LARAVEL DOCUMENTATION BASED ON OFFICIAL

Install, ConfigurE, Directory Structure, Frontend, Starter Kits & Deployment

Certainly! I'll provide you with a comprehensive explanation of each of these topics in Laravel, along with code examples where relevant. This will serve as a thorough guide for Laravel developers.

Getting Started with Laravel

Laravel is a PHP web application framework that simplifies and accelerates the development of web applications. Here are the key steps to get started with Laravel:

1. **Prerequisites:** Ensure you have PHP, Composer, and a web server (e.g., Apache or Nginx) installed on your system.
2. **Installation:** Use Composer to create a new Laravel project:

```
composer create-project --prefer-dist laravel/laravel projectName
```

3. **Configuration:** Laravel's configuration files are located in the `config` directory. You can customize various settings like database connections, caching, and more.

Directory Structure

Laravel follows a well-organized directory structure:

Laravel follows a structured directory layout:

- `app`: folder is one of the core folders in a Laravel project. It contains various subfolders and files that together make up the application's logic and functionality
- `bootstrap`: folder provides the necessary startup and configuration scripts to initialize the Laravel framework
- `config`: Configuration files.
- `database`: Database migrations and seeders.
- `public`: Publicly accessible assets.
- `resources`: Views and frontend assets.
- `routes`: Application routes.
- `storage`: Temporary files and logs.
- `tests`: PHPUnit tests.
- `vendor`: Composer dependencies.
- `.env`: Environment configuration file.
- `webpack.mix.js` and `package.json`: Frontend build configuration.

Frontend

Laravel provides tools for frontend development:

- **Blade Templates:** Laravel uses Blade as its templating engine. You can create dynamic views with Blade's expressive syntax.

```
<!-- Blade template example -->
<h1>Hello, {{ $name }}</h1>
```

- **Asset Compilation:** Laravel Mix simplifies asset compilation, including CSS and JavaScript.

```
// webpack.mix.js
mix.js('resources/js/app.js', 'public/js')
    .sass('resources/sass/app.scss', 'public/css');
```

Starter Kits

Laravel offers starter kits for common use cases:

- **Jetstream:** A customizable authentication system with user registration, login, and two-factor authentication.
- **Fortify:** A minimal authentication scaffolding that you can customize according to your needs.
- **Breeze:** A lightweight and minimalistic starter kit for authentication.

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

Deployment

When deploying a Laravel application:

- Set up a production server with PHP and a web server.
- Configure your web server to point to the `public` directory as the document root.
- Secure your environment by setting appropriate permissions and environment variables.
- Use a tool like Laravel Envoyer or deploy scripts to automate the deployment process.

Architecture Concepts

architecture concepts in Laravel, including "Request Lifecycle," "Service Container," "Service Providers," and "Facades," along with code examples for Laravel developers.

REQUEST LIFESTYLE

Certainly! Understanding the request lifecycle in Laravel is crucial for developers to comprehend how an HTTP request is processed and how different components of the framework work together. Here's a detailed explanation of the request lifecycle in Laravel, along with code examples:

Overview:

The request lifecycle in Laravel describes the sequence of events that occur when an HTTP request is received and processed by the framework. It involves several key components and stages.

Key Stages in the Request Lifecycle:

1. **Entry Point:**
    - The request enters Laravel through the `public/index.php` file.
    - Laravel's application instance is created.
2. **HTTP Kernel:**
    - The request is then passed to the HTTP Kernel, which serves as the central part of the request handling process.
    - The kernel loads middleware and routes.
3. **Middleware:**
    - Middleware are classes that can perform actions before or after the request reaches the application's core logic.
    - Middleware is defined in the `app/Http/Kernel.php` file.

    Example Middleware:

    ```
    public function handle($request, Closure $next)
    {
        // Perform actions before the request is handled.
        // ...

        $response = $next($request);

        // Perform actions after the request is handled.
        // ...

        return $response;
    }
    ```
4. **Routing:**
    - Laravel's router matches the incoming request to a route defined in the `routes/web.php` or `routes/api.php` file.
    - Controllers or closures handle the matched route.

Example Route:

```
Route::get('/home', 'HomeController@index');
```

5. **Controller Action:**

    - If a controller handles the route, it executes a specific method (action) within the controller.
    - The controller processes the request and returns a response.

    Example Controller:

```
class HomeController extends Controller
{
    public function index()
    {
        // Your controller logic here.
        return view('home');
    }
}
```

6. **Response:**

    - The response goes back through the middleware and kernel.
    - Middleware can modify the response before it's sent to the client.

    Example Response:

```
return view('home');
# for response to client side
return response()->json(['message' => 'Hello, world!'], 200);
```

7. **Sending to Browser:**

    - Finally, the response is sent to the client's browser, which displays the result.

Conclusion:

The request lifecycle in Laravel is a fundamental aspect of understanding how the framework handles incoming HTTP requests. Each stage, from the entry point to the response, is a part of the sequence that developers can leverage to build robust(□□□□□□) web applications. By customizing middleware, routes, controllers, and responses, Laravel provides a powerful foundation for building web applications. Developers can dive deeper into each stage to implement custom logic and achieve specific functionality in their Laravel applications.

Service Container

Overview:

The Service Container is a powerful tool in Laravel for managing class dependencies and achieving a high degree of flexibility in your application. It follows the concept of Inversion of Control (IoC) and allows you to perform Dependency Injection easily.

Binding a Class:

You can register a binding in Laravel's Service Container using the `bind` method. For example, let's bind an interface `PaymentGateway` to a concrete implementation `StripePaymentGateway`:

```
php artisan make:provider PaymentGatewayServiceProvider
use App\PaymentGateway;
use App\StripePaymentGateway;
// ...

public function register()
{
    $this->app->bind(PaymentGateway::class, StripePaymentGateway::class);
}
```

Resolving Dependencies:

When you need an instance of a class or interface, Laravel can automatically resolve it from the container. This is commonly used in controller constructors or method injections. For example:

```
use App\Contracts\PaymentGateway;

class OrderController extends Controller
{
    protected $paymentGateway;

    public function __construct(PaymentGateway $custom_paymentGateway)
    {
        $this->paymentGateway = $custom_paymentGateway;
    }

    // ...
}
```

Here, Laravel will automatically inject an instance of `StripePaymentGateway` when creating an `OrderController`.

Singleton Binding:

You can bind a class as a singleton, which means only one instance will be created and shared across the application. This is useful for maintaining a single instance of a class throughout the application's lifecycle:

```
app()->singleton('example', function () {
    return new ExampleService();
});
```

Resolving from the Container:

You can resolve a class or interface from the container using the `app()` helper function:

```
$paymentGateway = app(PaymentGateway::class);
```

Real-life Example:

Suppose you are building an e-commerce application and you want to implement different payment gateways. Here's how you can use the Service Container:

1. **Binding Payment Gateways:** In your service provider (e.g., `AppServiceProvider`), bind different payment gateways:

```
app()->bind(PaymentGateway::class, StripePaymentGateway::class);
app()->bind(PaymentGateway::class, PayPalPaymentGateway::class);
```

2. **Controller:** In your controller, you can inject the `PaymentGateway` interface and use it for processing payments:

```
use App\Contracts\PaymentGateway;

class OrderController extends Controller
{
    protected $paymentGateway;

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

    public function processPayment()
    {
        // Process payment using the injected PaymentGateway
        $this->paymentGateway->charge(100);
        // ...
    }
}
```

3. **Blade View:** In your Blade view, you can use the resolved instance to display the payment option:

```
<p>Payment Method: {{ $paymentGateway->getName() }}</p>
```

This real-life example demonstrates how the Service Container in Laravel can be used to handle different payment gateways with ease.

By leveraging the Service Container, you can achieve better code organization, maintainability, and testability in your Laravel applications. It's a powerful feature that enables you to manage dependencies efficiently throughout your project.

SERVICE CONTAINER : REAL-LIFE EXAMPLES

1. **Service Container**:
   - **Definition**: The service container, also known as the IoC (Inversion of Control) container, is a container for managing class dependencies and performing dependency injection. It resolves and provides instances of classes when needed.
   - **Real-life Example**: Think of it like a restaurant menu. You don't need to know how to cook each dish; you order from the menu, and the chef prepares it for you. The menu is the service container, and the chef is the service provider.
2. **Service Provider**:
   - **Definition**: A service provider in Laravel is a way to register services (classes) into the service container. It defines what the container should do when an application needs an instance of a particular class.
   - **Real-life Example**: In a real-world scenario, a car dealership can be seen as a service provider. They provide various car models (services) to customers. The dealership tells you which cars (services) are available, and when you choose one, they give you the keys (an instance of the class).

Here's a simple Laravel project example:

Let's say you have a Laravel e-commerce website. You want to create a service that calculates shipping costs.

1. Create a service class, e.g., `ShippingCalculator`, which calculates shipping costs based on parameters.

```
class ShippingCalculator
{
    public function calculateShippingCost($weight, $destination)
    {
        // Define shipping rates based on weight and destination
        $shippingRates = [
            'Local' => [
                'upTo1kg' => 5.00,
                'upTo5kg' => 8.00,
                'upTo10kg' => 12.00,
            ],
            'National' => [
                'upTo1kg' => 10.00,
                'upTo5kg' => 15.00,
                'upTo10kg' => 20.00,
            ],
            'International' => [
                'upTo1kg' => 20.00,
                'upTo5kg' => 30.00,
                'upTo10kg' => 40.00,
            ],
        ];

        // Check if the destination is valid
        if (!isset($shippingRates[$destination])) {
            throw new \InvalidArgumentException('Invalid destination');
        }

        // Determine the shipping rate based on weight
        if ($weight <= 1) {
            $rate = $shippingRates[$destination]['upTo1kg'];
        } elseif ($weight <= 5) {
            $rate = $shippingRates[$destination]['upTo5kg'];
        } elseif ($weight <= 10) {
            $rate = $shippingRates[$destination]['upTo10kg'];
        } else {
            // Handle weights over 10kg if needed
            throw new \InvalidArgumentException('Weight exceeds the supported range');
        }

        return $rate;
    }
}
```

2. Create a service provider, e.g., `ShippingServiceProvider`, and register the `ShippingCalculator` class in it.

```
use Illuminate\Support\ServiceProvider;

class ShippingServiceProvider extends ServiceProvider
{
    public function register()
    {
        $this->app->bind('shipping', function ($app) {
            return new ShippingCalculator();
        });
    }
}
```

3. Now, in your controller or wherever you need to calculate shipping costs, you can use the service container to get an instance of the `ShippingCalculator` class.

```
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;
use App\Services\ShippingCalculator; // Import the ShippingCalculator class

class ShippingController extends Controller
{
    public function calculateShipping(Request $request)
    {
        // Get weight and destination from the form or request
        $weight = $request->input('weight');
        $destination = $request->input('destination');

        // Validate input (you can add more validation if needed)
        $this->validate($request, [
            'weight' => 'required|numeric|min:0.1', // Adjust validation rules as needed
            'destination' => 'required|in:Local,National,International',
        ]);

        // Create an instance of the ShippingCalculator
        $shippingCalculator = new ShippingCalculator();

        // Calculate shipping cost
        $shippingCost = $shippingCalculator->calculateShippingCost($weight, $destination);

        // Pass the calculated cost to a Blade view
        return view('shipping.calculate', ['shippingCost' => $shippingCost]);
    }
}
```

4. Using in blade file

```
<!DOCTYPE html>
<html>
<head>
    <title>Shipping Calculator</title>
</head>
<body>
    <h1>Shipping Calculator</h1>

    <form method="POST" action="{{ route('shipping.calculate') }}">
        @csrf
        <label for="weight">Weight (kg):</label>
        <input type="number" name="weight" id="weight" step="0.1" required>
        <br>

        <label for="destination">Destination:</label>
        <select name="destination" id="destination" required>
            <option value="Local">Local</option>
            <option value="National">National</option>
            <option value="International">International</option>
        </select>
        <br>

        <button type="submit">Calculate Shipping Cost</button>
    </form>

    @if(isset($shippingCost))
        <p>Shipping Cost: ${{ $shippingCost }}</p>
    @endif
</body>
</html>
```

OVERVIEW OF SERVICE CONTAINER

Certainly, let's simplify the concept of the service container in the context of the provided codes.

Service Container in Simple Terms:

1. **What is a Service Container?**

   Think of the service container as a central place where Laravel keeps a list of all the tools (classes) it might need to use in your application.

2. **How Does It Work in the Code You Provided?**

   - In the code you provided, we have a `ShippingCalculator` class that calculates shipping costs. We created an instance of it in the controller like this:

     ```
     $shippingCalculator = new ShippingCalculator();
     ```

     Instead of directly creating it using `new`, we could have asked the service container to give it to us. It would look like this:

     ```
     $shippingCalculator = app(ShippingCalculator::class);
     ```

     Here, we're asking the service container to provide an instance of the `ShippingCalculator` class. The service container knows how to create and manage instances of classes.

Using the Service Container in Controller and Blade:

1. **Controller**:

   In the controller, you can use the service container to get instances of classes or services that your application needs. You do this using the `app()` function, like this:

   ```
   $shippingCalculator = app(ShippingCalculator::class);
   ```

   This way, you don't have to manually create instances; Laravel handles it for you.

2. **Blade View**:

   In Blade views, you don't directly interact with the service container. Instead, you use data passed from the controller. For example, in the Blade view, we displayed the shipping cost like this:

   ```
   @if(isset($shippingCost))
       <p>Shipping Cost: ${{ $shippingCost }}</p>
   @endif
   ```

   The `$shippingCost` variable was set in the controller and then passed to the Blade view. Blade is simply used to display data provided by the controller.

Service Provider

DEFINITION:

Service providers in Laravel are classes that contain methods for registering services, binding classes, and performing application bootstrapping. They act as the central place to configure various parts of your application.

Real-life Example:

Yes, you can create and provide instances of classes or services in the service container and then use them within a service provider. Let's create a simple example to illustrate this concept.

Step 1: Create a Class

Suppose you want to create a class called `TaxCalculator`, which calculates taxes based on a given amount.

```
php artisan make:class TaxCalculator

// TaxCalculator.php
class TaxCalculator
{
    public function calculateTax($amount)
    {
        // Your tax calculation logic here
        return $amount * 0.1; // For simplicity, we'll use a fixed tax rate of 10%
    }
}
```

Step 2: Create a Service Provider

Now, let's create a service provider called `TaxCalculatorServiceProvider`. This service provider will register the `TaxCalculator` class in the service container.

```
// TaxCalculatorServiceProvider.php
use Illuminate\Support\ServiceProvider;

class TaxCalculatorServiceProvider extends ServiceProvider
{
    public function register()
    {
        // Register the TaxCalculator class in the service container
        $this->app->bind('taxCalculator', function ($app) {
            return new TaxCalculator();
        });
    }
}
```

In this code, we're binding the `'taxCalculator'` identifier to an instance of the `TaxCalculator` class in the service container.

Step 3: Register the Service Provider

Don't forget to register the `TaxCalculatorServiceProvider` in the `config/app.php` file under the `providers` array:

```
// config/app.php

'providers' => [
    // ...
    App\Providers\TaxCalculatorServiceProvider::class,
],
```

Notes: Now, you can open the newly created `TaxCalculator.php` file in the `app` directory and define your `TaxCalculator` class within it.

Here's an example of the directory structure:

```
- app
  - Http
    - Controllers
      - TaxController.php   <-- Controllers are stored here
  - Providers
    - TaxCalculatorServiceProvider.php  <-- Service Providers are stored here
  - TaxCalculator.php       <-- Your custom classes, like TaxCalculator, can be stored here
  - Services
    - YourService.php       <-- Your custom service can be stored here
```

Step 4: Use the TaxCalculator in a Controller

Now, let's create a controller that uses the `TaxCalculator` class via the service container.

```
// TaxController.php
use App\Http\Controllers\Controller;
use Illuminate\Http\Request;

class TaxController extends Controller
{
    public function calculateTax(Request $request)
    {
        // Use the service container to get an instance of TaxCalculator
        $taxCalculator = app('taxCalculator');

        // Calculate tax based on the amount
        $amount = $request->input('amount');
        $tax = $taxCalculator->calculateTax($amount);

        return "Tax calculated: $tax";
    }
}
```

Step 5: Create a Route

Finally, define a route for your controller's method in `routes/web.php`:

```
Route::post('/calculate-tax', 'TaxController@calculateTax');
```

Now, when you access the `/calculate-tax` route and submit an amount through a form or request, the controller will use the `TaxCalculator` class, which is provided by the service container via the service provider, to calculate and display the tax.

In this example, we've created a class (`TaxCalculator`), registered it in the service container using a service provider (`TaxCalculatorServiceProvider`), and then used it in a controller. This demonstrates how you can provide instances of classes as services through the service container and service providers in Laravel.

Register() and boot() methods:

No problem, I'll explain the `register()` and `boot()` methods in Laravel service providers, and I'll provide code examples to clarify their roles.

1. `register()` Method:

- **What is it?** The `register()` method is used for registering services, bindings, or dependencies in the Laravel service container. It's typically where you define how Laravel should create and provide instances of classes or services.
- **Example:**

  Let's say you want to register a service named `'myService'`:

```
public function register()
{
    $this->app->bind('myService', function ($app) {
        return new MyService(); // Define how to create the service
    });
}
```

  In this example, the `register()` method tells Laravel how to create and provide an instance of the `MyService` class when someone asks for `'myService'`.

2. `boot()` Method:

- **What is it?** The `boot()` method is used for performing operations or tasks after all services have been registered and the application is fully booted. It's often used to set up event listeners, route bindings, or other application-specific logic.
- **Example:**

  Let's say you want to set up a route binding in your service provider:

```
public function boot()
{
    Route::bind('user', function ($value) {
        return User::where('name', $value)->firstOrFail();
    });
}
```

In this example, the `boot()` method sets up a route binding for the `'user'` route parameter. It tells Laravel how to find and bind a user based on the route parameter value.

In Summary:

- `register()`: Used for defining how to create and provide instances of classes or services in the service container.
- `boot()`: Used for performing operations or tasks that rely on registered services and are executed after the application is fully booted.

Both methods are important in Laravel service providers, but as a beginner, you'll often focus on the `register()` method for defining service bindings. The `boot()` method comes into play when you need to set up more complex interactions in your application.

SERVICE CONATAINER AND PROVIDER With (Class Dependency)

Scenario: Blog Comment System

Imagine you're building a blog application, and you have a `CommentService` class that handles creating and managing comments on blog posts. You want to use this service in your controller to handle comment submissions.

Step 1: Create the `CommentService` Class

1. **What is it?**

   The `CommentService` class contains methods for creating and managing comments.

2. **How to Create It?**

   Create a `CommentService.php` file in your application's directory (e.g., `app/Services`) and define the class:

```
namespace App\Services;

class CommentService
{
    public function createComment($postId, $content)
    {
        // Logic to create a comment and store it in the database
    }

    // Other methods for managing comments...
}
```

   This class has a method `createComment` to create new comments.

Step 2: Use Dependency Injection in Controller

1. **What is it?**

   Dependency injection allows you to inject instances of dependencies (like `CommentService`) into your controller's constructor or method.

2. **How to Do It?**

   In your controller (e.g., `CommentController.php`), use dependency injection in the constructor:

```
use App\Services\CommentService;

class CommentController extends Controller
{
    protected $commentService;

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

    // ...
}
```

Here, we're injecting an instance of `CommentService` into the controller via the constructor.

Step 3: Use the `CommentService` in Controller Method

1. **What is it?**

   Now that you have the `CommentService` instance available in your controller, you can use it to handle comment submissions.

2. **How to Do It?**

   In your controller method, you can use the `$this->commentService` instance:

```
public function store(Request $request)
{
    // Validate the comment data

    $postId = $request->input('post_id');
    $content = $request->input('content');

    // Use the CommentService to create the comment
    $this->commentService->createComment($postId, $content);

    // Redirect or return a response...
}
```

In this example, when a new comment is submitted, we use the `$this->commentService` instance to create the comment.

Summary:

In this professional project scenario, we've used dependency injection to inject an instance of the `CommentService` class into the `CommentController`. This allows the controller to use the methods of the `CommentService` to create and manage comments without worrying about how the `CommentService` is instantiated

Facades

Laravel facades are like shortcuts that allow you to access different parts of the Laravel framework in a straightforward way. Instead of creating objects or writing lengthy code, you can use facades to perform common tasks quickly. They provide a more organized and user-friendly approach to working with Laravel's features.

Laravel provides a set of facades, which are static proxies to underlying classes. These facades provide a convenient and expressive way to interact with Laravel's services and features. Below, I'll list some of the most important Laravel facades along with code examples of how to use them:

Notes : In Laravel Facades , both `Session::put('user_id', 1)` and `session()->put('user_id', 1)` are used to store a value in the session. They achieve the same result, but they use different syntax. but some can't use like `Db::table()` , but not `db()->table()`;

Facades are found from **use Illuminate\Support\Facades\FacadeName**

1. **Route Facade (`Route`):**

   - The `Route` facade allows you to define routes and generate URLs.
   - Example: Defining a named route and generating a URL.

```
Route::get('/profile', 'UserController@profile')->name('profile');

// Generating URL
$url = route('profile');
```

2. **View Facade (`View`):**
   - The `View` facade is used to render Blade templates.
   - Example: Rendering a view with data.

```
return View::make('welcome', ['name' => 'John']);
return view('products.all_products',['products'=>$products]);
```

3. **DB Facade (`DB`):**
   - The `DB` facade provides access to the database using Eloquent or the query builder.
   - Example: Querying the database with the query builder.

```
$users = DB::table('users')->where('status', 'active')->get();
```

4. **Session Facade (`Session`):**
   - The `Session` facade allows you to work with the session data.
   - Example: Storing and retrieving data from the session.

```
// Storing data in the session
Session::put('user_id', 1);
session()->put('user_id',auth()->id()); // alternative way Auth::id()

// Retrieving data from the session
$userId = Session::get('user_id');
```

5. **Request Facade (`Request`):**
   - The `Request` facade provides access to the current HTTP request.
   - Example: Getting request parameters and headers.

```
$input = Request::input('username');
$header = Request::header('User-Agent');
```

6. **Config Facade (`Config`):**
   - The `Config` facade allows you to access configuration values from `config` files.
   - Example: Getting a configuration value.

```
$timezone = Config::get('app.timezone');
```

7. **Log Facade (`Log`):**
   - The `Log` facade allows you to log messages.
   - Example: Logging a message with different log levels.

```
Log::info('This is an info message.');
Log::error('An error occurred: ' . $exception->getMessage());
```

8. **Auth Facade (`Auth`):**
   - The `Auth` facade provides authentication and authorization features.
   - Example: Checking if a user is authenticated.

```
if (Auth::check()) {
    // User is authenticated
}
```

9. **Mail Facade (`Mail`):**
    - The `Mail` facade is used to send emails.
    - Example: Sending an email.

```
Mail::to('example@example.com')->send(new MyMailClass($data));
```

10. **Cache Facade (`Cache`):**
    - The `Cache` facade allows you to work with Laravel's caching system.
    - Example: Caching data.

```
// Store data in cache for 60 minutes
Cache::put('key', 'value', 60);

// Retrieve data from cache
$data = Cache::get('key');
```

Certainly! Here are more important Laravel facades, extending the list beyond the initial 10 facades:

11. **Redirect Facade (`Redirect`):**
    - The `Redirect` facade simplifies the process of creating HTTP redirects.
    - Example: Redirecting to a specific URL or route.

```
return Redirect::to('new-page'); // redirect()->to()
```

12. **File Facade (`File`):**
    - The `File` facade provides methods for working with files and directories.
    - Example: Checking if a file exists and reading its contents.

```
if (File::exists('path/to/file.txt')) {
    $content = File::get('path/to/file.txt');
}
```

13. **Hash Facade (`Hash`):**
    - The `Hash` facade is used for hashing and verifying passwords.
    - Example: Hashing a password and checking it against a stored hash.

```
$hashedPassword = Hash::make('password123');
if (Hash::check('password123', $hashedPassword)) {
    // Password is correct
}
```

14. **Notification Facade (`Notification`):**
    - The `Notification` facade allows you to send notifications via various channels (email, SMS, etc.).
    - Example: Sending a notification to a user.

```
Notification::send($user, new InvoicePaid($invoice));
```

15. **Cookie Facade (`Cookie`):**

- The `Cookie` facade allows you to work with HTTP cookies.

- Example: Setting and getting cookies.

```
Cookie::queue('name', 'John', 60); // Set a cookie
$value = Cookie::get('name'); // Get a cookie
```

24. **Artisan Facade (`Artisan`):**

- The `Artisan` facade provides access to the Artisan command-line tool from within your application.

- Example: Running an Artisan command programmatically.

```
Artisan::call('migrate');
```

27. **Schema Facade (`Schema`):**

- The `Schema` facade provides methods for working with database schemas and tables.

- Example: Creating a new table using migrations.

```
Schema::create('users', function ($table) {
    $table->id();
    $table->string('name');
    $table->timestamps();
});
```

28. **URL Facade (`URL`):**

- The `URL` facade helps generate URLs for various parts of your application.

- Example: Generating URLs for routes.

```
$url = URL::to('profile'); // url('products', $produf->id)
```

29. **Validator Facade (`Validator`):**

- The `Validator` facade allows you to validate data easily.

- Example: Validating form data.

```
$validator = Validator::make($request->all(), [
    'email' => 'required|email',
    'password' => 'required|min:6',
]);
```

30. **Str Facade (`Str`):**

- The `Str` facade provides string manipulation methods.

- Example: Truncating a string.

```
$shortened = Str::limit('This is a long string', 10);
```

Custom Facade

Creating a real-life custom facade in Laravel involves several steps, including defining the facade class, creating a service provider, binding the facade to the underlying class, and using it in a controller and Blade file. We can create **CustomLoggingFacade**,**PaymentGatewayFacade**,**ImageProcessingFacade**

Step 1: Create the Facade Class

First, create a custom facade class that will act as a static proxy to an underlying service or functionality. For this example, let's create a simple facade for generating QR codes.

Create the facade class in the `app/Facades` directory. If the directory doesn't exist, you can create it:

```
mkdir app/Facades
touch app/Facades/QrCodeFacade.php
```

Now, define the `QrCodeFacade.php` class:

```
// app/Facades/QrCodeFacade.php

namespace App\Facades;

use Illuminate\Support\Facades\Facade;

class QrCodeFacade extends Facade
{
    protected static function getFacadeAccessor()
    {
        return 'qr-code'; // This should match the binding name in the service provider
    }
}
```

Step 2: Create the Service Provider

Next, create a service provider to bind the underlying functionality (in this case, a QR code generator) to the Laravel service container.

Generate a service provider using Artisan:

```
php artisan make:provider QrCodeServiceProvider
```

Edit the `QrCodeServiceProvider.php` file to bind the QR code generator:

```
// app/Providers/QrCodeServiceProvider.php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use SimpleSoftwareIO\QrCode\Facades\QrCode;

class QrCodeServiceProvider extends ServiceProvider
{
    public function register()
    {
        $this->app->bind('qr-code', function () {
            return QrCode::size(200);
        });
    }

    // ...
}
```

In this example, we're using the `SimpleSoftwareIO/QrCode` package for generating QR codes. Make sure to install the package using Composer:

```
composer require simplesoftwareio/simple-qrcode
```

Step 3: Register the Service Provider

Add the `QrCodeServiceProvider` to the `providers` array in your `config/app.php` configuration file:

```
// config/app.php

'providers' => [
    // ...
    App\Providers\QrCodeServiceProvider::class,
],
```

Step 4: Use the Facade in a Controller

Now, you can use your custom facade in a controller. For example, create a new controller using Artisan:

```
php artisan make:controller QRCodeController
```

Edit the `QRCodeController.php` file and use the facade to generate a QR code:

```
// app/Http/Controllers/QRCodeController.php

namespace App\Http\Controllers;

use App\Facades\QrCodeFacade;
use Illuminate\Http\Request;

class QRCodeController extends Controller
{
    public function generateQRCode(Request $request)
    {
        $qrCode = QrCodeFacade::generate($request->input('data'));

        return view('qrcode', compact('qrCode'));
    }
}
```

Step 5: Use the Facade in a Blade View

Create a Blade view file to display the generated QR code. For example, create a `qrcode.blade.php` file in the `resources/views` directory:

```
<!-- resources/views/qrcode.blade.php -->

<!DOCTYPE html>
<html>
<head>
    <title>QR Code Generator</title>
</head>
<body>
    <img src="data:image/png;base64,{{ base64_encode($qrCode) }}" alt="QR Code">
</body>
</html>
```

In this Blade view, we're using the `$qrCode` variable passed from the controller to display the generated QR code image.

Step 6: Create a Route

Define a route in your `routes/web.php` file to access the QR code generation feature:

```
// routes/web.php

use App\Http\Controllers\QRCodeController;

Route::get('/generate-qrcode', [QRCodeController::class, 'generateQRCode']);
```

Step 7: Access the QR Code Generator

Now, you can access the QR code generator by visiting the `/generate-qrcode` route in your application. It will use the custom facade to generate a QR code and display it on the `qrcode.blade.php` view.

More Custom Facades

1. **CustomLoggingFacade:** Create a custom facade for enhanced logging, allowing you to log specific types of messages or perform additional actions when logging.
2. **NotificationFacade:** Implement a custom facade to handle notifications, making it easier to send messages, alerts, and emails to users.
3. **GeolocationFacade:** Create a facade for geolocation services, enabling you to retrieve and manage location-related data easily.
4. **PaymentGatewayFacade:** Develop a custom facade for payment gateway integration, simplifying payment processing in your application.
5. **ImageProcessingFacade:** Implement a facade to handle image processing tasks, such as resizing, cropping, or watermarking images.

6. **SecurityFacade:** Create a security facade to encapsulate various security-related tasks, like authentication, authorization, and encryption.
7. **CacheManagementFacade:** Build a custom facade to manage caching, allowing you to cache and retrieve data efficiently.
8. **JobQueueFacade:** Develop a facade to manage background jobs and task queues, making it easy to dispatch and handle asynchronous tasks.
9. **FileStorageFacade:** Implement a facade for interacting with file storage services, such as cloud storage or local file systems.
10. **NotificationDispatcherFacade:** Create a facade to manage and dispatch notifications across multiple channels, such as email, SMS, and push notifications.
11. **SocialMediaIntegrationFacade:** Develop a facade for integrating with social media platforms, simplifying actions like posting to social media or fetching user data.
12. **SearchEngineIntegrationFacade:** Implement a facade for integrating with search engines like Elasticsearch or Algolia, enabling efficient search functionality in your application.
13. **EmailServiceFacade:** Build a custom email service facade for sending and managing emails with various providers or services.
14. **AnalyticsFacade:** Create a facade to handle analytics and tracking, allowing you to collect and analyze user data.
15. **DatabaseFacade:** Develop a custom database facade for interacting with databases efficiently, including custom query builders or data migration utilities.
16. **LocalizationFacade:** Implement a localization facade to manage translations and language-related tasks in multilingual applications.
17. **APIIntegrationFacade:** Build a facade for integrating with external APIs, making it easier to send requests, handle responses, and manage API keys.
18. **PDFGenerationFacade:** Create a facade for generating PDF documents from HTML templates or data, simplifying PDF generation tasks.
19. **WorkflowAutomationFacade:** Implement a facade for workflow automation, allowing you to define and manage complex business processes.
20. **CustomValidationFacade:** Develop a custom validation facade for handling specific validation rules or complex validation scenarios.

BASIC CONCEPTS OF LARAVEL

ROUTE

the HTTP Request by which any page is displayed or loaded based on appropriate controller is called routes.

Route Definition: A route in Laravel is like a map that tells the application how to respond to different HTTP requests. It defines the URL (Uniform Resource Locator) and associates it with a specific controller or action to determine what should be displayed or loaded when that URL is accessed.

Named Route: Named routes are like giving a nickname to a route. You can create a named route by chaining the `name()` method when defining the route. It makes it easier to refer to the route in your code.

For example:

```
Route::get('/students', 'StudentController@index')->name('student.all');
```

Now, you can refer to this route as 'student.all' in your code.

Route Parameters: Route parameters allow you to capture and use values from the URL within your application. There are two types:

1. **Required Parameter:** A required parameter must have a value in the URL's request. It's essential for the route to work correctly. For example, in a route like `/students/{id}`, `{id}` is a required parameter, and you need to provide a value for it in the URL.
2. **Optional Parameter:** An optional parameter does not need to have a value in the URL. You can use it if needed, but it's not mandatory for the route to function. To make a parameter optional, you can specify a default value for it in your route definition.

For example:

```
Route::get('/students/{id?}', 'StudentController@show')->where('id', '[0-9]+')->default('id', 1);
```

1. **Basic Route Definition:**

- The most common way to define a route is using the `Route::get()` method, which maps an HTTP GET request to a callback or controller method.
- Example: Defining a basic route and returning a view in a controller method.

```
// routes/web.php
Route::get('/home', 'HomeController@index');

// app/Http/Controllers/HomeController.php
public function index()
{
    return view('home');
}
```

2. **Route Parameters:**

- You can define dynamic route parameters that capture values from the URL.
- Example: Defining a route with a parameter and passing it to a controller method.

```
// routes/web.php
Route::get('/user/{id}', 'UserController@show');

// app/Http/Controllers/UserController.php
public function show($id)
{
    // Use $id to fetch and display user details
}
```

3. **Named Routes:**

- Named routes allow you to give a unique name to a route, making it easier to generate URLs and redirects.
- Example: Defining a named route and generating a URL.

```
// routes/web.php
Route::get('/profile', 'ProfileController@index')->name('profile');

// Generating URL
$url = route('profile');
```

4. **Route Groups:**

- Route groups allow you to apply middleware, prefixes, and namespaces to a group of routes.
- Example: Defining a route group with middleware.

```
// routes/web.php
Route::middleware(['auth'])->prefix('user/')->name('user.')->group(function () {

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

    // url = user/dashboard and name = user.dashboard

    // More routes for authenticated users
});
```

```
Route::middleware(['auth'])->prefix('user/')->name('user.')->controller(DashboardController::class)->group(function () {

     Route::get('/dashboard', 'index')->name('dashboard');
     // url = user/dashboard and name = user.dashboard

     // More routes for authenticated users
 });
 ```
```

## 5. **Fallback Routes:**

- Fallback routes are used to handle undefined routes or 404 errors.
```

- Example: Defining a fallback route.

```
// routes/web.php
Route::fallback(function () {
    return view('404');
});
```

6. **Resourceful Routes:**

- Resourceful routes generate multiple routes for CRUD operations on a resource (e.g., a model).
- Example: Defining resourceful routes for a `Post` model.

```
// routes/web.php
Route::resource('posts', 'PostController');
```

Controller Files

```
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Models\Post;

class PostController extends Controller
{

    public function index()
    {
        $posts = Post::all();
        return view('posts.index', compact('posts'));
    }

    public function create()
    {
        return view('posts.create');
    }


    public function store(Request $request)
    {
        // Validation rules can be added here
        $data = $request->validate([
            'title' => 'required|max:255',
            'content' => 'required',
        ]);

        Post::create($data);

        return redirect()->route('posts.index');
    }


    public function show($id)
    {
        $post = Post::findOrFail($id);
        return view('posts.show', compact('post'));
    }


    public function edit($id)
    {
        $post = Post::findOrFail($id);
        return view('posts.edit', compact('post'));
    }


    public function update(Request $request, $id)
    {
        // Validation rules can be added here
        $data = $request->validate([
            'title' => 'required|max:255',
            'content' => 'required',
        ]);

        $post = Post::findOrFail($id);
        $post->update($data);

        return redirect()->route('posts.index');
    }

    public function destroy($id)
    {
        $post = Post::findOrFail($id);
        $post->delete();
        return redirect()->route('posts.index');
    }
}

Route::resource('posts', 'PostController'); // by default prefix url is 'posts'
```

HTTP Method	URI	Action/methods	Route Name
GET	/posts	index	posts.index
GET	/posts/create	create	posts.create
POST	/posts	store	posts.store
GET	/posts/	show	posts.show
GET	/posts//edit	edit	posts.edit
PUT/PATCH	/posts/	update	posts.update
DELETE	/posts/	destroy	posts.destroy

Notes: To Show the route List

```
php artisan route:list
```

7. **API Routes:**

- API routes are typically used for building API endpoints and can be defined separately in `routes/api.php`.
- Example: Defining an API route.

```
     // routes/api.php
     Route::get('/user/{id}', 'UserController@show');
```

in Client Side/savascript

```

     import axios from 'axios';

const apiUrl = 'http://localhost:8000'; // Replace with your Laravel server URL

// Make an API request

http://localhost:8000/api/user/id

axios.get(`${apiUrl}/api/user/12`)
    .then(response => {
        // Handle the API response here
    })
    .catch(error => {
        // Handle any errors
    });
```

8. **Route Prefixes:**

- You can add a prefix to a group of routes to group them under a common URL segment.
- Example: Defining route prefixes.

```
// routes/web.php
Route::prefix('admin')->group(function () {
    Route::get('/dashboard', 'AdminController@index');
    // More admin routes
});
```

Certainly! Let's explore some advanced level routing concepts and techniques in Laravel:

9. **Route Model Binding:**

- Laravel provides a feature called "Route Model Binding" that allows you to automatically inject model instances into your routes.
- Example: Using route model binding to fetch a `Post` model by its `id` and display it.

```
// routes/web.php
Route::get('/post/{post}', 'PostController@show');

// app/Http/Controllers/PostController.php
public function show(Post $post)
{
    // $post is automatically injected based on the {post} parameter
    return view('posts.show', compact('post'));
}
```

10. **Route Prefixes with Parameters:**

- You can use route parameters within route prefixes to create dynamic URL segments.
- Example: Creating dynamic route prefixes based on user roles.

```
// routes/web.php
Route::prefix('{role}/dashboard')->group(function () {
    Route::get('/', 'DashboardController@index');
});
```

In this example, `{role}` is a route parameter that can be used to create role-based dashboard URLs like `/admin/dashboard` and `/user/dashboard`.

