## ****Module 6: Laravel Fundamentals [Routes, Controllers, Views & Models]****

### 📌 ****Course Overview****

This module introduces students to the core components of the Laravel PHP framework, focusing on building web applications using Laravel’s MVC (Model-View-Controller) architecture. It covers fundamental concepts such as defining routes, creating and managing controllers, utilizing Blade templating in views, and designing models to interact with databases. By the end of this module, students will be able to construct dynamic, structured, and secure Laravel-based applications using best development practices.

**🔀 Understanding Laravel Routes**

In Laravel, **routes** define the entry points for web requests. When a user visits a URL, Laravel determines which code to execute based on the defined route.

All routes are defined in files located in the routes/ directory, with the most commonly used one being:

* **routes/web.php** – For routes that return views or HTML (intended for web browsers).

Laravel uses the **MVC (Model-View-Controller)** pattern, and routes typically direct requests to **controllers**, where the logic lives.

**✅ How to Define Routes**

**1. Basic Route Example**

use Illuminate\Support\Facades\Route;

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

return view('welcome');

});

* **Route Method**: get() – handles GET requests.
* **URL Path**: '/' – the root of the site.
* **Closure**: The anonymous function that returns a view.

**2. Route with Controller**

Instead of defining logic in a closure, you can route to a controller method:

Route::get('/about', [PageController::class, 'about']);

* Laravel will look for the about method inside PageController.

**3. POST, PUT, DELETE Routes**

Laravel supports all HTTP verbs. Examples:

Route::post('/submit', [FormController::class, 'submit']);

Route::put('/update/{id}', [DataController::class, 'update']);

Route::delete('/delete/{id}', [DataController::class, 'delete']);

**🗂️ Organizing Routes Effectively**

**1. Named Routes**

Naming routes gives them a convenient alias for redirection and form actions.

Route::get('/contact', [PageController::class, 'contact'])->name('contact.page');

Use in Blade:

<a href="{{ route('contact.page') }}">Contact Us</a>

**2. Route Parameters**

**Required Parameters**

Route::get('/user/{id}', [UserController::class, 'show']);

**Optional Parameters**

Route::get('/user/{name?}', [UserController::class, 'profile']);

**3. Route Grouping**

**With Middleware**

Route::middleware(['auth'])->group(function () {

Route::get('/dashboard', [DashboardController::class, 'index']);

});

**With Prefix**

Route::prefix('admin')->group(function () {

Route::get('/users', [AdminController::class, 'users']);

});

URL becomes /admin/users.

**With Namespace (Laravel 7 and below)**

Route::namespace('Admin')->group(function () {

Route::get('/dashboard', 'DashboardController@index');

});

**4. Resource Routes**

Quickly generate a full set of CRUD routes for a controller:

Route::resource('posts', PostController::class);

This single line creates 7 routes: index, create, store, show, edit, update, and destroy.

Use:

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

**🔐 Route Middleware**

Routes can be protected with middleware like auth:

Route::get('/profile', [UserController::class, 'profile'])->middleware('auth');

**📦 Route Caching (Optimization)**

In production, cache routes for faster performance:

php artisan route:cache

Clear cache:

php artisan route:clear

**🧠 Best Practices for Laravel Routing**

* **Use named routes** for maintainability.
* **Group routes** by functionality or access level (e.g., admin, public).
* **Apply middleware** to protect sensitive routes.
* **Keep route files clean**; push logic to controllers.
* **Use RESTful resource routes** when appropriate.
* **Validate route parameters** using Route::where() constraints if needed.

Would you like a diagram or code scaffold of a complete Laravel routing file for a sample project (like a blog or e-commerce site)?

Absolutely! Let's dive deep into how to **develop and manage controllers in Laravel** to handle user input and encapsulate business logic effectively.

