***MODULE – (Introduction of Laravel)***

**1) Create Following Routes.(for frontend side application)**

**a) Home page**

**b) About us**

**c) Contact us**

**d) Gallery**

**e) Registration**

**f) Login**

**Ans.** Creating routes for a frontend application in PHP typically involves setting up a simple PHP framework or using plain PHP with a router. Below is a basic example of how you can set up routes for your application using plain PHP.

**Project Structure**

Here’s how your project might look:

php-app/

├── index.php

├── about.php

├── contact.php

├── gallery.php

├── register.php

└── login.php

**Step 1: Create the Entry Point**

**index.php**

This file will handle routing based on the URL.

<?php

// Start session if needed

session\_start();

// Get the requested URL

$requestUri = parse\_url($\_SERVER['REQUEST\_URI'], PHP\_URL\_PATH);

$requestUri = trim($requestUri, '/');

// Route to different pages

switch ($requestUri) {

case '':

case 'home':

include 'home.php';

break;

case 'about':

include 'about.php';

break;

case 'contact':

include 'contact.php';

break;

case 'gallery':

include 'gallery.php';

break;

case 'register':

include 'register.php';

break;

case 'login':

include 'login.php';

break;

default:

http\_response\_code(404);

include '404.php'; // You can create a custom 404 page

break;

}

?>

**Step 2: Create Page Files**

You will create individual PHP files for each route.

**home.php**:

<h1>Home Page</h1>

<p>Welcome to the home page!</p>

**about.php**:

<h1>About Us</h1>

<p>This is the about us page.</p>

**contact.php**:

<h1>Contact Us</h1>

<p>This is the contact us page.</p>

**gallery.php**:

<h1>Gallery</h1>

<p>This is the gallery page.</p>

**register.php**:

<h1>Registration</h1>

<p>This is the registration page.</p>

**login.php**:

<h1>Login</h1>

<p>This is the login page.</p>

**404.php** (optional):

<h1>404 Not Found</h1>

<p>The page you are looking for does not exist.</p>

**Step 3: .htaccess for URL Rewriting**

If you want to use clean URLs (without index.php), create an .htaccess file in your project root:

**.htaccess**

<IfModule mod\_rewrite.c>

RewriteEngine On

RewriteRule ^(.\*)$ index.php [QSA,L]

</IfModule>

**Summary**

With this setup, your application will respond to the following URLs:

* **Home Page**: /home or /
* **About Us**: /about
* **Contact Us**: /contact
* **Gallery**: /gallery
* **Registration**: /register
* **Login**: /login

You can further enhance each page with HTML, CSS, and PHP logic as needed!

**2) How to pass Multiple Variable in route?**

**Ans.** Passing multiple variables in a route can be achieved in various ways depending on your setup. Below, I'll show you how to handle this in a PHP application, both with a query string and using URL parameters.

**Method 1: Using Query Strings**

You can pass multiple variables using query strings in the URL. For example, a URL could look like this:

/page.php?param1=value1&param2=value2

**Example**

**index.php**

<?php

// Start session if needed

session\_start();

// Get the requested URL

$requestUri = parse\_url($\_SERVER['REQUEST\_URI'], PHP\_URL\_PATH);

$requestUri = trim($requestUri, '/');

// Check for route with query parameters

if ($requestUri === 'page.php' && isset($\_GET['param1']) && isset($\_GET['param2'])) {

$param1 = $\_GET['param1'];

$param2 = $\_GET['param2'];

echo "Param 1: " . htmlspecialchars($param1) . "<br>";

echo "Param 2: " . htmlspecialchars($param2);

} else {

echo "Invalid route or missing parameters.";

}

?>

**URL**

http://yourdomain.com/page.php?param1=value1&param2=value2

**Method 2: Using URL Parameters**

You can also pass multiple variables directly in the URL path. For example:

/page/value1/value2

**Example**

**index.php**

<?php

session\_start();