11. **Route Constraints:**

- You can define constraints on route parameters to control the values they can accept.
- Example: Using a route constraint to restrict a parameter to numeric values.

```
// routes/web.php
Route::get('/user/{id}', 'UserController@show')->where('id', '[0-9]+');

// Only matches URLs like /user/1, /user/2, etc.
```

12. **Route Model Binding with Custom Columns:**

- You can bind route parameters to model instances based on custom columns other than the primary key (`id`).
- Example: Using route model binding with a custom column, such as `slug`.

```
// routes/web.php
Route::get('/post/{post:slug}', 'PostController@show');

// Binds the {post} parameter to a Post model where 'slug' matches.
```

13. **Nested Resource Routes:**

- When dealing with relationships between models, you can use nested resource routes to represent those relationships.
- Example: Defining nested resource routes for `Post` and `Comment` models.

```
// routes/web.php
Route::resource('posts.comments', 'CommentController');

// Generates routes for creating, updating, and deleting comments related to posts.
```

14. **Route Middleware Groups:**

- You can create custom middleware groups to apply multiple middleware to a group of routes.
- Example: Creating a custom middleware group for admin routes.

```
// app/Http/Kernel.php
protected $middlewareGroups = [
    'admin' => [
        'auth',
        'admin.check',
    ],
];

// routes/web.php
Route::middleware(['admin'])->group(function () {
    Route::get('/admin/dashboard', 'AdminController@index');
    // More admin routes
});
```

15. **Optional Parameters:**

- You can make route parameters optional by providing default values.
- Example: Using optional parameters to handle different search scenarios.

```
// routes/web.php
Route::get('/search/{keyword?}', 'SearchController@index')->where('keyword', '.*');

// The {keyword} parameter is optional, and the regular expression .* allows for any characters.
```

Certainly! Here are some more advanced routing concepts and techniques in Laravel:

8. **Route Model Binding with Multiple Parameters:**

- You can use multiple route parameters for route model binding when needed.
- Example: Binding a `Post` model by both `id` and `slug` parameters.

```
// routes/web.php
Route::get('/post/{id}/{slug}', 'PostController@show')->where('id', '[0-9]+');

// app/Http/Controllers/PostController.php
public function show($id, $slug)
{
    $post = Post::where('id', $id)->where('slug', $slug)->firstOrFail();
    return view('posts.show', compact('post'));
}
```

9. **Route Model Binding with Custom Resolvers:**

- You can use custom resolvers to specify how a model should be retrieved from the database.
- Example: Using a custom resolver to fetch a `User` model by a unique field other than `id`.

```
// app/Providers/RouteServiceProvider.php
public function boot()
{
    parent::boot();

    Route::bind('user', function ($value) {
        return User::where('username', $value)->firstOrFail();
    });
}

// routes/web.php
Route::get('/user/{user}', 'UserController@show');

// The {user} parameter is resolved based on the 'username' field.
```

10. **Route Prefixes with Middleware:**

~~~
- Combine route prefixes with middleware to create dynamic routes with specific middleware applied.
- Example: Creating route groups with prefixes and middleware for user roles.

  ```php
  // routes/web.php
  Route::prefix('admin')->middleware('admin')->group(function () {
      Route::get('/dashboard', 'AdminController@index');
  });
  ```
~~~

11. **Nested Middleware Groups:**

~~~
- You can nest middleware groups within other groups to create complex middleware arrangements.
- Example: Nesting middleware groups for more granular control.

  ```php
  // app/Http/Kernel.php
  protected $middlewareGroups = [
      'web' => [
          // ...
      ],
      'admin' => [
          'auth',
          'admin.check',
      ],
  ];

  // routes/web.php
  Route::middleware(['web', 'admin'])->group(function () {
      Route::get('/admin/dashboard', 'AdminController@index');
      // More admin routes
  });
  ```
~~~

12. **Route Caching:**

~~~
- Laravel provides route caching to speed up route registration in production.
- Example: Generate a cached routes file for better performance.

  ```bash
  php artisan route:cache
  ```

  This can significantly improve the performance of your application by reducing the time needed to register routes.
~~~

13. **Route Model Binding with Closure:**

~~~
- You can use a closure-based route to bind a model based on a custom logic.
- Example: Binding a `User` model based on a custom logic.

  ```php
  // routes/web.php
  Route::get('/user/{user}', function (User $user) {
      return view('user.profile', compact('user'));
  })->where('user', '^(?!admin).*'); // Exclude 'admin' usernames
  ```
~~~

Certainly, let's explore some additional advanced routing concepts in Laravel:

14. **Route Resource Naming:**

- You can customize the resource naming conventions when defining resource routes.
- Example: Customizing the resource route names for a `Product` resource.

```
// routes/web.php
Route::resource('products', 'ProductController')->names([
    'create' => 'products.build',
    'store' => 'products.save',
]);
```

In this example, you've customized the route names for the create and store actions.

15. **Route Model Binding with Explicit Binding:**

- You can use explicit binding to manually specify how a model should be bound to a route.
- Example: Manually binding a `Category` model by its `slug` attribute.

```
// app/Providers/RouteServiceProvider.php
public function boot()
{
    parent::boot();

    Route::bind('category', function ($value) {
        return Category::where('slug', $value)->firstOrFail();
    });
}

// routes/web.php
Route::get('/category/{category}', 'CategoryController@show');
```

16. **API Resource Routes:**

- When building RESTful APIs, you can define resource routes exclusively for APIs.
- Example: Defining API resource routes for a `Product` resource.

```
// routes/api.php
Route::apiResource('products', 'ProductApiController');
```

17. **Route Model Binding with Implicit Binding:**

- Laravel provides implicit model binding where route parameters match the model's route key name.
- Example: Using implicit binding for a `Product` model by its default key `id`.

```
// routes/web.php
Route::get('/product/{product}', 'ProductController@show');

// The {product} parameter is automatically bound to a Product model.
```

18. **Route Model Binding with Custom Keys:**

- You can specify custom route keys for model binding, allowing you to bind models using different attributes.
- Example: Using a custom route key for a `Coupon` model.

```
// app/Coupon.php
protected $routeKey = 'code';

// routes/web.php
Route::get('/coupon/{coupon}', 'CouponController@show');
```

19. **Subdomain Routing:**

- Laravel supports subdomain routing, enabling you to define routes based on subdomains.
- Example: Defining routes for a subdomain-based feature.

```
// routes/web.php
Route::domain('admin.example.com')->group(function () {
    Route::get('/dashboard', 'AdminDashboardController@index');
});
```

20. **Route Model Binding with Optional Parameters:**

- You can use optional route parameters combined with model binding for more flexible routing.
- Example: Binding a `Product` model by both `id` and `optional` attribute.

```
// routes/web.php
Route::get('/product/{product}/{optional?}', 'ProductController@show');
```

```
#### 21. **Route Middleware with Parameters:**
```

- You can pass parameters to route middleware, allowing for more dynamic middleware behavior.
- Example: Using middleware with parameters to check user roles.

```
// app/Http/Middleware/CheckRole.php
public function handle($request, Closure $next, $role)
{
    if ($request->user()->role !== $role) {
        abort(403, 'Unauthorized');
    }

    return $next($request);
}

// routes/web.php
Route::middleware('role:admin')->group(function () {
    Route::get('/admin/dashboard', 'AdminController@index');
});
```

Route controller, Prefix, name Prefix and Middleware together

---

To assign controller, Prefix, name Prefix and Middleware to all routes within a group.We can assigned the methods by chaning operator(->) after the first static method(added with ::). you may use the middleware method before defining the group. Middleware are executed in the order they are listed in the array.['auth','admin'].

```
/* Poly Morphic Relation Route */ // If middleware is found
Route::middleware('auth')->prefix('polymorphic/')->name('polymorphic.')->controller(PolyMorphicController::class)->group(function(){
    Route::get('one-to-one','OneToOnePolyMorphic')->name('OneToOne');
     // This url is polymorphic/one-to-one and  name route is polymorphic.OneToOne
});
```

Custom Error with Code

```
Route::get('/show-view', function(){
abort_if(!isset($address),404);
return view('student.data',['my_data'=>$address]);
});
```

Route Caching

When deploying your application to production, you should take advantage of Laravel's route cache. Using the route cache will drastically decrease the amount of time it takes to register all of your application's routes. To generate a route cache, execute the route:cache Artisan command:

```
php artisan route:cache
```

After running this command, your cached routes file will be loaded on every request. Remember, if you add any new routes you will need to generate a fresh route cache. Because of this, you should only run the route:cache command during your project's deployment.

You may use the route:clear command to clear the route cache:

```
php artisan route:clear
```

Middleware

Middleware in Laravel is a series of filters or layers through which an HTTP request passes. Each middleware performs a specific task, such as authentication, logging, or data manipulation. Middleware is executed in a sequential order defined in your application, allowing you to process requests at various stages of the HTTP request lifecycle.

Middleware & Responses

Of course, a middleware can perform tasks before or after passing the request deeper into the application. For example, the following middleware would perform some task before the request is handled by the application:

```
<?php

namespace App\Http\Middleware;

use Closure;

class BeforeMiddleware
{
    public function handle($request, Closure $next)
    {
        // Perform action

        return $next($request);
    }
}
```

Here are some key aspects of middleware in Laravel:

CREATE MIDDLEWARE

1. **Middleware Creation:**

- You can create custom middleware using Artisan commands or manually in the `app/Http/Middleware` directory.
- Example: Creating a custom middleware that checks if a user is an admin.

```
php artisan make:middleware CheckAdmin
```

- After generating the middleware, you can define its logic in the `handle` method.

```
// app/Http/Middleware/CheckAdmin.php
public function handle($request, Closure $next)
{
    if (auth()->check() && auth()->user()->isAdmin()) {
        return $next($request);
    }

    return redirect('/home');
}
```

2. **Middleware Registration:**

- You must register custom middleware in the `app/Http/Kernel.php` file within the `$middleware` or `$routeMiddleware` property.
- Example: Registering the custom `CheckAdmin` middleware.

```
// app/Http/Kernel.php
protected $routeMiddleware = [
    'admin' => \App\Http\Middleware\CheckAdmin::class,
    // ...
];
```

3. **Global Middleware:**

- Global middleware is executed for every HTTP request in your application.
- Example: Using global middleware for HTTP logging.

```
// app/Http/Kernel.php
protected $middleware = [
    \App\Http\Middleware\HttpLogger::class,
    // ...
];
```

4. **Route Middleware:**

- Route-specific middleware is applied to specific routes or route groups.
- Example: Applying the `auth` middleware to a route.

```
// routes/web.php
Route::get('/dashboard', 'DashboardController@index')->middleware('auth');
```

5. **Middleware Groups:**

- You can group middleware for easy application to multiple routes.
- Example: Creating a middleware group for authentication.

```
// app/Http/Kernel.php
protected $middlewareGroups = [
    'web' => [
        \App\Http\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
        \Illuminate\Session\Middleware\StartSession::class,
        // ...
    ],
    'api' => [
        'throttle:60,1',
        'auth:api',
    ],
];
```

6. **Middleware Parameters:**

- Middleware can accept parameters, allowing for dynamic behavior.
- Example: Passing a role parameter to a custom `CheckRole` middleware.

```
// app/Http/Middleware/CheckRole.php
public function handle($request, Closure $next, $role)
{
    if (auth()->check() && auth()->user()->hasRole($role)) {
        return $next($request);
    }

    abort(403, 'Unauthorized');
}
```

- Usage in a route definition:

```
// routes/web.php
Route::get('/admin', 'AdminController@index')->middleware('role:admin');
```

7. **Middleware Execution Order:**

- Middleware is executed in the order it is listed in the `$middleware` and `$middlewareGroups` properties.
- Example: Controlling the execution order of middleware.

```
// app/Http/Kernel.php
protected $middleware = [
    \App\Http\Middleware\FirstMiddleware::class,
    \App\Http\Middleware\SecondMiddleware::class,
    // ...
];
```

8. **Terminable Middleware:**

- Some middleware may have a `terminate` method, allowing you to perform actions after the response is sent to the browser.
- Example: Logging requests after they are handled.

```
// app/Http/Middleware/LogRequest.php
public function terminate($request, $response)
{
    // Log the request details to a log file or database
}
```

Certainly! Let's create a SubscriptionMiddleware as an example. This middleware will check if a user has an active subscription to access certain routes. Here are the steps:

Step 1: Create the Middleware

1. **Generate the Middleware:**
   - Use the Artisan command to create the middleware. Replace `SubscriptionMiddleware` with the desired middleware name:

```
php artisan make:middleware SubscriptionMiddleware
```

This will create a new middleware file in the `app/Http/Middleware` directory.

2. **Edit the Middleware Logic:**
   - Open the newly created middleware file (`SubscriptionMiddleware.php`) in a text editor or your IDE.
3. **Implement the Middleware Logic:**
   - In the `handle` method, check if the authenticated user has an active subscription. You can customize the logic based on your subscription system.

```
// app/Http/Middleware/SubscriptionMiddleware.php

public function handle($request, Closure $next)
{
    // Check if the user is authenticated
    if (auth()->check()) {
        // Check if the user has an active subscription
        if (auth()->user()->hasActiveSubscription()) {
            // User has an active subscription, proceed to the next middleware or route handler
            return $next($request);
        }
    }

    // User does not have an active subscription, return a response (e.g., 403 Forbidden)
    return response('Unauthorized. You do not have an active subscription.', 403);
}
```

In this example, we assume there's a `hasActiveSubscription` method on the `User` model to check for an active subscription. You should adapt this logic to your subscription system.

Step 2: Register the Middleware

1. **Register the Middleware:**
    - Open the `app/Http/Kernel.php` file.
2. **Add the Middleware to `$routeMiddleware`:**
    - Inside the `$routeMiddleware` property, add an entry for your custom middleware. You can choose a key (e.g., 'subscription') to use when applying the middleware to routes.

```
// app/Http/Kernel.php

protected $routeMiddleware = [
    // ...
    'subscription' => \App\Http\Middleware\SubscriptionMiddleware::class,
];
```

Step 3: Apply the Middleware to Routes

1. **Use the Middleware in Routes:**
    - In your routes file (e.g., `routes/web.php` or `routes/api.php`), apply the middleware to specific routes or route groups that require an active subscription.

```
// routes/web.php

Route::middleware(['subscription'])->group(function () {
    // Routes that require an active subscription
    Route::get('/premium-content', 'PremiumContentController@index');
    // More subscription-based routes
});
```

In this example, the `'subscription'` middleware is applied to all routes within the group, ensuring that only users with an active subscription can access them.

Step 4: Implement Subscription Logic in the User Model

1. **Implement the `hasActiveSubscription` Method:**
    - In your `User` model (`app/User.php`), create a method to check if the user has an active subscription. You'll need to interact with your subscription system or database to determine this.

```
// app/User.php

public function hasActiveSubscription()
{
    // Implement your logic here to check for an active subscription
    return $this->subscriptions->where('status', 'active')->count() > 0;
}
```

Notes : Customize this method to fit your subscription model and logic.

CSRF Protection in Laravel:

Cross-Site Request Forgery (CSRF) is an attack that tricks a user into executing unwanted actions on a web application without their knowledge or consent. Laravel provides built-in CSRF protection to mitigate this threat.

Here are the key aspects of CSRF protection in Laravel:

1. **CSRF Tokens:**

- Laravel generates unique CSRF tokens for each user session.
- These tokens are included in forms as hidden fields or can be added to AJAX requests.
- Tokens are used to verify that the incoming request was made from your application and not from a malicious source.

2. **Middleware:**

- The `VerifyCsrfToken` middleware is included by default in Laravel's middleware stack.
- This middleware checks the CSRF token on every incoming POST, PUT, or DELETE request.

3. **CSRF Token Blade Directive:**

- You can use the `@csrf` Blade directive to generate a hidden input field containing the CSRF token.
- Include this directive in your forms to ensure CSRF protection.

4. **AJAX Requests:**

- When making AJAX requests, you should include the CSRF token in the request headers or data.
- Laravel provides a JavaScript variable, `csrf_token`, which you can use to fetch the token.

Example Usage in Blade Views:

Let's go through a real-life example of adding CSRF protection to a form in a Blade view:

```
    <form method="POST" action="/contact">
        @csrf <!-- Add the CSRF token -->
        <input type="text" id="name" name="name" required><br>
        <input type="email" id="email" name="email" required><br>
        <button type="submit">Submit</button>
    </form>
```

In this example:

- We include the `@csrf` Blade directive inside the `<form>` tag to generate a hidden input field containing the CSRF token.
- When the form is submitted, Laravel's `VerifyCsrfToken` middleware will check if the token in the request matches the one generated for the user's session.

AJAX Request Example:

Here's how you can include the CSRF token in an AJAX request using JavaScript and jQuery:

```
<script>
    // Fetch the CSRF token from the meta tag
    const csrfToken = document.querySelector('meta[name="csrf-token"]').getAttribute('content');

    // Make an AJAX POST request
    $.ajax({
        url: '/api/some-endpoint',
        type: 'POST',
        data: {
            _token: csrfToken, // Include the CSRF token
            // Other request data
        },
        success: function (response) {
            // Handle the response
        },
        error: function (xhr, status, error) {
            // Handle errors
        }
    });
</script>
```

In this AJAX request example:

- We fetch the CSRF token from a meta tag using JavaScript.
- The `_token` parameter is included in the request data with the CSRF token.

This ensures that the AJAX request is protected against CSRF attacks.

CSRF & XSRF

Certainly, let's dive deeper into CSRF (Cross-Site Request Forgery) and XSRF (Cross-Site Request Forgery) protection in Laravel, including more advanced aspects and use cases.

1. CSRF Token Validation in AJAX Requests:

- When making AJAX requests, you can include the CSRF token in the headers instead of form fields.

- Example using Axios in a Vue.js component:

```
// Include CSRF token in Axios headers
axios.defaults.headers.common['X-CSRF-TOKEN'] = document.querySelector('meta[name="csrf-token"]').getAttribute('content');

// Make an AJAX POST request
axios.post('/api/some-endpoint', {
    // Request data
})
.then(response => {
    // Handle the response
})
.catch(error => {
    // Handle errors
});
```

- By including the token in the headers, you ensure that AJAX requests are properly validated.

2. Excluding Routes from CSRF Protection:

- In some cases, you may need to exclude specific routes from CSRF protection, such as API routes used by external services.
- Example: Excluding an API route from CSRF protection by adding it to the `except` array in `VerifyCsrfToken` middleware.

```
// app/Http/Middleware/VerifyCsrfToken.php

protected $except = [
    '/api/webhook',
];
```

- Be cautious when excluding routes, and only do so when it's necessary and safe.

3. CSRF Protection in Blade Components:

- Laravel Blade components can also make use of the `@csrf` directive when they include forms.
- Example using a Blade component:

```
<!-- resources/views/components/contact-form.blade.php -->
<form method="POST" action="{{ route('contact.submit') }}">
    @csrf <!-- Add the CSRF token -->
    <!-- Form fields -->
</form>
```

4. Customizing the CSRF Token Field Name:

- By default, Laravel expects the CSRF token to be named `_token`. You can customize this field name.
- Example: Customizing the CSRF field name to `csrf_token` in a form.

```
<form method="POST" action="/some-route">
    @csrf_field('csrf_token') <!-- Customize the field name -->
    <!-- Form fields -->
</form>
```

5. Cross-Origin Request Forgery (XSRF) Tokens:

- Laravel allows you to use XSRF tokens for enhanced security in SPA (Single Page Applications) and API requests.
- Example: Using XSRF tokens in a Vue.js SPA:

```
// Include XSRF token in Axios headers
axios.defaults.headers.common['X-XSRF-TOKEN'] = Cookies.get('XSRF-TOKEN');

// Make an AJAX POST request
axios.post('/api/some-endpoint', {
    // Request data
})
.then(response => {
    // Handle the response
})
.catch(error => {
    // Handle errors
});
```

- XSRF tokens offer additional security for applications that separate the front-end and back-end.

6. CSRF Token Verification in Custom Controllers:

- If you have custom controllers that handle form submissions, you can verify the CSRF token using the `csrf` middleware.
- Example: Verifying the CSRF token in a custom controller method.

```
// app/Http/Controllers/CustomController.php

public function store(Request $request)
{
    $this->middleware('csrf');

    // Handle form submission
}
```

- This way, you can add CSRF protection to non-route-closure-based controllers.

Controllers in Laravel:

Controllers in Laravel are PHP classes responsible for handling incoming HTTP requests and returning appropriate responses. They act as intermediaries between the routes and the application's logic, making it easier to separate concerns and maintain a clean codebase.

Here are the key aspects of controllers in Laravel:

1. Creating Controllers:

- You can create controllers using Artisan commands. For example, to create a controller named `UserController`, run:

```
php artisan make:controller UserController
```

- This command generates a new controller file in the `app/Http/Controllers` directory.

2. Controller Methods:

- Controllers contain methods that correspond to different actions or routes. For instance, a `UserController` might have methods like `index`, `show`, `store`, `update`, and `destroy` to handle various CRUD (Create, Read, Update, Delete) operations.

3. Request Handling:

- Controllers receive incoming requests as method arguments, typically using type-hinted Request objects.
- Example method in a controller:

```
public function store(Request $request)
{
    // Handle the incoming request
    $data = $request->all();
    // Process and store data
}
```

4. Returning Responses:

- Controllers return responses to the client, which can be HTML views, JSON data, or other formats.
- Example of returning a view in a controller method:

```
public function index()
{
    return view('users.index');
}
```

5. Route Binding:

- Laravel provides route model binding, allowing you to automatically inject model instances into controller methods.
- Example of route model binding in a controller:

```
public function show(User $user)
{
    return view('users.show', compact('user'));
}
```

6. Middleware:

- You can apply middleware to controller methods to perform tasks like authentication, authorization, and input validation.
- Example of applying middleware to a controller:

```
public function __construct()
{
    $this->middleware('auth');
}
```

7. Dependency Injection:

- You can use dependency injection to inject services or other classes into your controllers, making them more testable and maintainable.
- Example of dependency injection in a controller constructor:

```
public function __construct(UserService $userService)
{
    $this->userService = $userService;
}

public function index()
{
    $users = $this->userService->getAllUsers();
    return view('users.index', compact('users'));
}
```

8. Resource Controllers:

- Laravel provides resource controllers to quickly generate CRUD routes and methods for a resource (e.g., users, posts).
- Example of creating a resource controller:

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

9. API Controllers:

- Laravel offers API controllers for building RESTful APIs, which often return JSON responses.
- Example of creating an API controller:

```
php artisan make:controller ApiController
```

10. Controller Middleware Groups:

- You can group middleware at the controller level, applying it to multiple methods within the controller.
- Example: Applying the `auth` middleware to all methods in a controller.

```
public function __construct()
{

   $this->middleware('auth');
   $this->middleware('log')->only('index');
   $this->middleware('subscribed')->except('store');
   $this->middleware('auth')->except('publicMethod');
}
```

- This allows you to specify middleware behavior for the entire controller while excluding specific methods.

11. Resourceful Resource Controllers:

- Laravel provides resourceful controllers for common CRUD operations. These controllers include methods like `index`, `create`, `store`, `show`, `edit`, `update`, and `destroy`.
- Example of using resourceful controller routes:

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

- This single route definition generates all the necessary routes for CRUD operations on the `Post` model.

12. Controller Resource Validation:

- You can validate request data within controller methods using validation rules.
- Example of validating data in a controller method:

```
public function store(Request $request)
{
    $validatedData = $request->validate([
        'title' => 'required|string|max:255',
        'content' => 'required|string',
    ]);

    // If validation fails, Laravel will automatically redirect back with errors
    // If validation passes, continue processing the request
}
```

- Laravel handles the validation process and can automatically return validation errors to the view.

13. Controller Dependency Injection with Custom Classes:

- You can inject your own custom classes or services into controller methods.
- Example of injecting a custom service into a controller method:

```
public function store(Request $request, MyService $service)
{
    $result = $service->processData($request->input('data'));
    // Handle the result
}
```

- This allows you to keep your controllers lean by delegating complex logic to dedicated services.

14. Invokable Controllers:

- Laravel supports invokable controllers, which are single-action controllers defined as classes with an `__invoke` method.
- Example of an invokable controller:

```
class MyController
{
    public function __invoke()
    {
        return 'This is an invokable controller action.';
    }
}
```

- You can route to invokable controllers directly, simplifying routes for single-action endpoints.

15. Controller Testing:

- Laravel's testing tools allow you to write tests for your controllers.
- Example of a controller test using PHPUnit:

```
public function testIndex()
{
    $response = $this->get('/posts');

    $response->assertStatus(200);
    $response->assertViewIs('posts.index');
}
```

- Controller testing helps ensure that your application's behavior is consistent and reliable.

16. Controller Namespace:

- You can organize your controllers into namespaces to better structure your application.
- Example of defining a controller within a namespace:

```
namespace App\Http\Controllers\Admin;

class AdminController extends Controller
{
    // Controller methods
}
```

- Namespaced controllers help keep your code organized, especially in larger applications.

Certainly, let's explore some real-life examples of controller methods in Laravel that involve various logic, such as multiple queries, route model binding, and filtering data. We'll use a hypothetical e-commerce application as an example.

Real-Life Controller Methods:

1. **Display Products with Filters:**

- Purpose: Display a list of products with filtering options by price, review rating, color, and size.
- Logic: Query the database to retrieve products based on user-selected filters.
- Code Example:

```
public function filterProducts(Request $request)
{
    $query = Product::query();

    // Filter by price range
    if ($request->has('price')) {
        $priceRange = explode('-', $request->input('price'));
        $query->whereBetween('price', $priceRange);
    }

    // Filter by review rating
    if ($request->has('rating')) {
        $rating = $request->input('rating');
        $query->where('rating', '>=', $rating);
    }

    // Filter by color
    if ($request->has('color')) {
        $color = $request->input('color');
        $query->where('color', $color);
    }

    // Filter by size
    if ($request->has('size')) {
        $size = $request->input('size');
        $query->where('size', $size);
    }

    $products = $query->get();

    return view('products.index', compact('products'));
}
```

2. **View Product Details with Route Model Binding:**

- Purpose: Display detailed information about a specific product.
- Logic: Use route model binding to fetch product details based on the product's slug or ID.
- Code Example:

```
public function show(Product $product)
{
    return view('products.show', compact('product'));
}
```

3. **Add or Update Product Information in One Method:**

- Purpose: Handle both adding and updating product information in a single method.
- Logic: Check if the request contains a product ID; if it does, update the existing product; otherwise, create a new product.
- Code Example:

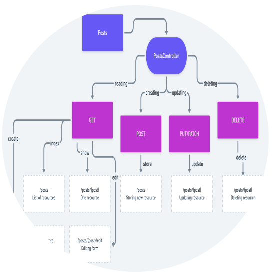
Beginner Level OF Add or Update Product Information in One Method

```
// Beginner Level

public function storeOrUpdate(Request $request, $id = null)
{
    $validatedData = $request->validate([
        'name' => 'required|string|max:255',
        'description' => 'required|string',
        'price' => 'required|numeric',
        'color' => 'required|string',
        'size' => 'required|string',
    ]);

    if ($id) {
        $product = Product::findOrFail($id);
        $product->update($validatedData);
    } else {
        Product::create($validatedData);
    }

    return redirect('/products')->with('success', 'Product saved successfully');
}
```

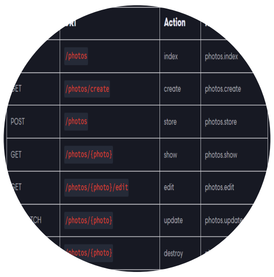
Advanced Level OF Add or Update Product Information in One Method

```
use Illuminate\Support\Facades\Auth;
use Illuminate\Validation\Rule;
use Illuminate\Support\Facades\Storage;

public function storeOrUpdate(Request $request, $id = null)
{
    // Validate the incoming request data
    $validatedData = $request->validate([
        'name' => [
            'required',
            'string',
            'max:255',
            Rule::unique('products')->ignore($id), // Ignore the current product if updating
        ],
        'description' => 'required|string',
        'price' => 'required|numeric',
        'color' => 'required|string',
        'size' => 'required|string',
        'image' => 'nullable|image|mimes:jpeg,png,jpg,gif|max:2048', // Optional image upload
    ]);

    // Check if the currently authenticated user is authorized to update the product
    if (Auth::id() != $id) {
        return redirect('/products')->with('error', 'Unauthorized. You can only update your own products.');
    }

    // Retrieve the existing product or create a new one
    $product = $id ? Product::findOrFail($id) : new Product();

    // Fill the product model with the validated data
    $product->fill($validatedData);

    // Handle optional image upload
    if ($request->hasFile('image')) {
        // Delete the previous image if updating and a new image is uploaded
        if ($id && $product->image) {
            Storage::delete($product->image);
        }

        // Store the new image and update the product's image path
        $imagePath = $request->file('image')->store('product_images', 'public');
        $product->image = $imagePath;
    }

    // Save the product model to the database
    $product->save();

    return redirect('/products')->with('success', 'Product saved successfully');
}


// Try Catch Block with Commit and Rollback

    if (Auth::id() != $id) {
        return redirect('/products')->with('error', 'Unauthorized. You can only update your own products.');
    }

    try {
        // Begin a database transaction
        DB::beginTransaction();

        if ($id) {
            $product = Product::findOrFail($id);
        } else {
            $product = new Product();
        }

        // Fill the product model with validated data
        $product->fill($validatedData);

        // Handle optional image upload
        if ($request->hasFile('image')) {
            // Delete the previous image if updating and a new image is uploaded
            if ($id && $product->image) {
                Storage::delete($product->image);
            }

            $imagePath = $request->file('image')->store('product_images', 'public');
            $product->image = $imagePath;
        }

        $product->save();

        // Commit the transaction
        DB::commit();

        return redirect('/products')->with('success', 'Product saved successfully');
    } catch (\Exception $e) {
        // If an error occurs, rollback the transaction and handle the error
        DB::rollback();
        return redirect('/products')->with('error', 'An error occurred while saving the product.');
    }
```

Resource Controller Images



Actions Handled By Resource Controller



	RI	Action	
	/photos	index	photos.index
ET	/photos/create	create	photos.create
POST	/photos	store	photos.store
GET	/photos/{photo}	show	photos.show
ET	/photos/{photo}/edit	edit	photos.edit
TCH	/photos/{photo}	update	photos.update
	/photos/{photo}	destroy	

Request and Response

Request(Dependency Injection) : Laravel's Illuminate\Http\Request class provides an object-oriented way to interact with the current HTTP request being handled by your application as well as retrieve the input, cookies, and files that were submitted with the request.

Handling Requests:

In Laravel, requests refer to the incoming HTTP requests made to your application. These requests can be of various types, such as GET, POST, PUT, DELETE, etc. Laravel provides a convenient way to handle and process these requests.

1. **Basic Request Handling:** Laravel provides a convenient way to access data from incoming HTTP requests. Here's an example of how to retrieve data from a GET request in a Laravel controller method:

```
public function index(Request $request)
{
    $name = $request->input('name');
    return "Hello, $name!";
}
```

2. **Validation of Requests:** You can validate incoming data using Laravel's validation rules. For instance, ensuring that an email field is present and valid:

```
$validatedData = $request->validate([
    'email' => 'required|email',
]);
```

3. **Uploading Files:** Handling file uploads is straightforward in Laravel. You can access and store uploaded files like this:

```
$file = $request->file('file');
$file->store('uploads');
```

Response

Response can be string, array or Object. All routes and controllers should return a response to be sent back to the user's browser. Laravel provides several different ways to return responses. The most basic response is returning a string from a route or controller. The framework will automatically convert the string into a full HTTP response:

```
Route::get('/', function () {
    return 'Hello World'; // string
    return [12,34,55,65]; // array

         return response('Hello World', 200)
                  ->header('Content-Type', 'text/plain'); // object
                  or
                  $model = Product::all();
                return response(Json($model)); // this response for ajax call request

            # Redirect Route with carrying Data

                return redirect()->route('profile', [$user]);

                # Redirecting With Flashed Session Data
                 return redirect('dashboard')->with('status', 'Profile updated!');

                 # redirect with downlaod
                 return response()->download($pathToFile);
});
```

Sending Responses:

1. **Returning JSON Responses:**
    - Used when you want to return data in JSON format, often for AJAX requests or APIs.
    - Example: Returning a JSON response with user data.

```
public function getUserData($id)
{
    $user = User::find($id);
    return response()->json(['user' => $user]);
}
```

2. **Redirecting to URLs/Redirect Response:**
    - Used to redirect the user to a different URL.
    - Example: Redirecting after a successful form submission.

```
public function store(Request $request)
{
    // Process form data
    return redirect('/thank-you');
}
```

3. **View Responses/ HTML Response:** To return a view as a response, you can use the `view()` method:

```
return view('welcome');
```

4. **Customizing Responses:** You can set custom HTTP headers and status codes:

```
return response('Unauthorized', 401)
    ->header('Content-Type', 'text/plain');
```

5. **Attachments (e.g., PDF):** To send a file as a response, like a PDF:

```
return response()->file($pathToFile);
```

VIEW In Laravel

To create View File

```
php artisan make:view frontend/welcome
```

Once you're inside the project directory, you can create a new view and folder in the frontend directory using the `mkdir` command (for creating a folder) and the `touch` command (for creating a view file). Replace `view_name` with the name of your view and `folder_name` with the name of the folder:

```
mkdir resources/views/frontend/folder_name
touch resources/views/frontend/folder_name/view_name.blade.php
```

Sharing Data Views

In Laravel, there are several ways to share data with views to pass information from the backend to the frontend. Here are some common methods:

1. **Using the `view` function:** The simplest way is to use the `view` function to create a view and pass data to it as an associative array.

```
$data = ['key' => 'value'];
return view('viewName', $data);
```

2. **Using the `with` method:** You can use the `with` method to pass individual variables to the view.

```
return view('viewName')->with('variableName', $variableValue);
```

3. **Using the `compact` function:** The `compact` function allows you to pass variables to the view by specifying their names as arguments.

```
$variable = 'some value';
return view('viewName', compact('variable'));
```

4. **Using the `@php` Blade directive:** In your Blade templates, you can use the `@php` directive to write PHP code and set variables.

```
@php
    $variable = 'some value';
@endphp
```

5. **Using view composers:** View composers allow you to bind data to specific views or view templates using service providers. This is helpful when you want to share data with multiple views.

```
View::composer('viewName', function ($view) {
    $view->with('variableName', $variableValue);
});
```

6. **Using view creators:** Similar to view composers, view creators let you share data with views, but they allow for more fine-grained control as they are tied to specific view classes.

```
View::creator('App\SomeViewClass', function ($view) {
    $view->with('variableName', $variableValue);
});
```

7. **Using the `@inject` Blade directive:** The `@inject` directive allows you to inject a service or value into your view directly from a service container.

```
@inject('serviceName', 'Namespace\ServiceClass')
```

8. **Using shared views:** You can create a shared view that includes data you want to share across multiple views. This can be included in other views using the `@include` directive.

9. **Basic View:** A basic view is a simple HTML template. Here's an example:

```
<!-- resources/views/welcome.blade.php -->
<!DOCTYPE html>
<html>
<head>
    <title>Welcome Page</title>
</head>
<body>
    <h1>Welcome to My Laravel App</h1>
</body>
</html>
```

10. **Blade Template Inheritance:**

Blade allows you to create a master layout and extend it in child views. This promotes code reusability. Here's an example:

```
<!-- resources/views/layouts/app.blade.php -->
<!DOCTYPE html>
<html>
<head>
    <title>@yield('title')</title>
</head>
<body>
    @yield('content')
</body>
</html>
```

```
<!-- resources/views/home.blade.php -->
@extends('layouts.app')

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

@section('content')
    <h1>Welcome to the Home Page</h1>
@endsection
```

11. **Conditional Views:**

You can conditionally display content in your views using Blade directives. For example:

```
<!-- resources/views/conditional.blade.php -->
@if($condition)
    <p>This content is displayed because the condition is true.</p>
@else
    <p>This content is displayed when the condition is false.</p>
@endif
```

12. **Looping in Views:**

    Blade provides directives for looping through data. Here's an example of looping through an array:

```
<!-- resources/views/loop.blade.php -->
<ul>
    @foreach($items as $item)
        <li>{{ $item }}</li>
    @endforeach
</ul>
```

13. **Including Sub-Views:**

    You can include sub-views within your main view using the `@include` directive. For example:

```
<!-- resources/views/main.blade.php -->
<div class="header">
    @include('partials.header')
</div>
<div class="content">
    @include('partials.content')
</div>
```

Blade Inheritance and Layout

1. **Basic Blade Template:** A basic Blade template consists of standard HTML mixed with Blade directives enclosed in double curly braces `{{ }}`. Blade expressions are used to display variables, execute PHP code, and control the flow of your templates.

    ```
    <html>
    <head>
        <title>{{ $title }}</title>
    </head>
    <body>
        <h1>Welcome, {{ $user->name }}</h1>
    </body>
    </html>
    ```

2. **Extending a Layout:** Blade allows you to create reusable layouts and extend them in child views. The `@extends` directive is used to inherit the layout, and the `@section` directive defines sections within the layout that can be customized in child views.

    `layout.blade.php` (Layout File):

    ```
    <html>
    <head>
        <title>@yield('title')</title>
    </head>
    <body>
        @yield('content')
    </body>
    </html>
    ```

    `child.blade.php` (Child View):

```
@extends('layout')

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

@section('content')
    <h1>Hello from Child Page</h1>
@endsection
```

3. **Including Subviews:** You can include subviews within your Blade templates using the `@include` directive. This is useful for reusing components across different views.

```
<div class="header">
    @include('partials.header')
</div>
```

4. **Conditional Statements:** Blade allows you to use conditional statements like `@if`, `@else`, `@elseif`, and `@endif` to conditionally display content based on certain conditions.

```
@if($isAdmin)
    <p>Welcome, Admin!</p>
@else
    <p>Welcome, Guest!</p>
@endif
```

5. **Loops:** You can use Blade directives for loops, such as `@foreach`, `@for`, and `@while`, to iterate through arrays or collections.

```
<ul>
    @foreach($items as $item)
        <li>{{ $item }}</li>
    @endforeach
</ul>
```

6. **Escaping Content:** Blade automatically escapes output by default to prevent XSS attacks. If you want to output unescaped content, you can use the `@html` directive.

```
<p>{!! $unescapedHtml !!}</p>
```

Advanced-Blade Template

1. **Blade Layouts and Nesting:** You can nest layouts within layouts to create a hierarchy of templates. This is useful for building complex page structures.

   `master.blade.php` (Master Layout):

```
<html>
<head>
    <title>@yield('title')</title>
</head>
<body>
    @yield('content')
</body>
</html>
```

   `sublayout.blade.php` (Sub-layout):

```
@extends('master')

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

@section('content')
    <div class="sub-content">
        @yield('sub-content')
    </div>
@endsection
```

`child.blade.php` (Child View):

```
@extends('sublayout')

@section('sub-content')
    <h1>Hello from Child Page</h1>
@endsection
```

2. **Blade Components and Slots:** Laravel introduced Blade components for creating reusable UI components. You can define a Blade component with slots to inject content.

   `button.blade.php` (Component):

```
<button {{ $attributes->merge(['class' => 'btn']) }}>
    {{ $slot }}
</button>
```

   Usage:

```
<x-button class="bg-blue-500">
    Click me
</x-button>
```

3. **Blade Directives for Authentication:** Laravel provides Blade directives for checking the authentication status of users.

```
@auth
    <!-- User is authenticated -->
@else
    <!-- User is not authenticated -->
@endauth
```

4. **Blade Includes with Data:** You can pass data to included views using Blade's `@include` directive.

   `header.blade.php` (Partial View):

```
<h1>{{ $pageTitle }}</h1>
```

   Usage:

```
@include('partials.header', ['pageTitle' => 'Welcome'])
```

5. **Blade Custom Directives:** You can create your custom Blade directives for more advanced functionality. Define these in the `AppServiceProvider`.

```
Blade::directive('myDirective', function ($expression) {
    return "<?php // Your custom code here ?>";
});
```

6. **Blade Comments:** You can add comments in Blade templates that won't be rendered in the HTML output.

```
{{-- This is a Blade comment --}}
```

7. **Blade Templates for Email:** You can use Blade templates to create HTML and plain text email templates. These templates can be sent using Laravel's email functionality.

