }
}
```

- Define a Route that Uses the Controller:
  - **open routes/web.php**
  - **Define route for controller**

```
<?php

use Illuminate\Support\Facades\Route;

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

    return view('filename');

});
```

Passing value to template in route

resources/views/welcome.blade.php

```
<body>
ID      : {{  $id  }} <br>
Name : {{ $name }} <br>
</body>
```

routes/we.php

```
<?php
use Illuminate\Support\Facades\Route;
Route::get('/retrive', function () {
    return view('welcome',['id'=>1,'name'
=>'Anvi']);
});
```

Method 1: with()

- The with() method is used in Laravel controllers to pass data to the view. It accepts an array where keys are variable names available in the view and values are the actual data you want to pass.

syntax:

```
return view('view_name')->with('variable_name', $value);
```

You can also pass multiple variables using an array:

```
return view('view_name')->with([
        'variable1' => $value1,
        'variable2' => $value2,
         // and so on
]);
```

```
return view('view_name')
        ->with('variable1' => $value1)
        ->with('variable2' => $value2)
         // and so on
        ;
]);
```

Example1:Passing an Associative Array Directly

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

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

    $data = ['id'=>1,'name' =>'Anvi'];

    return view('welcome')->with($data);

});
```

resources/views/welcome.blade.php

```
<body>

ID      : {{  $id  }} <br>

Name : {{ $name }} <br>

</body>
```

Example 2 : Using a Key-Value Pair

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

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

    $data = [‘id’=>1,’name’ =>’Anvi’];

    return view(‘welcome’)->with(‘data’,$data);

});
```

resources/views/welcome.blade.php

```
<body>

ID      : {{  $data[‘id’]  }} <br>

Name : {{ $data[‘name’] }} <br>

</body>
```

Example 3: Pass Individual Key-Value Pairs

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

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

      return view('welcome')->with('name','Riss')

->with('place','Kannur')

->with('more',['phone'=>8590059328,

'email':'risskannur@gmail.com']);

});
```

resources/views/welcome.blade.php

```
<body>

Name : {{  $name }} <br>

Place : {{ $place }} <br>

Email : {{ $more['phone'] }} <br>

Place : {{ $more['email'] }} <br>

</body>
```

Example 4: Pass multiple values in an array

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

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

 return view(‘welcome’) ->with([ “name”=>”Riss”,

“place”=>”Kannur”,

“more”=>

[ 'phone'=>8590059328,'email':'risskannur@gmail.com']

]);

});
```

resources/views/welcome.blade.php

```
<body>

Name : {{  $name }} <br>

Place : {{ $place }} <br>

Email : {{ $more[‘phone’] }} <br>

Place : {{ $more[‘email’] }} <br>

</body>
```

Method 2: compact()

- The compact() function is a PHP function that creates an array containing variables and their values.
- In Laravel, compact() is often used in conjunction with the view() function to pass variables to a view.

syntax:

```
return view('view_name', compact('variable1', 'variable2',....));
```

Note: This is equivalent to manually creating an array like ['variable1' => $variable1, 'variable2' => $variable2].

Example:

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

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

    $age  = 12;

    $data = [‘id’=>1,’name’ =>’Anvi’];

    return view(‘welcome’,compact(‘data’,’age’));

});
```

resources/views/welcome.blade.php

```
<body>

ID      : {{  $data[‘id’]  }} <br>

Name : {{ $data[‘name’] }} <br>

Age    : {{ $age }}

</body>
```

Note :

- ✓ **Both methods achieve the same result of passing data to views, so the choice often depends on personal preference and the specific context of your application.**
- ✓ **compact() can be more concise for passing a few variables, while with() is preferred for more complex scenarios or when passing non-variable data (like arrays).**

Passing values from the controller through HTML

app/Http/Controllers/EmpController

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class EmpController extends Controller
{
    ......
    public function function3{

 return view( 'welcome' ,['id'=>1,'name'
=>'Anvi'] );
    }
}
```

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

use App\Http\Controllers\EmpController;

......

Route::get('/function3',

[EmpController::class,'function3']);
```

Note: First, you need to create a controller named EmpController . Refer to the slide on controllers click here.

views/welcome.blade.php

```
<!DOCTYPE html>
<html>
<head>
    <title>Welcome to Laravel</title>
</head>
<body>
    <h1>Welcome to Laravel Controller </h1>
    <p>Employee ID : {{ id }}</p>
   <p>Employee NAme : {{ name }}</p>
</body>
</html>
```

Blade Templating Engine

Blade is a powerful templating engine in Laravel that provides features such as:

- Extending Layouts: Using @extends and @section directives to define and extend layouts.
- Including Sub-Views: Using @include directive to include sub-views within a view.
- Control Structures: Using @if, @foreach, @while, and @switch for conditional rendering and looping.

1.Conditional statements

- Blade provides **@if, @elseif, @else,@unless,@endunless,** and **@endif** directives to create conditional statements.

a) **@if**

resources/views/welcome.blade.php

```
<body>
    @if ($v == 5)
        <p>Condition true!</p>
    @endif
</body>
```

routes/web.php

```
<?php
use Illuminate\Support\Facades\Route;
Route::get('/', function () {
    return view('welcome',['v'=>5]);
});
```

b) **@else**

resources/views/welcome.blade.php

```
<body>
    @if ($v == 5)
        <p>Condition true!</p>
    @else
      <p>Condition false!</p>
    @endif
</body>
```

routes/web.php

```
<?php
use Illuminate\Support\Facades\Route;
Route::get('/', function () {
    return view('welcome',['v'=>15]);
});
```

c) @elseif

resources/views/welcome.blade.php

```
<body>
    @if ($v == 5)
        <p>Condition true!</p>
    @elseif ($v == 15)
        <p>Condition true!</p>
    @else
        <p>Condition false!</p>
    @endif
</body>
```

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

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

    return view('welcome',['v'=>15]);

});
```

d) @unless

- The @unless directive in Blade is the **inverse of the @if directive.**
- It executes the given block of code if the specified condition evaluates to false.
- It is essentially a syntactic sugar for @if (!condition).

Syntax: **resources/views/example.blade.php**

