Laravel Advanced

Laravel Advanced Topics

Blade Templating

Blade is the templating engine provided by Laravel. It allows you to write PHP code in your templates with clean and concise syntax. Here's a detailed guide to Blade Templating:

---

1. Basics of Blade

- **File Naming**: Blade templates have the .blade.php extension (e.g., welcome.blade.php).
- **Usage**: Store templates in the resources/views directory.
- **Syntax**: Use {{ }} for data output and @ for directives.

---

2. Displaying Data

- **Escaped Output**:
  `{{ $variable }}` ensures data is escaped to prevent **XSS attacks.**
- **Unescaped Output**:
  `{!! $variable !!}` outputs raw HTML.

---

3. Blade Directives

- **Conditionals**

```
@if ($condition)
    Content
@elseif ($anotherCondition)
    Another Content
@else
    Default Content
@endif
@unless ($condition)
    Content if false
@endunless
```

- **Loops**

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

```
    {{ $i }}
@endfor

@foreach ($items as $item)
    {{ $item }}
@endforeach

@while ($condition)
    Content
@endwhile
```

- **Switch Statement:**

```
@switch($value)
    @case(1)
        Case 1
        @break
    @default
        Default case
@endswitch
```

4. Layouts and Sections:

Blade layouts and sections allow you to create reusable templates and organize your views efficiently. This feature helps you avoid repeating common HTML structures like headers, footers, or navigation bars in multiple views.

- **Defining Layouts**:
  Create a base template (e.g., layouts.app):

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@yield('title')</title>
    <!-- Add your CSS or meta tags here -->
</head>
<body>
    <header>
        <h1>My Website Header</h1>
    </header>
```

```
    <nav>
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/about">About</a></li>
            <li><a href="/contact">Contact</a></li>
        </ul>
    </nav>

    <main>
        @yield('content')
    </main>

    <footer>
        <p>&copy; 2024 My Website</p>
    </footer>
</body>
</html>
```

- **Using Layouts**:
  Extend a layout in your view:
  Create a view file, e.g., `resources/views/home.blade.php`:

```
@extends('layouts.app')

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

@section('content')
    <h2>Welcome to My Website!</h2>
    <p>This is the homepage content.</p>
@endsection
```

Laravel Advanced Topics

5. Blade Components and Slots:

What are Blade Components?

Blade components are reusable pieces of HTML and Blade code. Instead of repeating similar code in multiple views,

- **Creating Components**:
  Use `php artisan make:component ComponentName`. It generates a class in `app/View/Components` and a template in `resources/views/components`.
- **Using Components**:

```
<x-component-name />
```

- **Slots**:

  Pass content to components:

```
<x-alert>
    <x-slot:name>Slot Content</x-slot>
</x-alert>
```

8. Debugging

Use `@dd($variable)` or `{{ $variable }}` to inspect data.

---

9. Security

- Escaping outputs with `{{ }}` prevents XSS.
- Use `{!! !!}` carefully for raw HTML.

---

10. Optimization

- **Compiled Views**: Blade templates are compiled into PHP and cached for better performance.
- **Avoid Logic in Views**: Keep complex logic in controllers or view composers. ?

Laravel Advanced Topics

Laravel APP Key Usage :

In Laravel, the **application key (APP_KEY)** is essential for ensuring security. It is used for:

1. **Data Encryption**: Secures sensitive data in storage.
2. **Session and Cookie Security**: Protects against session hijacking and tampering.
3. **CSRF Tokens**: Ensures secure form submissions.
4. **Password Reset Tokens**: Safeguards sensitive tokens.

What is Migration in Laravel?

- **Migrations** are like version control for your database. They allow developers to define and modify the database structure using code.
- Cmd :
    - Create Migration :php artisan make:migration create_users_table
    - Run Migration : php artisan migrate.

What is Seeder and Factory?

Seeder: Used to insert fake data into a table.
Factory: A factory is used to define a blueprint for generating fake data.
Faker Library: The Faker library is used to generate fake data like names, addresses, emails, etc. This is typically used within factories to create fake data.

So, to summarize:

- **Factories** define how the fake data should look.
- **Seeders** use the factory to insert the fake data into the database.
- **Faker** generates the fake data.

What is Eloquent ORM?

Eloquent ORM (Object-Relational Mapping) is a system in Laravel that interacts with your database using PHP, without writing raw SQL.

Key Features:

1. Model Usage: Each model represents a table in the database. We can use models to perform CRUD (Create, Read, Update, Delete) operations easily.
2. **CRUD Operations:**
    - Create: Use `create()` or `save()` to add new records.
    - Read: Use `all()`, `find()`, `where()`, and `first()` to retrieve records.
    - Update: Use `update()` or modify an existing model instance and call `save()`.

Laravel Advanced Topics

- Delete: Use `delete()` or `destroy()` to remove records.

3. **Accessors and Mutators:**
   - Accessor: Format data when retrieving it from the database (e.g., capitalize a title).
   - Mutator: Modify data before saving it (e.g., convert a title to lowercase).
4. **Relationships:** Eloquent makes it easy to define and work with relationships between models, such as:
   - One-to-One (`hasOne`, `belongsTo`) Ex: User and User Profile
   - One-to-Many (`hasMany`, `belongsTo`) Ex : Order and Product
   - Many-to-Many (`belongsToMany`) Ex : Post and Comment
5. **Eager Loading:** Load related records in one go to prevent multiple queries (e.g., `Post::with('comments')->get()`).
6. **Soft Deleting:** Instead of deleting records permanently, you can "soft delete" them, marking them as deleted but not removing them from the database.

   $activeRecords = Model::withoutTrashed()->get();

7. **Events:** Eloquent supports model events (e.g., when a model is created, updated, or deleted) to perform actions automatically.

Laravel's Routing System

1. Laravel's routing system is designed to handle all HTTP requests to your application.
2. **Named Routing :** Named routes allow you to assign a name to a route. If you use the route in your application for redirection, you won't need to change the redirection code if the URL changes.
3. **Resource Routes** - Laravel, **resource routes** allow you to define **CRUD (Create, Read, Update, Delete) operations** in a single line instead of defining each route manually.
4. Route Model Binding
   - **Route Model Binding** simplifies model access by binding it directly to the route, avoiding manual queries.
   - **Implicit Binding** uses the default `id` column for automatic resolution.
   - **Explicit Binding** allows you to use a custom column (like `email` instead of `id`) for model resolution.

Laravel Middleware:

1. Laravel Middleware is used to filter HTTP requests. For example, it can handle authorization, validate requests, or perform other checks before processing the HTTP request.

What is Log ?

1. Log Is Like A History Maintenance
2. Laravel Have Default Log Maintenance

   File Path Is Root Directory\storage\logs\laravel.log

Laravel Advanced Topics

3. Log ERROR Labels

```
Log::emergency($message);
Log::alert($message);
Log::critical($message);
Log::error($message);
Log::warning($message);
Log::notice($message);
Log::info($message);
Log::debug($message);
```

4. Log Config File Path config/logging.php
5. Laravel Used ThirdParty Monolog Logging libraries.

Facades - is Static service , we can access the service without create object any where in your application

Service Providers :

- Used to Register the server in the Service Container

Service Container :

- used to create object and dependency

Service Repository :

- Used to split up the DB query and Business Logic

Laravel Event & Listeners

- **Event**: Represents something that happens in your application (e.g., a user registers, an order is placed, etc.). It's like a signal or trigger that something significant has occurred.
- **Listener**: Reacts to the event and performs a specific action in response (e.g., sending a welcome email after a user registers, updating inventory after an order is placed).
- **EventServiceProvider**: Registers events and their listeners.

Laravel Observers

- Observers in Laravel are specifically designed to handle Eloquent ORM events. They allow you to listen for model events and perform actions when something happens to a model, like inserting, updating, deleting, etc.
- php artisan make:observer ModelObserver --model=ModelName
- Folder Path: app/Observers
- **Observers** in Laravel are **directly linked to the Eloquent model**, and you **do not need to manually dispatch events**. They automatically handle the model's events like `creating`, `created`, `updating`, `updated`, `deleting`, `deleted`, etc., without any additional intervention.
- You need to register the observer in the `AppServiceProvider` or another relevant service provider. In the boot method,

Laravel Advanced Topics

```php
namespace App\Providers;

use App\Models\User;
use App\Observers\UserObserver;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    public function boot()
    {
        User::observe(UserObserver::class);
    }
}
```

Laravel Notifications

Laravel Notifications is a feature that allows your application to send notifications across various delivery channels like email, SMS, Slack, database, and more.

1. Create a Notification

Run the following Artisan command to generate a notification class