Example: Creating an Email Template

```
@component('mail::message')
    # Hello, {{ $user->name }}
    Thank you for signing up!

    @component('mail::button', ['url' => $verificationUrl])
        Verify Email
    @endcomponent
@endcomponent
```

ADVANCED-Custom Blade Directives:

1. **Create a Service Provider:** First, create a custom service provider to register your Blade directive. Run the following Artisan command to generate a service provider:

```
php artisan make:provider CustomBladeDirectiveServiceProvider
```

2. **Define the Directive in the Service Provider:** Open the `CustomBladeDirectiveServiceProvider.php` file in the `app/Providers` directory. In the `boot` method, use the `Blade` facade to define your custom directive. For example, let's create a directive to display the current date:

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

public function boot()
{
    Blade::directive('currentDate', function () {
        return "<?php echo now()->format('Y-m-d'); ?>";
    });
}
```

In this example, we've defined a `@currentDate` directive that will output the current date in the 'Y-m-d' format when used in a Blade template.

3. **Register the Service Provider:** Add your custom service provider to the `providers` array in the `config/app.php` configuration file.

```
'providers' => [
    // Other service providers
    App\Providers\CustomBladeDirectiveServiceProvider::class,
],
```

4. **Use the Custom Directive in Blade Templates:** You can now use your custom Blade directive in your Blade templates. For instance, to display the current date, simply use `@currentDate` in your view:

```
<p>Today's Date: @currentDate</p>
```

5. **Compile Your Blade Templates:** If you've made changes to your Blade directives or service providers, run the following Artisan command to recompile your Blade templates:

```
php artisan view:clear
```

6. **Test Your Custom Blade Directive:** Finally, load a page that uses the custom directive in your Laravel application, and you should see the output generated by your custom Blade directive.

By following these steps, you can create and use custom Blade directives in Laravel to encapsulate logic and make your templates more expressive and maintainable.

7. **Stack** You can use the `@push` and `@prepend` directives in a Blade template to add content to a stack. Here's how you can use them together within one section:

```
@section('content')
    <!-- Your main content here -->
@endsection

@push('scripts')
    This will be second...
@endpush

@prepend('scripts')
    This will be first...
@endprepend
```

7. **@inject Directive:** `@inject` directive to inject a service (`MetricsService`) into a Blade view. It then displays the result of a method call from the injected service in the view. Here's a brief explanation with code examples:

   The `@inject` directive allows you to inject an instance of a class or service into a Blade view. In your example, it injects an instance of the `MetricsService` class into the view and assigns it to a variable named `$metrics`.

```
```

@inject('metrics', 'App\Services\MetricsService')

Monthly Revenue: {{ $metrics->monthlyRevenue() }}.

~~~
### **Asset Bundling (Vite)**
---
Asset bundling with Vite is a modern approach to managing and optimizing your frontend assets (JavaScript, CSS, and more) in Laravel

Here's a step-by-step explanation of how to set up asset bundling with Vite in a Laravel application:

1. **Installation**:
  First, make sure you have a Laravel project set up. Then, you can add Vite to your project using npm or yarn:

  ```bash
  npm install -D create-vite
~~~

2. **Create a Vite Configuration**: Run the following command to create a Vite configuration file:

```
npx create-vite
```

   This command will guide you through the setup process. Make sure to configure Vite to output assets in the Laravel public directory.

3. **Configure Laravel Mix**: Open your Laravel Mix configuration file (`webpack.mix.js`) and configure it to copy assets from the Vite output directory to the public directory. Here's an example configuration:

```
mix.copy('resources/assets/css', 'public/css')
   .copy('resources/assets/js', 'public/js');
```

4. **Usage in Blade Templates**: In your Blade templates, you can include the bundled assets like this:

```
<link rel="stylesheet" href="{{ mix('css/app.css') }}">
<script src="{{ mix('js/app.js') }}"></script>
```

   Laravel Mix's `mix()` function will automatically version your assets, ensuring that browsers cache them properly.

5. **Building Assets**: To build your assets during development, you can run Vite in development mode with hot-reloading:

```
npm run dev
```

   For production, you can build optimized assets:

```
npm run build
```

Now, let's provide a real-life and professional code example using Blade templates:

Suppose you have a Blade file for your application layout (`resources/views/layouts/app.blade.php`). You want to include Vite-bundled assets in this layout. Here's how you can do it:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Your Laravel App</title>
    <link rel="stylesheet" href="{{ mix('css/app.css') }}">
</head>
<body>
    <div id="app">
        <!-- Your application content goes here -->
    </div>
    <script src="{{ mix('js/app.js') }}"></script>
</body>
</html>
```

Using Vite with Vue.js:

1. **Install Vue.js**: If you haven't already, install Vue.js in your Laravel project:

```
npm install vue@next
```

2. **Create a Vue Component**: Create a Vue component, let's say `Example.vue`, in your resources directory. This component can be a simple example or a more complex one, depending on your needs.

```
<template>
  <div>
    <h1>Hello Vue.js</h1>
  </div>
</template>

<script>
export default {
  // Your Vue component logic goes here
}
</script>

<style scoped>
/* Your component-specific styles go here */
</style>
```

3. **Include the Vue Component in Blade**: In your Blade file, include the Vue component using a `vue` directive:

```
<div id="app">
  <example></example>
</div>
```

4. **Build Your Vite Configuration**: Configure Vite to handle Vue.js components. In your `vite.config.js`, you might need to use the `@vitejs/plugin-vue` plugin to handle `.vue` files:

```
import vue from '@vitejs/plugin-vue';

export default {
  plugins: [vue()],
  // Other Vite configuration settings
};
```

Using Vite with React:

1. **Install React**: If you're using React, you'll need to install it:

```
npm install react react-dom
```

2. **Create a React Component**: Create a React component, let's say `Example.js`, in your resources directory:

```
import React from 'react';

function Example() {
  return (
    <div>
      <h1>Hello React</h1>
    </div>
  );
}

export default Example;
```

3. **Include the React Component in Blade**: In your Blade file, include the React component:

```
<div id="app">
  <div id="react-app"></div>
</div>
```

4. **Build Your Vite Configuration**: Configure Vite to handle React components. Ensure you have the necessary plugins for React:

```
import react from '@vitejs/plugin-react';

export default {
  plugins: [react()],
  // Other Vite configuration settings
};
```

URL GENERATING

Definition: URL generation in Laravel refers to the process of generating URLs for routes defined in your Laravel application. It simplifies the task of creating links and redirects in your web application by abstracting the underlying URL structure.

Basic URL Generation: In Laravel, you can use the `url()` function or the `route()` function to generate URLs. Here's a basic example using the `url()` function:

```
<a href="{{ url('/about') }}">About Us</a>
```

This code generates a URL for the `/about` route and creates a link to the "About Us" page.

Named Routes: Named routes provide a cleaner and more maintainable way to generate URLs. You define a name for your routes in the `web.php` routes file like this:

```
Route::get('/about', 'AboutController@index')->name('about');
```

Then, in your Blade file, you can use the `route()` function with the route name:

```
<a href="{{ route('about') }}">About Us</a>
```

URL Parameters: You can also generate URLs with parameters. For example, if you have a route that accepts an ID:

```
Route::get('/user/{id}', 'UserController@show');
```

You can generate a URL with the `route()` function like this:

```
<a href="{{ route('user.show', ['id' => 1]) }}">User Profile</a>
```

Generating URLs with Controllers: Laravel allows you to generate URLs to controller actions. If you have a controller method like this:

```
public function about()
{
    // ...
}
```

You can generate the URL using the `action()` function:

```
<a href="{{ action('AboutController@about') }}">About Us</a>
```

URL Generation with Parameters: If your route has parameters, you can pass them as an array in the `action()` function:

```
<a href="{{ action('UserController@show', ['id' => 1]) }}">User Profile</a>
```

Real-life Example: Let's say you want to create a link to a user's profile page, and you have a named route 'user.profile'. You can do it like this in a Blade file:

```
<a href="{{ route('user.profile', ['id' => $user->id]) }}">View Profile</a>
```

This generates a URL to the user's profile, where `$user->id` contains the user's ID.

SESSION

Definition: In web development, a session is a mechanism that allows you to store data on the server temporarily, tied to a specific user. Sessions are crucial for maintaining stateful interactions with users across multiple HTTP requests.

Types of Sessions:

1. **File-based Sessions:** In Laravel, by default, sessions are stored as files on the server. Laravel manages these files, and you can store session data using the `session()` helper.
2. **Database Sessions:** You can also configure Laravel to store sessions in a database. This is useful when you want to persist session data between server restarts.
3. **Cache-based Sessions:** Sessions can be stored in a cache store like Redis for faster access and better scalability.

Using Sessions in Blade Files:

1. **Storing Data in Sessions:**

   To store data in a session, you can use the `session()` helper. Here's an example of storing a user's name in a session:

   ```
   // In a controller
   session(['user_name' => 'John']);
   ```

2. **Retrieving Data from Sessions:**

   You can retrieve session data in Blade files like this:

   ```
   <!-- In a Blade file -->
   <p>Welcome, {{ session('user_name') }}</p>
   ```

   This code will display "Welcome, John" if 'user_name' is stored in the session.

3. **Checking for Session Data:**

   You can check if a session variable exists using Blade directives:

   ```
   @if(session()->has('user_name'))
       <p>Welcome, {{ session('user_name') }}</p>
   @else
       <p>Welcome, Guest</p>
   @endif
   ```

4. **Removing Session Data:**

   To remove session data, use the `forget()` method or `pull()` method:

```
// Remove 'user_name' from the session
session()->forget('user_name');

// Retrieve and remove 'user_name' from the session
$userName = session()->pull('user_name');
```

Real-life Example:

Imagine you want to store the user's login status in a session. When the user logs in, you set a session variable to indicate they are authenticated. In a Blade file, you can use this to display a personalized message:

```
@if(session('is_authenticated'))
    <p>Welcome, {{ session('user_name') }}</p>
    <a href="{{ route('logout') }}">Logout</a>
@else
    <p>Welcome, Guest</p>
    <a href="{{ route('login') }}">Login</a>
@endif
```

In this example, `is_authenticated` is a session variable set when the user logs in, and `user_name` stores the user's name.

Retrieving Data:

You can retrieve data from sessions using the `session()` helper function or the `Request` object. Here's an example of how to retrieve data:

```
// Using the session() helper
$value = session('key');

// Using the Request object
$value = request()->session()->get('key');

// Retrieve all session data
$data = $request->session()->all();

// Determine if an item is present and not null using has
if ($request->session()->has('users')) {
    // Do something if 'users' is present and not null
    $users = $request->session()->get('users');
    // ...
}

// Determine if an item is present, including null values, using exists
if ($request->session()->exists('users')) {
    // Do something if 'users' is present, even if its value is null
    $users = $request->session()->get('users');
    // ...
}

// Determine if an item is missing from the session using missing
if ($request->session()->missing('users')) {
    // Do something if 'users' is not present in the session
    // ...
}
```

Storing Data:

You can store data in sessions using the `put` method or the `session()` helper. Here's an example:

```
// Using the put method
session()->put('key', 'value');

// Using the session() helper
session(['key' => 'value']);
```

Flash Data:

Flash data is a special type of session data that is available only for the next request. It's commonly used for displaying messages after a form submission. Here's how you can flash data:

```
session()->flash('message', 'This is a flash message.');

// Redirect to another route or page
return redirect()->route('some.route');
```

In your Blade file, you can display the flashed message like this:

```
@if(session('message'))
    <div class="alert alert-success">
        {{ session('message') }}
    </div>
@endif
```

Deleting Data:

To remove data from the session, you can use the `forget` method. Here's an example:

```
session()->forget('key');



<!-- Your registration form HTML goes here -->
```

Incrementing and Decrementing Session Values:

Absolutely, you can use Laravel's `increment` and `decrement` methods to manipulate integer values stored in your session data. These methods are quite handy for implementing features like counting user interactions or limiting certain actions. Here's how you can use them:

Incrementing Session Values:

1. Increment by 1 (default):

```
$request->session()->increment('count');
```

2. Increment by a specific value (e.g., 2):

```
$request->session()->increment('count', $incrementBy = 2);
```

Decrementing Session Values:

1. Decrement by 1 (default):

```
$request->session()->decrement('count');
```

2. Decrement by a specific value (e.g., 2):

```
$request->session()->decrement('count', $decrementBy = 2);
```

Here's a simple example of how you might use this in practice:

```
public function incrementCount(Request $request)
{
    // Get the current count from the session or set it to 0 if it doesn't exist
    $count = $request->session()->get('count', 0);

    // Increment the count by 1
    $request->session()->increment('count');

    return "Count: $count";
}

public function decrementCount(Request $request)
{
    // Get the current count from the session or set it to 0 if it doesn't exist
    $count = $request->session()->get('count', 0);

    // Decrement the count by 1
    $request->session()->decrement('count');

    return "Count: $count";
}
```

Advanced-SESSION

In Laravel, regenerating or invalidating the session ID is a useful security measure to prevent session fixation attacks. It changes the session ID, making it more challenging for malicious users to hijack a session. To do this, you can use the `invalidate()` method to regenerate the session ID. Here's how to do it:

```
// Invalidate the current session ID and regenerate it
$request->session()->invalidate();

// Regenerate the session ID
$request->session()->regenerate();
```

You can typically use this within a controller method or route closure. Here's a breakdown of the steps:

1. `invalidate()`: This method invalidates the current session, which means that the existing session data remains, but it's associated with a new session ID. It's like locking the old session data and creating a fresh session with a new ID.
2. `regenerate()`: This method generates a new session ID for the refreshed session. It's essential to regenerate the session ID after invalidating it to ensure that the old session ID can't be reused.

```
public function logout(Request $request)
{
    // Invalidate and regenerate the session ID
    $request->session()->invalidate();
    $request->session()->regenerate();

    // Perform any other logout actions
    // ...

    return redirect('/login');
}
```

3. The `flush()` method in Laravel's session handling allows you to clear all data stored in the session, effectively resetting it. When you call flush(), all session data, including variables and values, will be removed

```
// Clear all data stored in the session
$request->session()->flush();
```

Typically, you might use `flush()` when you want to log a user out of your application completely or clear all temporary data stored in the session. For example, when a user logs out, you can clear their session data like this:

```
public function logout(Request $request)
{
    // Clear all session data
    $request->session()->flush();

    // Perform any other logout actions
    // ...

    return redirect('/login');
}
```

VALIDATON

Validation in Laravel is the process of ensuring that user input or data adheres to a set of rules and criteria defined by your application. It's crucial to validate user input to maintain data integrity and security in your application.

Writing The Validation Logic:

In Laravel, you can write validation logic in your controllers or dedicated form request classes. Here's an example of validating a form request in a controller method:

```
public function store(Request $request)
{
   $validatedData = $request->validate([
    'name' => 'required|string|max:255',
    'email' => 'required|email|unique:users',
    'password' => 'required|min:6',
    'description' => 'string|max:500',
    'age' => 'numeric',
    'quantity' => 'integer',
    'username' => 'unique:users',
    'phone' => 'regex:/^[0-9]{10}$/',
    'gender' => 'in:Male,Female,Other',
    'birthdate' => 'date_format:Y-m-d',
    'event_date' => 'after:2023-01-01',
    'expiry_date' => 'before:2023-12-31',
    'password_confirmation' => 'required_with:password|same:password',
    'custom_field' => function ($attribute, $value, $fail) {
        if ($value != 'expected_value') {
            $fail($attribute.' is invalid.');
        }
    },
]);

// Alternatively, validation rules may be specified as arrays of rules instead of a single | delimited string:

$validatedData = $request->validate([
    'title' => ['required', 'unique:posts', 'max:255'],
    'body' => ['required'],
]);
       // Process the validated data
}
```

Displaying The Validation Errors:

When validation fails, Laravel provides an easy way to display error messages in your Blade views. Here's an example of how to display errors for a 'name' field in a Blade view:

```
@if ($errors->has('name'))
    <div class="alert alert-danger">
        {{ $errors->first('name') }}
    </div>
@endif
```

Repopulating Forms:

After form submission with validation errors, it's essential to repopulate the form fields with the user's input. Laravel's `old()` function helps with this. For example:

```
<input type="text" name="name" value="{{ old('name') }}">
```

This code repopulates the 'name' input field with the user's previous input.

Form Request Validation:

Form request validation is a recommended approach in Laravel for keeping your controller methods clean and organized.

Creating Form Requests:

You can create a form request class using the `artisan` command:

```
php artisan make:request StorePostRequest
```

In the generated form request class, define your validation rules in the `rules()` method:

```
public function rules()
{
    return [
        'title' => 'required|string|max:255',
        'content' => 'required|string',
    ];
}
```

Authorizing Form Requests:

You can also authorize the request by defining an `authorize()` method in your form request class:

```
public function authorize()
{
    return true; // By default, anyone is authorized; add your custom authorization logic here
}
```

Customizing The Error Messages:

You can customize validation error messages by overriding the `messages()` method in your form request class:

```
public function messages()
{
    return [
        'title.required' => 'The title field is required.',
        'content.required' => 'The content field is required.',
    ];
}
```

Performing Additional Validation:

You can perform additional custom validation beyond the built-in rules. Use the `Validator` facade to create custom validation rules:

```
use Illuminate\Support\Facades\Validator;

$data = [
    'name' => 'John',
    'email' => 'john@example.com',
    // Add other data fields here
];

$rules = [
    'name' => 'required|string|max:255',
    'email' => 'required|email|unique:users|max:255',
    // Define validation rules for other fields
];

$validator = Validator::make($data, $rules);

if ($validator->fails()) {
    // Validation failed, you can handle errors here
    $errors = $validator->errors();
    // Handle the errors, return a response, etc.
} else {
    // Validation passed, you can proceed with your logic here
}
```

Working With Validated Input:

After validation, you can access the validated input using the `validated()` method:

```
$validatedData = $request->validated();
```

This gives you access to the sanitized and validated input data.

Working With Error Messages:

You can display error messages in your Blade views using the `$errors` variable. For example:

```
@if ($errors->any())
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif
```

This code displays all validation error messages.

Specifying Custom Messages In Language Files:

You can define custom validation error messages in language files. For example, in `resources/lang/en/validation.php`, you can customize messages for specific rules:

```
'custom' => [
    'custom_field' => [
        'required' => 'The :attribute field must be "custom_value".',
    ],
],
```

Using Closures:

You can use closures for custom validation logic. For example:

```
'custom_field' => [
    'required',
    function ($attribute, $value, $fail) {
        if ($value != 'custom_value') {
            $fail('The custom field must be "custom_value".');
        }
    },
],
```

Blade: Validation Errors

Validation errors in Laravel are a crucial aspect of form handling. They allow you to display error messages to users when their input data doesn't meet the validation criteria you've defined. Let's explore how to handle validation errors in Laravel Blade templates with real-life examples:

1. **Controller Validation:** In your Laravel controller, you can define validation rules for incoming requests using the `validate` method. If validation fails, Laravel automatically redirects the user back to the previous page with the validation errors.

```
   public function store(Request $request)
   {
$validatedData = $request->validate([
    'name' => 'required|string|max:255',
    'email' => 'required|email|unique:users',
    'password' => 'required|min:6',
    'description' => 'string|max:500',
    'age' => 'numeric',
    'quantity' => 'integer',
    'username' => 'unique:users',
    'publish_at' => 'nullable|date',
    'phone' => 'regex:/^[0-9]{10}$/',
    'gender' => 'in:Male,Female,Other',
    'birthdate' => 'date_format:Y-m-d',
    'event_date' => 'after:2023-01-01',
    'expiry_date' => 'before:2023-12-31',
    'password_confirmation' => 'required_with:password|same:password',
    'custom_field' => function ($attribute, $value, $fail) {
        if ($value != 'expected_value') {
            $fail($attribute.' is invalid.');
        }
    },
]);

       // Process the validated data
   }
```

2. **Blade Template for Displaying Errors:** In your Blade template where you render the form, you can use the `@if` directive to check if there are validation errors for a specific field. You can then use the `@error` directive to display the error message.

```
<form action="{{ route('user.store') }}" method="POST">
    @csrf
    <div>
        <label for="name">Name:</label>
        <input type="text" id="name" name="name">
        @error('name')
            <div class="alert alert-danger">{{ $message }}</div>
        @enderror
    </div>
    <!-- Repeat for other form fields -->
    <button type="submit">Submit</button>
</form>
```

3. **Displaying All Errors:** To display all validation errors at once, you can use the `@if` directive to check if there are any errors and then loop through them.

```
@if ($errors->any())
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif
```

4. **Styling Error Messages:** You can customize the styling of error messages to fit your application's design by adding appropriate CSS classes.

```
<div class="form-group">
    <label for="email">Email:</label>
    <input type="email" id="email" name="email" class="@error('email') is-invalid @enderror">
    @error('email')
        <div class="invalid-feedback">{{ $message }}</div>
    @enderror
</div>
```

5. **Old Input Values:** When a validation error occurs, you can use the `old` helper function to repopulate the form fields with the user's previous input.

```
<input type="text" id="name" name="name" value="{{ old('name') }}">
```

6. **Customizing Error Messages:**

   You can customize error messages in the `resources/lang/en/validation.php` language file by defining custom error messages for specific rules or attributes.

```
'custom' => [
    'name' => [
        'required' => 'The name field is required.',
        'max' => 'The name field must not exceed :max characters.',
    ],
    // Define custom messages for other fields here
],
```

7. **Manually Creating Validators:** If you do not want to use the validate method on the request, you may create a validator instance manually using the Validator facade. The make method on the facade generates a new validator instance:

```
class PostController extends Controller
{
    /**
     * Store a new blog post.
     */
    public function store(Request $request): RedirectResponse
    {
        $validator = Validator::make($request->all(), [
            'title' => 'required|unique:posts|max:255',
            'body' => 'required',
        ]);

        if ($validator->fails()) {
            return redirect('post/create')
                        ->withErrors($validator)
                        ->withInput();
        }

        // Retrieve the validated input...
        $validated = $validator->validated();

        // Retrieve a portion of the validated input...
        $validated = $validator->safe()->only(['name', 'email']);
        $validated = $validator->safe()->except(['name', 'email']);

        // Store the blog post...

        return redirect('/posts');
    }
}
```

8. **Customizing The Error Messages** If needed, you may provide custom error messages that a validator instance should use instead of the default error messages provided by Laravel. There are several ways to specify custom messages. First, you may pass the custom messages as the third argument to the Validator::make method:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rules\Password;
$validator = Validator::make($input, $rules, $messages = [
    'required' => 'The :attribute field is required.',
]);

$validator = Validator::make($request->all(), [
    'password' => ['required', 'confirmed', Password::min(8)],
]);

Password::min(8)  // Require at least 8 characters...
    ->letters()  // Require at least one letter...
    ->mixedCase() // Require at least one uppercase and one lowercase letter...
    ->numbers() // Require at least one number...
    ->symbols()  // Require at least one symbol...
    ->uncompromised()
    /* uncompromised() :
    If there's a match, it suggests that the user's chosen password has been previously exposed in a data breach. This is a strong i

If there's no match, it means the password is not found in the known compromised password lists, which indicates it's less likely to
    */
```

In this example, the :attribute placeholder will be replaced by the actual name of the field under validation. You may also utilize other placeholders in validation messages. For example:

```
$messages = [
    'same' => 'The :attribute and :other must match.',
    'size' => 'The :attribute must be exactly :size.',
    'between' => 'The :attribute value :input is not between :min - :max.',
    'in' => 'The :attribute must be one of the following types: :values',
];
```

In the custom error messages, you can use placeholders like `:attribute`, `:other`, `:size`, `:min`, `:max`, and `:values` to include dynamic information about the validation rule. Here are some examples:

Error Handling

Error handling in Laravel is a crucial aspect of building robust and reliable applications. Laravel provides a comprehensive set of tools and features to help developers manage errors effectively. Let's explore different types of error handling in Laravel along with real-life and professional code examples.

1. **Exceptions in Laravel**:

   In Laravel, exceptions are used to handle errors gracefully. They are instances of the `Exception` class or its subclasses. Laravel provides several built-in exception classes, and you can create custom ones as needed.

   Example:

```
try {
    // Code that may throw an exception
    $result = 1 / 0; // This will throw a DivisionByZeroError
} catch (DivisionByZeroError $e) {
    // Handle the exception
    Log::error($e->getMessage());
    return view('error', ['message' => 'An error occurred.']);
}
```

2. **HTTP Exceptions**:

   Laravel has a set of HTTP exception classes to handle HTTP-related errors like 404 (Not Found) and 500 (Internal Server Error).

   Example:

```
public function show($id)
{
    $item = Item::find($id);
    if (!$item) {
        throw new \Symfony\Component\HttpKernel\Exception\NotFoundHttpException('Item not found.');
    }
    return view('item.show', ['item' => $item]);
}
```

3. **Custom Exception Handling**:

   You can create custom exception classes to handle specific application-related errors and provide meaningful error messages to users.

   Example:

```
class CustomException extends \Exception
{
    public function render($request)
    {
        return response()->view('errors.custom', ['message' => $this->getMessage()], 500);
    }
}
```

4. **Error Logging**:

   Laravel allows you to log errors and exceptions for debugging and monitoring purposes. You can configure different log channels, including files, databases, and external services like Loggly or Papertrail.

   Example (Logging to a file):

```
try {
    // Code that may throw an exception
} catch (\Exception $e) {
    Log::error('Error occurred: ' . $e->getMessage());
}
```

5. **Error Views**:

   Laravel provides default error views that can be customized to display user-friendly error pages. You can find these views in the `resources/views/errors` directory.

6. **Handling AJAX Errors**:

   When working with AJAX requests, it's essential to handle errors gracefully and return JSON responses. You can use Laravel's `Response` class to send appropriate HTTP status codes and error messages.

   Example:

```
public function ajaxRequest()
{
    try {
        // Code that may throw an exception
    } catch (\Exception $e) {
        return response()->json(['error' => 'An error occurred.'], 500);
    }
}
```

7. **Validation Errors**:

   Laravel offers robust validation handling using the Validator class. You can validate user inputs and handle validation errors easily.

   Example:

```
$validator = Validator::make($request->all(), [
    'email' => 'required|email',
    'password' => 'required|min:8',
]);

if ($validator->fails()) {
    return redirect('login')
        ->withErrors($validator)
        ->withInput();
}
```

8. **Database Transactions**:

   When working with database operations, you can use database transactions to ensure data consistency and handle errors gracefully.

   Example:

```
DB::beginTransaction();

try {
    // Database operations
    DB::commit();
} catch (\Exception $e) {
    DB::rollback();
    Log::error('Database error: ' . $e->getMessage());
}
```

Certainly! You can create a custom exception handler for both 404 (Not Found) and 500 (Internal Server Error) pages in Laravel. Here's how you can do it:

1. **Create Custom Exceptions**:

   First, create custom exception classes for handling these errors. You can place these classes in the `app/Exceptions` directory.

   For the 404 error, create a `NotFoundHttpException` handler:

```
// app/Exceptions/CustomNotFoundHttpException.php

namespace App\Exceptions;

use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

class CustomNotFoundHttpException extends NotFoundHttpException
{
    public function render($request)
    {
        return response()->view('errors.404', [], 404);
    }
}
```

For the 500 error, create a `CustomInternalServerErrorException` handler:

```
// app/Exceptions/CustomInternalServerErrorException.php

namespace App\Exceptions;

use Exception;

class CustomInternalServerErrorException extends Exception
{
    public function render($request)
    {
        return response()->view('errors.500', [], 500);
    }
}
```

2. **Custom Error Views**:

   Create custom error views for both 404 and 500 errors. These views should be placed in the `resources/views/errors` directory.

   - `resources/views/errors/404.blade.php` for the 404 error.
   - `resources/views/errors/500.blade.php` for the 500 error.

   Customize these views according to your application's design and requirements.

3. **Register Custom Exceptions**:

   Open the `app/Exceptions/Handler.php` file and register your custom exceptions in the `render` method:

```
public function render($request, Exception $exception)
{
    if ($exception instanceof CustomNotFoundHttpException) {
        return $exception->render($request);
    }

    if ($exception instanceof CustomInternalServerErrorException) {
        return $exception->render($request);
    }

    return parent::render($request, $exception);
}
```

4. **Test Your Custom Exceptions**:

   You can now test your custom exceptions by triggering them in your routes or controllers.

   For example, in a controller method:

```
public function customErrorExample()
{
    // Trigger a 404 error
    throw new CustomNotFoundHttpException('Custom 404 Error');

    // Trigger a 500 error
    // throw new CustomInternalServerErrorException('Custom 500 Error');
}
```

5. Create a Custom Exception Class:

   First, create a custom exception class that extends Laravel's `HttpException`. This class will allow you to handle both 404 and 500 errors.

```
// app/Exceptions/CustomException.php

namespace App\Exceptions;

use Symfony\Component\HttpKernel\Exception\HttpException;

class CustomException extends HttpException
{
    public function __construct($message = null, \Exception $previous = null, $code = 0)
    {
        parent::__construct(500, $message, $previous, [], $code);
    }
}
```

6. Create Custom Error Views:

   Next, create custom error views for both 404 and 500 errors. You can place these views in the `resources/views/errors` directory.

   - For 404 error, create a view file named `404.blade.php`.
   - For 500 error, create a view file named `500.blade.php`.

   Example 404 View (`resources/views/errors/404.blade.php`):

```
<!DOCTYPE html>
<html>
<head>
    <title>404 Not Found</title>
</head>
<body>
    <h1>404 Not Found</h1>
    <p>The page you requested could not be found.</p>
</body>
</html>
```

   Example 500 View (`resources/views/errors/500.blade.php`):

```
<!DOCTYPE html>
<html>
<head>
    <title>500 Internal Server Error</title>
</head>
<body>
    <h1>500 Internal Server Error</h1>
    <p>Something went wrong on the server.</p>
</body>
</html>
```

7. Use the Custom Exception:

   In your controller or route handler, you can throw the custom exception when you encounter an error that should result in a 500 error or a 404 error.

   Example (Throwing a 404 Error):

```
use App\Exceptions\CustomException;

public function show($id)
{
    $item = Item::find($id);
    if (!$item) {
        throw new CustomException('Item not found.', null, 404);
    }
    return view('item.show', ['item' => $item]);
}
```

```
use App\Exceptions\CustomException;

public function someMethod()
{
    try {
        // Code that may throw an exception
        if (/* Some error condition */) {
            throw new CustomException('An internal server error occurred.');
        }
    } catch (\Exception $e) {
        throw new CustomException('An internal server error occurred.', $e);
    }
}
```

LOGGING

1. What is Logging?

Logging is the process of recording events and messages from your application to a designated location, typically a log file. These logs are essential for debugging and monitoring your application's behavior, especially in production environments.

2. Laravel's Logging System

Laravel provides a powerful and flexible logging system through the use of the Monolog library. You can configure and use this system to capture and store log messages.

3. Log Levels

Laravel supports various log levels, each indicating the severity of a message:

- `emergency`: System is unusable.
- `alert`: Action must be taken immediately.
- `critical`: Critical conditions.
- `error`: Error conditions.
- `warning`: Warning conditions.
- `notice`: Normal but significant condition.
- `info`: Informational messages.
- `debug`: Debug-level messages.

4. Configuration

Laravel's logging configuration can be found in the `config/logging.php` file. You can set the log channel (where logs are stored), the log level, and other options.

5. Writing Log Messages

You can log messages in Laravel using the `Log` facade. Here's an example:

```
use Illuminate\Support\Facades\Log;

Log::info('This is an informational message.');
Log::error('An error occurred: ' . $exception->getMessage());
```

6. Real-Life Blade Example

Suppose you want to display a log message in your Blade template. You can pass the log message from your controller to the view and then display it. Here's an example:

In your controller:

```
use Illuminate\Support\Facades\Log;

public function showLogMessage()
{
    Log::info('This is a log message from the controller.');
    return view('logMessage', ['message' => 'Log message from controller sent to view.']);
}
```

In your Blade view (e.g., `logMessage.blade.php`):

```
<!DOCTYPE html>
<html>
<head>
    <title>Laravel Logging Example</title>
</head>
<body>
    <h1>Log Message Example</h1>
    <p>Message from controller: {{ $message }}</p>
</body>
</html>
```

When you visit the route associated with `showLogMessage`, you'll see the log message displayed in the browser.

7. Viewing Logs

By default, Laravel stores logs in the storage/logs directory. You can access these log files to review and troubleshoot issues in your application.

DIGGING DEEPERS : ADVANCED LEVEL

It is an advanced part of Laravel.

ARTISAN CONSOLE

The Laravel Artisan Console is a command-line tool included with the Laravel PHP framework. It provides a wide range of commands for various tasks related to Laravel application development, management, and maintenance. Artisan simplifies common development tasks and automates many processes, making it easier for developers to work with Laravel. Certainly, here are some of the Artisan commands in Laravel with their descriptions in PHP code view:

1. **Make a New Controller:**

```
php artisan make:controller MyController
```

2. **Create a New Model:**

```
php artisan make:model MyModel
```

3. **Create a Migration:**

```
php artisan make:migration create_table_name
```

4. **Run Migrations:**

```
php artisan migrate
```

5. **Create a Seeder:**

```
php artisan make:seeder MySeeder
```

6. **Run Seeders:**

```
php artisan db:seed
```

7. **Generate a Key for .env File:**

```
php artisan key:generate
```

8. **Clear Cache:**

```
php artisan cache:clear
```

9. **Create a New Middleware:**

```
php artisan make:middleware MyMiddleware
```

10. **Create a New Request:**

```
php artisan make:request MyRequest
```

11. **List All Available Routes:**

```
php artisan route:list
```

12. **Create a New Job:**

```
php artisan make:job MyJob
```

13. **Create a New Event:**

```
php artisan make:event MyEvent
```

14. **Create a New Listener:**

```
php artisan make:listener MyListener
```

15. **Create a New Policy:**

```
php artisan make:policy MyPolicy
```

Certainly! Here are some more important Laravel Artisan commands without repetition:

1. **Clear Configuration Cache:**

```
php artisan config:clear
```

2. **Optimize Class Loading:**

```
php artisan optimize
```

3. **Create a New Middleware with Handle Method:**

```
php artisan make:middleware MyMiddleware --invokable
```

4. **Create a New Factory:**

```
php artisan make:factory MyFactory
```

5. **Create a New Test:**

```
php artisan make:test MyTest
```

6. **Create a New Resource Controller:**

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

7. **Create a New Notification:**

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

8. **Rollback the Last Database Migration:**

```
php artisan migrate:rollback
```

9. **Create a New Livewire Component:**

```
php artisan make:livewire MyComponent
```

10. **Generate Documentation for API Routes:**

```
php artisan api:generate
```

11. **Create a New Channel Class:**

```
php artisan make:channel MyChannel
```

12. **Create a New Artisan Command:**

```
php artisan make:command MyCommand
```

Certainly! Here are a few more Laravel Artisan commands that you might find useful:

1. **Generate Authentication Scaffolding:**

```
php artisan make:auth
```

2. **Create a New Job Listener:**

```
php artisan queue:listen
```

3. **Schedule Task Execution:**

```
php artisan schedule:run
```

4. **Create a New Factory for Model Factories:**

```
php artisan make:factory MyModelFactory --model=MyModel
```

5. **Create a New Channel for Broadcasting:**

```
php artisan make:channel MyBroadcastChannel
```

6. **Optimize the Application for Production:**

```
php artisan optimize --force
```

7. **Create a New Policy with a Model:**

```
php artisan make:policy MyPolicy --model=MyModel
```

8. **Generate IDE Helper Files for Better Code Completion:**

```
php artisan ide-helper:generate
```

9. **Rebuild All IDE Helper Files:**

```
php artisan ide-helper:meta
```

10. **Generate a Sitemap:**

```
php artisan sitemap:generate
```

11. **Generate Application Encryption Key:**

```
php artisan key:generate
```

12. **List All Commands and Options:**

```
php artisan list
```

CUSTOM ARTISAN COMMAND

Creating a custom Artisan command is a great way to extend Laravel's functionality for your specific project needs. Let's walk through a simple example of creating a custom Artisan command for a real-life scenario. In this example, we'll create an Artisan command to send email reminders for upcoming appointments in a healthcare application.

Step 1: Create the Custom Artisan Command

1. Open your terminal and navigate to your Laravel project's root directory.
2. Use the Artisan command to create a new custom command. We'll call it `SendAppointmentReminders`:

```
php artisan make:command SendAppointmentReminders
```

This will generate a new file named `SendAppointmentReminders.php` in the `app/Console/Commands` directory.

Step 2: Define the Command Logic

Open the `SendAppointmentReminders.php` file in a code editor. You'll find a `handle()` method within the file. This is where you define the logic for your custom command. In our example, we want to send email reminders for upcoming appointments.

Here's a simplified code example:

```php
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;
use App\Appointment;
use Illuminate\Support\Facades\Mail;

class SendAppointmentReminders extends Command
{
    protected $signature = 'send:reminders';
    protected $description = 'Send email reminders for upcoming appointments';

    public function __construct()
    {
        parent::__construct();
    }

    public function handle()
    {
        $upcomingAppointments = Appointment::whereDate('date', now()->addDays(1))->get();

        foreach ($upcomingAppointments as $appointment) {
            $user = $appointment->user;
            $email = $user->email;
            $subject = 'Appointment Reminder';
            $message = 'Your appointment is scheduled for tomorrow.';

            Mail::raw($message, function ($message) use ($email, $subject) {
                $message->to($email)->subject($subject);
            });

            $this->info("Reminder email sent to: $email");
        }

        $this->info('Reminder emails sent successfully.');
    }
}
```

In this code:

- We define the signature and description for our command in the `protected $signature` and `protected $description` properties.
- In the `handle()` method, we retrieve upcoming appointments that are scheduled for the next day.
- We iterate through the appointments, retrieve the user's email, and send them a reminder email using Laravel's built-in `Mail` facade.
- Finally, we use `$this->info()` to display messages in the console.

Step 3: Register the Custom Command

To make your custom command accessible via Artisan, you need to register it. Open the `app/Console/Kernel.php` file and add your custom command to the `$commands` property:

```php
protected $commands = [
    // ...
    \App\Console\Commands\SendAppointmentReminders::class,
];
```

Step 4: Run the Custom Command

Now, you can run your custom Artisan command from the terminal:

```
php artisan send:reminders
```

This will execute the `handle()` method of your custom command, sending email reminders for upcoming appointments.

Laravel Broadcasting: An In-Depth Explanation

What is Laravel Broadcasting?

Laravel Broadcasting is a real-time messaging system that allows you to send data from your Laravel application to connected clients, such as web browsers, mobile apps, or other servers, in real-time. It enables you to build interactive, live-updating features like chat applications, notifications, and live feeds.

Key Components:

1. **Broadcasting Server:** Laravel uses broadcasting drivers like Pusher, Redis, or others as the broadcasting server to manage and broadcast events to connected clients.
2. **Events and Listeners:** In Laravel, you define events that represent something that has happened in your application (e.g., a new message). You also define event listeners that specify what should occur when an event is triggered.
3. **WebSockets:** WebSockets are a technology used to establish a full-duplex communication channel over a single TCP connection. Laravel Broadcasting often uses WebSockets to provide real-time communication.