```
<body>
    @unless(condition)
        <p>Executed when condition is  false</p>
    @else
        <p>Executed when condition is  true</p>
    @endunless
</body>
```

resources/views/welcome.blade.php

```
<body>
  @unless($isAdmin)
        <p>You are not an
administrator.</p>
    @else
        <p>Welcome, Admin!</p>
    @endunless
</body>
```

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

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

    return view('welcome', ['isAdmin'
=> false]);

});
```

resources/views/welcome.blade.php

```
<body>
  @unless(count($items))
      <p> No Data </p>
   @else
      <p>You have some data!</p>
   @endunless
</body>
```

routes/web.php

```
<?php
use Illuminate\Support\Facades\Route;
Route::get('/', function () {
   $items = [];
    return view('welcome',
compact('items') );
});
```

Note : The @unless directive is helpful for readability when you want to execute a block of code only when a condition is false.

2.Looping statements

- In Laravel's Blade templating engine, looping statements are used to iterate over arrays or collections and render HTML for each item.

- The most commonly used looping statements in Blade are **@foreach, @forelse, @for,** and **@while**.

2.1) @for :

- The @for directive is used to create a traditional for loop.

Syntax:

```
@for ($initialization; $condition; $increment)
                    <!-- Loop body -->
@endfor
```

Example:

views/example.php

```
<body>
    <ul>
        @for ($i = 0; $i < 5; $i++)
            <li>Item {{ $i }}</li>
        @endfor
    </ul>
</body>
```

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

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

    return view('example');

});
```

You can also use dynamic values for the loop control variables. Example:

views/example.php

```
<body>
    <ul>
        @for ($i = 0; $i < $count; $i++)
            <li>Item {{ $i }}</li>
        @endfor
    </ul>
</body>
```

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

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

    $count = 10;

    return view('example',  compact('count'));

});
```

2.2) @foreach :

- The @foreach directive is used to loop through an array or collection.

Syntax:

```
@foreach ($items as $item)
    <!-- Loop body -->
@endforeach
```

Example:

views/example.php

```
<body>
    <ul>
        @foreach ($users as $user)
            <li>{{ $user }}</li>
        @endforeach
    </ul>
</body>
```

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

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

    $users = ['Alice', 'Bob', 'Charlie', 'David'];

    return view('example',compact('users'));

});
```

2.3) @forelse :

- The @forelse directive is similar to @foreach, but it provides a convenient way to handle empty arrays or collections.

Syntax:

```
@forelse ($items as $item)
    <!-- Content to display for each item -->
@empty
    <!-- Content to display when $items is empty -->
@endforelse
```

Example1: Non-Empty Array

routes/web.php

```php
<?php

use Illuminate\Support\Facades\Route;

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

    $users = ['Alice', 'Bob', 'Charlie'];

    return view('example', compact('users'));

});
```

resources/views/example.blade.php

```
<body>
    <ul>
        @forelse ($users as $user)
            <li>{{ $user }}</li>
        @empty
            <li>No users found.</li>
        @endforelse
    </ul>
</body>
```

Example2: With Empty Array

routes/web.php

```php
<?php

use Illuminate\Support\Facades\Route;

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

    $users = [];

    return view('example', compact('users'));

});
```

resources/views/example.blade.php

```
<body>
    <ul>
        @forelse ($users as $user)
            <li>{{ $user }}</li>
        @empty
            <li>No users found.</li>
        @endforelse
    </ul>
</body>
```

2.3) @while:

- The @forelse directive is similar to @foreach, but it provides a convenient way to handle empty arrays or collections.

Syntax:

```
@while (condition)
    <!-- Content to display while the condition is true -->
@endwhile
```

Example:

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

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

    return view('welcome');

});
```

resources/views/welcome.blade.php

```
<body>
    <ul>
        @while(true)
            <p> Infinte loop </p>
        @endwhile
    </ul>
</body>
```

Note : **To know full while program first need to cover php execution in blade templates.click here.**

3.Looping control statements

- In Laravel's Blade templating engine, you can use several looping control statements to control the flow of loops, such as **@continue** and **@break**. These statements are useful for skipping iterations or breaking out of loops based on certain conditions.
- Looping Control Statements in Blade

  1. @continue: Skips the current iteration and proceeds to the next iteration of the loop.
  2. @break: Exits the loop immediately.

Example

views/example.php

```
<body>
    <ul>
        @for ($i = 0; $i < 5; $i++)
            @if($i==2)
                @break
            @endif
            <li>Item {{ $i }}</li>
        @endfor
    </ul>
</body>
```

views/example.php

```
<body>
    <ul>
        @for ($i = 0; $i < 5; $i++)
            @if($i==2)
                @continue
            @endif
            <li>Item {{ $i }}</li>
        @endfor
    </ul>
</body>
```

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

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

    return view('example');

});
```

4. php execution

- In Laravel's Blade templating engine, you can execute PHP code directly within your Blade templates using the @php directive.
- This allows you to include any PHP logic that you might need for rendering your views.

Syntax:

Using @php Directive

```
@php
    // Your PHP code here
@endphp
```

Using @php Directive

```
<?php
    // Your PHP code here
?>
```

Example 1:

views/example.php

```
<body>
    <ul>
    @php
        $name = 'John Doe';
        $age = 29;
    @endphp
    <p>Name: {{ $name }}</p>
    <p>Age: {{ $age }}</p>
    </ul>
</body>
```

views/example.php

```
<body>
    <ul>
        <?php
        $name = 'John Doe';
        $age = 29;
        ?>
        <p>Name: {{ $name }}</p>
        <p>Age: {{ $age }}</p>    </ul>
</body>
```

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

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

    return view('example');

});
```