// Get the requested URL

$requestUri = parse\_url($\_SERVER['REQUEST\_URI'], PHP\_URL\_PATH);

$requestUri = trim($requestUri, '/');

// Route handling

if (preg\_match('/^page\/([^\/]+)\/([^\/]+)$/', $requestUri, $matches)) {

$param1 = $matches[1];

$param2 = $matches[2];

echo "Param 1: " . htmlspecialchars($param1) . "<br>";

echo "Param 2: " . htmlspecialchars($param2);

} else {

echo "Invalid route.";

}

?>

**URL**

http://yourdomain.com/page/value1/value2

**Using .htaccess for Clean URLs**

If you're using the URL parameter method and want to avoid index.php in the URL, you can use an .htaccess file:

**.htaccess**

<IfModule mod\_rewrite.c>

RewriteEngine On

RewriteRule ^page/([^/]+)/([^/]+)$ index.php [QSA,L]

</IfModule>

**Summary**

* **Query Strings**: Pass variables via ?param1=value1&param2=value2.
* **URL Parameters**: Pass variables in the path, like /page/value1/value2.
* Use regular expressions to capture parameters in the URL.

These methods allow you to effectively manage multiple variables in your routes!

**3) How to pass variable which can be null in Route?**

**Ans.** In PHP, when working with routes (for example, in frameworks like Laravel, Symfony, or Slim), you can pass a variable that can be null by defining optional route parameters. Here’s how you can do it in a few popular frameworks:

**Laravel**

In Laravel, you can define optional parameters in your routes using a ? after the parameter name:

Route::get('/user/{id?}', function ($id = null) {

if ($id === null) {

return 'No user ID provided.';

}

return 'User ID: ' . $id;

});

**Symfony**

In Symfony, you can specify optional parameters in your routing configuration:

# config/routes.yaml

user:

path: /user/{id}

controller: App\Controller\UserController::show

defaults:

id: null

In your controller, you can handle the null value:

public function show($id = null)

{

if ($id === null) {

return new Response('No user ID provided.');

}

return new Response('User ID: ' . $id);

}

**Slim Framework**

In Slim, you can define optional parameters directly in the route:

$app->get('/user[/{id}]', function ($request, $response, $args) {

$id = $args['id'] ?? null; // Default to null if not provided

if ($id === null) {

return $response->write('No user ID provided.');

}

return $response->write('User ID: ' . $id);

});

**General PHP Approach**

If you’re not using a framework, you can handle query parameters directly:

$id = $\_GET['id'] ?? null; // This will set $id to null if not provided

if ($id === null) {

echo 'No user ID provided.';

} else {

echo 'User ID: ' . htmlspecialchars($id);

}

**Summary**

1. **Define optional route parameters** in your framework's routing configuration.
2. **Handle the null value** appropriately in your controller or route callback.

This approach ensures that your application can handle cases where the variable might not be passed in the route.

**4) Create custom auto using middleware.**

**Ans.** Creating a custom auto middleware in a PHP framework like Laravel involves defining a middleware class and then registering it. Here’s a step-by-step guide for Laravel, but the concept can be adapted for other frameworks as well.

**Step 1: Create Middleware**

First, you can create a middleware using the Artisan command line:

php artisan make:middleware CustomAutoMiddleware

This command creates a new middleware file in the app/Http/Middleware directory.

**Step 2: Implement Middleware Logic**

Open the newly created middleware file (app/Http/Middleware/CustomAutoMiddleware.php) and add your custom logic. Here’s a basic example:

<?php

namespace App\Http\Middleware;

use Closure;

use Illuminate\Http\Request;

class CustomAutoMiddleware

{

/\*\*

\* Handle an incoming request.