Setting Up Laravel Broadcasting:

1. **Configuration:** To use broadcasting, you need to configure the broadcasting driver in your `config/broadcasting.php` file. Laravel supports various broadcasting drivers, including Pusher, Redis, and more. For example, with Pusher:

In `.env` file configure

```
BROADCAST_DRIVER=pusher
PUSHER_APP_ID=your_app_id
PUSHER_APP_KEY=your_app_key
PUSHER_APP_SECRET=your_app_secret
PUSHER_APP_CLUSTER=your_app_cluster
```

```
'default' => env('BROADCAST_DRIVER', 'pusher'),
'connections' => [
    'pusher' => [
        'driver' => 'pusher',
        'key' => env('PUSHER_APP_KEY'),
        'secret' => env('PUSHER_APP_SECRET'),
        'app_id' => env('PUSHER_APP_ID'),
        'options' => [
            'cluster' => env('PUSHER_APP_CLUSTER'),
            'encrypted' => true,
        ],
    ],
    // Other broadcasting drivers...
],
```

2. **Event Creation:** Define your events using Laravel's Artisan command, `php artisan make:event EventName`. An event class typically includes a `broadcastOn` method that specifies the channel to which the event should be broadcast.
3. **Event Broadcasting:** In your application logic, trigger the event using `event(new EventName($data))`. This broadcasts the event to the specified channel.

Example: Real-Time Notifications

Let's create an example of real-time notifications using Laravel Broadcasting.

Step 1: Create an Event

Generate a new event using Artisan:

```
php artisan make:event NewNotification
```

In the `NewNotification` event class, define the event data:

```
public $message;

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

public function broadcastOn()
{
    return new Channel('notifications');
}
```

Step 2: Broadcast the Event

In your application logic, you can trigger this event when a new notification needs to be sent:

```
event(new NewNotification('You have a new message.'));
```

Step 3: Receive the Event in JavaScript (Blade File)

In your Blade file, include JavaScript code to listen for and handle the event:

```
<script src="https://js.pusher.com/7.0/pusher.min.js"></script>
<script>
    // Initialize Pusher
    var pusher = new Pusher('{{ config('broadcasting.connections.pusher.key') }}', {
        cluster: '{{ config('broadcasting.connections.pusher.options.cluster') }}',
        encrypted: true
    });

    // Subscribe to the 'notifications' channel
    var channel = pusher.subscribe('notifications');

    // Listen for the 'NewNotification' event
    channel.bind('App\\Events\\NewNotification', function(data) {
        // Handle the incoming data (e.g., show a notification)
        alert(data.message);
    });
</script>
```

or blade view file

Broadcasting in Blade Views:

To receive and display real-time updates in a Blade view, you can use the `@pusher` directive. Here's an example:

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

@section('content')
    <!-- Chat room content -->

    @pusher('chat')
        <div id="chat-room">
            <!-- Chat messages will be displayed here -->
        </div>
    @endpusher

    <!-- Chat input form -->
@endsection
```

In this example:

- `@pusher('chat')` specifies that this section will be updated using Pusher when new messages arrive.

- The content inside the `@pusher` directive will automatically update in real-time as new messages are broadcasted.

This JavaScript code initializes Pusher, subscribes to the 'notifications' channel, and listens for the 'NewNotification' event. When the event is received, it displays an alert with the message.

Step 4: Broadcast the Event

Back in your Laravel code, when you trigger the `NewNotification` event using `event(new NewNotification('You have a new message.'));`, it will be broadcasted to connected clients and trigger the JavaScript code in your Blade file.

EVENTS LISTENER

Certainly! Here's a more detailed and comprehensive example of setting up broadcasting in a Laravel application for a real-time chat feature.

Step 1: Install Laravel Echo and Configure WebSocket Server

1. **Install Laravel Echo and Pusher**

   Run the following command to install Laravel Echo and Pusher:

   ```
   npm install laravel-echo pusher-js
   ```

2. **Set Up Pusher**

   Sign up for a Pusher account (https://pusher.com/ (https://pusher.com/)), create a new app, and get your app credentials.

   Update your `.env` file with your Pusher credentials:

   ```
   BROADCAST_DRIVER=pusher
   PUSHER_APP_ID=your-app-id
   PUSHER_APP_KEY=your-app-key
   PUSHER_APP_SECRET=your-app-secret
   PUSHER_APP_CLUSTER=your-app-cluster
   ```

Step 2: Create a Chat Event

1. **Generate the Event**

   Run the following command to generate a chat event:

   ```
   php artisan make:event NewMessage
   ```

2. **Edit the Event Class**

   In `app/Events/NewMessage.php`, define the event class like this:

   ```
   use Illuminate\Broadcasting\InteractsWithSockets;
   use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
   use Illuminate\Foundation\Events\Dispatchable;
   use Illuminate\Queue\SerializesModels;

   class NewMessage implements ShouldBroadcast
   {
       use Dispatchable, InteractsWithSockets, SerializesModels;

       public $message;

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

       public function broadcastOn()
       {
           return new Channel('chat');
       }
   }
   ```

Step 3: Create a Broadcasting Channel

1. **Define the Channel**

   In `routes/channels.php`, define the broadcasting channel:

```
Broadcast::channel('chat', function () {
    return true; // For simplicity, allow everyone to join the 'chat' channel
});
```

Step 4: Broadcast the Event

1. **Trigger the Event**

   In your controller where a new message is sent, dispatch the `NewMessage` event:

```
event(new NewMessage($message));
```

Step 5: Listen to the Event in JavaScript

1. **Set Up Laravel Echo**

   In your JavaScript file, set up Laravel Echo to listen for the event:

```
import Echo from 'laravel-echo'

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: 'your-app-key',
    cluster: 'your-app-cluster',
    encrypted: true,
});
```

2. **Listen for the Event**

   Now, listen for the `NewMessage` event and handle incoming messages in your JavaScript code:

```
window.Echo.channel('chat')
    .listen('NewMessage', (e) => {
        // Handle the new message here
        console.log('New Message:', e.message);
        // Display the message in your chat interface
        // Example: append the message to the chat window
    });
```

Step 6: Display Real-Time Messages

1. **Display Messages in Your Chat Interface**

   Update your chat interface to display messages in real-time as they arrive, using the JavaScript code you set up in the previous step.

That's it! With these steps, you've created a real-time chat feature in your Laravel application using broadcasting. When a user sends a message, it's broadcasted to the 'chat' channel, and all connected clients receive the message in real-time, providing a real-time chat experience in your application.

This example should help you integrate broadcasting into your professional project for real-time features like chat, notifications, or live updates.

BRAODCASTING

Certainly! Here's a more detailed and comprehensive example of setting up broadcasting in a Laravel application for a real-time chat feature.

Step 1: Install Laravel Echo and Configure WebSocket Server

1. **Install Laravel Echo and Pusher**

   Run the following command to install Laravel Echo and Pusher:

```
npm install laravel-echo pusher-js
```

2. **Set Up Pusher**

   Sign up for a Pusher account (https://pusher.com/ (https://pusher.com/)), create a new app, and get your app credentials.

   Update your `.env` file with your Pusher credentials:

```
BROADCAST_DRIVER=pusher
PUSHER_APP_ID=your-app-id
PUSHER_APP_KEY=your-app-key
PUSHER_APP_SECRET=your-app-secret
PUSHER_APP_CLUSTER=your-app-cluster
```

Step 2: Create a Chat Event

1. **Generate the Event**

   Run the following command to generate a chat event:

```
php artisan make:event NewMessage
```

2. **Edit the Event Class**

   In `app/Events/NewMessage.php`, define the event class like this:

```
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Queue\SerializesModels;

class NewMessage implements ShouldBroadcast
{
    use Dispatchable, InteractsWithSockets, SerializesModels;

    public $message;

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

    public function broadcastOn()
    {
        return new Channel('chat');
    }
}
```

Step 3: Create a Broadcasting Channel

1. **Define the Channel**

   In `routes/channels.php`, define the broadcasting channel:

```
Broadcast::channel('chat', function () {
    return true; // For simplicity, allow everyone to join the 'chat' channel
});
```

Step 4: Broadcast the Event

1. **Trigger the Event**

   In your controller where a new message is sent, dispatch the `NewMessage` event:

```
event(new NewMessage($message));
```

Step 5: Listen to the Event in JavaScript

1. **Set Up Laravel Echo**

   In your JavaScript file, set up Laravel Echo to listen for the event:

```
import Echo from 'laravel-echo'

window.Echo = new Echo({
    broadcaster: 'pusher',
    key: 'your-app-key',
    cluster: 'your-app-cluster',
    encrypted: true,
});
```

2. **Listen for the Event**

   Now, listen for the `NewMessage` event and handle incoming messages in your JavaScript code:

```
window.Echo.channel('chat')
    .listen('NewMessage', (e) => {
        // Handle the new message here
        console.log('New Message:', e.message);
        // Display the message in your chat interface
        // Example: append the message to the chat window
    });
```

Step 6: Display Real-Time Messages

1. **Display Messages in Your Chat Interface**

   Update your chat interface to display messages in real-time as they arrive, using the JavaScript code you set up in the previous step.

That's it! With these steps, you've created a real-time chat feature in your Laravel application using broadcasting. When a user sends a message, it's broadcasted to the 'chat' channel, and all connected clients receive the message in real-time, providing a real-time chat experience in your application.

This example should help you integrate broadcasting into your professional project for real-time features like chat, notifications, or live updates.

CACHING IN LARAVEL APP

Certainly! Let's explore Laravel Caching with a simple example and easy-to-understand explanations for beginners.

Laravel Caching in Simple Terms:

- **What is Caching?**

  Caching in Laravel is a technique used to store and retrieve frequently accessed data from a fast, temporary storage space (the cache) to improve application performance. It helps reduce the load on your database and speeds up responses.

- **Why Use Caching?**

  Caching is beneficial when you have data that doesn't change frequently but is frequently requested by users. It saves time and resources by serving the data quickly from the cache instead of recalculating or fetching it from the database every time.

Scenario: Caching Frequently Accessed Blog Posts

Let's create a simplified example where we cache frequently accessed blog posts to improve the performance of a blog website.

Step 1: Set Up Caching Configuration

1. **Configure Cache Driver**

   Open your `.env` file and set the `CACHE_DRIVER` to your desired caching driver (e.g., `file`, `redis`, `memcached`, etc.):

```
CACHE_DRIVER=file
```

   For simplicity, we'll use the `file` driver, which stores cached data in the file system.

Step 2: Cache Frequently Accessed Data

1. **What is it?**

In our case, we want to cache frequently accessed blog posts.

2. **How to Do It?**

   In your controller method where you retrieve blog posts, you can use Laravel's caching functionality:

```
public function getBlogPosts()
{
    $key = 'blog_posts';
    $minutes = 30; // Cache for 30 minutes

    // Attempt to retrieve blog posts from the cache
    $posts = Cache::remember($key, $minutes, function () {
        return BlogPost::all(); // Fetch from the database if not in cache
    });

    return view('blog.posts', ['posts' => $posts]);
}
```

   Here, we're using the `Cache::remember` method to check if the 'blog_posts' data is in the cache. If it's not found or has expired, it will fetch the data from the database, cache it for 30 minutes, and then return it.

Step 3: Clear the Cache

1. **What is it?**

   Occasionally, you may need to clear the cache when data is updated.

2. **How to Do It?**

   To clear the cache for the 'blog_posts' key, you can use:

```
Cache::forget('blog_posts'); // Remove the 'blog_posts' cache
```

Summary:

In this example, we've demonstrated how to use caching in Laravel to improve the performance of a blog website. We cache frequently accessed blog posts, and if the data is not in the cache or has expired, we fetch it from the database and store it in the cache for 30 minutes. This reduces database queries and speeds up the website's response time, making it more efficient for users.

Caching is a valuable tool for optimizing your Laravel application's performance by storing and retrieving data efficiently. It's particularly useful for frequently accessed or slow-to-fetch data.

COLLECTIONS IN LARAVEL

Certainly! Let's explore Laravel Collections with a simple example and easy-to-understand explanations for beginners.

Laravel Collections in Simple Terms:

- **What are Collections?**

  Collections in Laravel are a powerful way to work with arrays or sets of data. They provide a wide range of methods for filtering, transforming, and manipulating data in a clean and concise way.

- **Why Use Collections?**

  Collections make it easier to perform common data operations, like filtering items, mapping values, and reducing arrays, without the need for complex loops or custom functions. They can help simplify code and improve readability.

Scenario: Processing Orders in an E-commerce Application

Let's create a simplified example of using Laravel Collections to process orders in an e-commerce application.

Step 1: Retrieve Orders

1. **What is it?**

   In our scenario, we want to retrieve a list of orders.

2. **How to Do It?**

In your controller, you can fetch orders from your database or any data source:

```
$orders = Order::all(); // Assuming 'Order' is your Eloquent model
```

Step 2: Use a Collection

1. **What is it?**

   Once you have your orders, you can convert them into a collection to take advantage of Laravel's collection methods.

2. **How to Do It?**

```
use Illuminate\Support\Collection;

$orderCollection = collect($orders);
```

   Now, you have a collection of orders.

Step 3: Filter Orders

1. **What is it?**

   Let's say you want to filter orders to find all the orders with a total amount greater than $100.

2. **How to Do It?**

   You can use the `filter` method:

```
$expensiveOrders = $orderCollection->filter(function ($order) {
    return $order->total > 100;
});
```

   `$expensiveOrders` now contains only the orders with a total amount greater than $100.

Step 4: Calculate Total Revenue

1. **What is it?**

   Now, you want to calculate the total revenue from these expensive orders.

2. **How to Do It?**

   You can use the `sum` method:

```
$totalRevenue = $expensiveOrders->sum('total');
```

   `$totalRevenue` contains the total revenue from expensive orders.

Step 5: Map Orders to Their Customer Names

1. **What is it?**

   Let's say you want to create a list of customer names who placed expensive orders.

2. **How to Do It?**

   You can use the `map` method:

```
$customerNames = $expensiveOrders->map(function ($order) {
    return $order->customer->name;
});
```

   `$customerNames` now contains a list of customer names.

Step 6: Display the Results

1. **What is it?**

   Finally, you can display the results in your view or return them from your controller.

2. **How to Do It?**

```
return view('orders', [
    'expensiveOrders' => $expensiveOrders,
    'totalRevenue' => $totalRevenue,
    'customerNames' => $customerNames->implode(', '), // Convert names to a comma-separated string
]);
```

In your view, you can then access these variables and display them.

Summary:

In this example, we used Laravel Collections to filter, calculate, and transform data related to orders in an e-commerce application. Collections simplify data manipulation and allow you to work with data in a more readable and efficient way, making your code cleaner and more maintainable.

Certainly! Here are some more important Laravel Collection methods along with descriptions:

1. **`map`**: Transforms each item in the collection using a callback function and returns a new collection with the modified items.

```
$doubledPrices = $prices->map(function ($price) {
    return $price * 2;
});
```

2. **`filter`**: Filters the collection based on a callback function and returns a new collection containing only the items that pass the filter.

```
$expensiveItems = $items->filter(function ($item) {
    return $item['price'] > 100;
});
```

3. **`reduce`**: Reduces the collection to a single value using a callback function.

```
$total = $numbers->reduce(function ($carry, $number) {
    return $carry + $number;
}, 0); // 0 is the initial value
```

4. **`take`**: Returns a new collection with a specified number of items from the beginning of the original collection.

```
$firstThree = $collection->take(3);
```

5. **`skip`**: Returns a new collection with a specified number of items skipped from the beginning of the original collection.

```
$skippedItems = $collection->skip(2);
```

6. **`first` and `last`**: Retrieve the first or last item from the collection.

```
$firstItem = $collection->first();
$lastItem = $collection->last();
```

7. **`count`**: Get the count of items in the collection.

```
$count = $collection->count();
```

8. **`isEmpty` and `isNotEmpty`**: Check if the collection is empty or not.

```
if ($collection->isEmpty()) {
    // Collection is empty
}

if ($collection->isNotEmpty()) {
    // Collection is not empty
}
```

9. **`all`**: Convert the collection to a plain array.

```
$array = $collection->all();
```

10. **`implode`**: Concatenate the values of a given key as a string.

```
$names = $collection->pluck('name')->implode(', ');
```

These are some of the commonly used methods in Laravel Collections. They allow you to perform various operations on your data with ease and efficiency.

Collection Methods (Important)

Certainly! Here are some of the important Laravel Collection methods along with examples and short definitions:

1. **`all`**: Get all the items from the collection.

```
$allItems = $collection->all();
```

2. **`avg`**: Calculate the average value of a given key in the collection.

```
$average = $collection->avg('score');
```

3. **`chunk`**: Split the collection into smaller chunks.

```
$chunks = $collection->chunk(3); // Split into chunks of 3 items each
```

4. **`collapse`**: Collapse a multi-dimensional collection into a single level.

```
$collapsed = $multiDimensional->collapse();
```

5. **`contains`**: Check if the collection contains a specific item.

```
$contains = $collection->contains('name', 'John');
```

6. **`count`**: Get the count of items in the collection.

```
$count = $collection->count();
```

7. **`each`**: Iterate over the collection and apply a callback function to each item.

```
$collection->each(function ($item) {
    // Do something with each item
});
```

8. **`filter`**: Filter the collection based on a callback function.

```
$filtered = $collection->filter(function ($item) {
    return $item['age'] > 18;
});
```

9. **`first`**: Get the first item from the collection.

```
$firstItem = $collection->first();
```

10. **`implode`**: Concatenate the values of a given key as a string.

```
$names = $collection->pluck('name')->implode(', ');
```

11. **`intersect`**: Get the items that are present in both the collection and another collection or array.

```
$intersection = $collection->intersect($otherCollection);
```

12. **`isEmpty`**: Check if the collection is empty.

```
if ($collection->isEmpty()) {
    // Collection is empty
}
```

13. **`join`**: Join the items in the collection using a delimiter.

```
$csv = $collection->pluck('name')->join(', ');
```

14. **`map`**: Transform each item in the collection using a callback function.

```
$doubled = $collection->map(function ($item) {
    return $item * 2;
});
```

15. **`max`**: Get the maximum value of a given key in the collection.

```
$maxValue = $collection->max('price');
```

16. **`merge`**: Merge the collection with another collection or array.

```
$merged = $collection->merge($otherCollection);
```

17. **`pluck`**: Extract a single column's value from the collection.

```
$names = $collection->pluck('name');
```

18. **`random`**: Get a random item(s) from the collection.

```
$randomItem = $collection->random();
```

19. **`reduce`**: Reduce the collection to a single value using a callback function.

```
$total = $collection->reduce(function ($carry, $item) {
    return $carry + $item['price'];
}, 0);
```

20. **`reverse`**: Reverse the order of items in the collection.

```
$reversed = $collection->reverse();
```

21. **`chunkWhile`**: Split the collection into chunks using a custom callback function.

```
$chunks = $collection->chunkWhile(function ($item, $key) {
    return $item['status'] === 'active';
});
```

22. **`collect`**: Create a new collection instance from an array or other iterable.

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

23. **`concat`**: Concatenate another collection or array onto the end of the current collection.

```
$combined = $collection->concat($otherCollection);
```

24. **`containsStrict`**: Check if the collection contains an item with a specific key and value, using strict comparison.

```
$contains = $collection->containsStrict('name', 'John');
```

25. **`countBy`**: Count the occurrences of values in the collection and return a new collection.

```
$counted = $collection->countBy('status');
```

26. **`crossJoin`**: Get the cross join of two or more arrays or collections.

```
$crossJoined = $collection->crossJoin($otherCollection);
```

27. **`duplicates`**: Get the items in the collection that have duplicates.

```
$duplicateItems = $collection->duplicates();
```

28. **`duplicatesStrict`**: Get the items in the collection that have duplicates using strict comparison.

```
$duplicateItems = $collection->duplicatesStrict();
```

29. **`every`**: Check if all items in the collection pass a given truth test.

```
$allPass = $collection->every(function ($item) {
    return $item['age'] >= 18;
});
```

30. **`except`**: Get all items in the collection except for those with the specified keys.

```
$filtered = $collection->except(['key1', 'key2']);
```

31. **`firstOrFail`**: Get the first item in the collection or throw an exception if it's empty.

```
$firstItem = $collection->firstOrFail();
```

32. **`groupBy`**: Group the collection's items by a given key.

```
$grouped = $collection->groupBy('category');
```

33. **`has`**: Check if the collection has an item with a specific key.

```
$hasKey = $collection->has('name');
```

34. **`intersectAssoc`**: Get the items that are present in both the collection and another collection using a strict comparison.

```
$intersection = $collection->intersectAssoc($otherCollection);
```

35. **`intersectByKeys`**: Get the items that have matching keys in both the collection and another collection.

```
$intersection = $collection->intersectByKeys($otherCollection);
```

36. **`isNotEmpty`**: Check if the collection is not empty.

```
if ($collection->isNotEmpty()) {
    // Collection is not empty
}
```

37. **`join`**: Join the items in the collection using a delimiter.

```
$csv = $collection->join(', ');
```

38. **`keys`**: Get all the keys from the collection.

```
$keys = $collection->keys();
```

39. **`last`**: Get the last item from the collection.

```
$lastItem = $collection->last();
```

40. **`map`**: Transform each item in the collection using a callback function.

```
$doubled = $collection->map(function ($item) {
    return $item * 2;
});
```

41. **`merge`**: Merge the collection with another collection or array.

```
$merged = $collection->merge($otherCollection);
```

42. **`pluck`**: Extract a single column's value from the collection.

```
$names = $collection->pluck('name');
```

43. **`random`**: Get a random item(s) from the collection.

```
$randomItem = $collection->random();
```

44. **`reduce`**: Reduce the collection to a single value using a callback function.

```
$total = $collection->reduce(function ($carry, $item) {
    return $carry + $item['price'];
}, 0);
```

45. **`reject`**: Remove items from the collection that do not pass a given truth test.

```
$rejected = $collection->reject(function ($item) {
    return $item['status'] === 'inactive';
});
```

46. **`replace`**: Replace items in the collection with another array or collection based on a key.

```
$replaced = $collection->replace(['key1' => 'value1', 'key2' => 'value2']);
```

47. **`reverse`**: Reverse the order of items in the collection.

```
$reversed = $collection->reverse();
```

48. **`shuffle`**: Shuffle the items in the collection randomly.

```
$shuffled = $collection->shuffle();
```

49. **`skip`**: Skip a specified number of items from the beginning of the collection.

```
$skipped = $collection->skip(2);
```

50. **`slice`**: Get a portion of the collection starting from a given index.

```
$sliced = $collection->slice(2);
```

CONTACTS IN LARAVEL

Certainly! Let's explore Laravel Contracts with an easy-to-understand definition, a simplified real-life example, and sample code.

Laravel Contracts in Simple Terms:

- **What are Contracts?**

Contracts in Laravel are a set of defined interfaces that provide a blueprint for specific functionalities. They act as agreements that classes must adhere to in order to work with Laravel's core services.

- **Why Use Contracts?**

  Contracts help ensure consistency and compatibility in Laravel by specifying what methods a class must implement. This allows for easier swapping of components and promotes clean, testable code.

Scenario: Image Upload in a Blogging Platform

Let's consider a simplified example where you're building a blogging platform in Laravel, and you want to allow users to upload images. We'll use Laravel's `Filesystem` contract for this.

Step 1: Define the Contract

1. **What is it?**

   We'll use Laravel's `Filesystem` contract, which defines methods for interacting with a filesystem (e.g., local disk or cloud storage).

2. **How to Use It?**

   Laravel already provides this contract, so we don't need to create it ourselves.

Step 2: Use the Contract in Your Code

1. **What is it?**

   We'll create an `ImageUploader` class that uses the `Filesystem` contract to upload images.

2. **How to Do It?**

   - First, create a new class `ImageUploader`:

     ```
     php artisan make:class ImageUploader
     ```

   - In the `ImageUploader` class, we'll use dependency injection to work with the `Filesystem` contract:

     ```
     use Illuminate\Contracts\Filesystem\Filesystem;

     class ImageUploader
     {
         protected $filesystem;

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

         public function upload($file)
         {
             // Use the filesystem contract to upload the file
             $path = 'images/' . $file->getClientOriginalName();
             $this->filesystem->put($path, file_get_contents($file));

             return $path;
         }
     }
     ```

     In this class, we inject the `Filesystem` contract into the constructor and use it to upload an image file.

Step 3: Bind the Contract to a Concrete Implementation

1. **What is it?**

   We need to specify which concrete implementation of the `Filesystem` contract to use. Laravel offers multiple options, such as local storage or cloud storage.

2. **How to Do It?**

   In your Laravel configuration (`config/filesystems.php`), specify the filesystem driver you want to use. For example, to use local storage:

```
'default' => 'local',
'disks' => [
    'local' => [
        'driver' => 'local',
        'root' => storage_path('app'),
    ],
],
```

Step 4: Use the `ImageUploader` Class

1. **What is it?**

   Now, you can use the `ImageUploader` class in your controllers or services to upload images.

2. **How to Do It?**

   In your controller:

```
use App\ImageUploader;
use Illuminate\Http\Request;

class ImageController extends Controller
{
    public function upload(Request $request, ImageUploader $imageUploader)
    {
        $file = $request->file('image');
        $path = $imageUploader->upload($file);

        // You can now save the image path in your database or use it as needed
        // ...

        return "Image uploaded to: $path";
    }
}
```

   This controller method accepts an `ImageUploader` instance through dependency injection and uses it to upload an image file.

Summary:

Laravel Contracts are like blueprints that define what methods a class must implement. In our example, we used the `Filesystem` contract to create an `ImageUploader` class that can work with various filesystem drivers, such as local storage. Contracts help ensure that your code adheres to specified interfaces, making it more flexible and maintainable.

Laravel **File Storage** and all functionalities of file storage

Laravel File Storage in Simple Terms:

Laravel's file storage system provides a way to manage and store files in your web application. It abstracts the underlying file system, making it easy to interact with files and directories. You can use it to store user-uploaded images, documents, and other files.

Scenario: User Profile Images for a Social Media App

Imagine you're building a social media application, and users can upload profile pictures. We'll use Laravel's file storage to handle these user profile images.

Step 1: Configure File System Disk

1. **What is it?**

   Laravel allows you to configure multiple "disk" connections, which represent different storage locations. In this example, we'll use the "public" disk, which typically maps to the `public` directory.

2. **How to Configure It?**

   Open `config/filesystems.php` and configure the "public" disk:

```
'public' => [
    'driver' => 'local',
    'root' => public_path('uploads'), // Store files in the "public/uploads" directory
    'url' => env('APP_URL').'/uploads',
],
```

This configures the "public" disk to store files in the `public/uploads` directory and provides a URL to access them.

Step 2: Create a Form for Uploading Profile Pictures

1. **What is it?**

   You need a way for users to upload their profile pictures.

2. **How to Create It?**

   In your Blade view or HTML form, create an input field for file uploads:

```
<form action="/upload-profile" method="POST" enctype="multipart/form-data">
    @csrf
    <input type="file" name="profile_picture">
    <button type="submit">Upload Profile Picture</button>
</form>
```

   Make sure to include the `enctype="multipart/form-data"` attribute for file uploads.

Step 3: Handle File Upload in a Controller

1. **What is it?**

   You need to handle the uploaded file and store it using Laravel's file storage.

2. **How to Do It?**

   In your controller, handle the file upload and store it:

```
public function uploadProfile(Request $request)
{
    $uploadedFile = $request->file('profile_picture');

    if ($uploadedFile) {
        $path = $uploadedFile->store('profile_pictures', 'public'); // Store the file in the "public/profile_pictures" director
        // Update the user's profile picture path in the database
        auth()->user()->update(['profile_picture' => $path]);
    }

    return redirect()->back()->with('success', 'Profile picture uploaded successfully');
}
```

   This code checks if a file was uploaded, stores it in the `public/profile_pictures` directory, and updates the user's profile picture path in the database.

Step 4: Display the User's Profile Picture

1. **What is it?**

   You need to display the user's profile picture on their profile page.

2. **How to Do It?**

   In your view, display the user's profile picture:

```
<img src="{{ asset(auth()->user()->profile_picture) }}" alt="Profile Picture">
```

   This code uses Laravel's `asset` function to generate the correct URL to the user's profile picture.

Step 5: Delete Profile Pictures

1. **What is it?**

You should allow users to delete their profile pictures if needed.

2. **How to Do It?**

   In your controller, handle profile picture deletion:

```
public function deleteProfilePicture()
{
    $user = auth()->user();

    if ($user->profile_picture) {
        Storage::disk('public')->delete($user->profile_picture);
        $user->update(['profile_picture' => null]);
    }

    return redirect()->back()->with('success', 'Profile picture deleted successfully');
}
```

   This code deletes the user's profile picture from storage and updates the database to remove the path.

The previous example provides a solid foundation for understanding Laravel's file storage system and covers the most common use case of handling file uploads and storage. However, there are more advanced features and techniques related to file storage in Laravel that can be useful for developers. Here are some additional topics with code examples:

1. File Validation:

It's important to validate uploaded files to ensure they meet your application's requirements. Laravel provides built-in validation rules for file uploads. Here's an example of how to validate file uploads:

```
$validatedData = $request->validate([
    'profile_picture' => 'required|image|max:2048', // Ensure it's an image and not larger than 2MB
]);
```

2. File Downloads and copy and move:

You may need to allow users to download files from your application. Here's how you can return a file download response in a controller:

```
class DownloadController extends Controller
{
    public function downloadFile($filename)
    {
        // Get the file path based on the provided filename
        $filePath = storage_path('app/public/' . $filename);

        // Check if the file exists
        if (!Storage::disk('public')->exists($filename)) {
            return abort(404); // File not found
        }

        // Determine the MIME type of the file
        $mimeType = mime_content_type($filePath);

  // Download the file with a custom name
        return Storage::disk('public')->download($filePath, $filename, ['Content-Type' => $mimeType]);

        // Copy the file to the destination
        Storage::disk('public')->copy($sourcePath, $destinationPath);

        // Set custom file names for download (optional)
        $customFilename = 'downloaded_file.' . pathinfo($filePath, PATHINFO_EXTENSION);

        // Prepare the response with appropriate headers
        $response = new Response(file_get_contents($filePath), 200);

        // Set the appropriate headers for file download
        $response->header('Content-Type', $mimeType);
        $response->header('Content-Disposition', 'attachment; filename="' . $customFilename . '"');

        return $response;
    }

    public function showDownloadPage()
    {
        return view('download');
    }


}

# Blade File
 <a href="{{ route('download.file', ['filename' => 'example.pdf']) }}" class="btn btn-primary">Download File</a>

# Route Files
Route::get('/download', [DownloadController::class, 'showDownloadPage'])->name('download.page');
Route::get('/download/{filename}', [DownloadController::class, 'downloadFile'])->name('download.file');
```

3. Custom File Storage Disks:

You can create custom file storage disks for different use cases. For example, if you want to store files on an Amazon S3 bucket, you can configure a custom disk like this:

```
AWS_ACCESS_KEY_ID=your-access-key-id
AWS_SECRET_ACCESS_KEY=your-secret-access-key
AWS_DEFAULT_REGION=your-aws-region
AWS_BUCKET=your-s3-bucket-name

's3' => [
    'driver' => 's3',
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION'),
    'bucket' => env('AWS_BUCKET'),
],
```

So, to clarify:

1. The S3 disk configuration is defined in `config/filesystems.php`.
2. The S3 bucket and the files within it are stored remotely on AWS S3 servers, not within your Laravel application's directory structure.
3. Laravel uses the S3 disk configuration to access and manage files in the specified S3 bucket via API calls to the AWS S3 service. Then, you can use this custom disk in your code:

```
Storage::disk('custom-s3')->put('file.txt', 'File contents');
```

4. File Deletion and Cleanup:

You might need to periodically delete or clean up files based on certain conditions. For example, you can delete old log files or temporary files. Here's an example of deleting files older than a specific date:

```
$expirationDate = now()->subDays(7); // Delete files older than 7 days
$filesToDelete = Storage::disk('public')->files('temporary');

foreach ($filesToDelete as $file) {
    $fileModificationDate = Storage::disk('public')->lastModified($file);
    if ($fileModificationDate < $expirationDate) {
        Storage::disk('public')->delete($file);
    }
}
```

Certainly! Here are a few more advanced file storage scenarios in Laravel with code examples:

5. File Uploads with Unique Filenames:

Sometimes, you might want to ensure that uploaded files have unique filenames to avoid overwriting existing files. You can achieve this by generating a unique filename before storing the file. Here's an example:

```
$uploadedFile = $request->file('file');
$originalFilename = $uploadedFile->getClientOriginalName();
$extension = $uploadedFile->getClientOriginalExtension();
$uniqueFilename = md5(uniqid()) . '.' . $extension;

$uploadedFile->storeAs('uploads', $uniqueFilename, 'public');
```

This code generates a unique filename based on the original filename and stores the file with the new name.

6. File Permissions:

You may need to set specific permissions for files or directories in storage. For example, you might want to make uploaded files readable by the public. Here's how to set file permissions:

```
Storage::disk('public')->setVisibility('uploads/file.txt', 'public');
```

This code sets the visibility of the file to "public," making it readable by anyone.

7. File Versioning:

In some cases, you may want to implement file versioning to keep track of changes to files. Here's a simple example of how to create file versions:

```
$originalFile = 'path/to/original.txt';
$versionedFile = 'path/to/versions/' . time() . '_version.txt';

Storage::copy($originalFile, $versionedFile);
```

This code copies the original file to a new location with a timestamp in the filename, creating a versioned copy.

8. File Streaming:

Laravel allows you to stream large files instead of loading them entirely into memory. This is useful for serving large downloads efficiently. Here's how to stream a file as a response:

```
public function streamFile($filename)
{
    $path = storage_path('app/public/' . $filename); // Adjust the path as needed

    return response()->stream(function () use ($path) {
        $stream = fopen($path, 'rb');
        fpassthru($stream);
        fclose($stream);
    }, 200, [
        'Content-Type' => 'application/octet-stream',
        'Content-Disposition' => 'attachment; filename="' . basename($path) . '"',
    ]);
}
```

9. Storing a File from a Resource:

```
use Illuminate\Support\Facades\Storage;

$contents = 'This is the content of the file.';
 // Or
$resource = fopen('path/to/your/local/file.jpg', 'r');

// Store the file from the resource in the 'public' disk
Storage::disk('public')->put('file.jpg', $resource Or $contents);

fclose($resource);

return 'File has been successfully stored.';
```

This code streams the file directly to the client's browser, which is more memory-efficient for large files.

HELPERS

- **What are Laravel Helpers?**

  Laravel Helpers are a set of utility functions provided by Laravel that simplify common tasks in web development. These functions can be used throughout your Laravel application to perform various tasks quickly and efficiently.

Certainly! Here are the Laravel Helpers for arrays and objects with short definitions and code examples:

Working with Arrays (Arr Helpers):

1. `Arr::get`:

   - Definition: Gets a value from an array or returns a default if not found.
   - Example:

   ```
   $value = Arr::get($array, 'key', 'default');
   ```

2. `Arr::has`:

   - Definition: Checks if a key exists in an array.
   - Example:

   ```
   $exists = Arr::has($array, 'key');
   ```

3. `Arr::only`:

   - Definition: Filters an array to only include specified keys.
   - Example:

   ```
   $filteredArray = Arr::only($array, ['key1', 'key2']);
   ```

4. `Arr::except`:

   - Definition: Filters an array to exclude specified keys.

- Example:

```
$filteredArray = Arr::except($array, ['key1', 'key2']);
```

5. `Arr::first`:

   - Definition: Gets the first item from an array.
   - Example:

```
$firstItem = Arr::first($array);
```

6. `Arr::last`:

   - Definition: Gets the last item from an array.
   - Example:

```
$lastItem = Arr::last($array);
```

7. `Arr::pluck`:

   - Definition: Extracts a list of values from an array of objects by key.
   - Example:

```
$values = Arr::pluck($array, 'key');
```

8. `Arr::shuffle`:

   - Definition: Shuffles the elements of an array randomly.
   - Example:

```
$shuffledArray = Arr::shuffle($array);
```

9. `Arr::sort`:

   - Definition: Sorts an array in ascending order.
   - Example:

```
$sortedArray = Arr::sort($array);
```

10. `Arr::sortDesc`:

    - Definition: Sorts an array in descending order.
    - Example:

```
$sortedArray = Arr::sortDesc($array);
```

11. `Arr::collapse`:

    - Definition: Collapses an array of arrays into a single array.
    - Example:

```
$collapsedArray = Arr::collapse($arrayOfArrays);
```

12. `Arr::dot`:

    - Definition: Flattens a multi-dimensional array into a single dot-notated array.
    - Example:

```
$flatArray = Arr::dot($multiDimensionalArray);
```

13. `Arr::exists`:

    - Definition: Checks if a key exists in a multi-dimensional array.
    - Example:

```
$exists = Arr::exists($multiDimensionalArray, 'key');
```

14. `Arr::isAssoc`:
    - Definition: Checks if an array is associative (has string keys) or not.
    - Example:

    ```
    $isAssoc = Arr::isAssoc($array);
    ```

15. `Arr::join`:
    - Definition: Joins the values of an array into a single string.
    - Example:

    ```
    $joinedString = Arr::join(', ', $array);
    ```

16. `Arr::set`:
    - Definition: Sets a value within a multi-dimensional array using dot notation.
    - Example:

    ```
    Arr::set($array, 'key.subkey', 'value');
    ```

Working with Objects (data Helpers):

11. `data_get`:
    - Definition: Gets a value from an object or returns a default if not found.
    - Example:

    ```
    $value = data_get($object, 'property', 'default');
    ```

12. `data_set`:
    - Definition: Sets a value in an object.
    - Example:

    ```
    data_set($object, 'property', $value);
    ```

13. `data_forget`:
    - Definition: Removes a property from an object.
    - Example:

    ```
    data_forget($object, 'property');
    ```

14. `data_fill`:
    - Definition: Fills a property in an object with a value if it's empty.
    - Example:

    ```
    data_fill($object, 'property', 'default');
    ```

Paths Helpers

1. `app_path()`:
    - Definition: Returns the path to the `app` directory.
    - Example:

    ```
    $appPath = app_path();
    ```

2. `base_path()`:
    - Definition: Returns the base path of the Laravel application.
    - Example:

```
$basePath = base_path();
```

3. `config_path()`:

   - Definition: Returns the path to the `config` directory.
   - Example:

   ```
   $configPath = config_path();
   ```

4. `database_path()`:

   - Definition: Returns the path to the `database` directory.
   - Example:

   ```
   $databasePath = database_path();
   ```

5. `lang_path()`:

   - Definition: Returns the path to the `resources/lang` directory.
   - Example:

   ```
   $langPath = lang_path();
   ```

6. `mix()`:

   - Definition: Generates a URL for a versioned Mix file.
   - Example:

   ```
   $mixUrl = mix('css/app.css');
   ```

7. `public_path()`:

   - Definition: Returns the path to the `public` directory.
   - Example:

   ```
   $publicPath = public_path();
   ```

8. `resource_path()`:

   - Definition: Returns the path to the `resources` directory.
   - Example:

   ```
   $resourcePath = resource_path();
   ```

9. `storage_path()`:

   - Definition: Returns the path to the `storage` directory.
   - Example:

   ```
   $storagePath = storage_path();
   ```

URLs

1. `url`:

   - **Definition:** Generates a fully qualified URL for a given path.
   - **Example:**

   ```
   $fullUrl = url('/dashboard');
   // Result: http://example.com/dashboard
   ```

2. `route`:

   - **Definition:** Generates a URL for a named route.
   - **Example:**

```
$routeUrl = route('profile');
// Result: http://example.com/profile
```

3. `to_route`:

   - **Definition:** Generates a URL for a named route.
   - **Example:**

   ```
   $routeUrl = to_route('profile');
   // Result: http://example.com/profile
   ```

4. `asset`:

   - **Definition:** Generates a URL for an asset file (e.g., CSS, JavaScript).
   - **Example:**

   ```
   $assetUrl = asset('css/styles.css');
   // Result: http://example.com/css/styles.css
   ```

5. `secure_url`:

   - **Definition:** Generates a fully qualified HTTPS URL for a given path.
   - **Example:**

   ```
   $secureFullUrl = secure_url('/login');
   // Result: https://example.com/login
   ```

6. `secure_asset`:

   - **Definition:** Generates a HTTPS URL for an asset file.
   - **Example:**

   ```
   $secureAssetUrl = secure_asset('js/app.js');
   // Result: https://example.com/js/app.js
   ```

7. `action`:

   - **Definition:** Generates a URL for a controller action.
   - **Example:**

   ```
   $actionUrl = action('HomeController@index');
   // Result: http://example.com/home
   ```

MOST IMPORTANT HELPERS

Certainly! Here are code view examples of some of the frequently used Laravel Helpers along with short explanations:

1. `app` Helper - Accessing the Application Container:

```
$mailer = app('mailer');
```

2. `auth` Helper - Checking Authentication:

```
if (auth()->check()) {
    // User is authenticated
}
```

3. `config` Helper - Accessing Configuration Values:

```
$timezone = config('app.timezone');
```

4. `csrf_field` Helper - Generating CSRF Token Field in a Form:

```
<form method="POST" action="/submit">
    @csrf
    <!-- Rest of the form -->
</form>
```

5. `env` Helper - Accessing Environment Variables:

```
$apiKey = env('API_KEY');
```

6. `event` Helper - Firing an Event:

```
event('user.registered', $user);
```

7. `redirect` Helper - Redirecting to a Different URL:

```
return redirect()->route('dashboard');
```

8. `request` Helper - Accessing Request Data:

```
$name = request('name');
```

9. `response` Helper - Creating an HTTP Response:

```
return response('Hello, World!', 200)
    ->header('Content-Type', 'text/plain');
```

10. `session` Helper - Storing Data in the Session:

```
session(['user_id' => 123]);
```

11. `view` Helper - Rendering a View Template:

```
return view('welcome', ['name' => 'John']);
```

12. `abort` Helper - Aborts the current request with an HTTP response.