Example 2: with Control Structures

views/example.php

```
<body>
    @php
        $users = ['Alice', 'Bob', 'Charlie'];
    @endphp
    <ul>
        @foreach ($users as $user)
            <li>{{ $user }}</li>
        @endforeach
    </ul>
    <p>Total Users: {{ count($users) }}</p>
</body>
```

views/example.php

```
<body>
    <ul>
        @php
        $i = 0;
        $max = 10;
    @endphp
    <ul>
        @while ($i < $max)
            <li>Number: {{ $i }}</li>
            @php
                $i++;
            @endphp
        @endwhile
 </ul>
</body>
```

views/example.php

```
<body>
    @php
        $i = 0;
        $max = 10;
    @endphp
    <ul>
        @while ($i < $max)
            <li>Number: {{ $i }}</li>
            {{ $i++ }}
        @endwhile
    </ul>
</body>
```

O/P

- Number: 0 0
- Number: 1 1
- Number: 2 2
- Number: 3 3
- Number: 4 4
- Number: 5 5
- Number: 6 6
- Number: 7 7
- Number: 8 8
- Number: 9 9

≠

views/example.php

```
<body>
    <ul>
        @php
        $i = 0;
        $max = 10;
    @endphp
    <ul>
        @while ($i < $max)
            <li>Number: {{ $i }}</li>
            @php
                $i++;
            @endphp
        @endwhile
 </ul>
</body>
```

O/P

- Number: 0
- Number: 1
- Number: 2
- Number: 3
- Number: 4
- Number: 5
- Number: 6
- Number: 7
- Number: 8
- Number: 9

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

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

    return view('example');

});
```

Note:

- ✓ In previous slide both snippets are intended to perform the same operation: generating a list of numbers from 0 to 9 using a while loop within a Laravel Blade template. However, there are slight differences in their implementations.
- ✓ Incrementing the Counter:
  - ✓ **First Snippet: Increments $i using {{ $i++ }}, which is Blade syntax for printing the value and then incrementing it.**
  - ✓ **Second Snippet: Increments $i using @php $i++; @endphp, which uses the @php directive to execute PHP code without printing anything.**

4. Including subviews

- Including subviews in Laravel Blade templates allows you to break down complex views into smaller, manageable pieces.
- This practice promotes reusability and maintainability of your code. The **@include** directive is used to include a subview within another view.

Syntax:

```
@include('view.name', ['key' => 'value'])
```

Example 1:include a header and a footer subview within a main view.

resources/views/header.blade.php

```
<div>
    <h1>Welcome to My Website</h1>
</div>
```

resources/views/footer.blade.php

```
<div>
    <p>&copy; 2024 My Website</p>
</div>
```

resources/views/main.blade.php

```
<body>
    @include('header')
    <div>
        <p>This is the main content of the
page.</p>
    </div>
    @include('footer')
</body>
```

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

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

    return view('main');

});
```

Example 2: Passing Data to Subviews

resources/views/header.blade.php

```
<div>
    <h1>{{ $title }}</h1>
   <h1>Welcome to My Website</h1>
</div>
```

resources/views/footer.blade.php

```
<div>
   <p>&copy; 2024 My Website</p>
</div>
```

resources/views/main.blade.php

```
<body>
    @include('header', ['title' => 'Welcome to My Website'])
    <div>
        <p>This is the main content of the page.</p>
    </div>
    @include('footer')
</body>
```

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

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

    return view('main');

});
```

5. Extending layouts

- Extending layouts in Laravel's Blade templating engine allows you to create a consistent look and feel across multiple pages of your application.
- This is achieved by defining a base layout and then extending this layout in your child views.
- The **@extends** directive is used to specify which layout a view should extend, and the **@section and @yield** directives are used to define and inject content into the layout.

resources/views/layout.blade.php

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>@yield('title')</title>
</head>
<body>
   <header>
      <!-- Navigation bar or header content -->
   </header>
   <div class="container">
      @yield('content')
   </div>
   <footer>
      <!-- Footer content -->
   </footer>
</body>
</html>
```

resources/views/main.blade.php

```
@extends('layouts.app')
@section('title', 'Home Page')
@section('content')
    <h1>Welcome to the Home Page</h1>
    <p>This is the content of the home page.</p>
@endsection
```

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

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

    return view('main');

});
```

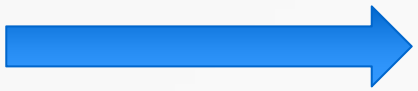
Loading images in blade

- ✓ To display images in a Laravel application, you need to ensure that your images are stored in a publicly accessible directory and then reference these images correctly in your Blade views.

  Step 1: Store Images in the Public Directory.

  Step 2: Reference Images in Blade Views.To display these images in your Blade views, use the **asset helper function**.

```
public/
    -images/
        -example.jpg
    -logo.png
```



File directory structure for saving images.

More info **Refer to Directory Structure**.

resources/views/main.blade.php

```
<body>
<img
src="{{ asset('/images/example.jpg') }}"
alt="" height="100" width="100">
<img src="{{ asset('logo.png') }}" alt=""
height="100" width="100">
</body>
```

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

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

    return view('main');

});
```

Redirect within a Blade view

- ✓ If you want to create a hyperlink in a Blade view that redirects the user to another route when clicked, you can use the route helper function to generate the URL for the route or <a> tag.

```
{{  route('urlname') }}
or
{{  route('routename',['variable1'=>'value1',...]) }}
or
<a href="/urlname">Link Name</a>
or
<a href="/urlname/value/value2">Link Name</a>
```

Example 1: using route()

resources/views/userhome.blade.php

```
<div>
    <h1>Welcome to My Website</h1>
   <a href =" {{ route('profile') }}"> View Profile </a> <br>
   <a href =" {{ route('category') }}"> View Category </a> <br>
</div>
```

resources/views/uprofile.blade.php

```
<div>
    <p>My Profile</p>
</div>
```

resources/views/ucategory.blade.php

```
<div>
    <p>Category</p>
</div>
```

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

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

    return view('userhome');

})->name('uhome');

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

    return view('uprofile');

})->name('profile');

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

    return view('ucategory');