**🧠 What is a Controller in Laravel?**

In Laravel's MVC (Model-View-Controller) architecture, a **controller** is responsible for:

* Receiving requests from routes.
* Handling user input (often via forms or APIs).
* Interacting with models to retrieve or store data.
* Returning responses (typically via views or JSON).

Think of a controller as the "traffic controller" — it coordinates the flow between the user interface (views) and the database (models).

**✍️ Creating a Controller**

You can create a controller using the Artisan CLI:

php artisan make:controller UserController

This generates a controller in app/Http/Controllers/UserController.php.

**📂 Types of Controllers**

**1. Basic Controller**

class UserController extends Controller

{

public function show($id)

{

$user = User::findOrFail($id);

return view('users.profile', compact('user'));

}

}

**2. Single-Action Controller**

Sometimes you want a controller to handle just one task:

php artisan make:controller ContactController --invokable

Usage:

class ContactController extends Controller

{

public function \_\_invoke()

{

return view('contact');

}

}

Route:

Route::get('/contact', ContactController::class);

**3. Resource Controller**

Generates all standard CRUD methods:

php artisan make:controller ProductController --resource

This will generate:

| **Method** | **HTTP Verb** | **Route** |
| --- | --- | --- |
| index() | GET | /products |
| create() | GET | /products/create |
| store() | POST | /products |
| show($id) | GET | /products/{id} |
| edit($id) | GET | /products/{id}/edit |
| update($id) | PUT/PATCH | /products/{id} |
| destroy($id) | DELETE | /products/{id} |

Register with:

Route::resource('products', ProductController::class);

**📝 Processing User Input**

**1. Retrieving Input**

In a controller, you can get form input via the Request object:

use Illuminate\Http\Request;

public function store(Request $request)

{

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

}

Or simply:

$name = $request->name;

**2. Validating Input**

Before processing, always validate user input:

$request->validate([

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

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

]);

Laravel automatically redirects back with errors if validation fails.

**3. Handling Form Submissions**

public function store(Request $request)

{

$validated = $request->validate([

'title' => 'required|string|max:100',

'body' => 'required',

]);

Post::create($validated);

return redirect()->route('posts.index')->with('success', 'Post created!');

}

**🧠 Managing Business Logic**

Controllers should delegate **complex logic** to **service classes** or **model methods**.

Instead of:

public function sendNewsletter(Request $request)

{

$users = User::all();

foreach ($users as $user) {

Mail::to($user->email)->send(new NewsletterMail());

}

}

Better:

public function sendNewsletter()

{

NewsletterService::sendToAllUsers();

}

This keeps controllers clean and maintainable.

**🔐 Middleware in Controllers**

You can apply middleware to controller methods:

class DashboardController extends Controller

{

public function \_\_construct()

{

$this->middleware('auth')->only(['index', 'settings']);

}

}

**🗃️ Grouping Controllers by Folder (Namespace)**

Organize large projects:

php artisan make:controller Admin/UserController

Route:

Route::get('/admin/users', [App\Http\Controllers\Admin\UserController::class, 'index']);

**🔁 Redirecting and Flash Messages**

After completing a task, redirect:

return redirect()->route('dashboard')->with('success', 'Profile updated!');

Display flash message in Blade:

@if(session('success'))

<div class="alert alert-success">{{ session('success') }}</div>

@endif

**✅ Best Practices for Laravel Controllers**

* Keep controllers **thin** and focused.
* Use **form requests** for validation (php artisan make:request).
* Avoid business logic in controllers — use **services** or **model scopes**.
* Use **resourceful routes and controllers** when possible.
* Apply appropriate **middleware** for authentication and authorization.

Would you like me to create a working code example or a scaffolded folder structure for a sample Laravel project using controllers, routes, and views?

Great question! Let’s explore **how to design Blade views in Laravel** to structure and present **dynamic content** effectively.

**🎨 What is Blade?**

**Blade** is Laravel's powerful, simple, and elegant templating engine used to build views with **dynamic content**.

* Blade files use the .blade.php extension.
* Blade syntax is clean and readable.
* It allows embedding PHP logic using easy directives (@if, @foreach, @include, etc.).