```
php artisan make:notification MyNotification
```

This creates a file in the app/Notifications directory.

Key Differences:

- **Observers**: Listen for Eloquent model events and allow you to handle model changes (e.g., after a model is created or updated).
- **Events & Listeners**: Handle application-wide events and can trigger actions in response (e.g., when a user registers, a listener may send a welcome email).
- **Notifications**: Used to send notifications (alerts/messages) to users based on events that occur in your application (e.g., sending an email when a new post is published).

Laravel Guest

guest is used to identify and handle users who access your website **without logging into your application**.

Key Points :

Laravel Advanced Topics

- **Middleware for Guests:** Laravel has a built-in guest middleware that ensures only unauthenticated users can access certain routes.
- **Checking if a User is a Guest:** You can use the guest() method provided by Laravel's Auth facade or helper.
- **Blade Templates:** In Blade templates, you can check if a user is a guest using the @guest directive.

Laravel Mix

Laravel Mix is used to **compile and bundle assets** (CSS, JavaScript, images, etc.) efficiently.

Laravel Echo

Laravel Horizon

Laravel Horizon is a package used to **manage** and **monitor** queues and jobs in Laravel.

- **Real-time Monitoring**: View the status of your queues and jobs in real time.
- **Dashboard Interface**: A user-friendly dashboard to track job performance, failures, and completion.
- **Queue Metrics**: Track metrics like job processing time, job throughput, and failed jobs.
- **Job Retry & Failures**: Easily retry failed jobs and see detailed failure information.
- **Worker Monitoring**: Track the health and status of your worker processes.
- **Queue Configuration**: Customize and configure your queues for different job types.
- **Tagging Jobs**: Categorize jobs using tags for better organization and filtering.
- **Redis Integration**: Full support for Redis as the queue backend.
- **Notifications**: Get notified when jobs fail, succeed, or experience issues.

Laravel JetStream

Laravel JetStream is an **inbuilt authentication tool** for Laravel applications.

- **Login/Registration**: Standard user authentication features.
- **Email Verification**: Automatically send verification emails to users.
- **Password Reset**: Out-of-the-box support for password reset functionality.
- **Two-Factor Authentication (2FA)**: Built-in support for adding an extra layer of security.
- **Session Management**: Users can view and manage their active sessions.
- **Team Management**: Allows users to create and manage teams (for multi-user applications).

Laravel PHPUnit

Laravel PHPUnit are used for testing in Laravel applications,

Laravel Pest

Pest is a testing framework for PHP

Laravel Tinker

Laravel Tinker is primarily used to interact with your Laravel application's database from the **command line** using **models** and **DB queries**.

Laravel Advanced Topics

Laravel Breeze

Laravel Breeze is a simple authentication package for Laravel applications.

- User Registration
- Login
- Password Reset
- Email Verification

Laravel Skeleton

- Laravel Skeleton is like a light version of Laravel application . It provides the basic setup and the core files to run Laravel
- If you need **specific features**, you can install them **manually** via Composer or set them up as you go.

Laravel Resource:

- **Resources** are used to format and transform data structures, making it easier to control the output of API responses.
- Locate App\Http\Resources

Laravel Collection:

- Use **Collections** for data manipulation, like filtering or mapping arrays.

Creating a Collection:

```
use Illuminate\Support\Collection;

$collection = collect([1, 2, 3, 4, 5]);
```

Common Collection Methods:

- **filter()**: Filter items based on a condition.

```
$filtered = $collection->filter(function ($value) {
    return $value > 2;
});
// [3, 4, 5]
```

- **map()**: Transform each item.

```
$mapped = $collection->map(function ($value) {
    return $value * 2;
});
// [2, 4, 6, 8, 10]
```

Laravel Advanced Topics

- **pluck()**: Extract values from a key in an associative array or object.

```
$data = collect([
    ['name' => 'Alice', 'age' => 25],
    ['name' => 'Bob', 'age' => 30]
]);
$names = $data->pluck('name');
// ['Alice', 'Bob']
```

- **sum()**: Get the sum of items.

```
$sum = $collection->sum();
// 15
```

- **toArray() and toJson()**: Convert the collection to an array or JSON.

```
$array = $collection->toArray();
$json = $collection->toJson();
```

Gates:

Questions:

1. **What is XSS Attack ?**
   If attackers insert malicious scripts into a website through user input fields or URLs. it call Xss Attack
2. **How to prevent the XSS Attack ?**
   a. Validate the user input before storing it in the database.
   b. Use the Laravel blade template method to display the data.
   c. Use the htmlentities method .
3. **How to work behind the screen in the Anonymous Components ?**

   Laravel Blade first checks if a corresponding **Component Class** exists. If it does, Laravel uses the class to determine the component's behavior and render the Blade view. If the class does not exist, Laravel directly renders the Blade view file from the resources/views/components directory.
4. booting" typically means the **process of preparing everything to be ready for use**

Blade Components

Blade Components and Slots: Full Details

Blade components and slots in Laravel provide a clean and reusable way to structure and render parts of your views. They are particularly useful for creating modular, maintainable code.

---

1. What are Blade Components?

Blade components are reusable pieces of HTML and Blade code. Instead of repeating similar code in multiple views, you can create a component and use it wherever needed.

---

2. How to Create a Blade Component?

Laravel allows you to create components using Artisan commands.

Command:

php artisan make:component ComponentName

This creates:

- **Component class**: `App/View/Components/ComponentName.php`
- **View file**: `resources/views/components/component-name.blade.php`

---

3. Using Blade Components

Example 1: A Simple Alert Component

Component Class: `app/View/Components/Alert.php`