})->name('category');
```

Note:

- ✓ In Laravel, the **Route::get('/uprofile', function () { ... })-> name (' profile');** statement defines a route for handling GET requests to the **/uprofile** URL. The name('**profile**') part assigns a name to this route.

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

- ✓ The **routename** is used in **route helper function.**

Example 2: Defining a Route with Parameters

To pass values to a route in a Blade view, you use the route helper and pass the necessary parameters as an array.

resources/views/userhome.blade.php

```
<div>
    <h1>Welcome to My Website</h1>
   <a href =" {{ route('profile', [ 'm' =>
'Message from user home' ]) }}"> View
Profile </a> <br>
   <a href =" {{ route('category') }}"> View
Category </a> <br>
</div>
```

resources/views/uprofile.blade.php

```
<div>
    <p>My Profile {{ m }}</p>
</div>
```

resources/views/ucategory.blade.php

```
<div>
    <p>Category</p>
</div>
```

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;


Route::get('/uhomee', function () {
    return view('userhome');
})->name('uhome');

Route::get('/uprofile/{m}', function ($m) {

    return view('uprofile',['m' => $m]);

})->name('profile');

Route::get('/category_user', function () {
    return view('ucategory');
})->name('category');
```

Example 3: using anchor tag

resources/views/userhome.blade.php

```
<div>
    <h1>Welcome to My Website</h1>
   <a href =" /uprofile"> View Profile </a>
<br>
   <a href =" /category_user"> View
Category </a> <br>
</div>
```

resources/views/uprofile.blade.php

```
<div>
    <p>My Profile</p>
</div>
```

resources/views/ucategory.blade.php

```
<div>
    <p>Category</p>
</div>
```

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

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

    return view('userhome');

})->name('uhome');

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

    return view('uprofile');

})->name('profile');

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

    return view('ucategory');

})->name('category');
```

Example 4: Generating URLs with Parameters

To pass values to a route in a Blade view, you use the route helper and pass the necessary parameters as an array.

resources/views/userhome.blade.php

```
<div>
    <h1>Welcome to My Website</h1>
   <a href =" /uprofile/message form
home"> View Profile </a> <br>
   <a href =" /category_user"> View
Category </a> <br>
</div>
```

resources/views/uprofile.blade.php

```
<div>
    <p>My Profile {{ m }}</p>
</div>
```

resources/views/ucategory.blade.php

```
<div>
    <p>Category</p>
</div>
```

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;


Route::get('/uhomee', function () {
    return view('userhome');
})->name('uhome');

Route::get('/uprofile/{m}', function ($m) {

    return view('uprofile',['m' => $m]);

})->name('profile');

Route::get('/category_user', function () {
    return view('ucategory');
})->name('category');
```

Note:

1. The two route definitions you provided are almost identical in their functionality, but there is a key difference related to the naming of the route.
2. A named route is used in the route() function, and an unnamed route is used in an <a> tag.
3. Using named routes is generally recommended for better maintainability and flexibility, especially in larger applications.
4. Same procedures in controller

Accessing Form elements

- ✓ Create a Form in a Blade Template.
- ✓ Use the @csrf directive in your forms to protect against CSRF attacks.
- ✓ Define a Route.
- ✓ Create a controller method or route to handle the form submission logic.
- ✓ Validate form data and handle any validation errors.

```
request->input('fieldname')
or
request->file('fieldname')
```

Method 1 : With help of routing

resources/views/form.blade.php

```
<body>
    <h1>Submit Your Information</h1>
    <form action="{{ route('form.submit') }}" method="POST">
        @csrf
            <button type="submit">Submit</button>
    </form>
</body>
```

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

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

    return view('form');

});

Route::post('/form_submit', function () {

    return 'Button clicked worked';

})->name('form.submit');
```

Method 2 : Using Controller

resources/views/form.blade.php

```
<body>
    <h1>Submit Your Information</h1>
    <form action="{{ route('form.submit') }}" method="POST">
        @csrf
            <button type="submit">Submit</button>
    </form>
</body>
```

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

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

    return view('form');

});

Route::post('/form-submit', [FormController ::

class, 'submit'])->name('form.submit');
```

app/Http/Controllers/FormController.php

```
namespace App\Http\Controllers;


class FormController extends Controller
{
    public function submit()
    {

        return 'Button clicked worked';
    }
}
```

1) Accessing value from textfield

resources/views/form.blade.php

```
<body>
   <h1>Submit Your Information</h1>
   <form action="{{ route('form.submit') }}" method="POST">
      @csrf
 <div>
         <label for="name">Name:</label>
         <input type="text" id="name" name="name">
      </div>
          <button type="submit">Submit</button>
   </form>
</body>
```

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

use Illuminate\Http\Request;

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

    return view('form');

});

Route::post('/form_submit',

function (Request $request) {

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

     return response()->json([

            'textfield' => $name;

            ]);

})->name('form.submit');
```

Note: Same procedure in controller

Note1:

- **$request->all()**
  - get all input values as an associative array
- **$request->only(['field1','field2',...])**
  - get only specified
- **$request->except(['field1','field2',...])**
  - get all values except specified

Note2:

- In the case of radio buttons, textareas, and dropdowns, you can fetch the value using request->input('fieldname').

2) Accessing value from filefield

resources/views/form.blade.php

```
<body>
    <h1>Submit Your Information</h1>
    <form action="{{ route('form.submit') }}" method="POST" enctype="multipart/form-data">
        @csrf
        <div>
            <label for="file">Choose a file:</label>
            <input type="file" id="file" name="file">
        </div>
            <button type="submit">Submit</button>
    </form>
</body>
```

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

use Illuminate\Http\Request;

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

    return view('form');

});
```

```
Route::post('/form_submit',

function (Request $request) {

 if ($request->hasFile('photo')) {

$photoPath = $request->file('file')->

store('photos', 'public');

            }

return response()->json([

                'file path=> $photoPath ?? null;

                ]);

})->name('form.submit');
```