```
abort(404, 'Page not found');
```

13. `back` Helper - Redirects the user to the previous URL.

```
return back();
```

14. `bcrypt` Helper - Hashes a password using the bcrypt algorithm.

```
$hashedPassword = bcrypt('password');
```

15. `blank` Helper - Checks if a variable is empty or considered "blank."

```
if (blank($value)) {
    // Variable is empty or blank
}
```

16. `cache` Helper - Accesses the Laravel cache system.

```
$cachedData = cache('key');
```

17. `cookie` Helper - Gets or sets a cookie value.

```
$cookieValue = cookie('name', 'value', $minutes);
```

18. `dump` Helper - Dumps the contents of a variable or expression for debugging.

```
dump($variable);
```

19. `env` Helper - Retrieves values from the `.env` configuration file.

```
$apiKey = env('API_KEY');
```

20. `old` Helper - Retrieves old input data from a previous form submission.

```
$oldValue = old('input_name');
```

21. `redirect` Helper - Redirects the user to a different URL.

```
return redirect()->route('dashboard');
```

22. `session` Helper - Accesses or stores data in the session.

```
session(['user_id' => 123]);
```

23. `throw_if` Helper - Throws an exception if a condition is true.

```
throw_if($condition, Exception::class, 'Condition is true');
```

Certainly! Here are additional frequently used Laravel Helpers without any repeats:

1. `cookie` Helper - Gets or sets a cookie value.

```
$cookieValue = cookie('name', 'value', $minutes);
```

2. `csrf_token` Helper - Retrieves the CSRF token for use in forms.

```
$token = csrf_token();
```

3. `dd` Helper - Dumps the contents of a variable or expression and terminates the script for debugging.

```
dd($variable);
```

4. `dispatch` Helper - Dispatches a job to be processed by a queue worker.

```
dispatch(new ProcessTask($task));
```

5. `method_field` Helper - Generates an HTML hidden input field for specifying an HTTP method in a form.

```
@method('PUT')
```

6. `report` Helper - Reports an exception or error to the Laravel error handling system.

```
report($exception);
```

7. `response` Helper - Creates an HTTP response with a given content and status code.

```
return response('Hello, World!', 200);
```

8. `throw_unless` Helper - Throws an exception if a condition is false.

```
throw_unless($condition, Exception::class, 'Condition is false');
```

9. `today` Helper - Retrieves the current date as a Carbon instance.

```
$today = today();
```

10. `transform` Helper - Transforms an array or object using a callback.

```
$transformed = transform($data, function ($item) {
    return $item->toArray();
});
```

11. `validator` Helper - Creates a new Validator instance for validating data.

```
$validator = validator($data, $rules);
```

12. `with` Helper - Passes data to a view.

```
return view('welcome')->with('name', 'John');
```

Custom Helpers:

Step 1: Create the Helper Function

1. **What is it?**

   A helper function is a custom function that you can use throughout your Laravel application.

2. **How to Create It?**

   In your Laravel project, navigate to the `app` directory. Create a new PHP file in this directory and name it something like `CartHelper.php`. This file will contain your custom helper function.

3. **What's Inside?**

   In `CartHelper.php`, define your custom helper function:

```
<?php

if (!function_exists('calculateTotalPrice')) {
    function calculateTotalPrice($items)
    {
        $total = 0;
        foreach ($items as $item) {
            $total += $item['price'] * $item['quantity'];
        }
        return $total;
    }
}
```

function formatCurrency($amount, $currency = 'USD') { return number_format($amount, 2) . ' ' . $currency; }

function truncateString($string, $length = 100, $append = '...') { if (strlen($string) > $length) { return substr($string, 0, $length) . $append; } return $string; }

function generateRandomString($length = 10) { return Str::random($length); }

function formatDate($date, $format = 'Y-m-d H:i:s') { return \Carbon\Carbon::parse($date)->format($format); }

function generateSlug($string) { // Convert the string to lowercase $slug = strtolower($string);

```
// Replace spaces with hyphens
$slug = str_replace(' ', '-', $slug);

// Remove special characters
$slug = preg_replace('/[^A-Za-z0-9\-]/', '', $slug);

return $slug;
```

}

~~~
  This function, `calculateTotalPrice`, takes an array of items with prices and quantities and calculates the total price.

**Step 2: Autoload the Helper Function**

1. **What is it?**

  You need to tell Laravel to autoload your custom helper function.

2. **How to Do It?**

  In your `composer.json` file, find the `autoload` section and add an `"files"` section to autoload your helper file:

  ```json
  "autoload": {
      "files": [
        "app/CartHelper.php",
       "app/CustomHelper.php"
      ],
      // ...
  }
~~~

}

After adding this, run the following command to update the autoloader:

```
composer dump-autoload
```

Step 3: Use the Helper Function

1. **What is it?**

   You can now use your custom helper function in your Laravel application.

2. **How to Do It?**

   In a controller or view, you can call `calculateTotalPrice` like this:

   ```
   $items = [
       ['name' => 'Product 1', 'price' => 10, 'quantity' => 2],
       ['name' => 'Product 2', 'price' => 15, 'quantity' => 1],
   ];

   $totalPrice = calculateTotalPrice($items);

   return view('cart', ['totalPrice' => $totalPrice]);
   ```

$title = "This is a Sample Title"; $slug = generateSlug($title); // $slug now contains "this-is-a-sample-title"

~~~
   In this example, we calculate the total price of items in the shopping cart and pass it to the view.

**Summary:**

Creating a custom helper function in Laravel is a way to encapsulate commonly used code into reusable functions. In this example, we




### **Laravel HTTP Clients & all types of Http**
---
Certainly! Let's explore Laravel HTTP Clients and the different HTTP methods with a beginner-friendly example.

**Laravel HTTP Clients in Simple Terms**:

- **What are HTTP Clients?**

  In Laravel, HTTP clients are a way to send HTTP requests to external APIs or web services. Think of them as messengers that allow

- **Why Use Them?**

  You use HTTP clients to fetch data from other services, send data to them, or perform various operations over the internet. For ex

Now, let's look at a beginner-friendly example of using Laravel HTTP clients in a real-life scenario.

**Scenario: Fetching Weather Data from an API**

Imagine you're building a weather application, and you want to retrieve weather data from a weather API using Laravel's HTTP client.

**Step 1: Install Laravel HTTP Client**

1. **What is it?**

   Laravel's HTTP client is built-in and doesn't require a separate installation.

**Step 2: Create a Controller**

1. **What is it?**

   A controller handles web requests and responses.

2. **How to Create It?**

   In your terminal, run:

   ```bash
   php artisan make:controller WeatherController
~~~

This generates a `WeatherController.php` file in the `app/Http/Controllers` directory.

3. **What's Inside?**

   In `WeatherController.php`, define a method to fetch weather data from the API:

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Http;

class WeatherController extends Controller
{
    public function getWeather(Request $request)
    {
        // Define the API endpoint and query parameters
        $endpoint = 'https://api.example.com/weather';
        $params = [
            'city' => $request->input('city'),
            'apikey' => 'your-api-key',
        ];

        // Make an HTTP GET request to the API
        $response = Http::get($endpoint, $params);

        // Check if the request was successful
        if ($response->successful()) {
            // Parse the JSON response
            $weatherData = $response->json();
            return view('weather', ['weatherData' => $weatherData]);
        } else {
            return view('error', ['message' => 'Unable to fetch weather data']);
        }
    }
}
```

In this code, we use Laravel's HTTP client (`Http::get`) to send a GET request to a weather API and handle the response.

Step 3: Create Views

1. **What is it?**

   Views are templates that display data to the user.

2. **How to Create Them?**

   Create two Blade views, `weather.blade.php` and `error.blade.php`, to display weather data or an error message.

   `weather.blade.php`:

```
<!DOCTYPE html>
<html>
<head>
    <title>Weather Report</title>
</head>
<body>
    <h1>Weather Report</h1>
    <p>City: {{ $weatherData['city'] }}</p>
    <p>Temperature: {{ $weatherData['temperature'] }}°C</p>
</body>
</html>
```

   `error.blade.php`:

```
<!DOCTYPE html>
<html>
<head>
    <title>Error</title>
</head>
<body>
    <h1>Error</h1>
    <p>{{ $message }}</p>
</body>
</html>
```

Step 4: Define Routes

1. **What is it?**

   Routes map URLs to controller methods.

2. **How to Do It?**

   In `routes/web.php`, define a route that points to the `getWeather` method in the `WeatherController`:

```
Route::get('/get-weather', 'WeatherController@getWeather');
```

Step 5: Make a Request

1. **What is it?**

   You make a request by accessing a URL.

2. **How to Do It?**

   You can now visit `/get-weather` in your browser and enter a city name. The controller will use Laravel's HTTP client to fetch weather data from the API and display it on the `weather.blade.php` view.

Localisation In Laravel

Certainly! Let's walk through Laravel Localization step by step with a real-life example of a professional project.

Scenario: Localizing a Blog Platform

Imagine you're building a blog platform that you want to make available in both English and Spanish. Here's how you can implement Laravel Localization:

Step 1: Publish Language Files

1. **What is it?**

   Laravel provides language files that contain translations for various phrases in your application.

2. **How to Do It?**

   In your terminal, run:

```
php artisan vendor:publish --tag=laravel-lang
```

   This command publishes Laravel's default language files to your `resources/lang` directory.

Step 2: Configure the Locale

1. **What is it?**

   The locale determines which language your application should use.

2. **How to Do It?**

   In `config/app.php`, set the `'locale'` to your desired default language, e.g., `'en'` for English:

```
'locale' => 'en',
```

Step 3: Define Translation Strings

1. **What is it?**

   Translation strings are the phrases you want to translate.

2. **How to Do It?**

   Open `resources/lang/en/messages.php` (or create a new one for Spanish, `resources/lang/es/messages.php`). Define translation strings like this:

```
return [
    'welcome' => 'Welcome to our Blog',
    'read_more' => 'Read More',
];
```

Step 4: Use Translation Strings in Views

1. **What is it?**

   You use translation strings in your views.

2. **How to Do It?**

   In your Blade view, use the `@lang` directive to display translated text:

```
<h1>@lang('messages.welcome')</h1>
<a href="/post/1">@lang('messages.read_more')</a>
```

   This displays "Welcome to our Blog" and "Read More" in the respective languages.

Step 5: Use Translation Strings in Controllers

1. **What is it?**

   You can retrieve translation strings programmatically in your controllers or PHP code.

2. **How to Do It?**

   In your controller or PHP code, use the `trans` function:

```
$welcomeMessage = trans('messages.welcome');
$readMoreText = trans('messages.read_more');
```

Step 6: Pluralization

1. **What is it?**

   Pluralization is handling singular and plural forms.

2. **How to Do It?**

   Define translation strings with plural forms in `messages.php`:

```
'apple' => '{0} No apples|{1} :count apple|[2,*] :count apples',
```

   Use them with the `trans_choice` function:

```
$appleMessage = trans_choice('messages.apple', 0); // "No apples"
$appleMessage = trans_choice('messages.apple', 1); // "1 apple"
$appleMessage = trans_choice('messages.apple', 5); // "5 apples"
```

   This allows you to handle pluralization correctly.

Step 7: Switching Locale Dynamically

1. **What is it?**

   You may want users to switch between languages dynamically.

2. **How to Do It?**

   Create a language switcher in your view, and use a route to change the locale. For example:

```
<a href="{{ route('setLocale', 'en') }}">English</a>
<a href="{{ route('setLocale', 'es') }}">Español</a>
```

In your routes, define a route to set the locale:

```
Route::get('setLocale/{locale}', 'LocalizationController@setLocale')->name('setLocale');
```

Then, in your `LocalizationController`, set the locale:

```
public function setLocale($locale)
{
    app()->setLocale($locale);
    return redirect()->back();
}
```

This allows users to switch between English and Spanish dynamically.

Summary:

Laravel Localization is essential for making your application accessible to users in different languages and regions. It allows you to define translation strings, use them in views and code, handle pluralization, and even switch between languages dynamically. This is crucial for creating a user-friendly, global web application.

Laravel Mail

Certainly! Let's explore Laravel Mail step by step with easy explanations and a real-life example for beginners.

Laravel Mail in Simple Terms:

- **What is Mail in Laravel?**

  Laravel's Mail feature allows you to send email notifications from your application. It's like having a built-in email sender for your web application.

- **Why Use It?**

  You can use Laravel Mail to send various types of emails, including welcome emails, password reset links, order confirmations, and more. It's essential for keeping users informed and engaged.

Now, let's dive into the key aspects of Laravel Mail using a real-life example:

Scenario: Sending a Welcome Email to New Users

Imagine you're building a registration system for a blog platform, and you want to send a welcome email to new users when they sign up.

Step 1: Configure the Sender

1. **What is it?**

   You need to set up the sender's email address and name.

2. **How to Do It?**

   In `config/mail.php`, configure the sender details:

```
'from' => [
    'address' => 'noreply@example.com',
    'name' => 'Your App',
],
```

Step 2: Configure the View

1. **What is it?**

   You need to create an email template.

2. **How to Do It?**

   Create an email view file, e.g., `welcome.blade.php`, in the `resources/views/emails` directory. Design your email template.

```
<p>Welcome to Our Blog Platform!</p>
<p>Thank you for joining our community.</p>
```

Step 3: View Data

1. **What is it?**

   You can pass data to your email view.

2. **How to Do It?**

   In your controller, prepare the data you want to send to the view:

```
$data = [
    'username' => 'John',
];
```

Step 4: Sending the Email

1. **What is it?**

   You send the email using Laravel's `Mail` facade.

2. **How to Do It?**

   In your controller, use the `Mail` facade to send the email:

```
use Illuminate\Support\Facades\Mail;
use App\Mail\WelcomeEmail;

Mail::to('john@example.com')->send(new WelcomeEmail($data));
```

   Here, `WelcomeEmail` is a Mailable class we'll create next.

Step 5: Create a Mailable Class

1. **What is it?**

   A Mailable class defines how the email should be constructed.

2. **How to Do It?**

   Run this command to create a Mailable class:

```
php artisan make:mail WelcomeEmail
```

   This generates a `WelcomeEmail.php` file in the `app/Mail` directory.

Step 6: Configure the Mailable

1. **What is it?**

   You define how the email should be built in the Mailable class.

2. **How to Do It?**

   In `WelcomeEmail.php`, configure the sender, subject, and attach the view data:

```
public function build()
{
    return $this->from('noreply@example.com')
                ->subject('Welcome to Our Blog')
                ->view('emails.welcome')
                ->with([
                    'username' => $this->data['username'],
                ]);
}
```

We're using the view we created earlier and passing the `$data` array to it.

Step 7: Sending Attachments

1. **What is it?**

   You can attach files to your emails.

2. **How to Do It?**

   In your Mailable class, use the `attach` method:

```
public function build()
{
    return $this->from('noreply@example.com')
                ->subject('Welcome to Our Blog')
                ->view('emails.welcome')
                ->with([
                    'username' => $this->data['username'],
                ])
                ->attach(public_path('files/welcome.pdf'));
}
```

   This attaches the `welcome.pdf` file to the email.

Step 8: Sending Inline Attachments

1. **What is it?**

   You can embed images or files within your email content.

2. **How to Do It?**

   In your Mailable class, use the `embed` method:

```
public function build()
{
    return $this->from('noreply@example.com')
                ->subject('Welcome to Our Blog')
                ->view('emails.welcome')
                ->with([
                    'username' => $this->data['username'],
                ])
                ->attach(public_path('files/welcome.pdf'), [
                    'as' => 'welcome.pdf',
                ])
                ->embed(public_path('images/logo.png'), 'logo');
}
```

Note: The email would look like this:

```
From: noreply@example.com
Subject: Welcome to Our Blog

-- HTML Content --
<p>Welcome to Our Blog!</p>
<p>Thank you for joining our community.</p>
-- HTML Content --

-- Embedded Image --
<img src="cid:logo" alt="Logo">
-- Embedded Image --

-- Attachments --
Attachment: welcome.pdf
```

This embeds the `logo.png` image in your email.

Step 9: Customizing Headers, Tags, and Metadata

1. **What is it?**

   You can customize email headers, add tags for categorization, and set metadata.

2. **How to Do It?**

   In your Mailable class, use methods like `withSwiftMessage`, `tag`, and `metadata`:

```
  public function build()
  {
      return $this->from('noreply@example.com')
                  ->

subject('Welcome to Our Blog')
                  ->view('emails.welcome')
                  ->with([
                      'username' => $this->data['username'],
                  ])
                  ->withSwiftMessage(function ($message) {
                      $message->getHeaders()
                              ->addTextHeader('X-Custom-Header', 'Hello');
                  })
                  ->tag(['registration', 'welcome'])
                  ->metadata([
                      'user_id' => 123,
                      'account_type' => 'free',
                  ]);
  }
```

Note: The email would look like this:

```
From: noreply@example.com
Subject: Welcome to Our Blog
X-Custom-Header: Hello
Tags: registration, welcome
Metadata: {"user_id":123,"account_type":"free"}

-- HTML Content --
<p>Welcome to Our Blog!</p>
<p>Thank you for joining our community.</p>
-- HTML Content --
```

Step 10: Sending the Email

1. **What is it?**

   You send the email with all configurations in place.

2. **How to Do It?**

   In your controller, use the `Mail` facade to send the email:

```
use Illuminate\Support\Facades\Mail;
use App\Mail\WelcomeEmail;

Mail::to('john@example.com')->send(new WelcomeEmail($data));
```

   This sends the welcome email to the user.

Summary:

Laravel Mail is a powerful tool for sending various types of emails in your web application. You configure the sender, create email views, pass data to them, send attachments, customize headers and metadata, and more. This example demonstrates sending a welcome email to new users, but you can adapt these concepts to other email notifications in your professional projects.

Custom Email

Step 1: Generate a Custom Mail Notification

Run the following command to generate a custom mail notification class:

```
php artisan make:mail NewCommentNotification
```

This command will create a `NewCommentNotification.php` file in the `app/Mail` directory.

Step 2: Configure the Mail Notification

In the generated `NewCommentNotification.php` file, you can configure the mail notification. Here's an example configuration:

```
<?php

namespace App\Mail;

use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Contracts\Queue\ShouldQueue;

class NewCommentNotification extends Mailable
{
    use Queueable, SerializesModels;

    public $user;
    public $comment;

    /**
     * Create a new message instance.
     *
     * @param User $user
     * @param Comment $comment
     */
    public function __construct($user, $comment)
    {
        $this->user = $user;
        $this->comment = $comment;
    }

    /**
     * Build the message.
     *
     * @return $this
     */
    public function build()
    {
        return $this->from('noreply@example.com')
                    ->subject('New Comment Notification')
                    ->markdown('emails.new-comment-notification');
    }
}
```

In this example:

- We pass the user and comment data to the constructor so they can be used in the email content.
- The `build` method defines the email's sender, subject, and the Markdown template `emails.new-comment-notification`.

Step 3: Create a Markdown Email Template

Next, you should create a Markdown email template. Create a new Blade Markdown file named `new-comment-notification.blade.php` in the `resources/views/emails` directory. This is where you design the email content.

Here's a simple example of the template:

```
@component('mail::message')
# New Comment Notification

Hi {{ $user->name }},

You have received a new comment on your blog post.

Comment:
{{ $comment->content }}

Thanks,<br>
{{ config('app.name') }}
@endcomponent
```

This template uses the `mail::message` component to structure the email. It displays the user's name, the new comment's content, and a closing message.

Step 4: Send the Custom Mail Notification

In your application logic (e.g., when a new comment is added to a post), you can send the custom mail notification like this:

```
use Illuminate\Support\Facades\Mail;
use App\Mail\NewCommentNotification;

// ...

$user = User::find(1); // Replace with the actual user receiving the notification
$comment = Comment::find(1); // Replace with the actual comment

Mail::to($user->email)->send(new NewCommentNotification($user, $comment));
```

This code sends the `NewCommentNotification` mail to the specified user's email address with the user and comment data.

Notification In Laravel

Certainly! Let's create a custom database notification in Laravel using the `php artisan notification:table` command. We'll assume you want to notify a user when they receive a new follower on a social media platform in a professional project. Here's how you can do it:

Step 1: Generate the Notification Table Migration

Run the following command to generate the migration for the notification table:

```
php artisan notifications:table
```

This command will generate a migration file for the notification table. You can find it in the `database/migrations` directory.

Step 2: Run the Migration

Next, run the migration to create the notification table in your database:

```
php artisan migrate
```

This will create the `notifications` table in your database.

Step 3: Create a Custom Notification

Now, create a custom notification class. You can use the following command to generate it:

```
php artisan make:notification NewFollowerNotification
```

This command will create a `NewFollowerNotification.php` file in the `app/Notifications` directory.

Step 4: Configure the Notification

In the generated `NewFollowerNotification.php` file, you can configure the notification. Here's an example configuration:

```php
<?php

namespace App\Notifications;

use Illuminate\Notifications\Notification;
use Illuminate\Notifications\Messages\MailMessage;

class NewFollowerNotification extends Notification
{
    public $follower;

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

    public function toDatabase($notifiable)
    {
        return [
            'follower_id' => $this->follower->id,
            'follower_name' => $this->follower->name,
        ];
    }
}
```

In this example:

- We pass the follower data to the constructor so it can be used in the notification.
- The `toDatabase` method defines the data to be stored in the `notifications` table. In this case, we store the follower's ID and name.

Step 5: Use the Custom Notification

In your application logic (e.g., when a user gains a new follower), you can send the custom notification like this:

```php
use App\Notifications\NewFollowerNotification;

// ...

$user->notify(new NewFollowerNotification($follower));
```

This code sends the `NewFollowerNotification` to the specified user with the follower data.

Step 6: Retrieve Notifications

You can retrieve a user's notifications from the `notifications` table and display them in your application. For example, you might show a list of notifications in the user's profile.

Here's how you can retrieve unread notifications for a user:

```php
$unreadNotifications = auth()->user()->unreadNotifications;
```

You can then loop through these notifications and display them as needed.

DB Notification In Read unRead

Certainly! Let's complete the example of a database notification system in Laravel, including displaying notifications in a Blade file, marking them as read, and allowing users to mark them as read when clicked.

Step 1: Generate the Notification Table

We've already covered this step. You can refer to the previous response to create the `notifications` table.

Step 2: Generate a Custom Notification

We've already generated the `NewFollowerNotification` class. You can refer to the previous response for the code example.

Step 3: Use the Custom Notification

In your application logic (e.g., when a user gains a new follower), send the custom notification like this:

```
use App\Notifications\NewFollowerNotification;

// ...

$user->notify(new NewFollowerNotification($follower));
```

Step 4: Display Notifications in a Blade File

Create a Blade view (e.g., `notifications.blade.php`) to display the user's notifications. Here's how to loop through and display them:

```
@extends('layouts.app') {{-- Use your layout file here --}}

@section('content')
<div class="container">
    <h1>Notifications</h1>

    @forelse($notifications as $notification)
        <div class="notification {{ $notification->read_at ? 'read' : 'unread' }}">
            <p>{!! $notification->data['message'] !!}</p>
            <small>{{ $notification->created_at->diffForHumans() }}</small>
            @if(!$notification->read_at)
                <a href="{{ route('markAsRead', $notification->id) }}" class="mark-as-read">Mark as Read</a>
            @endif
        </div>
    @empty
        <p>No notifications</p>
    @endforelse
</div>
@endsection
```

In this Blade view:

- We loop through the `$notifications` variable, which should be passed from your controller.
- Each notification is displayed with a message, timestamp, and a "Mark as Read" link if it's unread.
- When clicking "Mark as Read," it will send a request to the `markAsRead` route.

Step 5: Mark Notifications as Read

Define a route and a controller method to handle marking notifications as read:

```
// routes/web.php
Route::get('/mark-as-read/{id}', 'NotificationController@markAsRead')->name('markAsRead');
```

```
// app/Http/Controllers/NotificationController.php
public function markAsRead($id)
{
    $notification = auth()->user()->notifications()->where('id', $id)->first();

    if ($notification) {
        $notification->markAsRead();
    }

    return redirect()->back();
}
```

This code marks a notification as read when the user clicks the "Mark as Read" link.

Step 6: Retrieve and Pass Notifications to the Blade View

In your controller, retrieve the user's notifications and pass them to the Blade view:

```
use Illuminate\Support\Facades\Auth;

// ...

public function index()
{
    $user = Auth::user();
    $notifications = $user->notifications()->latest()->paginate(10); // Change the pagination settings as needed

    return view('notifications', compact('notifications'));
}
```

Now, when you visit the `/notifications` route (or your preferred route), you'll see the user's notifications displayed. Users can mark notifications as read, and unread notifications will have a "Mark as Read" link.

SMS notifications To User

Sending SMS notifications in a Laravel project typically involves using an SMS gateway service. These services allow you to send SMS messages programmatically. In this example, we'll use Twilio, a popular SMS gateway. Here's how to send SMS notifications to users in a Laravel professional project:

Step 1: Set Up a Twilio Account

1. Sign up for a Twilio account: Go to the Twilio website (https://www.twilio.com/) and create an account.
2. Get your Twilio credentials: After signing in, you'll find your Account SID and Auth Token in your Twilio Dashboard. Keep these secure; you'll need them to send SMS messages.
3. Obtain a Twilio phone number: Purchase a Twilio phone number that you can use to send SMS messages. You can do this from your Twilio Dashboard.

Step 2: Install the Twilio SDK

In your Laravel project, install the Twilio SDK using Composer:

```
composer require twilio/sdk
```

Step 3: Configure Twilio in Laravel

In your `.env` file, add your Twilio credentials:

```
TWILIO_SID=your_twilio_account_sid
TWILIO_AUTH_TOKEN=your_twilio_auth_token
TWILIO_PHONE_NUMBER=your_twilio_phone_number
```

Then, in your `config/services.php` file, add the Twilio configuration:

```
'twilio' => [
    'sid' => env('TWILIO_SID'),
    'token' => env('TWILIO_AUTH_TOKEN'),
    'from' => env('TWILIO_PHONE_NUMBER'),
],
```

Step 4: Create a Notification

Generate a new notification class for SMS notifications:

```
php artisan make:notification SMSNotification
```

In the generated `SMSNotification.php` file, customize it to send SMS messages. For example:

```
use Illuminate\Notifications\Notification;
use Illuminate\Notifications\Messages\NexmoMessage;

class SMSNotification extends Notification
{
    public function toNexmo($notifiable)
    {
        return (new NexmoMessage)
            ->content('This is your SMS notification message.');
    }
}
```

Here, we're using the Nexmo message provider, but you can replace it with Twilio by using `TwilioMessage` instead.

Step 5: Use the SMS Notification

In your application logic (e.g., when you want to send an SMS notification), use the notification like this:

```
use App\Notifications\SMSNotification;

// ...

$user->notify(new SMSNotification());
```

This code sends an SMS notification to the user.

Step 6: Send SMS Notifications from Events

You can also send SMS notifications from events in your Laravel project. For example, when a user signs up, you can trigger an event that sends a welcome SMS. First, create an event and listener:

```
php artisan make:event UserRegistered
php artisan make:listener SendWelcomeSMS --event=UserRegistered
```

In the `SendWelcomeSMS` listener, send the SMS notification:

```
use App\Notifications\SMSNotification;

public function handle(UserRegistered $event)
{
    $user = $event->user;
    $user->notify(new SMSNotification());
}
```

Dispatch this event when a user registers.

With these steps, you can send SMS notifications to users in your Laravel professional project using the Twilio SMS gateway or any other SMS gateway service you prefer. Remember to adapt the code to your specific project needs and requirements.

SMS notifications To User By Vonage website (https://www.vonage.com/)

Certainly! To send SMS notifications using the Vonage (formerly Nexmo) SMS gateway with the `laravel/vonage-notification-channel` package, you can follow these steps. In this example, we'll create a professional project code to send SMS notifications.

Step 1: Install Required Packages

Install the `laravel/vonage-notification-channel` package along with `guzzlehttp/guzzle` for HTTP requests:

```
composer require laravel/vonage-notification-channel guzzlehttp/guzzle
```

Step 2: Set Up a Vonage Account

1. Sign up for a Vonage account: Go to the Vonage website (https://www.vonage.com/) and create an account.
2. Get your Vonage API credentials: After signing in, you'll find your API Key and API Secret in your Vonage Dashboard. Keep these secure; you'll need them to send SMS messages.

Step 3: Configure Vonage in Laravel

In your `.env` file, add your Vonage API credentials:

```
VONAGE_API_KEY=your_vonage_api_key
VONAGE_API_SECRET=your_vonage_api_secret
VONAGE_PHONE_NUMBER=your_vonage_phone_number
```

Step 4: Create a Notification

Generate a new notification class for SMS notifications:

```
php artisan make:notification SMSNotification
```

In the generated `SMSNotification.php` file, customize it to send SMS messages using the Vonage notification channel:

```
use Illuminate\Notifications\Notification;
use NotificationChannels\Vonage\VonageMessage;

class SMSNotification extends Notification
{
    public function toVonage($notifiable)
    {
        return VonageMessage::create()
            ->content('This is your SMS notification message.');
    }
}

 // OR
 use Illuminate\Notifications\Messages\VonageMessage;

class SMSNotification extends Notification
{
    protected $content;
    protected $from;

    public function __construct($content, $from = null)
    {
        $this->content = $content;
        $this->from = $from;
    }

    public function toVonage($notifiable): VonageMessage
    {
        $message = (new VonageMessage)->content($this->content);

        if ($this->from) {
            $message->from($this->from);
        }

        return $message;
    }
}
```

Step 5: Use the SMS Notification

In your application logic (e.g., when you want to send an SMS notification), use the notification like this:

```
use App\Notifications\SMSNotification;

// ...

$user->notify(new SMSNotification());

// or
$user->notify(new SMSNotification('Dynamic SMS content', '15554443333'));
```

From: 15554443333 Message: Dynamic SMS content

This code sends an SMS notification to the user using the Vonage SMS gateway.

Step 6: Send SMS Notifications from Events

You can also send SMS notifications from events in your Laravel project. For example, when a user signs up, you can trigger an event that sends a welcome SMS. First, create an event and listener:

```
php artisan make:event UserRegistered
php artisan make:listener SendWelcomeSMS --event=UserRegistered
```

In the `SendWelcomeSMS` listener, send the SMS notification:

```
use App\Notifications\SMSNotification;

public function handle(UserRegistered $event)
{
    $user = $event->user;
    $user->notify(new SMSNotification());
}
```

Dispatch this event when a user registers.

Laravel Package Development

Package development in the context of Laravel, and in software development in general, refers to the process of creating reusable and distributable bundles of code, known as packages or libraries, that extend the functionality of a software framework or application. These packages can contain code, configuration files, views, assets, and more. Laravel, being a popular PHP web framework, encourages package development to modularize and share functionality across different Laravel projects.

Step 1: Create a New Laravel Package

Start by creating a new Laravel package using Composer. Replace `vendorname` and `packagename` with your own vendor and package names.

```
composer create-package vendorname/packagename
```

Step 2: Package Directory Structure

Inside the package directory, you should have a typical Laravel package directory structure, including `src` for your package's source code, and `resources` for any views, assets, or configuration files.

```
packagename/
    ├── src/
    │   ├── PackagenameServiceProvider.php
    │   └── ...
    ├── resources/
    │   ├── views/
    │   └── ...
    └── ...
```

Step 3: Create the Service Provider

Inside the `src` directory, create a service provider. This provider will be used to register your package's components with Laravel.

```
// src/PackagenameServiceProvider.php

namespace Vendorname\Packagename;

use Illuminate\Support\ServiceProvider;

class PackagenameServiceProvider extends ServiceProvider
{
    public function boot()
    {
        // View
        $this->loadViewsFrom(__DIR__.'/../resources/views', 'packagename');

            // Route
           $this->loadRoutesFrom(__DIR__.'/../routes/web.php');

            // Migration
            $this->loadMigrationsFrom(__DIR__.'/../database/migrations');

            // publishes
            $this->publishes([
                __DIR__.'/../public' => public_path('vendor/courier'),
            ], 'public');

            // Language
              $this->loadTranslationsFrom(__DIR__.'/../lang', 'courier');
              //  For this Language, using like This
              echo trans('courier::messages.welcome');

              // Publishing File Groups
                $this->publishes([
                    __DIR__.'/../config/package.php' => config_path('package.php')
                ], 'courier-config');

                $this->publishes([
                    __DIR__.'/../database/migrations/' => database_path('migrations')
                ], 'courier-migrations');

    }

    public function register()
    {
        // Register any package-specific services or bindings here
    }
}
```

Step 4: Create Views

Inside the `resources/views` directory, you can create Blade views that your package will use. For example, let's create a simple welcome view.

```
// resources/views/welcome.blade.php

<h1>Welcome to My Package</h1>
```

Step 5: Publish Configuration (Optional)

If your package has configuration files, you can publish them using Artisan. For example, if you have a `config.php` file inside your package's `config` directory:

```
php artisan vendor:publish --tag=packagename-config
```

Step 6: Register the Service Provider

In your Laravel application's `config/app.php`, add your package's service provider to the `providers` array.

```
'providers' => [
    // ...
    Vendorname\Packagename\PackagenameServiceProvider::class,
],
```

Step 7: Use the Package in Your Laravel Application

Now, you can use the package in your Laravel application. For example, you can create a route to display the welcome view from your package.

```
// routes/web.php

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

Step 8: Publish Assets (Optional)

If your package has assets (CSS, JavaScript, etc.), you can publish them using Artisan. For example, if you have assets in your package's `public` directory:

```
php artisan vendor:publish --tag=packagename-assets --force
```

Step 9: Test Your Package

You should write tests for your package using Laravel's testing facilities. This ensures that your package functions as expected and allows you to catch any issues early on.

Step 10: Share Your Package (Optional)

If you want to share your package with the Laravel community, you can consider publishing it on Packagist and GitHub.

Processes : invoking, asynchronous & concurrent process

In Laravel, the term "processes" typically refers to the execution of tasks or operations within the framework. Let me break down the processes you mentioned and provide explanations with some basic code examples:

1. **Invoking Process**:

    - Invoking a process in Laravel means triggering a specific action, usually through an HTTP request. This can be done through routes, controllers, and views.

    Example:

    ```
    // Define a route that invokes a controller method
    Route::get('/example', 'ExampleController@index');

    // Controller method to handle the request
    public function index()
    {
        // Your code logic here
    }
    ```

2. **Asynchronous Process**:

    - An asynchronous process in Laravel allows you to perform tasks in the background without blocking the main application flow. This is often achieved using Laravel's built-in queue system, which utilizes tools like Redis or database for queuing and processing jobs.

    Example:

    ```
    // Queue a job for asynchronous processing
    dispatch(new SomeJob);
    ```

3. **Concurrent Process**:

    - Concurrent processing in Laravel refers to the ability to handle multiple requests or tasks simultaneously. Laravel's underlying server (e.g., Apache or Nginx) typically manages concurrency, allowing multiple users to interact with your application concurrently.

    Example:

```
// Laravel handles concurrent requests by default
```

Queue In Laravel

In Laravel, a queue is a mechanism for performing tasks or jobs asynchronously. It allows you to offload time-consuming and non-blocking tasks, such as sending emails, processing images, or generating reports, to be executed in the background without slowing down your main application.

Here's a step-by-step guide to using queues in Laravel:

1. **Setting Up Laravel Queue**:

   - To use queues, you need to configure your Laravel application. Laravel supports multiple queue drivers like Redis, Beanstalk, and more. Choose one and configure it in your `.env` file.

   Example `.env` configuration for Redis:

   ```
   QUEUE_CONNECTION=redis
   ```

2. **Creating a Job**:

   - Jobs are the individual tasks that you want to run asynchronously. You can generate a new job using Laravel's artisan command:

   ```
   php artisan make:job SendEmail
   ```

3. **Defining the Job Logic**:

   - Inside the generated job class (`SendEmail` in this example), define the task you want to perform. For instance, sending an email.

   ```
   public function handle()
   {
       // Your email sending logic here
   }
   ```

4. **Dispatching the Job**:

   - To enqueue a job for processing, you can dispatch it. This can be done from anywhere in your application.

   ```
   dispatch(new SendEmail);
   ```

5. **Running the Queue Worker**:

   - Laravel provides a queue worker that processes jobs from the queue. Run the worker using the `queue:work` artisan command:

   ```
   php artisan queue:work
   ```

6. **Monitoring and Configuration**:

   - You can monitor the queue status and configure various settings in Laravel's `config/queue.php` file. You can also set up retry and failure strategies for jobs.

Now, here's a simple example of using a queue in a Laravel project:

```
// Create a new job
php artisan make:job SendEmail

// Define the email sending logic in the job class

public function handle()
{
    // Send an email
    Mail::to('example@example.com')->send(new WelcomeEmail);
}

// Dispatch the job
dispatch(new SendEmail);
```

In this example, when you dispatch the `SendEmail` job, it will be processed asynchronously in the background by the queue worker, allowing your main application to remain responsive.

Rate Limiting

Rate Limiting in Laravel for Beginners:

Rate limiting is a way to control how often users or clients can make requests to your Laravel application. It's like setting a speed limit on a road to ensure safe and fair usage. In Laravel, you can implement rate limiting to prevent abuse or overuse of your application's resources, such as API endpoints.

Here's how to understand and implement rate limiting step by step:

1. Concept of Rate Limit:

- Rate limiting sets a maximum number of requests a user or client can make within a specific time period (e.g., 10 requests per minute).

2. Options:

- When setting up rate limiting, you define options like:
  - The key for the rate limit: A unique identifier for this rate limit.
  - The maximum number of requests allowed.
  - The time period for the limit (in minutes).

3. Middleware Configuration:

- Laravel's rate limiting middleware is already included by default. To configure it, open the `app/Http/Kernel.php` file and add the `throttle` middleware to the `$middlewareGroups` array:

```
protected $middlewareGroups = [
    'web' => [
        // Other middleware...
        \Illuminate\Routing\Middleware\ThrottleRequests::class,
    ],
];
```

4. Defining Rate Limiting Options:

- Next, you need to define the rate limiting options in your routes or controllers. This includes the maximum number of requests and the time frame (in minutes) for rate limiting.