\*

\* @param \Illuminate\Http\Request $request

\* @param \Closure $next

\* @return mixed

\*/

public function handle(Request $request, Closure $next)

{

// Custom logic before the request is handled

if (!$request->user()) {

return response()->json(['error' => 'Unauthorized'], 401);

}

// Continue to the next middleware or request handler

return $next($request);

}

}

In this example, the middleware checks if a user is authenticated. If not, it returns a JSON response indicating an unauthorized access.

**Step 3: Register Middleware**

You need to register your middleware in app/Http/Kernel.php. You can add it to the routeMiddleware array for use in specific routes or to the globalMiddleware array if you want it to run on all requests.

protected $routeMiddleware = [

// Other middleware...

'custom.auto' => \App\Http\Middleware\CustomAutoMiddleware::class,

];

**Step 4: Use Middleware in Routes**

You can now use your middleware in your routes. For example, in your routes/web.php or routes/api.php file:

Route::middleware(['custom.auto'])->group(function () {

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

return 'User profile';

});

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

return 'Settings page';

});

});

**Step 5: Testing the Middleware**

You can test your middleware by accessing the routes you defined. If the user is not authenticated, you should receive the JSON response from your middleware.

**Example with Other Frameworks**

The concept is similar in other PHP frameworks:

* **Symfony**: Create a service, configure it in services.yaml, and use it in your controller or route annotations.
* **Slim**: Create a middleware class and add it to your application instance using $app->add().

**Summary**

1. Create a middleware class using the framework's CLI.
2. Implement the logic in the handle method.
3. Register the middleware in the kernel or middleware configuration.
4. Apply it to your routes.

This gives you a reusable piece of functionality that can be applied across your application! If you have a specific use case in mind, feel free to ask!

**5) Generate Resource Controller for employee.**

**Ans.** Creating a resource controller for an Employee model in PHP, particularly using the Laravel framework, involves generating a controller that can handle standard CRUD (Create, Read, Update, Delete) operations. Here’s a simple guide on how to create an Employee resource controller.

**Step 1: Generate the Controller**

Run the following Artisan command in your terminal to generate a resource controller named EmployeeController:

php artisan make:controller EmployeeController --resource

**Step 2: Define Routes**

Next, you'll want to define the routes for this controller. Open the routes/web.php file (or routes/api.php for API routes) and add the following line:

Route::resource('employees', EmployeeController::class);

**Step 3: Implement the Controller Methods**

Open the generated EmployeeController.php file located in the app/Http/Controllers directory. Here's a basic implementation of the CRUD methods:

<?php

namespace App\Http\Controllers;

use App\Models\Employee; // Make sure to import the Employee model

use Illuminate\Http\Request;

class EmployeeController extends Controller

{

// Display a listing of the employees

public function index()

{

$employees = Employee::all();

return view('employees.index', compact('employees'));

}

// Show the form for creating a new employee

public function create()

{

return view('employees.create');

}

// Store a newly created employee in storage

public function store(Request $request)

{

$request->validate([

'name' => 'required|string|max:255',

'email' => 'required|email|unique:employees,email',

'position' => 'required|string|max:255',

]);

Employee::create($request->all());

return redirect()->route('employees.index')->with('success', 'Employee created successfully.');

}

// Display the specified employee

public function show(Employee $employee)

{

return view('employees.show', compact('employee'));

}

// Show the form for editing the specified employee

public function edit(Employee $employee)

{

return view('employees.edit', compact('employee'));

}

// Update the specified employee in storage

public function update(Request $request, Employee $employee)

{

$request->validate([

'name' => 'required|string|max:255',

'email' => 'required|email|unique:employees,email,' . $employee->id,

'position' => 'required|string|max:255',

]);

$employee->update($request->all());

return redirect()->route('employees.index')->with('success', 'Employee updated successfully.');

}

// Remove the specified employee from storage

public function destroy(Employee $employee)

{

$employee->delete();

return redirect()->route('employees.index')->with('success', 'Employee deleted successfully.');