2) Accessing value from checkbox

resources/views/form.blade.php

```
<body>
    <h1>Submit Your Information</h1>
    <form action="{{ route('form.submit') }}" method="POST" enctype="multipart/form-data">
        @csrf
            <div class="form-group">
                <div class="form-check">
                    <input class="form-check-input" type="checkbox" id="newsletter" name="newsletter[]">
                    <label class="form-check-label" for="newsletter">
                        Subscribe to Newsletter
                    </label>
                    <input class="form-check-input" type="checkbox" id="newsletter1" name="newsletter[]">
                    <label class="form-check-label" for="newsletter1">
                        Subscribe to Newsletter1
                    </label>
                </div>
            </div>
        <button type="submit">Submit</button>
    </form>
</body>
```

routes/web.php

```
<?php

use Illuminate\Support\Facades\Route;

use Illuminate\Http\Request;

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

    return view('form');

});

Route::post('/form_submit',

function (Request $request) {

    $checkbox = $request->input('newsletter',[]);

return response()->json([

            'checkbox'=> $checkbox

            ]);

})->name('form.submit');
```

Database Connection

- ✓ Laravel, a popular PHP framework, provides a simple and elegant way to interact with databases.
- ✓ It supports multiple database systems such as MySQL, PostgreSQL, SQLite, and SQL Server.

Connecting with Database

❑ Step 1 : Configure Database Connection

Laravel uses environment variables to configure database settings. These are defined in the .env file in the root of your Laravel project.

```
DB_CONNECTION = mysql
DB_HOST       = 127.0.0.1
DB_PORT       = 3306
DB_DATABASE   = your_database_name
DB_USERNAME   = your_database_user
DB_PASSWORD   = your_database_password
```

❑ Step 2 : Run Migrations

Migrations are a way to version control your database schema. They allow you to define your database structure in code and easily apply changes.

Step 2.1: Create a Migration

```
php artisan make:migration migrationname
or
php artisan make:migration migrationname --create=tablename
```

- ✓ **php artisan: The command-line interface for interacting with Laravel.**
- ✓ **make:migration: The Artisan command to create a new migration file.**
- ✓ **migrationname: The name of the migration. It's a good practice to use a descriptive name.**
- ✓ **--create=tablename: This option tells Laravel that this migration will create a new table named users.**

Example : php artisan make:migration reg

- This command creates a new migration file in the **database/migrations** directory. Open this file to define the schema for the “reg” table.
- If we skip the --create command, the default schema name will be the same as the migration name; otherwise, it will be assigned as specified with --create.

- **Note : To create a table, we need to know the data types required to define the table's attributes.**
  - **string('fieldname', length) : Used to define varchar field. Here length is optional parameter. if it is not mentioned by default to 255 ; other wise the specified value is used.**
  - **$table->text('fieldname') : Used to define text field**
  - **$table->integer('fieldname') : Used to define integer field**
  - **$table->date('fieldname') : Used to define date field**

- **$table->id()** : Create an auto-incrementing unsigned big int column named “id”

- **unique() and nullable()**

1. unique() : The unique() modifier ensures that all values in the column are unique, meaning no two rows can have the same value for this column. This is useful for fields like email addresses, usernames, or any other attribute where duplicate values should be prevented.

2. nullable(): The nullable() modifier allows the column to accept NULL values. By default, columns are not nullable unless explicitly specified. This is useful for optional fields where the absence of a value is acceptable.

- `$table->string('email')->unique(); // Ensures unique email addresses`
- `$table->string('middle_name')->nullable(); // Middle name can be NULL`
- `$table->text('feedback')->nullable(); // Middle name can be NULL`
- `$table->integer('user_id')->unique(); // Ensures unique user IDs`
- `$table->integer('age')->nullable(); // Age can be NULL`
- `$table->timestamp('email_verified_at')->nullable(); // Timestamp can be NULL`
- `$table->boolean('is_active')->nullable(); // Boolean value can be NULL`

Note:

- ✓ In the case of string and integer unique and nullable is used.
- ✓ In the case of text nullable only used.

– **default() :** In Laravel migrations, the default() method is used to set a default value for a column in a database table. This default value is assigned to the column if no explicit value is provided during the insertion of a new row.

```
$table->string('column_name')->default('default_value');
```

Note : You can use default() with data types such as string, integer, boolean, and date, but not with text.

database/migrations/2024_06_10_112302_laravel_user.php

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    public function up(): void
    {

        Schema::create('reg', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('email');
            $table->string('bio');
            $table->string('gender');
            $table->string('country');
            $table->string('language');
            $table->string('photo');

        });
    }

    public function down(): void
    {
        //
    }
};
```

Step 2.2: Create a Migration

To apply the migrations and create the database tables, run the following command:

```
php artisan migrate
```

Crud Operation : Insert

- **Step 1: Create a form for add/register**

resources/views/add.blade.php

```
<form action="{{ route('add_post') }}" method="POST" enctype="multipart/form-data">
    @csrf
    <fieldset>
        <legend>Personal Information</legend>
        <label for="name">Name:</label>
        <input type="text" id="name" name="name" required>
        <label for="email">Email:</label>
        <input type="email" id="email" name="email" required>
        <label for="bio">Biography:</label>
        <textarea id="bio" name="bio" rows="4"></textarea>
    </fieldset>
```

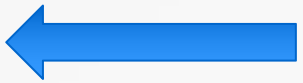
```
<fieldset>
    <legend>Preferences</legend>

    <label for="gender">Gender:</label>
    <input type="radio" id="male" name="gender" value="male">
    <label for="male" style="display: inline;">Male</label>
    <input type="radio" id="female" name="gender" value="female">
    <label for="female" style="display: inline;">Female</label>

    <label for="country">Country:</label>
    <select id="country" name="country">
        <option value="usa">United States</option>
        <option value="canada">Canada</option>
        <option value="uk">United Kingdom</option>
    </select>
</fieldset>
```