**📁 1. Creating Blade View Files**

Views are typically stored in the resources/views/ directory.

Example:

resources/views/home.blade.php

You return this view from a controller:

return view('home');

**🧱 2. Blade Syntax Essentials**

**a. Displaying Data**

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

* Escaped: {{ }} is safe and prevents XSS.
* Unescaped: {!! $htmlContent !!} for raw HTML.

**b. Control Structures**

**Conditionals**

@if($user->isAdmin())

<p>Welcome, admin!</p>

@elseif($user->isMember())

<p>Welcome, member!</p>

@else

<p>Please log in.</p>

@endif

**Loops**

@foreach($posts as $post)

<h2>{{ $post->title }}</h2>

@endforeach

**🧩 3. Layout and Template Inheritance**

**a. Creating a Master Layout**

Example: layouts/app.blade.php

<!DOCTYPE html>

<html>

<head>

<title>@yield('title')</title>

</head>

<body>

@include('partials.navbar')

<div class="container">

@yield('content')

</div>

</body>

</html>

**b. Using the Layout in a View**

@extends('layouts.app')

@section('title', 'Home Page')

@section('content')

<h1>Welcome to My Site</h1>

@endsection

This keeps your views DRY (Don’t Repeat Yourself) by reusing layout files.

**🧩 4. Includes and Components**

**a. @include**

Re-use partials:

@include('partials.footer')

This pulls in resources/views/partials/footer.blade.php.

**b. @component and Slots**

Reusable Blade components (Laravel 7+):

php artisan make:component Alert

resources/views/components/alert.blade.php:

<div class="alert alert-{{ $type }}">

{{ $slot }}

</div>

Use it in a view:

<x-alert type="success">

User created successfully!

</x-alert>

**📥 5. Form Handling with Blade**

Use CSRF protection:

<form method="POST" action="/submit">

@csrf

<input type="text" name="title">

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

</form>

For PUT, PATCH, DELETE:

@method('PUT')

**🔁 6. Passing Data from Controller to Blade**

Controller:

public function index() {

$posts = Post::all();

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

}

View (posts/index.blade.php):

@foreach($posts as $post)

<h3>{{ $post->title }}</h3>

@endforeach

**🎛️ 7. Blade and Bootstrap (for Styling)**

You can use any CSS framework like Bootstrap inside Blade:

<div class="card">

<div class="card-body">

<h5 class="card-title">{{ $post->title }}</h5>

<p class="card-text">{{ $post->content }}</p>

</div>

</div>

**🧠 8. Blade Tips & Best Practices**

* Use @each for rendering partials in loops.
* Use @isset and @empty to check variable presence.
* Always escape output unless intentionally showing raw HTML.
* Organize views in folders to match controller structure.
* Use x-component or @includeIf, @includeWhen for conditional loading.

**✅ Example: Complete Flow**

**Route**

Route::get('/posts', [PostController::class, 'index']);

**Controller**

public function index()

{

$posts = Post::latest()->paginate(5);

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

}

**Blade View (posts/index.blade.php)**

@extends('layouts.app')

@section('title', 'All Posts')

@section('content')

<h1>Blog Posts</h1>

@foreach($posts as $post)

<div class="post">

<h2>{{ $post->title }}</h2>

<p>{{ Str::limit($post->content, 150) }}</p>

</div>

@endforeach

{{ $posts->links() }}

@endsection

Would you like a sample mini Laravel project structure using Blade views, including controllers and models (like a blog or task manager)?

Absolutely! Let's dive deep into how to **construct Laravel models to handle database interactions and define relationships**. Models in Laravel are essential to interacting with your database in a clean, organized, and object-oriented way.