```
Route::middleware('throttle:api_rate_limit,10,1')->group(function () {
    Route::get('/api/resource', 'ApiController@getResource');
});
```

In this example, `rate_limit` is the key you define for this rate limit configuration, and `1` specifies that the user is limited to one request within the time frame.

5. ustomizing Response:

- Laravel will automatically handle rate-limited requests by responding with an error message if the limit is exceeded. However, you can customize the response in the `app/Exceptions/Handler.php` file:

```
protected function render($request, Throwable $exception)
{
    if ($exception instanceof ThrottleRequestsException) {
        return response()->json(['error' => 'Too many requests. Please try again later.'], 429);
    }

    return parent::render($request, $exception);
}
```

6. Testing Rate Limiting:

- To test rate limiting, you can use tools like Postman or cURL to send multiple requests within the specified time frame. You should observe that once the limit is reached, you'll receive a `429 Too Many Requests` response.

Here's a simple example of rate limiting applied to an API route:

```
Route::middleware('throttle:api_rate_limit,5,1')->group(function () {
    Route::get('/api/resource', 'ApiController@getResource');
});
```

In this example, `api_rate_limit` is the key, `5` is the maximum number of requests allowed, and `1` is the time frame (in minutes).

Task Scheduling

Task Scheduling in Laravel - Simplified Explanation:

Think of task scheduling in Laravel like setting up automated reminders or chores for your Laravel application. These reminders can be things like sending emails, cleaning up files, or performing routine tasks. You set a schedule, and Laravel takes care of executing these tasks automatically.

Step-by-Step Guide:

1. Identify a Task to Automate:

- Start by identifying a task in your Laravel project that you want to automate. For example, let's say you want to send a daily email to your users.

2. Define the Task:

- Determine what the task should do. In our example, the task is to send a daily email.

3. Create an Artisan Command:

- In Laravel, you create an Artisan command to define the task. Think of it as giving your task a name and a set of instructions. Use this command to encapsulate the logic for the task.

```
   php artisan make:command SendDailyEmail

   public function handle()
{
    // Get the list of users you want to send emails to
    $users = User::where('subscribed', true)->get();

    // Loop through the users and send a daily email
    foreach ($users as $user) {
        // Use Laravel's built-in Mail facade to send an email
        Mail::to($user->email)->send(new DailyEmail());

        // Log the email sent for monitoring purposes
        Log::info("Sent a daily email to {$user->name} at {$user->email}");
    }
}
```

All The Steps

1. Concepts:

- Task scheduling is about automating repetitive tasks that your Laravel application needs to perform at specified times or intervals.

2. Scheduling Configuration:

- Laravel's task scheduling is configured in the `app/Console/Kernel.php` file. This is where you define the tasks and their schedules.

3. Task Definitions:

- You can define tasks as Artisan commands or closures (anonymous functions). Artisan commands are predefined actions you can run, while closures allow you to define custom actions.

4. Schedule Definition:

- Use the `$schedule` property within the `Kernel.php` file to define the tasks and their schedules. You specify the frequency and timing for each task.

5. Cron Syntax:

- Task scheduling in Laravel uses the familiar Cron syntax to define when tasks should run. This syntax includes minute, hour, day, month, and day of the week fields.

6. Example:

- Here's an example of how to schedule a task to run every day at midnight:

```
protected function schedule(Schedule $schedule)
{
    $schedule->command('email:send')->dailyAt('00:00');
}
```

In this example, `email:send` is an Artisan command that will run every day at midnight.

7. Running the Scheduler:

- You need to set up a Cron job on your server to run Laravel's scheduler. This job calls the `schedule:run` Artisan command at regular intervals to check for scheduled tasks.

8. Monitoring and Logging:

- Laravel provides logs and notifications to help you monitor task scheduling and track any errors or failures.

9. Advanced Scheduling:

- Laravel's task scheduling offers advanced features like task retries, preventing task overlap, and even customizing the output and notifications for scheduled tasks.

10. Real-World Use Cases: - Common use cases for task scheduling include sending daily emails, performing database backups, and clearing old cache files.

Implementing task scheduling in Laravel allows you to automate routine maintenance and repetitive tasks, making your application more efficient and reliable.

Here's a basic example of scheduling a custom task in Laravel's `Kernel.php` file:

```
protected function schedule(Schedule $schedule)
{
    $schedule->call(function () {
        // Your custom task logic here
    })->daily();
}
```

In this example, a custom closure is scheduled to run daily. You can replace the closure with any custom logic you need.

String

Laravel includes a variety of functions for manipulating string values. Many of these functions are used by the framework itself; however, you are free to use them in your own applications if you find them convenient.

In Laravel, working with strings is made more convenient and powerful through the use of the `Illuminate\Support\Str` class. This class provides a wide range of string manipulation methods that simplify common string operations. Here are some essential methods and examples of how to use them:

1. **`Str::length($string)`**:
    - Returns the length (number of characters) of a string.

    ```
    $length = Str::length("Hello, Laravel"); // $length = 14
    ```

2. **`Str::lower($string)`**:
    - Converts a string to lowercase.

    ```
    $lower = Str::lower("Hello, Laravel"); // $lower = "hello, laravel"
    ```

3. **`Str::upper($string)`**:
    - Converts a string to uppercase.

    ```
    $upper = Str::upper("Hello, Laravel"); // $upper = "HELLO, LARAVEL"
    ```

4. **`Str::title($string)`**:
    - Converts a string to title case (each word capitalized).

    ```
    $title = Str::title("hello, laravel"); // $title = "Hello, Laravel"
    ```

5. **`Str::ucfirst($string)`**:
   - Converts the first character of a string to uppercase.

```
$ucfirst = Str::ucfirst("hello, laravel"); // $ucfirst = "Hello, laravel"
```

6. **`Str::camel($string)`**:
   - Converts a string to camelCase.

```
$camel = Str::camel("hello_world"); // $camel = "helloWorld"
```

7. **`Str::snake($string, $delimiter = '_')`**:
   - Converts a string to snake_case.

```
$snake = Str::snake("Hello, Laravel"); // $snake = "hello_laravel"
```

8. **`Str::slug($string, $separator = '-')`**:
   - Generates a URL-friendly "slug" from a string.

```
$slug = Str::slug("Hello, Laravel"); // $slug = "hello-laravel"
```

9. **`Str::startsWith($string, $prefix)`**:
   - Checks if a string starts with a given prefix.

```
$startsWith = Str::startsWith("Hello, Laravel", "Hello"); // $startsWith = true
```

10. **`Str::endsWith($string, $suffix)`**:
    - Checks if a string ends with a given suffix.

```
$endsWith = Str::endsWith("Hello, Laravel", "Laravel"); // $endsWith = true
```

Certainly! Here are some more useful string manipulation methods provided by `Illuminate\Support\Str` in Laravel:

11. **`Str::limit($string, $limit = 100, $end = '...')`**:
    - Truncates a string to a specified length and appends an optional ending.

```
$truncated = Str::limit("This is a long text", 10); // $truncated = "This is a..."
```

12. **`Str::plural($string, $count = 2)`**:
    - Pluralizes a string based on a count.

```
$plural = Str::plural("apple", 3); // $plural = "apples"
```

13. **`Str::singular($string)`**:
    - Singularizes a string.

```
$singular = Str::singular("apples"); // $singular = "apple"
```

14. **`Str::random($length = 16)`**:
    - Generates a random string of the specified length.

```
$random = Str::random(8); // Generates an 8-character random string
```

15. **`Str::replace($search, $replace, $subject)`**:
    - Replaces all occurrences of a search string with a replacement in a given subject string.

```
$replaced = Str::replace("quick", "slow", "The quick brown fox"); // $replaced = "The slow brown fox"
```

16. **`Str::contains($haystack, $needles)`**:
    - Checks if a string contains any of the given substrings.

```
$contains = Str::contains("Hello, world", ["world", "universe"]); // $contains = true
```

17. **`Str::before($string, $search)`**:
    - Gets the portion of a string before the first occurrence of a given search string.

```
$before = Str::before("Hello, world", ","); // $before = "Hello"
```

18. **`Str::after($string, $search)`**:
    - Gets the portion of a string after the first occurrence of a given search string.

```
$after = Str::after("Hello, world", ","); // $after = " world"
```

Certainly! Here are more useful string manipulation methods provided by `Illuminate\Support\Str` in Laravel without repeating the ones mentioned earlier:

19. **`Str::replaceFirst($search, $replace, $subject)`**:
    - Replaces the first occurrence of a search string with a replacement in a given subject string.

```
$replacedFirst = Str::replaceFirst("apple", "banana", "apple, apple, cherry"); // $replacedFirst = "banana, apple, cherry"
```

20. **`Str::replaceLast($search, $replace, $subject)`**:
    - Replaces the last occurrence of a search string with a replacement in a given subject string.

```
$replacedLast = Str::replaceLast("apple", "banana", "apple, apple, cherry"); // $replacedLast = "apple, banana, cherry"
```

21. **`Str::startsWithAny($string, array $prefixes)`**:
    - Checks if a string starts with any of the values in the provided array.

```
$startsWithAny = Str::startsWithAny("Hello, world", ["Hi", "Hello", "Hey"]); // $startsWithAny = true
```

22. **`Str::endsWithAny($string, array $suffixes)`**:
    - Checks if a string ends with any of the values in the provided array.

```
$endsWithAny = Str::endsWithAny("Hello, world", ["world", "universe", "planet"]); // $endsWithAny = true
```

23. **`Str::pluralStudly($string, $count = 2)`**:
    - Generates a plural, studly (PascalCase) form of a string based on a count.

```
$pluralStudly = Str::pluralStudly("apple", 3); // $pluralStudly = "Apples"
```

24. **`Str::singularStudly($string)`**:
    - Generates a singular, studly (PascalCase) form of a string.

```
$singularStudly = Str::singularStudly("apples"); // $singularStudly = "Apple"
```

25. **`Str::beforeLast($string, $search)`**:
    - Gets the portion of a string before the last occurrence of a given search string.

```
$beforeLast = Str::beforeLast("apple, banana, cherry", ","); // $beforeLast = "apple, banana"
```

26. **`Str::afterLast($string, $search)`**:

- Gets the portion of a string after the last occurrence of a given search string.

```
$afterLast = Str::afterLast("apple, banana, cherry", ","); // $afterLast = " cherry"
```

Certainly! Here are more `Illuminate\Support\Str` methods in Laravel that haven't been mentioned yet:

27. **`Str::startsWithAll($string, array $prefixes)`**:

    - Checks if a string starts with all of the values in the provided array.

```
$startsWithAll = Str::startsWithAll("Hello, world", ["Hello", "world"]); // $startsWithAll = true
```

28. **`Str::endsWithAll($string, array $suffixes)`**:

    - Checks if a string ends with all of the values in the provided array.

```
$endsWithAll = Str::endsWithAll("Hello, world", ["Hello", "world"]); // $endsWithAll = false
```

29. **`Str::replaceArray($search, array $replace, $subject)`**:

    - Replaces a given array of search values with an array of replace values in the given subject string.

```
$replacedArray = Str::replaceArray(':name', ['John', 'Doe'], 'My name is :name :last'); // $replacedArray = "My name is John Do
```

30. **`Str::ascii($value, $language = 'en')`**:

    - Transliterates a UTF-8 encoded string to ASCII characters.

```
$ascii = Str::ascii("Café"); // $ascii = "Cafe"
```

31. **`Str::pluralStudly($string, $count = 2)`**:

    - Generates a plural, studly (PascalCase) form of a string based on a count.

```
$pluralStudly = Str::pluralStudly("apple", 3); // $pluralStudly = "Apples"
```

32. **`Str::singularStudly($string)`**:

    - Generates a singular, studly (PascalCase) form of a string.

```
$singularStudly = Str::singularStudly("apples"); // $singularStudly = "Apple"
```

33. **`Str::padBoth($string, $length, $padding)`**:

    - Pads both sides of a string with a specified padding character.

```
$padded = Str::padBoth("123", 5, "0"); // $padded = "01230"
```

Fluent String

Certainly! I'll provide examples for some of the most important methods from the Fluent Strings category:

1. **`after($search)`**:

    - Gets the portion of a string that comes after the first occurrence of a given search string.

```
$after = Str::of("Hello, world")->after(", "); // $after = "world"
```

2. **`before($search)`**:

    - Gets the portion of a string that comes before the first occurrence of a given search string.

```
$before = Str::of("Hello, world")->before(", "); // $before = "Hello"
```

3. **`contains($needles)`**:

- Checks if the string contains any of the given substrings.

```
$contains = Str::of("Hello, world")->contains(["Hello", "universe"]); // $contains = true
```

4. **`explode($delimiter, $limit = null)`**:

   - Splits a string into an array using a delimiter.

```
$parts = Str::of("apple,banana,cherry")->explode(","); // $parts = ["apple", "banana", "cherry"]
```

5. **`replace($search, $replace)`**:

   - Replaces all occurrences of a search string with a replacement.

```
$replaced = Str::of("The quick brown fox")->replace("quick", "lazy"); // $replaced = "The lazy brown fox"
```

6. **`startsWith($needles)`**:

   - Checks if the string starts with any of the given prefixes.

```
$startsWith = Str::of("Hello, world")->startsWith(["Hi", "Hello", "Hey"]); // $startsWith = true
```

7. **`length()`**:

   - Returns the length (number of characters) of the string.

```
$length = Str::of("Hello, world")->length(); // $length = 12
```

8. **`limit($limit, $end = '...')`**:

   - Truncates the string to a specified length and appends an optional ending.

```
$limited = Str::of("This is a long text")->limit(10); // $limited = "This is a..."
```

9. **`upper()`**:

   - Converts the string to uppercase.

```
$upper = Str::of("Hello, world")->upper(); // $upper = "HELLO, WORLD"
```

10. **`lower()`**:

    - Converts the string to lowercase.

```
$lower = Str::of("Hello, world")->lower(); // $lower = "hello, world"
```

Certainly! Here are more important Fluent Strings methods without any repetition:

11. **`basename($suffix = '')`**:

    - Gets the trailing part of a path, similar to the PHP `basename` function.

```
$baseName = Str::of("/path/to/file.txt")->basename(".txt"); // $baseName = "file"
```

12. **`dirname()`**:

    - Gets the directory name from a path.

```
$dirName = Str::of("/path/to/file.txt")->dirname(); // $dirName = "/path/to"
```

13. **`replaceFirst($search, $replace)`**:

    - Replaces the first occurrence of a search string with a replacement.

```
$replacedFirst = Str::of("apple, apple, cherry")->replaceFirst("apple", "banana"); // $replacedFirst = "banana, apple, cherry"
```

14. **`replaceLast($search, $replace)`**:

    - Replaces the last occurrence of a search string with a replacement.

    ```
    $replacedLast = Str::of("apple, apple, cherry")->replaceLast("apple", "banana"); // $replacedLast = "apple, banana, cherry"
    ```

15. **`singular()`**:

    - Converts the string to its singular form.

    ```
    $singular = Str::of("apples")->singular(); // $singular = "apple"
    ```

16. **`plural()`**:

    - Converts the string to its plural form.

    ```
    $plural = Str::of("apple")->plural(); // $plural = "apples"
    ```

17. **`camel()`**:

    - Converts the string to camelCase.

    ```
    $camel = Str::of("hello_world")->camel(); // $camel = "helloWorld"
    ```

18. **`snake($delimiter = '_')`**:

    - Converts the string to snake_case.

    ```
    $snake = Str::of("HelloWorld")->snake(); // $snake = "hello_world"
    ```

19. **`ucfirst()`**:

    - Converts the first character of the string to uppercase.

    ```
    $ucfirst = Str::of("hello, world")->ucfirst(); // $ucfirst = "Hello, world"
    ```

20. **`ucsplit($delimiter = ' ', $limit = null)`**:

    - Splits a string into an array using a delimiter and converts the first character of each word to uppercase.

    ```
    $ucsplit = Str::of("hello world")->ucsplit(); // $ucsplit = ["Hello", "World"]
    ```

Certainly! Here are more important Fluent Strings methods without any repetition:

21. **`start($prefix)`**:

    - Prepends a string with a given prefix if it doesn't already start with it.

    ```
    $prefixed = Str::of("world")->start("Hello, "); // $prefixed = "Hello, world"
    ```

22. **`endsWith($needles)`**:

    - Checks if the string ends with any of the given suffixes.

    ```
    $endsWith = Str::of("Hello, world")->endsWith(["world", "universe"]); // $endsWith = true
    ```

23. **`containsAll($needles)`**:

    - Checks if the string contains all of the given substrings.

    ```
    $containsAll = Str::of("Hello, world")->containsAll(["Hello", "world"]); // $containsAll = true
    ```

24. **`is($value)`**:

    - Compares the string to another string to check if they are equal.

```
$isEqual = Str::of("Hello, world")->is("Hello, world"); // $isEqual = true
```

25. **`trim($characters = null)`**:

    - Removes leading and trailing whitespace (or specified characters) from the string.

```
$trimmed = Str::of("  Hello, world  ")->trim(); // $trimmed = "Hello, world"
```

26. **`slice($start, $length = null)`**:

    - Retrieves a portion of the string, starting from a specified position and optionally with a specified length.

```
$sliced = Str::of("Hello, world")->slice(7, 5); // $sliced = "world"
```

27. **`replaceMatches($pattern, $replacement)`**:

    - Replaces all occurrences of a regular expression pattern with a replacement.

```
$replacedMatches = Str::of("apple, banana, cherry")->replaceMatches("/a\w+/", "fruit"); // $replacedMatches = "fruit, fruit, fr
```

28. **`scan($pattern, $callback = null)`**:

    - Searches the string for matches to a regular expression pattern and optionally applies a callback function to each match.

```
$matches = Str::of("The price is $10 and $20.")->scan("/\$\d+/", function ($match) {
    return (int)str_replace('$', '', $match);
});
// $matches = [10, 20]
```

DATABASE

Laravel has made processing with database very easy. Laravel currently supports following 4 databases −

MySQL Postgres SQLite SQL Server

Database

Almost every modern web application interacts with a database. Laravel makes interacting with databases extremely simple across a variety of supported databases using raw SQL, a fluent query builder, and the Eloquent ORM. Currently, Laravel provides first-party support for five databases:

MariaDB 10.3+ (Version Policy) MySQL 5.7+ (Version Policy) PostgreSQL 10.0+ (Version Policy) SQLite 3.8.8+ SQL Server 2017+ (Version Policy)

RAW SQL QUERY

The DB facade provides methods for each type of query: select, update, insert, delete, and statement.

```
use Illuminate\Support\Facades\DB;
SELECT
$users = DB::select('select * from users where active = ?', [1]); // [1] is active value

INSERT
DB::insert('insert into users (id, name) values (?, ?)', [1, 'Marc']);

UPDATE
DB::update(
    'update users set votes = 100 where name = ?',
    ['Anita']
);

DELETE
DB::delete('delete from users where id = ?',[2]);
```

Database Transactions

You may use the transaction method provided by the DB facade to run a set of operations within a database transaction. If an exception is thrown within the transaction closure, the transaction will automatically be rolled back and the exception is re-thrown. If the closure executes successfully, the transaction will automatically be committed. You don't need to worry about manually rolling back or committing while using the transaction method:

```
DB::transaction(function () {
    DB::update('update users set votes = 1');

    DB::delete('delete from posts');
});
```

Database: Query Builder

You may use the table method provided by the DB facade to begin a query. The table method returns a fluent query builder instance for the given table, allowing you to chain more constraints onto the query and then finally retrieve the results of the query using the get method:

```
DB::table('users')->get();

DB::table('users')
->where('id', $user->id)
->update(['active' => true]);
```

Aggregate Function

The query builder also provides a variety of methods for retrieving aggregate values like count, max, min, avg, and sum.

```
DB::table('orders')
    ->where('finalized', 1)
     ->avg('price');
 DB::table('users')->count() or ->max('price');
```

Select Statements Of Query Builder

Select Statement is used to retrive specific column values From database table;

```
DB::table('users')
            ->select('name', 'email as user_email')
            ->get();
```

Raw Expression Of Query Builder

- Raw Methods

```
$orders = DB::table('orders')
                ->select('department', DB::raw('SUM(price) as total_sales'))
                ->groupBy('department')
                ->havingRaw('SUM(price) > ?', [2500])
                ->get();
```

Joins Of Query Builder

Inner Join will be used only join that method add with -> (chaining);

```
$users = DB::table('users')
            ->join('contacts', 'users.id', '=', 'contacts.user_id')
            ->join('orders', 'users.id', '=', 'orders.user_id')
            ->select('users.*', 'contacts.phone', 'orders.price')
            ->get();
```

Left Join / Right Join Clause

```
$users = DB::table('users')
            ->leftJoin('posts', 'users.id', '=', 'posts.user_id')
            ->get();

$users = DB::table('users')
            ->rightJoin('posts', 'users.id', '=', 'posts.user_id')
            ->get();
```

Where Clauses of Query Builder

Multiple Condition is used in query builder by where();

```
$users = DB::table('users')->where('votes', 100)->get();
```

With Operator

> = , <>, *'like','T%'*

```
DB::table('users')
        ->where('name', 'like', 'T%')
        ->get();
```

or

- Multiple Condition may also with array

```
$users = DB::table('users')->where([
    ['status', '=', '1'],
    ['subscribed', '<>', '1'],
])->get();
```

- orWhere Clauses for or condition in query

```
$users = DB::table('users')
                    ->where('votes', '>', 100)
                    ->orWhere('name', 'John')
                    ->get();

// another query
                $users = DB::table('users')
            ->where('votes', '>', 100)
            ->orWhere(function($query) {
                $query->where('name', 'Abigail')
                      ->where('votes', '>', 50);
            })
            ->get();

            same as raw query follow as

select * from users where votes > 100 or (name = 'Abigail' and votes > 50)

// another query

$users = DB::table('users')
           ->where('name', '=', 'John')
           ->where(function ($query) {
               $query->where('votes', '>', 100)
                     ->orWhere('title', '=', 'Admin');
           })
           ->get();

           same as
select * from users where name = 'John' and (votes > 100 or title = 'Admin')
```

Where not Clause

The whereNot and orWhereNot methods may be used to negate a given group of query constraints.

```
$products = DB::table('products')
                ->whereNot(function ($query) {
                    $query->where('clearance', true)
                          ->orWhere('price', '<', 10);
                })
                ->get();
```

Additional Where Clauses

```
->whereBetween('votes', [1, 100]) for between query
->whereNotBetween('votes', [1, 100]) for not between query
->whereIn('id', [1, 2, 3]) for keyword In query
->whereNotIn('id', [1, 2, 3]) for keyword Not In query
 ->whereNull('updated_at') for null constraints value
 ->whereNotNull('updated_at') for not null constraints value
 ->whereDate('created_at', '2016-12-31') for the specific date
```

Ordering, Grouping, Limit & Offset

orderBy : The orderBy method allows you to sort the results of the query by a given column. It may also multiple Ordering ->orderBy();

```
$users = DB::table('users')
                ->orderBy('name', 'desc')->get();
```

- The latest & oldest Methods

latest() or oldest() is for ordering by date.

```
$user = DB::table('users')->latest()->first();
```

- The groupBy & having Methods

```
$users = DB::table('users')
                ->groupBy('account_id')
                ->having('account_id', '>', 100)->get();
```

- Limit & Offset The limit() and offset() methods are used to define perspectively skip() and take() methods in laravel.

```
$users = DB::table('users')->skip(10)->take(5)->get();

or it may also used follow as

$users = DB::table('users')->offset(10)->limit(5)->get();
```

Insert Statements in Query Builder

The query builder also provides an insert method that may be used to insert records into the database table. The insert method accepts an array of column names and values:

```
DB::table('users')->insert([
    'email' => 'kayla@example.com',
    'votes' => 0
]);

or multiple records by passing as arrrays of arrays

DB::table('users')->insert([
    ['email' => 'picard@example.com', 'votes' => 0],
    ['email' => 'janeway@example.com', 'votes' => 0],
]);
```

Update Statements in Query Builder

the query builder can also update existing records using the update method. The update method.

```
$affected = DB::table('users')
             ->where('id', 1)
             ->update(['votes' => 1]);
```

Update Or Insert Sometimes you may want to update an existing record in the database or create it if no matching record exists.

```
DB::table('users')
   ->updateOrInsert(
       ['email' => 'john@example.com', 'name' => 'John'],
       ['votes' => '2']
   );
```

Delete Statements in Query Builder

The query builder's delete method may be used to delete records from the table. The delete method returns the number of affected rows.

```
$deleted = DB::table('users')->delete();
$deleted = DB::table('users')->where('votes', '>', 100)->delete();

to remove entire table with stracture and all records
DB::table('users')->truncate();
```

Eloquent ORM

Eloquent ORM (Object Relation Mapper) is easy to use for users who know how to use objects in PHP. The ORM is an important feature of the Laravel framework, considered a powerful and expensive feature of Laravel. The ORM works with database objects and is used to make relationships with database tables. Each table of the database is mapped with a particular eloquent model. The model object contains various methods to retrieve and update data from the database table. Eloquent ORM can be used with multiple databases by implementing ActiveMethod. This feature makes database-related tasks, such as defining relationships, simpler by defining the database tables.

Is "DB Model" worked with Qeury Builder / Raw Sql Query ?

We won't have to use Model if we're using Query Builder / Raw Sql Query.

```
Using Model in ORM
$users = Users::get(['first_name']);

Using Query Builder
$users = DB::table('users')->get(['first_name']);
```

Eloquent Model Conventions

It is known that ELOQEUNT ORM is used for MODEL.

```
class Flight extends Model
{
}
```

Table Names

If your model's corresponding database table does not fit this convention, you may manually specify the model's table name by defining a table property on the model:

```
class Flight extends Model
{
   # corresponding table name 'flights';
   # for not corresponding, use following.
    protected $table = 'my_flights';
}
or
 # corresponding table Model User;
class UserModel extends Model
{
   # for not corresponding Model, use following.
     protected $table = 'users';
}
```

Primary Keys

The eloquent model considers that each table has a primary key named 'id'. We can override this convention by providing a different name to the $primarykey attribute.

```
class Flight extends Model
{
   protected $primaryKey = 'flight_id';

}
```

Retrieving Models for Select in ORM

It may be used in Blade/controller file by calling Model Namespace. The model's all method will retrieve all of the records from the model's associated database table.

```
use App\Models\Flight;
# $flights = Flight::all();
foreach (Flight::all() as $flight) {
    echo $flight->name;
}
```

Building Queries

We can add additional constraints to queries and then invoke the get method to retrieve the results.The First Method or Constraints must be joined with model by scope resulotion operator(::).the next constraints or methods will use chainging operator (->).

```
$flights = Flight::where('active', 1)->orderBy('name')->take(10)->get();

or

Flight::orderBy('name')->get();
```

Chunking Results - chunk()

Your application may run out of memory if you attempt to load tens of thousands of Eloquent records via the all or get methods. Instead of using these methods, the chunk method may be used to process large numbers of models more efficiently.

```
Flight::orderBy('id')->chunk(200, function ($flights) {
    foreach ($flights as $flight) {
    }
});

or it may be used in following system
 $users = User::all();
      $chunkedUsers = $users->chunk(10);
      foreach ($chunkedUsers as $records) {
         foreach($records as $user) {
            echo $user->id."=>".$user->name."";
         }
      }
```

Chunk may be used in query builder

```
DB::table('users')->orderBy('id')->chunk(100, function ($users) {
    foreach ($users as $user) {
        echo $user->id."=>".$user->name." ";
         }
      });
```

Retrieving Single Models / Aggregates

In addition to retrieving all of the records matching a given query, you may also retrieve single records using the "find", "first", or "firstWhere" methods.

```
 Retrieve a model by its primary key...
Flight::find(1);

# Retrieve the first model matching the query constraints.

Flight::where('active', 1)->first();
same as
Flight::firstWhere('active', 1);
```

Not Found Exception in find() or fist()

The findOrFail and firstOrFail methods will retrieve the first result of the query; however, if no result is found, an Illuminate\Database\Eloquent\ModelNotFoundException will be thrown:

```
 Flight::findOrFail(1);
 Flight::where('legs', '>', 3)->firstOrFail();
```

In Retrieving with Aggregates Function

```
$count = Flight::where('active', 1)->count();
$max = Flight::where('active', 1)->max('price');
```

Select specifice data of some column with their specific condition with ORM

```
                 $data = Page::where('about_status', '0')
                         ->orWhere('faq_status', '0')
                         ->orWhere('contact_status', '0')
                        ->select('about_details', 'faq_details', 'contact_details')
                          ->get();
```

Inserting Models/Insert Method with ORM

To insert a new record into the database, you should instantiate a new model instance and set attributes on the model. Then, call the save method on the model instance. It's not mandatory to save $fillable or $guard Properties for object instantiate of Model class.

```
  public function store(Request $request)
    {
        # Validate the request...

        $flight = new Flight; # or new Flight()
        $flight->name = $request->name;
        $flight->email = $request->email;
        $flight->save();
    }
```

Note : The model's "created_at" and "updated_at" timestamps will automatically be set when the save method is called, so there is no need to set them manually.

UPDATE IN ELOQUENT ORM

We should call the model's save method in Update. Again, the "updated_at" timestamp will automatically be updated.

```
$flight = Flight::find(1);
$flight->name = 'Paris to London';
$flight->email = 'm.karimcu@gmail.com';
$flight->save();
```

Mass Updates

all flights that are active and have a destination of San Diego will be marked as delayed.

```
Flight::where('active', 1)
      ->where('destination', 'San Diego')
      ->update(['delayed' => 1]);
```

Note : The update method expects an array of column and value pairs representing the columns that should be updated. The update method returns the number of affected rows.

Mass Assignment in Eloquent ORM

You may use the create method to "save" a new model using a single PHP statement.

```
$flight = Flight::create([
    'name' => 'London to Paris',
    'email' => 'm.karimcu@gmail.com',
]);
```

- Note : However, before using the create method, you will need to specify either a "fillable" or "guarded" property on your model class

fillable vs $guard methods

The guarded attribute is the opposite of fillable attributes. In Laravel, fillable attributes are used to specify those fields which are to be mass assigned. Guarded attributes are used to specify those fields which are not mass assignable.

```
protected $guard = [] // all fields will be  mass assigned
protected $fillable = ['name','email'] // Only email & name will be  mass assigned but not others.
```

Upserts In Eloquent ORM

Occasionally, you may need to update an existing model or create a new model if no matching model exists. Like the firstOrCreate method, the updateOrCreate.

```
$flight = Flight::updateOrCreate(
    ['departure' => 'Oakland', 'destination' => 'San Diego'],
    ['price' => 99, 'discounted' => 1]
);
```

Note : if a flight exists with a departure location of Oakland and a destination location of San Diego, its price and discounted columns will be updated.

Deleting Models In Eloquent ORM

To delete a model, you may call the delete method on the model instance.

```
$flight = Flight::find(1);
$flight->delete();
or
Flight::find(1)->delete();

 or
 Truncate to remove all records with stractures
 Flight::truncate();
```

Deleting An Existing Model By Its Primary Key

In addition to accepting the single primary key, the destroy method will accept multiple primary keys, an array of primary keys, or a collection of primary keys:

```
Flight::destroy(1);
Flight::destroy(1, 2, 3);
Flight::destroy([1, 2, 3]);
Flight::destroy(collect([1, 2, 3]));
```

Deleting Models Using Queries

In this example, we will delete all flights that are marked as inactive. Like mass updates, mass deletes will not dispatch model events for the models that are deleted.

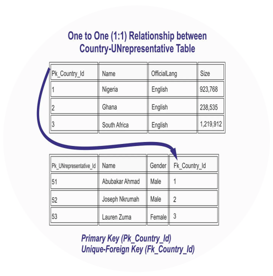
```
$deleted = Flight::where('active', 0)->delete();
```

Eloquent: Table Relationships

Database hase 3 types of Table Relationships.

One To One RelationShip

A one-to-one relationship in a database occurs when each row in table 1 has only one related row in table 2.


One to One (1:1) Relationship between
Country-UNrepresentative Table

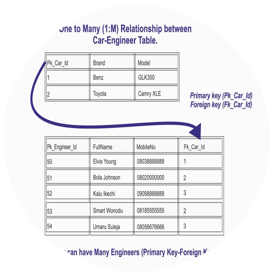
Pk_Country_Id	Name	OfficialLang	Size
1	Nigeria	English	923,768
2	Ghana	English	238,535
3	South Africa	English	1,219,912

Pk_UNrepresentative_Id	Name	Gender	Fk_Country_Id
51	Abubakar Ahmad	Male	1
52	Joseph Nkrumah	Male	2
53	Lauren Zuma	Female	3

Primary Key (Pk_Country_Id)
Unique-Foreign Key (Fk_Country_Id)

One To Many RelationShip

A one-to-many relationship occurs when one record in table 1 is related to one or more records in table 2.


ne to Many (1:M) Relationship between
Car-Engineer Table.

Pk_Car_Id	Brand	Model
1	Benz	GLK350
2	Toyota	Camry XLE

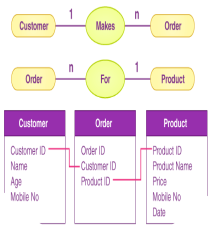
Primary key (Pk_Car_Id)
Foreign key (Fk_Car_Id)

Pk_Engineer_Id	FullName	MobileNo	Fk_Car_Id
50	Elvis Young	08038888888	1
51	Bola Johnson	08020000000	2
52	Kalu Ikechi	09098888888	3
53	Smart Wonodu	08185555555	2
54	Umaru Suleja	08056676666	3

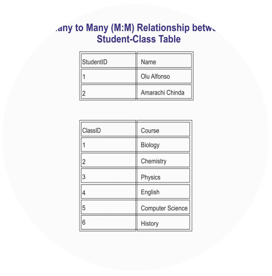
can have Many Engineers (Primary Key-Foreign

Many To Many RelationShip

This type of relationship exists when each of the records of the first table can be associated with one or more records of the second table, as well as a single record of the second table may be related to one or more records of the first table. A Many-to-Many relationship is formed by two one-to-many relationships that are connected by an 'associate table' or 'linking table.' By having fields that are the primary keys of the other two tables, the bridging table connects two tables. The following example will help us comprehend this.



or

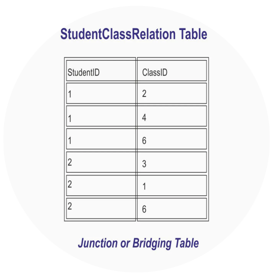

any to Many (M:M) Relationship betw
Student-Class Table

StudentID	Name
1	Olu Alfonso
2	Amarachi Chinda

ClassID	Course
1	Biology
2	Chemistry
3	Physics
4	English
5	Computer Science
6	History

two table and associated table/junction table/bridging table.

Junction table/bridging Table of Many To Many RelationShip


StudentClassRelation Table

StudentID	ClassID
1	2
1	4
1	6
2	3
2	1
2	6

Junction or Bridging Table

- hasOne() for one to one relation table to access data a User model might be associated with one Phone model. To define this relationship, we will place a phone method on the User model. The phone method should call the hasOne method and return its result.

```
class User extends Model
{
    public function phone()
    {
        // Get the phone associated with the user.
        return $this->hasOne(Phone::class);
        or better return $this->hasOne(Phone::class,'foreign_key','local_key');
        // foreign_key defines phone table and local key defines user table.
        // foreign_key = user_id , local_key = id of user table.
    }
}
```

Note : The first argument passed to the hasOne method (phone()) is the name of the related model class. Once the relationship is defined, we may retrieve the related record using Eloquent's dynamic properties.

```
// to retrieve data from phones table by user id.
$phone = User::find(1)->phone;
```

Defining The Inverse Of The Relationship

So, we can access the Phone model from our User model. Next, let's define a relationship on the Phone model that will let us access the user that owns the phone. We can define the inverse of a hasOne relationship using the belongsTo method:

```
class Phone extends Model
{
     * Get the user that owns the phone.
    public function user()
    {
        return $this->belongsTo(User::class);
        or
         return $this->belongsTo(User::class, 'foreign_key'); // foreign_key = user_id
    }
}
```

Note : If the parent model does not use id as its primary key, or you wish to find the associated model using a different column, you may pass a third argument to the belongsTo method specifying the parent table's custom key:

```
return $this->belongsTo(User::class, 'foreign_key', 'owner_key');
```

One To Many Relationship

```
class Post extends Model
{
     * Get the comments for the blog post.
    public function comments()
    {
        return $this->hasMany(Comment::class);
    }
}
```

Important Note : Eloquent will automatically determine the proper foreign key column for the Comment model. By convention, Eloquent will take the "snake case" name of the parent model and suffix it with _id. So, in this example, Eloquent will assume the foreign key column on the Comment model is post_id.

- To Retreive Data from above Relation

```
$comments = Post::find(1)->comments; // ->comments is method name from above
```

Since all relationships also serve as query builders, you may add further constraints to the relationship query by calling the comments method and continuing to chain conditions onto the query:

```
$comment = Post::find(1)->comments()
                   ->where('title', 'foo')
                   ->first();
```

Like the hasOne method, you may also override the foreign and local keys by passing additional arguments to the hasMany method:

```
return $this->hasMany(Comment::class, 'foreign_key');
return $this->hasMany(Comment::class, 'foreign_key', 'local_key');
```

One To Many (Inverse) / Belongs To

Now that we can access all of a post's comments, let's define a relationship to allow a comment to access its parent post. To define the inverse of a hasMany relationship, define a relationship method on the child model which calls the belongsTo method.