```
<fieldset>
        <legend>Languages Known</legend>
        <label for="languages">Select the languages you know:</label>
        <input type="checkbox" id="english" name="languages[]" value="English">
        <label for="english" style="display: inline;">English</label>
        <input type="checkbox" id="spanish" name="languages[]" value="Spanish">
        <label for="spanish" style="display: inline;">Spanish</label>
    </fieldset>
     <fieldset>
        <legend>Upload File</legend>
        <label for="file">Choose a file:</label>
        <input type="file" id="file" name="file">
     </fieldset>
      <div class="buttons">
        <button type="submit">Submit</button>
        <button type="reset">Reset</button>
    </div>
  </form>
```

- **Step 2: Create a route for fetching and saving data**

routes/web.php

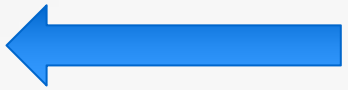
```
<?php

use Illuminate\Support\Facades\Route;

use Illuminate\Support\Facades\DB;

use Illuminate\Http\Request;

Route::get('/add', function (Request $request) {

    return view('add');

})->name('add');
```



Import DB for handing db functions

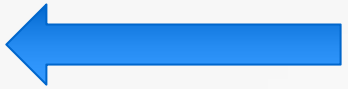
```
Route::post('/add_post', function (Request $request) {
    $photoPath = null;
    $name = $request->input('name');
    $email = $request->input('email');
    $bio = $request->input('bio');
    $gender = $request->input('gender');
    $country = $request->input('country');
    $languages = $request->input('languages',[]);
    if($request->hasFile('file')){
        $photoPath = $request->file('file')->store('photos','public') ;
    }
```

```
DB::statement(" INSERT INTO `login` (`username`, `password`, `usertype`)
    VALUES (?, ?, ?)",  [$email,'123','user'] );
```



method to inserted row.

```
    $id = DB::getPdo()->lastInsertId();
```



method to get the ID of the inserted row.

```
    DB::statement("
    INSERT INTO `reg` (id,`name`, `email`, `bio`, `gender`, `country`, `language`, `photo`)
    VALUES (?,?, ?, ?, ?, ?, ?, ?)",
    [$id,$name, $email, $bio, $gender, $country, json_encode($languages), $photoPath]
    );
    return "<script>alert('Added Successfully');window.location='/add'</script>";


})->name('add_post');
```

Note:

- By default, Laravel's store method stores files in the storage/app/public directory when you specify the public disk. To store the file directly in the public directory (which is typically public/ at the root of your Laravel project), you need to use a custom disk configuration.
- Other wise execute the command

```
php artisan storage:link
```

- If we do not execute the command, the image will not load.

Crud Operation : Read

- **resources/views/view.blade.php**

```
<table>
    <tr>
        <th>#</th>
        <th>Name</th>
        <th>Email</th>
        <th>Biography</th>
        <th>Gender</th>
        <th>Country</th>
        <th>Languages Known</th>
        <th>Photo</th>
        <th></th>
        <th></th>
    </tr>
```

```
@foreach($users as $user)
    <tr>
        <td>{{ $user->id }}</td>
        <td>{{ $user->name }}</td>
        <td>{{ $user->email }}</td>
        <td>{{ $user->bio }}</td>
        <td>{{ $user->gender }}</td>
        <td>{{ $user->country }}</td>
        <td>{{ implode(', ', json_decode($user->language)) }}</td>
```

Convert the resulting array into a readable string format with elements separated by commas.

to specify the default directory

```
<td><img src="{{ asset('storage/' . $user->photo ) }}" height="100" width="100"/></td>
```

```
  <td><a href="{{ route('edit', ['id' => $user->id]) }}" style="color:green">Edit</a></td>
   <td><a href="{{ route('delete_users', ['id' => $user->id]) }}" style="color:red">Delete</a>
</td>
            </tr>
        @endforeach
    </table>
```

routes/web.php

```
Route::get('/view_users', function (Request $request) {

    $users = DB::select('SELECT * FROM reg');
    return view('view',compact('users'));

})->name('view_users');
```

Crud Operation : Update

- **resources/views/edit.blade.php**

```
<form action="{{ route('edit_post',['id' => $user->id]) }}" method="POST" enctype="multipart/form-data">
    @csrf
    <fieldset>
        <legend>Personal Information</legend>
        <label for="name">Name:</label>
        <input type="text" id="name" name="name" required value="{{ $user->name }}">
        <label for="email">Email:</label>
        <input type="email" value="{{ $user->email }}" id="email" name="email" required>
        <label for="bio">Biography:</label>
        <textarea id="bio" name="bio" rows="4">{{ $user->bio }}</textarea>
    </fieldset>
```

```
 <fieldset><legend>Preferences</legend>  <label for="gender">Gender:</label>
<input type="radio" id="male" name="gender" value="male" {{ $user->gender == 'male' ? 'checked' : '' }}>
<label for="male" style="display: inline;">Male</label>
<input type="radio" id="female" name="gender" value="female" {{ $user->gender == 'female' ? 'checked' : '' }}>
        <label for="female" style="display: inline;">Female</label>
        <label for="country">Country:</label>
        <select id="country" name="country">
            <option value="usa" {{ $user->country == 'usa' ? 'selected' : '' }}>United States</option>
            <option value="canada" {{ $user->country == 'canada' ? 'selected' : '' }}>Canada</option>
            <option value="uk" {{ $user->country == 'uk' ? 'selected' : '' }}>United Kingdom</option>
        </select>
    </fieldset>
```

```
<fieldset>
        <legend>Languages Known</legend><label for="languages">Select the languages you know:</label>
<input type="checkbox" id="english" name="languages[]" value="English"  {{ in_array('English', json_decode($user->language)) ? 'checked' : '' }}>
        <label for="english" style="display: inline;">English</label>

        <input type="checkbox" id="spanish" name="languages[]" value="Spanish" {{ in_array('Spanish', json_decode($user->language)) ? 'checked' : '' }}>
        <label for="spanish" style="display: inline;">Spanish</label>

        <input type="checkbox" id="japanese" name="languages[]" value="Japanese" {{ in_array('Japanese', json_decode($user->language)) ? 'checked' : '' }}>
        <label for="japanese" style="display: inline;">Japanese</label>
    </fieldset>
```