}

}

**Step 4: Create the Employee Model**

If you haven't already, you can create the Employee model using the following command:

php artisan make:model Employee -m

The -m option creates a migration file as well. Update the migration file located in database/migrations/ to define the schema for the employees table:

public function up()

{

Schema::create('employees', function (Blueprint $table) {

$table->id();

$table->string('name');

$table->string('email')->unique();

$table->string('position');

$table->timestamps();

});

}

Run the migration with:

php artisan migrate

**Step 5: Create Views (Optional)**

You will need to create corresponding views for each method (e.g., index, create, edit, etc.) inside a directory called employees. Here’s an example for the index.blade.php view:

{{-- resources/views/employees/index.blade.php --}}

@extends('layouts.app')

@section('content')

<div class="container">

<h1>Employees</h1>

<a href="{{ route('employees.create') }}" class="btn btn-primary">Add Employee</a>

<table class="table">

<thead>

<tr>

<th>Name</th>

<th>Email</th>

<th>Position</th>

<th>Actions</th>

</tr>

</thead>

<tbody>

@foreach ($employees as $employee)

<tr>

<td>{{ $employee->name }}</td>

<td>{{ $employee->email }}</td>

<td>{{ $employee->position }}</td>

<td>

<a href="{{ route('employees.edit', $employee) }}" class="btn btn-warning">Edit</a>

<form action="{{ route('employees.destroy', $employee) }}" method="POST" style="display:inline;">

@csrf

@method('DELETE')

<button type="submit" class="btn btn-danger">Delete</button>

</form>

</td>

</tr>

@endforeach

</tbody>

</table>

</div>

@endsection

**Conclusion**

You now have a complete resource controller for managing employees in a Laravel application. Make sure to customize the views and validation rules according to your needs!

**6) In Employee Controller’s action Call Middleware.**

**Ans.** To call middleware in your EmployeeController actions in Laravel, you can apply middleware to specific controller methods or to the entire controller. Here's how you can do both.

**Step 1: Create Middleware**

If you don’t already have middleware created, you can create it using the Artisan command:

php artisan make:middleware CheckEmployeeAccess

In the generated middleware located at app/Http/Middleware/CheckEmployeeAccess.php, you can define your logic. For example:

<?php

namespace App\Http\Middleware;

use Closure;

use Illuminate\Http\Request;

class CheckEmployeeAccess

{

public function handle(Request $request, Closure $next)

{

// Your logic here (e.g., check if the user has access)

if (!auth()->user() || !auth()->user()->canAccessEmployees()) {

return redirect()->route('home')->with('error', 'Access denied.');

}

return $next($request);

}

}

**Step 2: Register Middleware**

Next, you need to register your middleware in app/Http/Kernel.php. You can register it either as a route middleware or as a global middleware. Here’s how to register it as route middleware:

protected $routeMiddleware = [

// Other middleware

'checkEmployeeAccess' => \App\Http\Middleware\CheckEmployeeAccess::class,

];

**Step 3: Apply Middleware to Controller Methods**

Now, you can apply the middleware to specific actions in your EmployeeController. Open EmployeeController.php and add the middleware in the constructor:

<?php

namespace App\Http\Controllers;

use App\Models\Employee;

use Illuminate\Http\Request;

class EmployeeController extends Controller

{

public function \_\_construct()

{

// Apply middleware to all methods

$this->middleware('checkEmployeeAccess')->except(['index', 'show']);

}

// The rest of your controller methods...

}

In this example, the checkEmployeeAccess middleware will be applied to all methods except for index and show, meaning only authenticated users with the right permissions can create, edit, or delete employees.

**Step 4: Applying Middleware to Specific Actions**

If you want to apply the middleware to specific methods instead, you can do so like this:

public function \_\_construct()

{

// Apply middleware only to the store, update, and destroy methods

$this->middleware('checkEmployeeAccess')->only(['store', 'update', 'destroy']);

}

**Conclusion**

You now have middleware applied to your EmployeeController actions. This will help ensure that only authorized users can perform certain actions, enhancing the security of your application. Feel free to customize the middleware logic according to your application's requirements!