```
class Comment extends Model
{
    * Get the post that owns the comment.
   public function post()
   {
       return $this->belongsTo(Post::class);
   }
}
```

Once the relationship has been defined, we can retrieve a comment's parent post by accessing the post "dynamic relationship property":

```
$comment = Comment::find(1);
return $comment->post->title;
```

Note : In the example above, Eloquent will attempt to find a Post model that has an id which matches the post_id column on the Comment model. Or

```
return $this->belongsTo(Post::class, 'foreign_key');
```

Note : If your parent model does not use "id" as its primary key, or you wish to find the associated model using a different column, you may pass a third argument to the belongsTo method specifying your parent table's custom key:

```
* Get the post that owns the comment.
public function post()
{
   return $this->belongsTo(Post::class, 'foreign_key', 'owner_key');
}
```

Querying Belongs To Relationships

When querying for the children of a "belongs to" relationship, you may manually build the where clause to retrieve the corresponding Eloquent models:

```
$posts = Post::where('user_id', $user->id)->get();
$posts = Post::whereBelongsTo($user)->get();
```

Has One Of Many Of Eloquent Relationship

```
 * Get the user\'s most recent order.
public function latestOrder()
{
    return $this->hasOne(Order::class)->latestOfMany();
}

 * Get the user\'s oldest order.

public function oldestOrder()
{
    return $this->hasOne(Order::class)->oldestOfMany();
}
```

Many To Many Relationships

Many-to-many relations are slightly more complicated than hasOne and hasMany relationships. An example of a many-to-many relationship is a user that has many roles and those roles are also shared by other users in the application. For example, a user may be assigned the role of "Author" and "Editor"; however, those roles may also be assigned to other users as well. So, a user has many roles and a role has many users.

Remember, since a role can belong to many users, we cannot simply place a user_id column on the roles table. This would mean that a role could only belong to a single user. In order to provide support for roles being assigned to multiple users, the role_user table is needed. We can summarize the relationship's table structure like so.

```
users
    id - integer
    name - string

roles
    id - integer
    name - string

role_user // junction/linking/associated table
    user_id - integer
    role_id - integer
```

Many-to-many relationships are defined by writing a method that returns the result of the belongsToMany method.

```
class User extends Model
{
     * The roles that belong to the user.
    public function roles()
    {
        return $this->belongsToMany(Role::class);
    }
}
```

Once the relationship is defined, you may access the user's roles using the roles dynamic relationship property:

```
$user = User::find(1); // for single/primary key data

foreach ($user->roles as $role) {
    //
}
to retrieve data ,use
$roles = User::find(1)->roles()->orderBy('name')->get();
```

To determine the table name of the relationship's intermediate table, Eloquent will join the two related model names in alphabetical order. However, you are free to override this convention. You may do so by passing a second argument to the belongsToMany method:

```
return $this->belongsToMany(Role::class, 'role_user');
```

In addition to customizing the name of the intermediate table, you may also customize the column names of the keys on the table by passing additional arguments to the belongsToMany method.

```
return $this->belongsToMany(Role::class, 'role_user', 'user_id', 'role_id');
```

Defining The Inverse Of The Relationship

To define the "inverse" of a many-to-many relationship, you should define a method on the related model which also returns the result of the belongsToMany method.

```
class Role extends Model
{
     * The users that belong to the role.
    public function users()
    {
        return $this->belongsToMany(User::class);
    }
}
```

Retrieving Intermediate Table Columns

let's assume our User model has many Role models that it is related to. After accessing this relationship, we may access the intermediate table using the pivot attribute on the models:

```
$user = User::find(1);
foreach ($user->roles as $role) {
    echo $role->pivot->created_at;
}
```

Notice that each Role model we retrieve is automatically assigned a pivot attribute. This attribute contains a model representing the intermediate table.

By default, only the model keys will be present on the pivot model. If your intermediate table contains extra attributes, you must specify them when defining the relationship:

```
return $this->belongsToMany(Role::class)->withPivot('active', 'created_by');
```

If the user has to retrieve the standard timestamps in the fields updated_at, created_at in the attribute of the pivot table which is used along with timestamps.

```
Return
$this -> belongs ToMany (Role::class)
->with Timestamps ()
->with Pivot( ‘updated_by’, ‘created_by’);
```

Filtering Queries Via Intermediate Table Columns

You can also filter the results returned by belongsToMany relationship queries using the wherePivot, wherePivotIn, wherePivotNotIn, wherePivotBetween, wherePivotNotBetween, wherePivotNull, and wherePivotNotNull methods when defining the relationship:

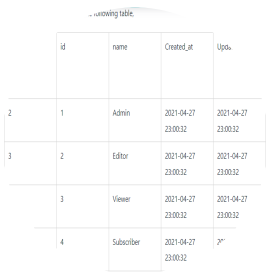
```
return $this->belongsToMany(Role::class)
                ->wherePivot('approved', 1);
```

Ordering Queries Via Intermediate Table Columns

```
return $this->belongsToMany(Badge::class)
                ->where('rank', 'gold')
                ->orderByPivot('created_at', 'desc');
```

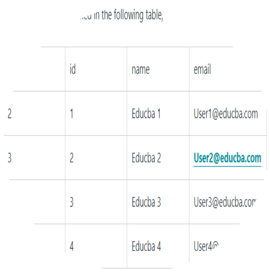
Defining Custom Intermediate Table Models

to know more about pivot (https://laravel.com/docs/9.x/eloquent-relationships#filtering-queries-via-intermediate-table-columns)


.. following table,

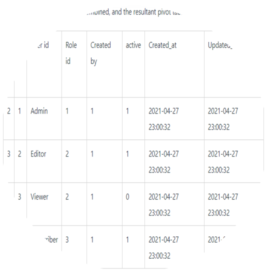
	id	name	Created_at	Upda.
2	1	Admin	2021-04-27 23:00:32	2021-04-27 23:00:32
3	2	Editor	2021-04-27 23:00:32	2021-04-27 23:00:32
	3	Viewer	2021-04-27 23:00:32	2021-04-27 23:00:32
	4	Subscriber	2021-04-27 23:00:32	20

- See The Diagram How Pivote works with the two tables. Role Table data


..u in the following table,

	id	name	email
2	1	Educba 1	User1@educba.com
3	2	Educba 2	User2@educba.com
	3	Educba 3	User3@educba.cor
	4	Educba 4	User4@

- and with The User table


..mbined, and the resultant pivot ta..

		er id	Role id	Created by	active	Created_at	Updated_
2	1	Admin	1	1	1	2021-04-27 23:00:32	2021-04-27 23:00:32
3	2	Editor	2	1	1	2021-04-27 23:00:32	2021-04-27 23:00:32
	3	Viewer	2	1	0	2021-04-27 23:00:32	2021-04-27 23:00:32
		riber	3	1	1	2021-04-27 23:00:32	2021-

- See The intermediate Table on role and user tables

Querying Relationship Existence

When retrieving model records, you may wish to limit your results based on the existence of a relationship. For example, imagine you want to retrieve all blog posts that have at least one comment. To do so, you may pass the name of the relationship to the has and orHas methods:

```
// Retrieve all posts that have at least one comment...
$posts = Post::has('comments')->get();
```

You may also specify an operator and count value to further customize the query:

```
// Retrieve all posts that have three or more comments...
$posts = Post::has('comments', '>=', 3)->get();
```

Nested has statements may be constructed using "dot" notation. For example, you may retrieve all posts that have at least one comment that has at least one image:

```
// Retrieve posts that have at least one comment with images...
$posts = Post::has('comments.images')->get();
```

If you need even more power, you may use the whereHas and orWhereHas methods to define additional query constraints on your has queries, such as inspecting the content of a comment:

```
use Illuminate\Database\Eloquent\Builder;

// Retrieve posts with at least one comment containing words like code%...
$posts = Post::whereHas('comments', function (Builder $query) {
    $query->where('content', 'like', 'code%');
})->get();

// Retrieve posts with at least ten comments containing words like code%...
$posts = Post::whereHas('comments', function (Builder $query) {
    $query->where('content', 'like', 'code%');
}, '>=', 10)->get();
```

Querying Relationship Absence

When retrieving model records, you may wish to limit your results based on the absence of a relationship. For example, imagine you want to retrieve all blog posts that don't have any comments. To do so, you may pass the name of the relationship to the doesntHave and orDoesntHave methods:

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

If you need even more power, you may use the whereDoesntHave and orWhereDoesntHave methods to add additional query constraints to your doesntHave queries, such as inspecting the content of a comment:

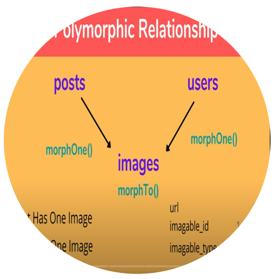
```
$posts = Post::whereDoesntHave('comments', function (Builder $query) {
    $query->where('content', 'like', 'code%');
})->get();
```

PolyMorphic Relation in Eloquent ORM

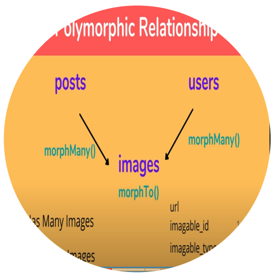
A polymorphic relationship allows the child model to belong to more than one type of model using a single association. For example, imagine you are building an application that allows users to share blog posts and videos. In such an application, a Comment model might belong to both the Post and Video models.

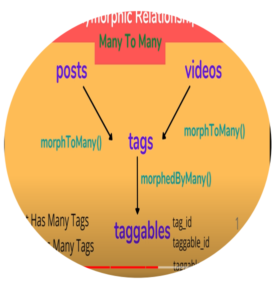
The diagram of Polymorphic Relationship

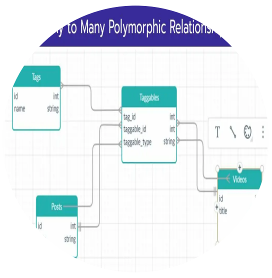
- One To One Polymorphic Relationship



- One To Many Polymorphic Relationship







Many To Many PolyMorphic Relation

A large number of Polymorphic relationships are also a little difficult to grasp. For example, if you have posts, videos, and tag tables, you'll need to connect them all to meet your needs, such as having many tags on each post and the same for videos. In addition, several tags are associated with multiple posts or videos. However, we can easily accomplish this with just one table, "taggables. To Know Details ,Click Here (https://www.codesolutionstuff.com/laravel-many-to-many-polymorphic-relationship/)

Model Structure

Eager Loading

When accessing Eloquent relationships as properties, the related models are "lazy loaded". This means the relationship data is not actually loaded until you first access the property. However, Eloquent can "eager load" relationships at the time you query the parent model.

```
<?php

class Book extends Model
{
    /**
     * Get the author that wrote the book.
     */
    public function author()
    {
        return $this->belongsTo(Author::class);
    }
}
```

```
$books = Book::all();
foreach ($books as $book) {
    echo $book->author->name;
}
```

if we have 25 books, the code above would run 26 queries: one for the original book, and 25 additional queries to retrieve the author of each book.

Thankfully, we can use eager loading to reduce this operation to just two queries. When building a query, you may specify which relationships should be eager loaded using the with method:

```
$books = Book::with('author')->get();
foreach ($books as $book) {
    echo $book->author->name;
}

To eager Load several pass an array of relationships to the with method:

$books = Book::with(['author', 'publisher'])->get();
```

Nested Eager Load

To eager load a relationship's relationships, you may use "dot" syntax. For example, let's eager load all of the book's authors and all of the author's personal contacts:

```
$books = Book::with('author.contacts')->get();

To do nested eager Load several pass an array of relationships to the with method:
$books = Book::with([
    'author' => [
        'contacts',
        'publisher',
    ],
])->get();
```

Nested Eager Loading morphTo Relationships

to know about this ,Click Here (https://laravel.com/docs/9.x/eloquent-relationships#eager-loading)

Eager Loading Specific Columns

Eloquent allows you to specify which columns of the relationship you would like to retrieve:

```
$books = Book::with('author:id,name,book_id')->get();
```

Note : When using this feature, you should always include the "id" column and any relevant foreign key columns in the list of columns you wish to retrieve.

Eager Loading By Default

```
   // The relationships that should always be loaded.

     protected $with = ['author'];

    // Get the author that wrote the book.

    public function author()
    {
        return $this->belongsTo(Author::class);
    }
```

If you would like to remove an item from the `$with` property for a single query, you may use the without method:

```
$books = Book::without('author')->get();
$books = Book::withOnly('genre')->get(); // withonly is to override with
```

Constraining Eager Loads

Sometimes you may wish to eager load a relationship but also specify additional query conditions for the eager loading query. You can accomplish this by passing an array of relationships to the with method where the array key is a relationship name and the array value is a closure that adds additional constraints to the eager loading query:

```
$users = User::with(['posts' => function ($query) {
    $query->where('title', 'like', '%code%');
    or it may use
     $query->orderBy('created_at', 'desc');
     or it may use
     $query->where('is_active',1);
}])->get();
```

Inserting & Updating Related Models for PolyMorphic

The save Method

Next, let's examine the model definitions needed to build this relationship:

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;
 # Comment Model Class
class Comment extends Model
{

    # Get the parent commentable model (post or video).

    public function commentable()
    {
        return $this->morphTo();
    }
}

# Post MOdel Class

class Post extends Model
{

     # Get all of the post's comments.

    public function comments()
    {
        return $this->morphMany(Comment::class, 'commentable');
    }
}

 # Video Model Class
class Video extends Model
{

     # Get all of the video's comments.

    public function comments()
    {
        return $this->morphMany(Comment::class, 'commentable');
    }
}
```

```
use App\Models\Comment;
use App\Models\Post;

$comment = new Comment(['message' => 'A new comment.']);

$post = Post::find(1);

$post->comments()->save($comment);
same as
$comment = $post->comments()->create([
    'message' => 'A new comment.',
]);

or Multiple data saving
$post->comments()->saveMany([
    new Comment(['message' => 'A new comment.']),
    new Comment(['message' => 'Another new comment.']),
]);
same as
$post->comments()->createMany([
    ['message' => 'A new comment.'],
    ['message' => 'Another new comment.'],
]);
```

Database: Pagination

There are several ways to paginate items. The simplest is by using the paginate method on the query builder or an Eloquent query.

Paginating Query Builder Results

The paginate method automatically takes care of setting the query's "limit" and "offset" based on the current page being viewed by the user. By default, the current page is detected by the value of the page query string argument on the HTTP request.

```
<?php

class UserController extends Controller
{
     * Show all application users.
    public function index()
    {
        return view('user.index', [
            'users' => DB::table('users')->paginate(15)
        ]);
        or
        $users = DB::table('users')->paginate(15);
        return view('user.index',compact('users'));
    }
}
```

Simple Pagination for both query builder and Eloquent

The paginate method counts the total number of records matched by the query before retrieving the records from the database. This is done so that the paginator knows how many pages of records there are in total. However, if you do not plan to show the total number of pages in your application's UI then the record count query is unnecessary.

Therefore, if you only need to display simple "Next" and "Previous" links in your application's UI, you may use the simplePaginate method to perform a single, efficient query:

```
$users = DB::table('users')->simplePaginate(15);
```

Paginating Eloquent Results

You may also paginate Eloquent queries.In that we plan to display 15 records per page. As you can see, the syntax is nearly identical to paginating query builder results:

```
$users = User::paginate(15);
```

Of course, you may call the paginate method after setting other constraints on the query, such as where clauses:

```
$users = User::where('votes', '>', 100)->paginate(15);
or
$users = User::where('votes', '>', 100)->simplePaginate(15);
or
  $employees = Employee::paginate(8);
      return view('home', compact('employees'));
```

Cursor vs. Offset Pagination

To illustrate the differences between offset pagination and cursor pagination, let's examine some example SQL queries. Both of the following queries will both display the "second page" of results for a users table ordered by id:

```
# Offset Pagination...
select * from users order by id asc limit 15 offset 15;

# Cursor Pagination...
select * from users where id > 15 order by id asc limit 15;
```

The cursor pagination query offers the following advantages over offset pagination:

For large data-sets, cursor pagination will offer better performance if the "order by" columns are indexed. This is because the "offset" clause scans through all previously matched data. For data-sets with frequent writes, offset pagination may skip records or show duplicates if results have been recently added to or deleted from the page a user is currently viewing.

Displaying Pagination Results

once you have retrieved the results, you may display the results and render the page links using Blade:

```
<div class="container">
    @foreach ($users as $user)
        {{ $user->name }}
    @endforeach
</div>

{{ $users->links() }}

or
Using the onEachSide method, you may control how many additional links are displayed on each side of the current page within the midd
 </tbody>
        </table>
        {{-- Pagination --}}
        <div class="d-flex justify-content-center">
           {{ $users->onEachSide(5)->links() }}
        </div>
    </div>
</body>
</html>
```

Note : You will notice that the links may not be aesthetically pleasing because Laravel uses Tailwind by default, while this example application uses Bootstrap.

The links method will render the links to the rest of the pages in the result set. Each of these links will already contain the proper page query string variable. Remember, the HTML generated by the links method is compatible with the Tailwind CSS framework.

Now, in your AppServiceProvider boot method, tell Paginator to use Bootstrap, as shown below.

```
public function boot()
{
    Paginator::useBootstrap();
    or
    Paginator::useBootstrapFive();
    Paginator::useBootstrapFour();
}
```

To Know More about Pagination from documentation Page (https://laravel.com/docs/9.x/pagination#displaying-pagination-results); To Watch The Works from youtube ,CLick Here (https://www.youtube.com/watch?v=LCOaWiGGptQ)

Mail Sending laravel 8 and 9 version

Step on how to send email or mail from localhost using laravel 9 apps:

Step 1 – Install Laravel 9 App.

Step 2 – Configuration SMTP in .env

Step 3 – Create Mailable Class

Step 4 – Add Email Send Route

Step 5 – Create Directory And Blade View

Step 6 – Create Email Controller

Step 7 – Run Development Server

Step 2 : In first step, you have to add send mail configuration with mail driver, mail host, mail port, mail username, mail password so laravel 8/9 will use those sender configuration for sending email. So you can simply add as like following.

```
MAIL_MAILER=smtp
MAIL_HOST=smtp.gmail.com
MAIL_PORT=465
MAIL_USERNAME=mygoogle@gmail.com
MAIL_PASSWORD=rrnnucvnqlbsl
MAIL_ENCRYPTION=tls
MAIL_FROM_ADDRESS=mygoogle@gmail.com
MAIL_FROM_NAME="${APP_NAME}"
```

Step 3 :

```
php artisan make:mail Mail/NotifyMail
```

Open the file and update the code below.

```
<?php

namespace App\Mail\Mail;

use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\SerializesModels;

class Websitemail extends Mailable
{
    use Queueable, SerializesModels;
    public $subject,$body,$pdf;
    /**
     * Create a new message instance.
     *
     * @return void
     */
    public function __construct($mail_subject,$mail_body,$mail_pdf)
    {
       $this->subject = $mail_subject;
       $this->body = $mail_body;
       $this->pdf = $mail_pdf;
    }



    /**
     * Get the attachments for the message.
     *
     * @return array
     */

    public function build()
    {
       return $this->view('mail.mail')->with([
        'subject'=>$this->subject // this is for dynamic subject in  calling the class
       ]);

       or
       return $this->subject('Test Email')->view('email.email'); // no dynamic Subject

       or
        return $this->view('mail.invoiceEmail')->attachData($this->pdf, 'customer-invoice.pdf', [
            'mime' => 'application/pdf',
        ]);
    }
}
```

Note : By whatever name you will create an email template. That you want to send. Do not forget to add an email template name in build class of the above created notifymail class.

Step 4: In next step, we will create email template named mail.blade.php inside resources/views/emails directory. That's why we have added view name email

Step 4 – Add Send Email Route

```
Route::get('send-email', [SendEmailController::class, 'index']);
```

Step 5 – Create Directory And Blade View

```
<!DOCTYPE html>
<html>
<head>
 <title>Laravel 9 Send Email Example</title>
</head>
<body>

 <h1>This is test mail from Tutsmake.com</h1>
 <p>Laravel 9 send email example</p>

</body>
</html>
```

Step 6 Use The mail in Controller.for example like following. Or With PDF , use the following.

```
use Barryvdh\DomPDF\Facade\Pdf;
$pdf = Pdf::loadView('pdf.invoice', compact('billingName', 'date', 'orderId', 'selectedProducts', 'totalPrice'));

//*MAIL SEND
 Mail::to(auth()->user()->email)->send(new Websitemail($subject,$message, $pdf->output())); // if the pdf needs to send.

 // this error handling for if failed while mailing

  if (Mail::failures()) {
           return response()->Fail('Sorry! Please try again latter');
      }else{
           return response()->success('Great! Successfully send in your mail');
         }
```

Database : Migrations

The Laravel Schema facade provides database agnostic support for creating and manipulating tables across all of Laravel's supported database systems. Typically, migrations will use this facade to create and modify database tables and columns.

You may use the make:migration Artisan command to generate a database migration. The new migration will be placed in your database/migrations directory. Each migration filename contains a timestamp that allows Laravel to determine the order of the migrations:

```
php artisan make:migration create_flights_table
```

Migration Structure

A migration class contains two methods: up and down. The up method is used to add new tables, columns, or indexes to your database, while the down method should reverse the operations performed by the up method.

```
<?php

return new class extends Migration
{
     * Run the migrations.
    public function up()
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->id();
            $table->timestamps();
        });
    }
     * Reverse the migrations.

    public function down()
    {
        Schema::drop('flights');
    }
};
```

Setting The Migration Connection

If your migration will be interacting with a database connection other than your application's default database connection, you should set the $connection property of your migration:

```

 # The database connection that should be used by the migration.

protected $connection = 'my_db'; // used to connect another database in this table
 # Run the migrations.
public function up()
{
    //
}
```

Running Migrations

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

```
php artisan migrate

# or migrating a specific table
php artisan migrate --path=/database/migrations/fileName.php

# or for refreshing after updating the file
php artisan migrate:refresh --path=/database/migrations/fileName.php
```

If you would like to see which migrations have run thus far, you may use the migrate:status Artisan command:

```
php artisan migrate:status
```

Note : this command describes how many files are already migrated and pending to be migrated.

If you would like to see the SQL statements that will be executed by the migrations without actually running them, you may provide the --pretend flag to the migrate command:

```
#first : write the commad.
php artisan make:migration create_test_table

#second : write the command to see the sql.
php artisan migrate --pretend

result of the command below
 # create table `test` (`id` bigint unsigned not null auto_increment primary key, `created_at` timestamp null, `updated_at` timestam
```

Note : this will show the SQL command of table after the making migration file, but not after the migrate.

Forcing Migrations To Run In Production

Some migration operations are destructive, which means they may cause you to lose data. In order to protect you from running these commands against your production database, you will be prompted for confirmation before the commands are executed. To force the commands to run without a prompt, use the --force flag:

```
php artisan migrate --force
```

Rolling Back Migrations

To roll back the latest migration operation, you may use the rollback Artisan command. This command rolls back the last "batch" of migrations, which may include multiple migration files:

```
php artisan migrate:rollback
```

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

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

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

```
php artisan migrate:reset
```

Roll Back & Migrate Using A Single Command

The migrate:refresh command will roll back all of your migrations and then execute the migrate command. This command effectively re-creates your entire database:

```
php artisan migrate:refresh

# Refresh the database and run all database seeds...
php artisan migrate:refresh --seed
# the following command will roll back and re-migrate the last five migrations:
php artisan migrate:refresh --step=5

# The migrate:fresh command will drop all tables from the database and then execute the migrate command:
php artisan migrate:fresh

php artisan migrate:fresh --seed
```

Database : Tables

To create a new database table, use the create method on the Schema facade. The create method accepts two arguments: the first is the name of the table, while the second is a closure which receives a "Blueprint" object that may be used to define the new table:

Dependency Object : In software engineering, dependency injection (Blueprint) is a design pattern in which an object (Blueprint) or function receives other objects ($table) or functions that it (Blueprint) depends on.

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

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

Note : When creating the table, you may use any of the schema builder's column methods to define the table's columns.

Checking For Table / Column Existence

You may check for the existence of a table or column using the hasTable and hasColumn methods:

```
if (Schema::hasTable('users')) {
    // The "users" table exists...
    Schema::create('users', function (Blueprint $table) {
    $table->id();
    $table->timestamps();
});
}

if (Schema::hasColumn('users', 'email')) {
    // The "users" table exists and has an "email" column...
}
```

Database Connection & Table Options

If you want to perform a schema operation on a database connection that is not your application's default connection, use the connection method.

```
Schema::connection('sqlite')->create('users', function (Blueprint $table) {
    $table->id();
});
```

Updating Tables

To rename an existing database table, use the rename method:

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

Schema::rename($from, $to);
# To drop an existing table, you may use the drop or dropIfExists methods:
Schema::drop('users');

Schema::dropIfExists('users');
```

Columns

The schema builder blueprint offers a variety of methods that correspond to the different types of columns you can add to your database tables. Each of the available methods are listed in the table below:

bigIncrements() ,id() and increments()

The bigIncrements , increments and id method create an auto-incrementing UNSIGNED BIGINT (primary key) equivalent column. increments and id() and method is also same as bigIncrements() with just simple difference in the following.

```
# bigIncrements() takes BIGINT(20)
$table->bigIncrements('id');
# id() methods takes INT(10)
$table->id();  # not user defined column name.
# same as
$table->increments('my_id'); # user defined int(10) column name
```

bigInteger()

The bigInteger method creates a "BIGINT" equivalent column(without primary key , auth-increments and UNSIGNED):

```
$table->bigInteger('votes');
```

boolean()

The boolean method creates a BOOLEAN equivalent column:

```
# tinyint(1) for true false checking.
$table->boolean('confirmed');
```

dateTime()

The dateTime method creates a DATETIME equivalent column with an optional precision (total digits):

```
$table->dateTime('created_at', $precision = 0);
```

date()

The date method creates a DATE equivalent column:

```
$table->date('created_at');
```

Difference Date, Datetime and timestamp

DATE : The DATE type is used for values with a date part but no time part. MySQL retrieves and displays DATE values in 'YYYY-MM-DD' format. The supported range is '1000-01-01' to '9999-12-31'.

DATETIME : The DATETIME type is used for values that contain both date and time parts. MySQL retrieves and displays DATETIME values in 'YYYY-MM-DD hh:mm:ss' format. The supported range is '1000-01-01 00:00:00' to '9999-12-31 23:59:59'.

TIMESTAMP : The TIMESTAMP data type is used for values that contain both date and time parts. TIMESTAMP has a range of '1970-01-01 00:00:01' UTC to '2038-01-19 03:14:07' UTC.

decimal()

The decimal method creates a DECIMAL equivalent column with the given precision (total digits) and scale (decimal digits):

```
$table->decimal('amount', $precision = 8, $scale = 2);
# Output decimal(8,2) means amount is 232452.50
```

NOTE :

```
[Full Documentation of Database all data types](https://zetcode.com/mysql/datatypes/)

or follow the link below.

https://www.w3resource.com/mysql/mysql-data-types.php
```

Float has 32 bit (4 bytes) with 8 places accuracy. Double has 64 bit (8 bytes) with 16 places accuracy.

enum()

The enum method creates a ENUM equivalent column with the given valid values:

```
$table->enum('difficulty', ['easy', 'hard']);
```

float()

The float method creates a FLOAT equivalent column with the given precision (total digits) and scale (decimal digits):

```
$table->float('amount', 8, 2);
```

foreignId()

The foreignId method creates an UNSIGNED BIGINT equivalent column:

```
$table->foreignId('user_id');
```

foreignIdFor()

The foreignIdFor method adds a _id UNSIGNED BIGINT equivalent column for a given model class:

```
$table->foreignIdFor(User::class);
```

constrained()

Create a foreign key constraint on this column referencing the "id" column of the conventionally related table.It will show the record list in the column of the related table.

```
$table->foreignIdFor(User::class)->constrained();
```

json()

The json method creates a JSON equivalent column:

```
$table->json('options'); # longtext data type
```

longText()

The longText method creates a LONGTEXT equivalent column:

```
$table->longText('description');
```

mediumInteger()

The mediumInteger method creates a MEDIUMINT equivalent column:

```
$table->mediumInteger('votes');
```

mediumText()

The mediumText method creates a MEDIUMTEXT equivalent column:

```
$table->mediumText('description');
```

morphs()

The morphs method is a convenience method that adds a _id UNSIGNED BIGINT equivalent column and a _type VARCHAR equivalent column.

This method is intended to be used when defining the columns necessary for a polymorphic Eloquent relationship. In the following example, taggable_id and taggable_type columns would be created:

```
$table->morphs('taggable');
```

rememberToken()

The rememberToken method creates a nullable, VARCHAR(100) equivalent column that is intended to store the current "remember me" authentication token:

```
$table->rememberToken();
```

set()

The set method creates a SET equivalent column with the given list of valid values:

```
$table->set('flavors', ['strawberry', 'vanilla']);
```

smallIncrements()

The smallIncrements method creates an auto-incrementing UNSIGNED SMALLINT equivalent column as a primary key:

```
$table->smallIncrements('id');
```

smallInteger()

The smallInteger method creates a SMALLINT equivalent column:

```
$table->smallInteger('votes');
```

softDeletes()

The softDeletes method adds a nullable deleted_at TIMESTAMP equivalent column with an optional precision (total digits). This column is intended to store the deleted_at timestamp needed for Eloquent's "soft delete" functionality:

```
$table->softDeletes($column = 'deleted_at', $precision = 0);
```

string()

The string method creates a VARCHAR equivalent column of the given length:

```
$table->string('name', 100);
```

text()

The text method creates a TEXT equivalent column:

```
$table->text('description');
```

time()

The time method creates a TIME equivalent column with an optional precision (total digits):

```
$table->time('sunrise', $precision = 0);
```

timestamp()

The timestamp method creates a TIMESTAMP equivalent column with an optional precision (total digits):

```
$table->timestamp('added_at', $precision = 0);
```

timestamps()

The timestamps method creates created_at and updated_at TIMESTAMP equivalent columns with an optional precision (total digits):

```
$table->timestamps($precision = 0);
```

tinyIncrements()

The tinyIncrements method creates an auto-incrementing UNSIGNED TINYINT equivalent column as a primary key:

```
$table->tinyIncrements('id');
```

tinyInteger()

The tinyInteger method creates a TINYINT equivalent column:

```
$table->tinyInteger('votes');
```

tinyText()

The tinyText method creates a TINYTEXT equivalent column:

```
$table->tinyText('notes');
```

unsignedBigInteger()

The unsignedBigInteger method creates an UNSIGNED BIGINT equivalent column:

```
$table->unsignedBigInteger('votes');
```

unsignedDecimal()

The unsignedDecimal method creates an UNSIGNED DECIMAL equivalent column with an optional precision (total digits) and scale (decimal digits):

```
$table->unsignedDecimal('amount', $precision = 8, $scale = 2);
```

unsignedInteger()

The unsignedInteger method creates an UNSIGNED INTEGER equivalent column:

```
$table->unsignedInteger('votes');
```

unsignedMediumInteger()

The unsignedMediumInteger method creates an UNSIGNED MEDIUMINT equivalent column:

```
$table->unsignedMediumInteger('votes');
```

unsignedSmallInteger()

The unsignedSmallInteger method creates an UNSIGNED SMALLINT equivalent column:

```
$table->unsignedSmallInteger('votes');
```

unsignedTinyInteger()

The unsignedTinyInteger method creates an UNSIGNED TINYINT equivalent column:

```
$table->unsignedTinyInteger('votes');
```

Indexes

The Laravel schema builder supports several types of indexes. The following example creates a new email column and specifies that its values should be unique. To create the index, we can chain the unique method onto the column definition:

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

You may even pass an array of columns to an index method to create a compound (or composite) index:

```
$table->index(['account_id', 'created_at']);
```

you may pass a second argument to the method to specify the index name yourself:

```
$table->unique('email', 'unique_email'); # Custom Index Name
```

to drop index

```
 $table->dropIndex(['state']);
```

Foreign Key Constraints

Laravel also provides support for creating foreign key constraints, which are used to force referential integrity at the database level. For example, let's define a user_id column on the posts table that references the id column on a users table:

```
 $table->unsignedBigInteger('user_id');
 $table->string('name');
 $table->foreign('user_id')->references('id')->on('users');
```

But in XML System

```
Post XML data using JavaScript
$.ajax({
  type: "POST",
  url: "https://reqbin.com/echo/post/xml",
  data: "",
  contentType: "text/xml",
  dataType: "text",
  success: function (result) {
    console.log(result);
  },
  error: function (result, status) {
    console.log(result);
  }
});
```

When using the foreignId method to create your column, the example above can be rewritten like so:

```
$table->foreignId('user_id')->constrained();
```

If your table name does not match Laravel's conventions, you may specify the table name by passing it as an argument to the constrained method:

```
    $table->foreignId('user_id')->constrained('users');
```

You may also specify the desired action for the "on delete" and "on update" properties of the constraint:

```
$table->foreignId('user_id')
      ->constrained()
      ->onUpdate('cascade')
      ->onDelete('cascade');
```

Dropping Foreign Keys

Foreign key constraints use the same naming convention as indexes. In other words, the foreign key constraint name is based on the name of the table and the columns in the constraint, followed by a "_foreign" suffix:

```
$table->dropForeign('posts_user_id_foreign');
```

Alternatively, you may pass an array containing the column name that holds the foreign key to the dropForeign method. The array will be converted to a foreign key constraint name using Laravel's constraint naming conventions:

```
$table->dropForeign(['user_id']);
```

Database: Seeding

It generates fake or dummy data for you and incorporate the data in the database table. With it, you can simultaneously create a single seeder or multiple seeders. You can test your laravel app with the data, which seems real to some extent.

Laravel includes the ability to seed your database with data using seed classes. All seed classes are stored in the database/seeders directory. By default, a DatabaseSeeder class is defined for you. From this class, you may use the call method to run other seed classes, allowing you to control the seeding order.

Writing Seeder

To generate a seeder, execute the make:seeder Artisan command.

```
php artisan make:seeder UserSeeder
```

As an example, let's modify the default DatabaseSeeder class and add a database insert statement to the run method:

```
class DatabaseSeeder extends Seeder
{
    public function run()
    {
        DB::table('users')->insert([
            'name' => Str::random(10),
            'email' => Str::random(10).'@gmail.com',
            'password' => Hash::make('password'),
        ]);
    }
}
```

Using Model Factories

you can use model factories to conveniently generate large amounts of database records. For example, let's create 50 users that each has one related post:

```
public function run()
{
    User::factory()
            ->count(50)
            ->hasPosts(1)
            ->create();
}
```

Calling Additional Seeders

Within the DatabaseSeeder class, you may use the call method to execute additional seed classes. The call method accepts an array of seeder classes that should be executed:

```
public function run()
{
    $this->call([
        UserSeeder::class,
        PostSeeder::class,
        CommentSeeder::class,
    ]);
}
```

Running Seeders

You may execute the db:seed Artisan command to seed your database. By default, the db:seed command runs the Database\Seeders\DatabaseSeeder class, which may in turn invoke other seed classes. However, you may use the --class option to specify a specific seeder class to run individually:

```
php artisan db:seed # all seeder files running
php artisan db:seed --class=UserSeeder # Specific single seeder file running
```

You may also seed your database using the migrate:fresh command in combination with the --seed option, which will drop all tables and re-run all of your migrations. This command is useful for completely re-building your database. The --seeder option may be used to specify a specific seeder to run:

```
php artisan migrate:fresh seed # all seeder files running
php artisan  migrate:fresh --seed --seeder=UserSeeder # Specific single seeder file running
```

To force the seeders to run without a prompt, use the --force flag:

```
php artisan db:seed --force
```

Database : Redis

- What is Redis : Redis is a NoSQL database which follows the principle of key-value store. The key-value store provides ability to store some data called a value, inside a key. You can recieve this data later only if you know the exact key used to store it.

Redis is a flexible, open-source (BSD licensed), in-memory data structure store, used as database, cache, and message broker. Redis is a NoSQL database so it facilitates users to store huge amount of data without the limit of a Relational database.

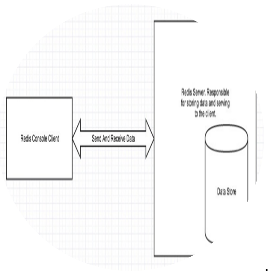
Redis supports various types of data structures like strings, hashes, lists, sets, sorted sets, bitmaps, hyperloglogs and geospatial indexes with radius queries.

- Redis Architecture : There are two main processes in Redis architecture.

1. Redis Client.
2. Redis Server.

These client and server can be on same computer or two different computers.

Redis Diagram.


Redis Console Client
Send And Receive Data
Redis Server: Responsible for storing data and serving to the client.
Data Store

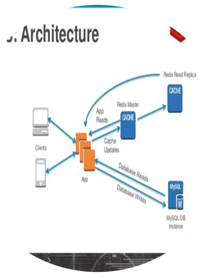


Diagram of How Redis works from cache.

Redis server is used to store data in memory . It controls all type of management and forms the main part of the architecture. You can create a Redis client or Redis console client when you install Redis application or you can use.

Features of Redis :

Following is the list of main features of Redis:

Speed: Redis stores the whole dataset in primary memory that's why it is extremely fast. It loads up to 110,000 SETs/second and 81,000 GETs/second can be retrieved in an entry level Linux box. Redis supports Pipelining of commands and facilitates you to use multiple values in a single command to speed up communication with the client libraries.

Persistence: While all the data lives in memory, changes are asynchronously saved on disk using flexible policies based on elapsed time and/or number of updates since last save. Redis supports an append-only file persistence mode. Check more on Persistence, or read the AppendOnlyFileHowto for more information.

Data Structures: Redis supports various types of data structures such as strings, hashes, sets, lists, sorted sets with range queries, bitmaps, hyperloglogs and geospatial indexes with radius queries.

Atomic Operations: Redis operations working on the different Data Types are atomic, so it is safe to set or increase a key, add and remove elements from a set, increase a counter etc.

Supported Languages: Redis supports a lot of languages such as ActionScript, C, C++, C#, Clojure, Common Lisp, D, Dart, Erlang, Go, Haskell, Haxe, Io, Java, JavaScript (Node.js), Julia, Lua, Objective-C, Perl, PHP, Pure Data, Python, R, Racket, Ruby, Rust, Scala, Smalltalk and Tcl.

Master/Slave Replication: Redis follows a very simple and fast Master/Slave replication. It takes only one line in the configuration file to set it up, and 21 seconds for a Slave to complete the initial sync of 10 MM key set on an Amazon EC2 instance.

Sharding: Redis supports sharding. It is very easy to distribute the dataset across multiple Redis instances, like other key-value store.

Portable: Redis is written in ANSI C and works in most POSIX systems like Linux, BSD, Mac OS X, Solaris, and so on. Redis is reported to compile and work under WIN32 if compiled with Cygwin, but there is no official support for Windows currently.

Redis Get and Set Methods

Get: This gets a key example blogs from the local cache and return its value. The Get method only stores strings and will return an error—the key stores hashes or others.

Set: This method sets a key example blogs to hold string values. If a key already exists, then it will override the value(s) of that key, regardless of its type, unless instructed otherwise.

we need to install the predis package in our application using composer prior to using Redis on our application. Predis is a Redis client written entirely in PHP and does not require any additional extensions. Run the following command on your terminal to install Predis:

Installing Redis :

```
composer require predis/predis
```

When the installation is complete, we can find our Redis configuration settings in config/database.php. In the file, you will see a redis array containing the Redis server. By default, the client is set to phpredis, and since predis is being used in this tutorial, the client will be changed to predis:

```
'redis' => [

  'client' => env('REDIS_CLIENT', 'predis'),

  'options' => [
      'cluster' => env('REDIS_CLUSTER', 'redis'),
      'prefix' => env('REDIS_PREFIX', Str::slug(env('APP_NAME', 'laravel'), '_').'_database_'),
  ],

  'default' => [
      'url' => env('REDIS_URL'),
      'host' => env('REDIS_HOST', '127.0.0.1'),
      'password' => env('REDIS_PASSWORD', null),
      'port' => env('REDIS_PORT', '6379'),
      'database' => env('REDIS_DB', '0'),
  ],

],
```

You can modify the default settings to your own custom settings depending on your URL and add a password, but for this tutorial, we won't be doing that.

After that is done, go ahead to register your REDIS_CLIENT in the .env file:

REDIS_CLIENT=predis Seeding I've gone ahead and created a database connection in my .env file and seeded a couple of blogs in the database.

Note: This is not a compulsory step but rather a maneuver to avoid going the long route to make a create request.

Use the command below to create a seeder. Then, add a couple of blogs, which we run with the artisan command to seed the data:

php artisan make:seeder BlogSeeder This will create a new BlogSeeder in your database folder. Next, we will add a couple of blogs:

~~~
DB::table('blogs')->insert([
          'title' => 'First blog',
          'sub_header' => 'This is the first sub header',
          'content' => 'BLOG_CONTENT'
      ]);
      ```
...
Then, we run the artisan command to seed the data to the database under the blogs table:
```bash
php artisan db:seed --class=BlogSeeder
~~~

Fetching : To fetch a blog from the database, we create a blog controller and add a function to fetch a blog using id from the database. On the first request, the blog will be fetched from the database and cached in Redis. However, on subsequent requests, the blog will be retrieved directly from Redis and formatted in JSON.