```
<fieldset>
        <legend>Upload File</legend>


        <label for="file">Choose a file:</label>
        <img src="{{ asset('storage/' . $user->photo ) }}" height="100" width="100"/>
        <br>
        <input type="file" id="file" name="file">
    </fieldset>
    <div class="buttons">
        <button type="submit">Update</button>
        <button type="reset">Reset</button>
    </div>


</form>
```

routes/web.php

```
Route::get('/edit/{id}', function (Request $request,$id) {
    $user = DB::select('select * from reg where id = ?', [$id]);
    return view('edit',['user' => $user[0] ]);
})->name('edit');

Route::post('/edit_post/{id}', function (Request $request,$id) {
    $photoPath = null;
    $name = $request->input('name');
    $email = $request->input('email');
    $bio = $request->input('bio');
    $gender = $request->input('gender');
    $country = $request->input('country');
    $languages = $request->input('languages',[]);
```

Note : In Laravel, when you use DB::select() to fetch data from the database, it returns an array of results matching your query. Even if your query is expected to return only one row (like in your case where you're fetching a user by their ID), Laravel still returns an array containing that single result.

```
    if($request->hasFile('file')){
        $photoPath = $request->file('file')->store('photos','public') ;
        DB::update('UPDATE `reg` SET `name` = ?, `email` = ?, `bio` = ?, `gender` = ?, `country` = ?,
`language` = ?, `photo` = ? WHERE `id` = ?',
        [$name, $email, $bio, $gender, $country, json_encode($languages), $photoPath, $id]
        );
    }
    else{
        DB::update('UPDATE `reg` SET `name` = ?, `email` = ?, `bio` = ?, `gender` = ?, `country` = ?,
`language` = ? WHERE `id` = ?',
        [$name, $email, $bio, $gender, $country, json_encode($languages),   $id]
        );
    }
return "<script>alert('Updated Successfully');window.location='/view_users'</script>";

})->name('edit_post');
```

Crud Operation : Delete

routes/web.php

```
Route::get('/delete_users/{id}', function (Request $request,$id) {
      $users = DB::delete('delete from reg where id=?',[$id]);
    if ( $users > 0) {
                return "<script>alert('Deleted successfully');window.location='/view_users'</script>";
    } else {
                return "<script>alert('Some thing went wrong try again');window.location='/view_users'</script>";
    }
})->name('delete_users');
```

Session in Laravel

- In Laravel, a session refers to a way of persisting data across multiple HTTP requests.
- It enables you to store user-specific information and make it accessible throughout the user's browsing session on your application.

```
Session::put('key', 'value');

$value = Session::get('key');
```

routes/web.php

```
<?php
use Illuminate\Support\Facades\Route;
use Illuminate\Support\Facades\Session;
Route::get('/', function () {
    Session::put('name', 'Riss');
    return view('startingpage');
})->name('/');
Route::get('/url1', function () {
    $value = Session::get('name');
    return  $value;
})->name('url1');
```

views/startingpage.blade.php

```
<div>
    <a href = '/url1'> Click Here </a>
<div>
```

Note : We can also session in blade template

routes/web.php

```
<?php
use Illuminate\Support\Facades\Route;
use Illuminate\Support\Facades\Session;
Route::get('/', function () {
    Session::put('name', 'Riss');
    return view('startingpage');
})->name('/');
Route::get('/url1', function () {
    return view('startingpage2');
})->name('url1');
```

views/startingpage.blade.php

```
<div>
    <a href = '/url1'> Click Here </a>
<div>
```

views/startingpage.blade.php

```
<div>
    User is {{ Session('name') }}
 <div>
```

Custom Templating

❑Custom templating in Laravel with Blade empowers developers to create organized, reusable, and dynamic views effortlessly.

❑ It enhances code readability and promotes efficient front-end development practices.

Custom Template Design

- Download the template and extract it.
- Copy **css / js / image....** file and paste it into the **“public”** folder.

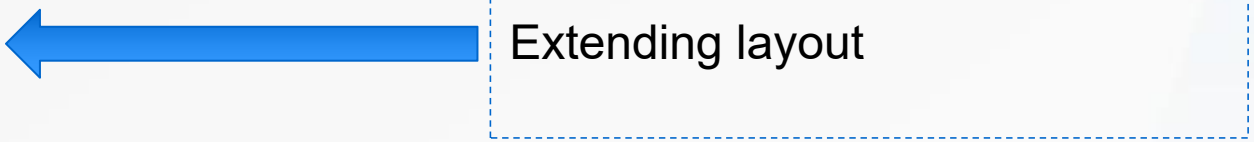
```
laravel_project_root/
├── app/
├── bootstrap/
├── config/
├── database/
├── public/
│   ├── css/
│   ├── js/
│   └── images
```

- Copy **index.html** and paste into “**resources/views/**” and rename to index.blade.php or otherwise another name.

views/index.blade.php

```
<!DOCTYPE html>
<html lang="en">
<head>
    <link href="{{ asset('lib/owlcarousel/assets/owl.carousel.min.css')}}" rel="stylesheet">
    <link href="{{ asset('lib/tempusdominus/css/tempusdominus-bootstrap-4.min.css')}}" rel="stylesheet" />
    <!-- Customized Bootstrap Stylesheet -->
    <link href="{{ asset('css/bootstrap.min.css')}}" rel="stylesheet">
    <!-- Template Stylesheet -->
    <link href="{{ asset('css/style.css')}}" rel="stylesheet">
</head>
```

```
<body>
.....................................................
<div class="container-fluid pt-4 px-4">
            <div class="bg-secondary text-center rounded p-4">
                <div class="d-flex align-items-center justify-content-between mb-4">