**7) How to remove route caching?**

**Ans.** To remove route caching in Laravel, you can follow these steps:

**Step 1: Clear Route Cache**

To clear the route cache, use the following Artisan command:

php artisan route:clear

This command removes the cached routes from the application.

**Step 2: Optional: Clear Other Caches**

If you're having issues related to cached views or configurations, you might also want to clear those caches. Here are the commands for that:

* **Clear Application Cache:**

php artisan cache:clear

* **Clear Config Cache:**

php artisan config:clear

* **Clear View Cache:**

php artisan view:clear

**Step 3: Verify Changes**

After clearing the caches, you can verify that your routes are updated. You can check your routes by running:

php artisan route:list

This will display the current routes that are available in your application, reflecting any changes made.

**Summary**

To remove route caching, simply use php artisan route:clear. For comprehensive maintenance, consider clearing other caches as needed. This helps ensure that your application is running with the most up-to-date information.

**8) Create Custom Macro For search User**

**Ans.** Creating a custom macro in Laravel for searching users involves defining a reusable piece of functionality that can be used throughout your application. Here’s how to create a custom macro for searching users in a User model.

**Step 1: Define the Macro**

You typically define custom macros in the AppServiceProvider or create a dedicated service provider. For this example, we'll add it to the AppServiceProvider.

1. Open app/Providers/AppServiceProvider.php.
2. In the boot method, define the macro for searching users.

Here's an example of how you might implement this:

<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

use App\Models\User;

class AppServiceProvider extends ServiceProvider

{

public function boot()

{

// Define a custom macro for searching users

User::macro('search', function ($query) {

return $this->where('name', 'like', "%{$query}%")

->orWhere('email', 'like', "%{$query}%");

});

}

public function register()

{

//

}

}

**Step 2: Use the Macro**

Now you can use this macro anywhere in your application to search for users. For example, in a controller:

<?php

namespace App\Http\Controllers;

use App\Models\User;

use Illuminate\Http\Request;

class UserController extends Controller

{

public function search(Request $request)

{

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

// Use the custom macro to search for users

$users = User::search($query)->get();

return view('users.index', compact('users'));

}

}

**Step 3: Define a Route**

You’ll need to define a route for the search action in your routes/web.php or routes/api.php file:

use App\Http\Controllers\UserController;

Route::get('/users/search', [UserController::class, 'search'])->name('users.search');

**Step 4: Create a Search Form (Optional)**

You may also want to create a form to input the search query. Here’s a simple example in a Blade view:

{{-- resources/views/users/search.blade.php --}}

<form action="{{ route('users.search') }}" method="GET">

<input type="text" name="query" placeholder="Search users..." required>

<button type="submit">Search</button>

</form>

**Conclusion**

You've now created a custom macro for searching users in Laravel. This allows you to encapsulate the search logic in one place and reuse it throughout your application. Adjust the search conditions and fields as needed to suit your application's requirements!

**Practical:**

**1) Create Photo Gallery for above route.**

**Note: upload files from admin side. Show all the images in user side**

**Ans.** Creating a photo gallery in Laravel involves several steps, including file uploads from the admin side and displaying the images on the user side. Below is a step-by-step guide to help you set this up.