**🧱 1. What is a Model in Laravel?**

In Laravel, a **model** is a PHP class that represents a **database table**. Models use **Eloquent ORM** (Object Relational Mapper) to interact with the database, so you can retrieve, insert, update, and delete records without writing raw SQL.

For example, a User model typically maps to a users table.

**⚙️ 2. Creating a Model**

Use Artisan to generate a model:

php artisan make:model Product

This creates:  
app/Models/Product.php

If you want a migration along with it:

php artisan make:model Product -m

**🧾 3. Basic Model Structure**

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Product extends Model

{

// Optional: table name if it doesn't follow naming convention

protected $table = 'products';

// Optional: fillable fields for mass assignment

protected $fillable = ['name', 'price', 'description'];

}

**💡 4. Performing Database Operations**

**a. Inserting Data**

Product::create([

'name' => 'iPhone 15',

'price' => 999,

'description' => 'Latest Apple smartphone'

]);

Make sure the fields are listed in $fillable.

**b. Retrieving Data**

// All records

$products = Product::all();

// Find by ID

$product = Product::find(1);

// Filtered data

$phones = Product::where('category', 'Phones')->get();

**c. Updating Data**

$product = Product::find(1);

$product->price = 950;

$product->save();

**d. Deleting Data**

Product::destroy(1); // Delete by ID

// Or

$product = Product::find(1);

$product->delete();

**🔗 5. Defining Relationships Between Models**

Laravel Eloquent makes it easy to define relationships like:

**a. One-to-One**

A User has one Profile.

**User model:**

public function profile()

{

return $this->hasOne(Profile::class);

}

**Profile model:**

public function user()

{

return $this->belongsTo(User::class);

}

**b. One-to-Many**

A Post has many Comments.

**Post model:**

public function comments()

{

return $this->hasMany(Comment::class);

}

**Comment model:**

public function post()

{

return $this->belongsTo(Post::class);

}

**c. Many-to-Many**

A User belongs to many Roles, and vice versa.

**User model:**

public function roles()

{

return $this->belongsToMany(Role::class);

}

**Role model:**

public function users()

{

return $this->belongsToMany(User::class);

}

This setup needs a pivot table: role\_user.

**d. Has Many Through**

Example: A Country has many Posts through Users.

public function posts()

{

return $this->hasManyThrough(Post::class, User::class);

}

**🧮 6. Accessing Related Data**

$post = Post::find(1);

// Get all comments for the post

$comments = $post->comments;

// Get post's author's name

$authorName = $post->user->name;

You can **eager load** relationships to optimize performance:

$posts = Post::with('comments')->get();

**🔄 7. Timestamps and Soft Deletes**

* Laravel automatically manages created\_at and updated\_at timestamps.
* Use SoftDeletes to keep deleted records.

use Illuminate\Database\Eloquent\SoftDeletes;

class Product extends Model

{

use SoftDeletes;

}

Run migration with softDeletes() column.

**🔐 8. Mass Assignment Protection**

Define $fillable to protect against mass assignment:

protected $fillable = ['title', 'content'];

Alternatively, guard fields:

protected $guarded = ['id'];

**📦 9. Custom Accessors and Mutators**

**Accessor – modify data when retrieving:**

public function getNameAttribute($value)

{

return ucfirst($value);

}

**Mutator – modify data when setting:**

public function setPriceAttribute($value)

{

$this->attributes['price'] = round($value, 2);

}

**✅ Example Summary**

**Model**

class Post extends Model

{

protected $fillable = ['title', 'body'];

public function comments()

{

return $this->hasMany(Comment::class);

}

}

**Controller**

public function show($id)

{

$post = Post::with('comments')->findOrFail($id);

return view('posts.show', compact('post'));

}

**Blade View**

<h1>{{ $post->title }}</h1>

@foreach($post->comments as $comment)

<p>{{ $comment->body }}</p>

@endforeach

Would you like me to generate an example mini-project (e.g., blog or task manager) using models, controllers, routes, and views to tie everything together?