```
namespace App\View\Components;

use Illuminate\View\Component;
```

```php
class Alert extends Component
{
    public $type;
    public $message;

    public function __construct($type, $message)
    {
        $this->type = $type;
        $this->message = $message;
    }

    public function render()
    {
        return view('components.alert');
    }
}
```

Component View: `resources/views/components/alert.blade.php`

```blade
<div class="alert alert-{{ $type }}">
    {{ $message }}
</div>
```

Using the Component: In your Blade file:

```blade
<x-alert type="success" message="Operation completed successfully!" />
<x-alert type="danger" message="An error occurred!" />
```

Rendered Output:

```html
<div class="alert alert-success">
    Operation completed successfully!
</div>
<div class="alert alert-danger">
    An error occurred!
</div>
```

---

4. Slots in Blade Components

Slots are placeholders for content that you want to pass into a component.

Default Slot:

Component View: `resources/views/components/card.blade.php`

```
<div class="card">
    <div class="card-header">
        {{ $title }}
    </div>
    <div class="card-body">
        {{ $slot }}
    </div>
</div>
```

Using the Component:

```
<x-card :title="$cardTitle">
    This is the body of the card.
</x-card>
```

Rendered Output:

```
<div class="card">
    <div class="card-header">
        My Card Title
    </div>
    <div class="card-body">
        This is the body of the card.
    </div>
</div>
```

Named Slots:

You can define multiple slots with names.

Component View: `resources/views/components/modal.blade.php`

```
<div class="modal">
    <div class="modal-header">
        {{ $header }}
    </div>
    <div class="modal-body">
        {{ $slot }}
    </div>
    <div class="modal-footer">
        {{ $footer }}
    </div>
</div>
```

Using the Component:

```
<x-modal>
    <x-slot:header>
        <h2>Modal Header</h2>
    </x-slot:header>
    This is the modal body content.
    <x-slot:footer>
        <button>Close</button>
    </x-slot:footer>
</x-modal>
```

Rendered Output:

```
<div class="modal">
    <div class="modal-header">
        <h2>Modal Header</h2>
    </div>
    <div class="modal-body">
        This is the modal body content.
    </div>
    <div class="modal-footer">
        <button>Close</button>
    </div>
</div>
```

---

5. Passing Data to Components

Using Props:

You can pass data to a component using props.

Example:

```
<x-alert :type="$alertType" :message="$alertMessage" />
```

Using `with()`:

In the component class:

```
public function with()
{
    return ['additionalData' => 'Some extra data'];
}
```

---

6. Anonymous Components

Anonymous components are **components without a corresponding PHP class**. They are simple and only require the Blade view file.

How Anonymous Components Work:

You simply create a Blade file inside the `resources/views/components/` directory, and Laravel will treat it as an anonymous component.

Creating an Anonymous Component:

Save the file in the `resources/views/components` directory:
`resources/views/components/button.blade.php`

```
<button class="btn btn-{{ $type }}">
    {{ $slot }}
</button>
```

Using the Component:

```
<x-button type="primary">
    Click Me
</x-button>
```

Rendered Output:

```
<button class="btn btn-primary">
    Click Me
</button>
```

---

7. Inline Components

Inline components are similar to anonymous components, but they are defined directly in your Blade component class using the `render()` method with inline Blade syntax.

This means instead of using a separate file, you can define the entire component content **inline** inside the component class.

How Inline Components Work:

You create the component class just like a normal component but return the Blade code **directly** in the `render()` method instead of pointing to a separate view file.

Command:

php artisan make:component InlineAlert --inline

This generates:

```
namespace App\View\Components;
```

```php
use Illuminate\View\Component;

class InlineAlert extends Component
{
    public $type;

    public function __construct($type)
    {
        $this->type = $type;
    }

    public function render()
    {
        return <<<\'blade\'
        <div class="alert alert-{{ $type }}">
            {{ $slot }}
        </div>
        blade;
    }
}
```

---

8. Customizing Component Paths

You can customize the default path for Blade components in the `AppServiceProvider`:

```php
use Illuminate\Support\Facades\Blade;

public function boot()
{
    Blade::componentNamespace('App\\View\\CustomComponents', 'custom');
}
```

Now use:

```blade
<x-custom::alert type="info" message="Custom path works!" />
```

---

9. Advantages of Blade Components

- **Reusability:** Write once, use everywhere.
- **Readability:** Clean and organized code.
- **Custom Logic:** Components can include logic in their class.
- **Usage :** Alert Message,Modal Popup,Breadcrumbs ,Table , Form Elements

---