**Step 1: Create Migration for Photos**

First, you’ll need to create a migration for storing photo information. Run the following command:

php artisan make:migration create\_photos\_table

In the generated migration file, define the structure of the photos table:

<?php

use Illuminate\Database\Migrations\Migration;

use Illuminate\Database\Schema\Blueprint;

use Illuminate\Support\Facades\Schema;

class CreatePhotosTable extends Migration

{

public function up()

{

Schema::create('photos', function (Blueprint $table) {

$table->id();

$table->string('filename');

$table->timestamps();

});

}

public function down()

{

Schema::dropIfExists('photos');

}

}

Run the migration:

php artisan migrate

**Step 2: Create Photo Model**

Next, create a model for the photos table:

php artisan make:model Photo

**Step 3: Create Controller for Photo Gallery**

php artisan make:controller PhotoController

In PhotoController.php, add methods for uploading and displaying photos:

<?php

namespace App\Http\Controllers;

use App\Models\Photo;

use Illuminate\Http\Request;

class PhotoController extends Controller

{

// Show upload form

public function create()

{

return view('photos.create');

}

// Store the uploaded photo

public function store(Request $request)

{

$request->validate([

'photo' => 'required|image|mimes:jpeg,png,jpg,gif|max:2048',

]);

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

$file = $request->file('photo');

$filename = time() . '.' . $file->getClientOriginalExtension();

$file->storeAs('photos', $filename, 'public');

Photo::create(['filename' => $filename]);

}

return redirect()->route('photos.index')->with('success', 'Photo uploaded successfully.');

}

// Display all photos

public function index()

{

$photos = Photo::all();

return view('photos.index', compact('photos'));

}

}

**Step 4: Define Routes**

Add the routes for the photo gallery in routes/web.php:

use App\Http\Controllers\PhotoController;

Route::get('/photos/create', [PhotoController::class, 'create'])->name('photos.create');

Route::post('/photos', [PhotoController::class, 'store'])->name('photos.store');

Route::get('/photos', [PhotoController::class, 'index'])->name('photos.index');

**Step 5: Create Views**

**Create Photo Upload View**

Create a view file at resources/views/photos/create.blade.php:

@extends('layouts.app')

@section('content')

<div class="container">

<h1>Upload Photo</h1>

<form action="{{ route('photos.store') }}" method="POST" enctype="multipart/form-data">

@csrf

<div class="form-group">

<label for="photo">Choose Photo</label>

<input type="file" name="photo" class="form-control" required>

</div>

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

</form>

</div>

@endsection

**Create Photo Gallery View**

Create another view file at resources/views/photos/index.blade.php:

@extends('layouts.app')

@section('content')

<div class="container">

<h1>Photo Gallery</h1>

<div class="row">

@foreach ($photos as $photo)

<div class="col-md-4">

<div class="card mb-4">

<img src="{{ asset('storage/photos/' . $photo->filename) }}" class="card-img-top" alt="Photo">

</div>

</div>

@endforeach

</div>

</div>

@endsection

**Step 6: Configure Filesystem**

Ensure that the filesystem is set up correctly for file storage. In config/filesystems.php, make sure the public disk is configured like this:

'disks' => [

'public' => [

'driver' => 'local',

'root' => storage\_path('app/public'),

'url' => env('APP\_URL') . '/storage',

'visibility' => 'public',

],

// other disks...

],

Run the command to create a symbolic link for accessing the uploaded files:

php artisan storage:link

**Step 7: Test the Application**

1. Navigate to /photos/create to upload photos from the admin side.
2. After uploading, you can view the uploaded photos at /photos.

**Conclusion**

You now have a basic photo gallery setup in Laravel, where an admin can upload photos, and users can view all uploaded images. You can further enhance this by adding features such as deleting photos, editing metadata, or implementing pagination.

**2) Download admin panel for employee management and convert it in Laravel blades.**

**Ans.** To create an employee management admin panel using Laravel and convert it into Blade templates, you can follow these steps:

**Step 1: Set Up Your Laravel Project**

1. **Install Laravel**: Make sure you have Composer installed. Open your terminal and run:

composer create-project --prefer-dist laravel/laravel employee-management

cd employee-management

1. **Set Up Environment**: Update the .env file with your database connection details.