                </div>
                <div class="table-responsive">
                    @yield('content')
                </div>
            </div>
</div>
.....................................................
 <script src="{{ asset('lib/tempusdominus/js/moment-timezone.min.js') }}"></script>
    <script src="{{ asset('lib/tempusdominus/js/tempusdominus-bootstrap-4.min.js') }}"></script>
    <script src="{{ asset('js/main.js') }}"></script>
</body>
</html>
```



Note:

1) Using the asset helper function is not mandatory to load templates. If there is a style folder in the public directory, we can load the template directly. In case there is any issue loading styles directly, use the asset helper function.
2) You must use the yield function for extending the layout.

Login / Logout

- ✓ Laravel, a popular PHP framework, provides a simple and elegant way to interact with databases.
- ✓ It supports multiple database systems such as MySQL, PostgreSQL, SQLite, and SQL Server.

routes/web.php

```
<?php
use Illuminate\Support\Facades\Route;
use Illuminate\Support\Facades\DB;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Session;

Route::get('/', function () {
    return view('mylogin');
})->name('/');
```

```
Route::post('/log_post', function (Request $request) {
    $u = $request->input('email');
    $p = $request->input('password');
    $logindata = DB::select("select * from login where username = ? and password = ?", [$u,$p]);
    if (count($logindata) > 0){
        $logindata = $logindata[0];
        Session::put('login_id', $logindata->id);
        Session::put('username', $logindata->username);
        if( $logindata->usertype == 'admin'){
            return "<script>alert('Welcome to admin Home page');window.location='/adminhome'</script>";
        }
        elseif ($logindata->usertype == 'user') {
                    return "<script>alert('Welcome to user Home page');window.location='/myprofile'</script>";
        }
    }
    else{
        return "<script>alert('Invalid username/ Password');window.location='/'</script>";
    }
})->name('log_post');
```

Storing login details

```
Route::get('/adminhome', function (Request $request) {
    if (! $request->session()->has('login_id')) {
        return redirect('/');
    }
    return view('admin');
})->name('adminhome');
```

Checking whether the user is logged in or not

Middleware

- ✓ In Laravel, middleware is used to filter HTTP requests entering your application.
- ✓ Middleware acts as a bridge between a request and the response, handling tasks such as authentication, logging, and request manipulation.
- ✓ When it comes to login and logout processes, middleware plays an essential role in ensuring that these processes are secure and efficient.

Single user Login and Logout with middleware

- Middleware is a crucial aspect of handling login and logout processes in Laravel.
- It ensures that your application is secure, only authenticated users can access certain routes, and user sessions are managed effectively.
- By leveraging middleware, you can create a robust and secure authentication system in your Laravel application.

Step 1 : Create a project

- Create a new project

```
composer create-project laravel/laravel:^10 project-name
```

Note:

In some cases, Laravel misses some files in the latest version. Consider **downgrading** to Laravel **10**. In the latest version, some middleware files are missing.

Step 2: Install laravel/ui

- The laravel/ui package provides a quick and easy way to scaffold the frontend authentication views, routes, controllers, and more in a Laravel application.
- This package is particularly useful for setting up user authentication, including login, registration, password reset, and email verification views. It supports different frontend frameworks like Bootstrap, Vue, and React.

```
composer require laravel/ui
```

Step 3: Generate Bootstrap Auth Scaffolding

```
php artisan ui bootstrap --auth
```

- The command **php artisan ui bootstrap --auth** is used in Laravel to generate frontend scaffolding for authentication with Bootstrap styling.
- This command is part of the **laravel/ui** package, which provides simple authentication scaffolding for Laravel applications.
- When you run php artisan ui bootstrap --auth, Laravel generates the following:

- **Frontend Views**: Located in resources/views/auth and resources/views/layouts, the command creates views for login, register, password reset, and email verification, all styled with Bootstrap.
- **Authentication Routes**: The necessary routes for authentication are added to the routes/web.php file. These include routes for login, registration, password reset, and email verification.
- **Controllers**: Controllers for handling authentication logic are set up. These include:
  - LoginController
  - RegisterController
  - ForgotPasswordController
  - ResetPasswordController
  - ConfirmPasswordController
  - VerificationController
- **Blade Layout**: A basic Blade layout file using Bootstrap is created in resources/views/layouts/app.blade.php.

Step 4 : Install NPM Dependencies

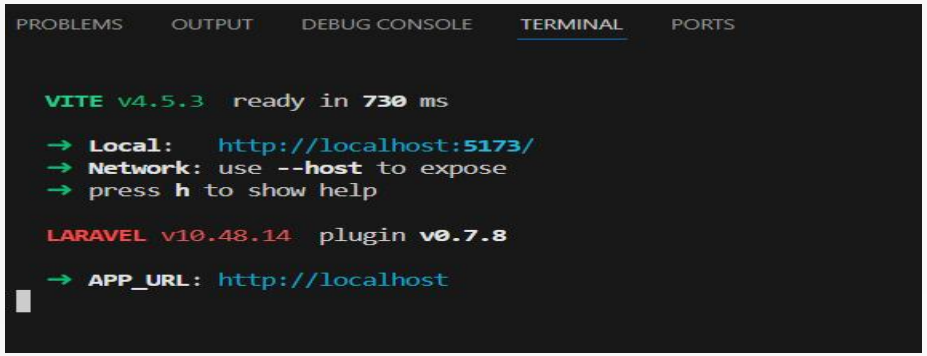
- Download Node.js from Official website and Install
- Open project directoty cmd and execute below command

```
npm install
```

- Compile the Assets:

```
npm run dev
```

- To stop npm run dev, press **Ctrl+C** when you see a screen like the one below:



- Then execute below command for ensuring a smooth and standardized build process for your Laravel project's frontend assets

```
npm run build
```

Step 5 : Run Migrations

```
php artisan migrate
```

- Ensure your database is set up, then run the migrations to create the necessary table.
- Refer Slide 125 - 126

Step 6 : Serve the Application

```
php artisan serve
```

- After running these steps, you'll have a working authentication system styled with Bootstrap in your Laravel application.
- This setup includes all necessary views, routes, and controllers for handling user authentication, registration, and password resets, providing a solid foundation for further customization.

Link

- Download Composer : Click Here
- Download Xaamp : Click Here
- Connecting With DB : Click Here
- Download Node : Click Here
- Single User authentication : Click Here

THANKS FOR WATCHING