**Step 2: Create the Employee Model and Migration**

1. **Create Model and Migration**:

php artisan make:model Employee -m

1. **Update Migration**: Open the migration file in database/migrations/ and define your employee table structure:

public function up()

{

Schema::create('employees', function (Blueprint $table) {

$table->id();

$table->string('name');

$table->string('email')->unique();

$table->string('position');

$table->timestamps();

});

}

1. **Run Migration**:

php artisan migrate

**Step 3: Create a Controller**

1. **Create Controller**:

php artisan make:controller EmployeeController

1. **Define CRUD Methods**:

Open app/Http/Controllers/EmployeeController.php and add methods for handling employees:

use App\Models\Employee;

use Illuminate\Http\Request;

public function index()

{

$employees = Employee::all();

return view('employees.index', compact('employees'));

}

public function create()

{

return view('employees.create');

}

public function store(Request $request)

{

$request->validate([

'name' => 'required',

'email' => 'required|email|unique:employees',

'position' => 'required',

]);

Employee::create($request->all());

return redirect()->route('employees.index');

}

public function edit(Employee $employee)

{

return view('employees.edit', compact('employee'));

}

public function update(Request $request, Employee $employee)

{

$request->validate([

'name' => 'required',

'email' => 'required|email|unique:employees,email,' . $employee->id,

'position' => 'required',

]);

$employee->update($request->all());

return redirect()->route('employees.index');

}

public function destroy(Employee $employee)

{

$employee->delete();

return redirect()->route('employees.index');

}

**Step 4: Define Routes**

Open routes/web.php and add the following routes:

use App\Http\Controllers\EmployeeController;

Route::resource('employees', EmployeeController::class);

**Step 5: Create Blade Views**

Create a folder named employees in the resources/views directory. Inside that folder, create the following Blade files:

1. **index.blade.php**:

@extends('layouts.app')

@section('content')

<h1>Employees</h1>

<a href="{{ route('employees.create') }}">Add Employee</a>

<table>

<tr>

<th>Name</th>

<th>Email</th>

<th>Position</th>

<th>Actions</th>

</tr>

@foreach($employees as $employee)

<tr>

<td>{{ $employee->name }}</td>

<td>{{ $employee->email }}</td>

<td>{{ $employee->position }}</td>

<td>

<a href="{{ route('employees.edit', $employee) }}">Edit</a>

<form action="{{ route('employees.destroy', $employee) }}" method="POST" style="display:inline;">

@csrf

@method('DELETE')

<button type="submit">Delete</button>

</form>

</td>

</tr>

@endforeach

</table>

@endsection

1. **create.blade.php**:

@extends('layouts.app')

@section('content')

<h1>Add Employee</h1>

<form action="{{ route('employees.store') }}" method="POST">

@csrf

<input type="text" name="name" placeholder="Name" required>

<input type="email" name="email" placeholder="Email" required>

<input type="text" name="position" placeholder="Position" required>

<button type="submit">Add Employee</button>

</form>

@endsection

1. **edit.blade.php**:

@extends('layouts.app')

@section('content')

<h1>Edit Employee</h1>

<form action="{{ route('employees.update', $employee) }}" method="POST">

@csrf

@method('PUT')

<input type="text" name="name" value="{{ $employee->name }}" required>

<input type="email" name="email" value="{{ $employee->email }}" required>

<input type="text" name="position" value="{{ $employee->position }}" required>

<button type="submit">Update Employee</button>

</form>

@endsection

**Step 6: Create a Layout**

Create a layouts folder in resources/views, then create app.blade.php:

<!DOCTYPE html>

<html>

<head>

<title>Employee Management</title>

</head>

<body>

<div class="container">

@yield('content')

</div>

</body>

</html>

**Step 7: Run the Application**

You can start your Laravel application with:

php artisan serve

Visit http://127.0.0.1:8000/employees to see your employee management panel in action.

**Conclusion**

This is a basic employee management admin panel using Laravel and Blade templates. You can enhance it further by adding features like authentication, validation messages, and front-end styling with Bootstrap or Tailwind CSS. Let me know if you need help with anything specific!