First, we will create a new controller using the following command:

```
php artisan make:controller BlogController
```

Then, we add our function to get all blogs:

```
use App\Models\Blog;
use Illuminate\Support\Facades\Redis;

public function index($id) {

  $cachedBlog = Redis::get('blog_' . $id);



  if(isset($cachedBlog)) {
      $blog = json_decode($cachedBlog, FALSE);

      return response()->json([
          'status_code' => 201,
          'message' => 'Fetched from redis',
          'data' => $blog,
      ]);
  }else {
      $blog = Blog::find($id);
      Redis::set('blog_' . $id, $blog);

      return response()->json([
          'status_code' => 201,
          'message' => 'Fetched from database',
          'data' => $blog,
      ]);
  }
}
```

In the above code, we are first checking Redis for the key with blog_ + id and returning it in JSON format if it exists. If the key has not been set, then we go to the database, get the key, and set it in Redis using blog_ + id.

Next, we create a route in routes/web.php that our API can interact with when used in Postman. This route will point to our BlogController and, specifically, our function that is fetching the blog:

```
Route::get('/blogs/{id}', 'BlogController@index');
```

- Updating : In this process, we will get a single blog and update it in the database. When this is done, we will look up the key in Redis and delete it, and then create a new key with the id of the updated blog.

Before we proceed, I commented out \App\Http\Middleware\VerifyCsrfToken::class, in the app/http/kernel.php file to disable CSRF verification. You shouldn't do this in an application ready for production, as this is necessary for the protection of the application routes. You can read more about CSRF verification and its usefulness here.

First, we create our update function in the BlogController:

```
public function update(Request $request, $id) {

  $update = Blog::findOrFail($id)->update($request->all());

  if($update) {

      // Delete blog_$id from Redis
      Redis::del('blog_' . $id);

      $blog = Blog::find($id);
      // Set a new key with the blog id
      Redis::set('blog_' . $id, $blog);

      return response()->json([
          'status_code' => 201,
          'message' => 'User updated',
          'data' => $blog,
      ]);
  }
}
```

In the above code, we got the id of the blog, updated it in the database, and then deleted the key with blog_ + id. After the key is deleted, we then fetch the blog from the database using the id and create a new key in Redis.

After this is done, we can create our route:

```
Route::post('/blogs/update/{id}', 'BlogController@update');
```

- Delete : For the last part, we will delete a blog from the database and then delete the key from Redis. First, add the delete function:

```
public function delete($id) {

  Blog::findOrFail($id)->delete();
  Redis::del('blog_' . $id);

  return response()->json([
      'status_code' => 201,
      'message' => 'Blog deleted'
  ]);
}
```

This will get the id and delete the blog from the database. Then, it will delete the key from Redis. After adding our function, we can then create a route.

```
Route::delete('/blogs/delete/{id}', 'BlogController@delete');
```

Notes : The redis crud source collected from this LINK (https://www.honeybadger.io/blog/laravel-caching-redis/)

To Learn Redis with Practical Example

Follow This steps to See the How Redis works in Laravel (https://youtu.be/3mAX8pjAtyU)

Eloquent: Collections

Defining Laravel collections:

Laravel collections can be regarded as modified versions of PHP arrays. They are located in the Illuminate\Support\Collection class directory and provide a wrapper to work with data arrays.

```
use App\Models\User;

$users = User::where('active', 1)->get();

foreach ($users as $user) {
    echo $user->name;
}
```

However, as previously mentioned, collections are much more powerful than arrays and expose a variety of map / reduce operations that may be chained using an intuitive interface. For example, we may remove all inactive models and then gather the first name for each remaining user:

```
$names = User::all()->reject(function ($user) {
    return $user->active === false;
})->map(function ($user) {
    return $user->name;
});
```

The Illuminate\Support\Collection class provides a fluent, convenient wrapper for working with arrays of data. For example, check out the following code. We'll use the collect helper to create a new collection instance from the array, run the strtoupper function on each element, and then remove all empty elements:

```
$collection = collect(['taylor', 'abigail', null])->map(function ($name) {
    return strtoupper($name);
})->reject(function ($name) {
    return empty($name);
});
```

Creating Collections

As mentioned above, the collect helper returns a new Illuminate\Support\Collection instance for the given array. So, creating a collection is as simple as:

```
$collection = collect([1, 2, 3]); # collect() is just for example function to check collection methods
by model
User::all()->toJson();
```

Available Methods

For the majority of the remaining collection documentation, we'll discuss each method available on the Collection class. Remember, all of these methods may be chained to fluently manipulate the underlying array. Furthermore, almost every method returns a new Collection instance, allowing you to preserve the original copy of the collection when necessary:

```
all
average
avg
chunk
chunkWhile
collapse
collect
combine
concat
contains
containsOneItem
containsStrict
count
countBy
crossJoin
dd
diff
diffAssoc
diffKeys
doesntContain
dump
duplicates
duplicatesStrict
each
eachSpread
every
except
filter
first
firstOrFail
firstWhere
flatMap
flatten
flip
forget
forPage
get
groupBy
has
hasAny
implode
intersect
intersectByKeys
isEmpty
isNotEmpty
join
keyBy
keys
last
lazy
macro
make
map
mapInto
mapSpread
mapToGroups
mapWithKeys
max
median
merge
mergeRecursive
min
mode
nth
only
```

```
pad
partition
pipe
pipeInto
pipeThrough
pluck
pop
prepend
pull
push
put
random
range
reduce
reduceSpread
reject
replace
replaceRecursive
reverse
search
shift
shuffle
skip
skipUntil
skipWhile
slice
sliding
sole
some
sort
sortBy
sortByDesc
sortDesc
sortKeys
sortKeysDesc
sortKeysUsing
splice
split
splitIn
sum
take
takeUntil
takeWhile
tap
times
toArray
toJson
transform
undot
union
unique
uniqueStrict
unless
unlessEmpty
unlessNotEmpty
unwrap
value
values
when
whenEmpty
whenNotEmpty
where
whereStrict
whereBetween
whereIn
whereInStrict
```

```
whereInstanceOf
whereNotBetween
whereNotIn
whereNotInStrict
whereNotNull
whereNull
wrap
zip
```

```
   $sidebar = new sidebarAdvertisement();
     dd($sidebar->all());
      $sidebar->dd();   # returns "select * from `sidebar_advertisements`"
$sidebar_ad->except([4,5]);
  $sidebar_ad->count();
```

Click Here To Know More About Collection (https://laravel.com/docs/9.x/collections)

Eloquent: Mutators, Accessors & Casting

laravel mutator and accessor also called setter and getter method. These two concepts are used when we get db value by model and save values throw modal."Accessors create a custom attribute on the object which you can access as if it were a database column." "Mutator is a way to change data when it is going set into database table.".

Attribute: Each methods of model or Column of Model Table is an attribute.

Mutators: Mutators are used to modify the value of an attribute before it is stored in the database. To define a mutator, you need to create a method with the naming convention "setAttribute". For example, if you have an attribute named "email", you can define a mutator for it like this:

```
public function setEmailAttribute($value) :  Attribute //
{
    $this->attributes['email'] = strtolower($value);
}

This mutator converts the email value to lowercase before storing it in the database.
# or

 protected function dateOfBirth(): Attribute  # date_of_birth , updated_at updatedAt , subCategory = sub_category
    {
        return new Attribute(
            get: fn ($value) =>  Carbon::parse($value)->format('m/d/Y'), # 12/05/2023
            set: fn ($value) =>  Carbon::parse($value)->format('Y-m-d'), # 2023-12-05
        );
    }

    # In Controller
     public function create()
    {
        $input = [
            'name' => 'Hardik',
            'email' => 'hardik2@gmail.com',
            'password' => bcrypt('123456'),
            'date_of_birth' => '07/21/1994'
        ];
        $user = User::create($input);
        dd($user);
    }

 # or

 class Product extends Model
{
    public function setPriceAttribute($value)
    {
        $this->attributes['price'] = $value * 100;
    }
}

$product = new Product;
$product->price = 10; # 10 * 100
$product->save();
```

Accessors: Accessors are used to retrieve the value of an attribute after it has been retrieved from the database. To define an accessor, you need to create a method with the naming convention "getAttribute". For example, if you have an attribute named "full_name", you can define an accessor for it like this:

```
public function getFullNameAttribute()
{
    return "{$this->first_name} {$this->last_name}";
}

This accessor concatenates the first_name and last_name attributes to create a full name.
 Retrieve Data.
 $user = new User();
 $user->full_name;  # full_name from getFullNameAttribute (full_name) method
```

Attribute Casting: Attribute casting allows you to define how a model attribute should be cast to and from a certain data type. This is useful when you want to ensure that the attribute is always of a certain data type, such as an integer or boolean. To define attribute casting, you need to define a protected $casts property on your model class. For example

```
protected $casts = [
    'is_admin' => 'boolean',
    'age' => 'integer',
   'created_at' => 'datetime:Y-m-d',
    'updated_at' => 'datetime',
];
```

Notes : This defines the is_admin attribute as a boolean and the age attribute as an integer. This ensures that any value stored in these attributes will be cast to the appropriate data type when retrieved from the database.

Full Example of Mutator, Accessors and Attribute Casting.

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    protected $casts = [
        'is_admin' => 'boolean',
        'age' => 'integer',
        'address' => 'array',
        'phone_number' => 'string',
    ];

    public function getFullNameAttribute()
    {
        return "{$this->first_name} {$this->last_name}";
    }



    public function setEmailAttribute($value)
    {
        $this->attributes['email'] = strtolower($value);
    }

    public function setAgeAttribute($value)
    {
        $this->attributes['age'] = $value < 0 ? 0 : $value;
    }
}

Notes : getter Method/get attribute for retrieving data from db table is called mutator.setter Method/set attribute for storing data
cast is data type by which system data is stored in column/attribute.
```

Eloquent: API Resources

- What does API Resources mean?

API Resources acts as a transformation layer that sits between our Eloquent models and the JSON responses that are actually returned by our API.

API resources present a way to easily transform our models into JSON responses.

The Steps to use API Resource

1. create a resource class

```
php artisan make:resource UserResource
```

2.

create a resource collection using either

```
php artisan make:resource Users --collection
php artisan make:resource UserCollection
```

3. Creating the book resource

Before we move on to create theBooksController, let’s create a book resource class. We’ll make use of the artisan command make:resource to generate a new book resource class. By default, resources will be placed in the app/Http/Resources directory of our application.

```
$ php artisan make:resource BookResource
```

Once that is created, let’s open it and update the toArray() method as below:

```
app/Http/Resources/BookResource.php
 public function toArray($request)
    {
      return [
        'id' => $this->id,
        'title' => $this->title,
        'description' => $this->description,
        'created_at' => (string) $this->created_at,
        'updated_at' => (string) $this->updated_at,
        'user' => $this->user,
        'ratings' => $this->ratings,
      ];
    }
```

As the name suggests, this will transform the resource into an array.

We can access the model properties directly from the $this variable.

Now we can make use of the BookResource class in our controller.

4. Creating the book controller

Let’s create the BookController. For this, we’ll make use of the API controller generation feature

```
$ php artisan make:controller BookController --api
```

Next, open it up and paste the following code into it:

```
app/Http/Controllers/BookController.php
    use App\Book;
    use App\Http\Resources\BookResource;

// ...

    public function index()
    {
      return BookResource::collection(Book::with('ratings')->paginate(25));
    }

    public function store(Request $request)
    {
      $book = Book::create([
        'user_id' => $request->user()->id,
        'title' => $request->title,
        'description' => $request->description,
        'comments' => CommentResource::collection($this->comments), # included a relationship
      ]);

      return new BookResource($book);
    }

    public function show(Book $book) # Book::findOrFail(primary_key); primary_key = id Route::get('show-book/{book}')

    {
      return new BookResource($book);
    }

    public function update(Request $request, Book $book)
    {
      // check if currently authenticated user is the owner of the book
      if ($request->user()->id !== $book->user_id) {
        return response()->json(['error' => 'You can only edit your own books.'], 403);
      }

      $book->update($request->only(['title', 'description']));

      return new BookResource($book);
    }

    public function destroy(Book $book)
    {
      $book->delete();

      return response()->json(null, 204);
    }
```

Difference between Normal Controller and API Resource:

A normal controller typically directly fetches data from the database and returns it as a JSON or HTML response. It might not follow a consistent structure and might expose internal data structures to the outside world.

On the other hand, using an API resource helps in:

1. **Data Transformation:** You can define exactly how the data should be presented, making it consistent and structured.
2. **Customization:** You can control which fields to include or exclude in the API response.
3. **Relationships:** You can easily include related data along with the main resource data.
4. **Reusability:** Resources can be reused across multiple endpoints, ensuring a consistent data format.
5. **Versioning:** It makes versioning of your API responses easier by encapsulating the data transformation logic.

In summary, API resources offer a more structured and controlled way to expose your data, especially in an API context, compared to a traditional controller approach.

Functionalities :

The *index()* method fetches and returns a list of the books that have been added.

The *store()* method creates a new book with the ID of the currently authenticated user along with the details of the book, and persists it to the database. Then we return a book resource based on the newly created book.

The *show()* method accepts a Book model (we are using route model binding here) and simply returns a book resource based on the specified book.

The *update()* method first checks to make sure the user trying to update a book is the owner of the book . If the user is not the owner of the book, we return an appropriate error message and set the HTTP status code to 403. Otherwise we update the book with the new details and return a book resource with the updated details.

Lastly, the *destroy()* method deletes a specified book from the database. Since the specified book has been deleted and no longer available, we set the HTTP status code of the response returned to 204.

ELOQUENT : Serialization

In Laravel, Eloquent is the built-in ORM (Object-Relational Mapping) that simplifies database interactions by allowing you to work with database records as if they were regular PHP objects. Eloquent provides a powerful and intuitive way to perform database operations, including retrieving, storing, updating, and deleting records.

Eloquent serialization is a feature that allows you to easily convert Eloquent model instances into different data formats, such as JSON or arrays, so that you can send them as responses in your API or use them in other parts of your application.

Let's explain Eloquent serialization step by step for beginner-level developers:

1. **Understanding Eloquent Models**:

   - In Laravel, each database table is typically associated with an Eloquent model. These models represent the structure and behavior of your data.

2. **Serializing Data**:

   - Serialization is the process of converting a complex data structure, like an Eloquent model, into a format that can be easily represented or transmitted. This format is often a JSON or array representation of the data.

3. **Eloquent Serialization Methods**:

   - Eloquent provides two primary methods for serializing data: `toArray()` and `toJson()`.
     - `toArray()`: This method converts the Eloquent model instance and its related data into an array.
     - `toJson()`: This method converts the Eloquent model instance and its related data into a JSON string.

4. **Basic Usage**:

   - Let's say you have an Eloquent model representing a "User" and you want to serialize a user's data:

```
// Fetch a user record
$user = User::find(1);

// Convert the user model to an array
$userArray = $user->toArray();

// Convert the user model to JSON
$userJson = $user->toJson();
```

   Now, `$userArray` will contain the user's data as an associative array, and `$userJson` will contain the user's data as a JSON string.

5. **Customizing Serialization**:

   - You can customize how your models are serialized by defining a `toArray()` method in your Eloquent model. This method allows you to specify which attributes should be included and how they should be formatted. For example:

```
class User extends Model
{
    public function toArray()
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'email' => $this->email,
            // Add more attributes as needed
        ];
    }
}
```

6. **Related Models and Eager Loading**:

- When you have relationships between models (e.g., a User has many Posts), Eloquent allows you to easily include related data in the serialized output using eager loading. This helps avoid the N+1 query problem. For example:

```
$userWithPosts = User::with('posts')->find(1);

// Serialize the user with their posts
$userArrayWithPosts = $userWithPosts->toArray();
```

In this example, the `posts` relationship is eager loaded, so the serialized user data will also include an array of their posts.

7. **Response Formatting**:

- You can use the serialized data to send responses in your API controllers, making it easy to return data in a consistent format.

```
public function getUserData($userId)
{
    $user = User::find($userId);
    return response()->json($user->toArray());
}
```

This code fetches a user by ID and returns their data as a JSON response.

Eloquent: Factories

In Laravel, factories are used to generate fake data for your application's database records during development and testing. They are especially useful when you need to populate your database with sample data for testing purposes or when you want to quickly create records for use in your application.

Faker all Methods and attributes Link (https://github.com/fzaninotto/Faker)

1. First, make sure you have a `Product` model created. If you don't have one, you can create it using the following command:

```
php artisan make:model Product
php artisan make:migration create_table_products

public function up()
{
    Schema::create('products', function (Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->text('description');
        $table->decimal('price', 10, 2);
        $table->integer('quantity');
        $table->string('brand');
        $table->string('manufacturer');
        $table->string('category');
        $table->string('color');
        $table->string('size');
        $table->string('weight');
        $table->string('material');
        $table->string('condition');
        $table->boolean('is_available')->default(true);
        $table->string('image_path')->nullable();
        $table->timestamps();
    });
}
```

2. Next, create a factory for the `Product` model. You can generate a factory using the following command:

```
php artisan make:factory ProductFactory
```

3. Open the generated `ProductFactory.php` file located in the `database/factories` directory and define your factory. Here's an example factory file for the `Product` model:

```
// database/factories/ProductFactory.php

use Faker\Generator as Faker;
use Illuminate\Support\Str;
use App\Models\Product;
use App\Models\User; // Assuming you have a User model for the seller

$factory->define(Product::class, function (Faker $faker) {
    return [
        'name' => $faker->sentence(3),
        'email' => $faker->unique()->safeEmail, // Generates a unique email address
        'password' => bcrypt('password'), // Sets a static password (useful for testing)
        'description' => $faker->paragraph(4),
        'price' => $faker->randomFloat(2, 10, 1000), // Example: 256.75 // random floating-point number between $10 and $1000.random
        'quantity' => $faker->numberBetween(1, 100),
        'seller_id' => User::inRandomOrder()->first()->id, // Assign a random seller from the User model
        'category' => $faker->randomElement(['Electronics', 'Clothing', 'Home & Kitchen', 'Books', 'Toys']), // Category::inRandoOrde
        'brand' => $faker->word,
        'rating' => $faker->randomFloat(1, 1, 5),
        'seller' => $faker->company,
        'image' => $faker->imageUrl(400, 400, 'product', true),
        'color' => $faker->colorName,
        'size' => $faker->randomElement(['S', 'M', 'L', 'XL']),
        'weight' => $faker->randomFloat(2, 0.1, 100),
        'material' => $faker->word,
        'condition' => $faker->randomElement(['New', 'Used', 'Featured']),
        'is_available' => $faker->boolean(90), // 90% chance of being available
        'address' => $faker->address, // Example: "123 Main St, Suite 456, New York, NY 10001"
        'email' => $faker->email, // Example: "user@example.com"
        'username' => $faker->unique()->userName, // Generates a unique username
        'created_at' => $faker->dateTime, // Example: "2023-09-15 14:30:00"
        'is_active' => $faker->boolean, // Example: true or false
        'bio' => $faker->text, // $faker->text(200) // Generates random text
        'website' => $faker->url, // Generates a random URL
        'phone_number' => $faker->phoneNumber, // Generates a random phone number
        'address' => $faker->address, // Generates a random address
        'city' => $faker->city, // Generates a random city
        'state' => $faker->state, // Generates a random state
        'zip_code' => $faker->postcode, // Generates a random postal code
        'country' => $faker->country, // Generates a random country name
        'latitude' => $faker->latitude, // Generates a random latitude value
        'longitude' => $faker->longitude, // Generates a random longitude value
        'remember_token' => str_random(10), // Generates a random 10-character string
        'created_at' => now(),
        'created_at' => $faker->dateTimeBetween('-2 years', 'now'), // Generates a random creation date.
        'updated_at' => $faker->dateTimeThisDecade,
    ];
});
```

4. To use this factory, you can use Laravel's Seeder to populate your database with sample product records. Create a seeder using the following command:

```
php artisan make:seeder ProductSeeder
```

5. Open the generated `ProductSeeder.php` file in the `database/seeders` directory and use the factory to create product records. For example:

```
// database/seeders/ProductSeeder.php

use Illuminate\Database\Seeder;
use App\Models\Product;

class ProductSeeder extends Seeder
{
    public function run()
    {
        // Create 50 product records using the ProductFactory  => Product::factory()
        Product::factory()->count(50)->create();
    }
}
```

6. Finally, run the seeder to populate your database with sample products:

```
php artisan db:seed --class=ProductSeeder
```

Now, your database will be populated with 50 sample products, each with randomly generated data, making it easier to test and develop your application. You can adjust the number of products created by changing the `count()` method in the seeder file.

Note : The Information above is enough for basic/beginner level developer. The following about factory is of advanced level studetns.

Factory Relationship

In Laravel, you can use factories to define relationships between models. Let's go through examples of three types of relationships: `hasMany`, `belongsToMany`, and `morphToMany`.

1. **hasMany Relationship**:

In a `hasMany` relationship, one model can have multiple related records in another model. For example, a `User` can have multiple `Post` records.

```
// User Factory
use App\Models\User;

$factory->define(User::class, function (Faker $faker) {
    return [
        'name' => $faker->name,
    ];
});

// Post Factory
use App\Models\Post;

$factory->define(Post::class, function (Faker $faker) {
    return [
        'title' => $faker->sentence,
        'content' => $faker->paragraph,
        'user_id' => factory(User::class)->create()->id,
    ];
});
```

In the `Post` factory, we associate each post with a user by creating a new user using the `factory(User::class)->create()` method and assigning its `id` to the `user_id` attribute of the post.

2. **belongsToMany Relationship**:

In a `belongsToMany` relationship, two models are related through an intermediate table. For example, `User` and `Role` models may have a many-to-many relationship.

```
// Role Factory
use App\Models\Role;

$factory->define(Role::class, function (Faker $faker) {
    return [
        'name' => $faker->word,
    ];
});

// User Factory (with belongsToMany relationship)
use App\Models\User;

$factory->define(User::class, function (Faker $faker) {
    return [
        'name' => $faker->name,
    ];
});

$factory->afterCreating(User::class, function ($user, $faker) {
    $user->roles()->attach(factory(Role::class, 2)->create()->pluck('id'));
});
```

In the `User` factory, we define a `belongsToMany` relationship with the `Role` model. After creating a user, we attach two random roles to the user.

3. **morphToMany Relationship**:

In a `morphToMany` relationship, a model can belong to multiple other models on a single association. For example, a `Comment` can belong to both `Post` and `Video` models.

```
// Comment Factory
use App\Models\Comment;

$factory->define(Comment::class, function (Faker $faker) {
    return [
        'content' => $faker->paragraph,
    ];
});

// Post and Video Factories (with morphToMany relationship)
use App\Models\Post;
use App\Models\Video;

$factory->define(Post::class, function (Faker $faker) {
    return [
        'title' => $faker->sentence,
    ];
});

$factory->define(Video::class, function (Faker $faker) {
    return [
        'title' => $faker->sentence,
    ];

});

$factory->afterCreating(Post::class, function ($post, $faker) {
    $post->comments()->attach(factory(Comment::class, 3)->create()->pluck('id'), ['commentable_type' => Post::class]);
});

$factory->afterCreating(Video::class, function ($video, $faker) {
    $video->comments()->attach(factory(Comment::class, 3)->create()->pluck('id'), ['commentable_type' => Video::class]);
});
```

In the above code, we define a `morphToMany` relationship between `Comment`, `Post`, and `Video`. After creating a `Post` or `Video`, we attach three random comments to them, specifying the `commentable_type` to indicate whether it's associated with a `Post` or a `Video`.

Factory Polymorphice RelationShip

Certainly! Polymorphic relationships in Laravel allow you to associate a model with multiple other models on a single association. They are typically used when you have multiple models that can belong to or associate with another model. I'll provide examples of each type of polymorphic relationship with factories.

1. Polymorphic One-to-One Relationship:

In this example, we'll have a `Comment` model that can be associated with various types of content (e.g., `Post` and `Video`).

```
// CommentFactory.php
$factory->define(App\Comment::class, function (Faker $faker) {
    return [
        'body' => $faker->sentence,
        'commentable_id' => factory(App\Post::class)->create()->id,
        'commentable_type' => 'App\Post',
    ];
});
```

2. Polymorphic One-to-Many Relationship:

Let's assume a `Tag` model that can be associated with various types of content.

```
// TagFactory.php
$factory->define(App\Tag::class, function (Faker $faker) {
    return [
        'name' => $faker->word,
    ];
});

// PostFactory.php
$factory->define(App\Post::class, function (Faker $faker) {
    return [
        'title' => $faker->sentence,
        'body' => $faker->paragraph,
    ];
});

// VideoFactory.php
$factory->define(App\Video::class, function (Faker $faker) {
    return [
        'title' => $faker->sentence,
        'url' => $faker->url,
    ];
});
```

3. Polymorphic Many-to-Many Relationship:

In this example, we'll create a `Tag` model that can be associated with both `Post` and `Video` models.

```
// TagFactory.php
$factory->define(App\Tag::class, function (Faker $faker) {
    return [
        'name' => $faker->word,
    ];
});

// PostFactory.php
$factory->define(App\Post::class, function (Faker $faker) {
    return [
        'title' => $faker->sentence,
        'body' => $faker->paragraph,
    ];
});

// VideoFactory.php
$factory->define(App\Video::class, function (Faker $faker) {
    return [
        'title' => $faker->sentence,
        'url' => $faker->url,
    ];
});

// TaggableFactory.php
$factory->define(App\Taggable::class, function (Faker $faker) {
    return [
        'tag_id' => function () {
            return factory(App\Tag::class)->create()->id;
        },
        'taggable_id' => function () {
            $taggableType = $faker->randomElement(['App\Post', 'App\Video']);
            return factory($taggableType)->create()->id;
        },
        'taggable_type' => $taggableType,
    ];
});
```

4. Polymorphic Has Many Relationship:

In this example, we'll create a `Comment` model that can have multiple comments on various types of content.

```
// CommentFactory.php
$factory->define(App\Comment::class, function (Faker $faker) {
    return [
        'body' => $faker->sentence,
    ];
});

// PostFactory.php
$factory->define(App\Post::class, function (Faker $faker) {
    return [
        'title' => $faker->sentence,
        'body' => $faker->paragraph,
    ];
});

// VideoFactory.php
$factory->define(App\Video::class, function (Faker $faker) {
    return [
        'title' => $faker->sentence,
        'url' => $faker->url,
    ];
});
```

These examples demonstrate various types of polymorphic relationships using factories in Laravel. You can use these factories to generate test data for your polymorphic relationships easily.

Architecture Concepts : Request Lifecycle, Service Container, Service Providers, Facades

Foreign id set with table in laravel

The code must be write with unsignedBigInteger('user_id')->unsigned() . like the following

```
            $table->id();
            $table->unsignedBigInteger('user_id')->unsigned();
            $table->unsignedBigInteger('news_id')->unsigned();
            $table->text('comment');

            $table->foreign('user_id')->references('id')->on('users')->onDelete('cascade');
            $table->foreign('news_id')->references('id')->on('news_posts')->onDelete('cascade');
            $table->enum('status',[1,0])->default(0);
            $table->timestamps();
```

manage a crud of a comment with database in laravel using ajax

Managing a CRUD (Create, Read, Update, Delete) system for comments in Laravel using Ajax involves setting up routes, controllers, models, and JavaScript code to handle the interactions. Below, I'll provide you with a step-by-step guide and example code snippets to achieve this. Please note that this is a high-level overview, and you might need to adjust the code according to your project's specific needs.

Step 1: Set up the database

Create a migration for the comments table:

```
php artisan make:migration create_comments_table
```

Edit the migration file to define the structure of the comments table:

```
// database/migrations/xxxx_xx_xx_create_comments_table.php

public function up()
{
    Schema::create('comments', function (Blueprint $table) {
        $table->id();
        $table->unsignedBigInteger('user_id');
        $table->unsignedBigInteger('post_id');
        $table->text('body');
        $table->timestamps();
    });
}
```

Run the migration to create the comments table:

```
php artisan migrate
```

Step 2: Create the Comment model

Generate a Comment model:

```
php artisan make:model Comment
```

Define the relationships in the Comment model:

```
// app/Models/Comment.php

class Comment extends Model
{
    protected $fillable = ['user_id', 'post_id', 'body'];

    public function user()
    {
        return $this->belongsTo(User::class);
    }

    public function post()
    {
        return $this->belongsTo(Post::class);
    }
}
```

Step 3: Set up routes

Define routes for CRUD operations in the `routes/web.php` file:

```
Route::resource('comments', CommentController::class)->middleware('auth');
```

Step 4: Create the CommentController

Generate a CommentController:

```
php artisan make:controller CommentController
```

Implement the CRUD operations in the controller:

```
// app/Http/Controllers/CommentController.php

use App\Models\Comment;
use Illuminate\Http\Request;

class CommentController extends Controller
{
    public function store(Request $request)
    {
        // Validation

        $comment = Comment::create([
            'user_id' => auth()->id(),
            'post_id' => $request->post_id,
            'body' => $request->body,
        ]);

        return response()->json($comment);
    }

    public function update(Request $request, Comment $comment)
    {
        // Validation

        $comment->update(['body' => $request->body]);

        return response()->json($comment);
    }

    public function destroy(Comment $comment)
    {
        $comment->delete();

        return response()->json(['message' => 'Comment deleted']);
    }
}
```

Step 5: Create the Blade view

Create a Blade view to display comments and handle Ajax interactions:

```
<!-- resources/views/posts/show.blade.php -->

<div id="comments">
    <!-- Display existing comments here -->
</div>

<form id="comment-form">
    @csrf
    <input type="hidden" name="post_id" value="{{ $post->id }}">
    <textarea name="body" placeholder="Add a comment"></textarea>
    <button type="submit">Submit</button>
</form>

<script>
    // Use JavaScript to handle AJAX requests and update the UI
</script>
```

Step 6: Implement JavaScript for AJAX

In your JavaScript code, you can use libraries like Axios or jQuery to send AJAX requests to the server for creating, updating, and deleting comments. Update the `script` tag in the Blade view to include your AJAX logic.

Here's a simplified example using Axios:

```
<!-- resources/views/posts/show.blade.php -->

<!-- ... existing HTML ... -->

<script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
<script>
    const commentForm = document.getElementById('comment-form');
    const commentsContainer = document.getElementById('comments');

    commentForm.addEventListener('submit', async (event) => {
        event.preventDefault();

        const formData = new FormData(commentForm);
        const response = await axios.post('/comments', formData);

        // Update the UI with the new comment
        const newComment = response.data;
        const commentElement = document.createElement('div');
        commentElement.innerHTML = `<p>${newComment.body}</p>`;
        commentsContainer.appendChild(commentElement);

        commentForm.reset();
    });

    // Implement similar logic for updating and deleting comments using AJAX
</script>
```

CRUD system for comments in Laravel using

Managing a CRUD (Create, Read, Update, Delete) system for comments in Laravel using AJAX involves several components, including routes, controllers, models, views, and JavaScript. Below, I'll provide you with a step-by-step guide along with code examples for each component.

Assuming you have a model named `Comment` to represent the comments and a table named `comments` in your database.

1. **Routes (routes/web.php):**

```
Route::get('/comments', 'CommentController@index'); // To fetch all comments
Route::post('/comments', 'CommentController@store'); // To create a new comment
Route::put('/comments/{id}', 'CommentController@update'); // To update a comment
Route::delete('/comments/{id}', 'CommentController@destroy'); // To delete a comment
```

2. **Controller (app/Http/Controllers/CommentController.php):**

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Models\Comment;

class CommentController extends Controller
{
    public function index()
    {
        $comments = Comment::all();
        return response()->json($comments);
    }

    public function store(Request $request)
    {
        $comment = new Comment();
        $comment->content = $request->input('content');
        $comment->save();

        return response()->json($comment);
    }

    public function update(Request $request, $id)
    {
        $comment = Comment::findOrFail($id);
        $comment->content = $request->input('content');
        $comment->save();

        return response()->json($comment);
    }

    public function destroy($id)
    {
        $comment = Comment::findOrFail($id);
        $comment->delete();

        return response()->json(['message' => 'Comment deleted']);
    }
}
```

3. **Blade View (resources/views/comments.blade.php):**

```
<!DOCTYPE html>
<html>
<head>
    <title>Comments</title>
</head>
<body>
    <div>
        <h1>Comments</h1>
        <form id="addCommentForm">
            <input type="text" id="commentContent" placeholder="Add a comment">
            <button type="submit">Add Comment</button>
        </form>
    </div>
    <ul id="commentList">
        <!-- Comments will be displayed here dynamically -->
    </ul>

    <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
    <script>
        $(document).ready(function () {
            // Fetch and display comments
            function fetchComments() {
                $.ajax({
                    url: '/comments',
                    type: 'GET',
                    success: function (data) {
                        var commentsHtml = '';
                        data.forEach(function (comment) {
                            commentsHtml += '<li>' + comment.content +
                                            ' <button class="deleteBtn" data-id="' + comment.id + '">Delete</button></li>';
                        });
                        $('#commentList').html(commentsHtml);
                    }
                });
            }

            fetchComments(); // Initial fetch

            // Add a new comment
            $('#addCommentForm').submit(function (e) {
                e.preventDefault();
                var content = $('#commentContent').val();

                $.ajax({
                    url: '/comments',
                    type: 'POST',
                    data: { content: content },
                    success: function () {
                        $('#commentContent').val('');
                        fetchComments(); // Fetch and update the list
                    }
                });
            });

            // Delete a comment
            $(document).on('click', '.deleteBtn', function () {
                var id = $(this).data('id');

                $.ajax({
                    url: '/comments/' + id,
                    type: 'DELETE',
                    success: function () {
                        fetchComments(); // Fetch and update the list
                    }
                });
            });
        });
    </script>
</body>
</html>
```

This example provides you with a basic implementation of a comment CRUD system in Laravel using AJAX. The provided code assumes that you have already set up the necessary database migrations, models, and have included jQuery for AJAX functionality. Remember that this is a simplified example, and in a real-world scenario, you might want to implement additional validation, error handling, and security measures.

Laravel Set Notification with Read Unread Methods

Setting up user review notifications and implementing read and unread notifications functionality in the admin dashboard of a Laravel application involves several steps. I'll guide you through the process and provide code examples along the way.

Step 1: Create Notification

First, let's create a notification for user reviews. Run the following command to generate a new notification:

```
php artisan make:notification UserReviewNotification
```

This will create a new notification class at `app/Notifications/UserReviewNotification.php`.

In this notification class, you can customize the content of the notification message. For example:

```
use Illuminate\Notifications\Notification;
use Illuminate\Notifications\Messages\MailMessage;

class UserReviewNotification extends Notification
{
    public function toMail($notifiable)
    {
        return (new MailMessage)
            ->line('A new review has been posted.')
            ->action('View Review', url('/reviews/' . $this->reviewId))
            ->line('Thank you for using our service!');
    }
}
```

Step 2: Trigger Notification

When a new review is posted, you'll need to trigger the notification. This might be done in your review controller or wherever reviews are stored. For instance:

```
use App\Notifications\UserReviewNotification;

// After saving a new review
$user->notify(new UserReviewNotification($reviewId));
```

Step 3: Admin Dashboard

Now, let's set up the admin dashboard to display both read and unread notifications.

Assuming you have a route and a controller method for your admin dashboard, you can fetch the notifications like this:

```
use Illuminate\Support\Facades\Auth;

public function adminDashboard()
{
    $user = Auth::user(); // Assuming the admin is logged in
    $unreadNotifications = $user->unreadNotifications;
    $readNotifications = $user->readNotifications;

    return view('admin.dashboard', compact('unreadNotifications', 'readNotifications'));
}
```

Step 4: Display Notifications in the View

In your admin dashboard view, you can iterate through the notifications to display them:

```
<!-- Unread notifications -->
@foreach ($unreadNotifications as $notification)
    <div class="notification unread">
        {!! $notification->data['message'] !!}
        <a href="{{ route('mark.notification.read', ['id' => $notification->id]) }}">Mark as Read</a>
    </div>
@endforeach

<!-- Read notifications -->
@foreach ($readNotifications as $notification)
    <div class="notification read">
        {!! $notification->data['message'] !!}
    </div>
@endforeach
```

Step 5: Mark Notification as Read

Create a route and controller method to mark a notification as read:

```
public function markNotificationAsRead($id)
{
    $notification = auth()->user()->notifications()->where('id', $id)->first();

    if ($notification) {
        $notification->markAsRead();
    }

    return redirect()->back();
}
```

Step 6: Styling and AJAX (Optional)

You can enhance the user experience by using AJAX to mark notifications as read without refreshing the page and by styling the notifications to fit your application's design.

Remember that this is a high-level guide, and you might need to adapt the code to fit your specific application structure and requirements.

Mail and Databse Notification sent togather

```php
<?php

namespace App\Notifications;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Notifications\Messages\MailMessage;
use Illuminate\Notifications\Notification;

class AppraisalGoalPublish extends Notification implements ShouldQueue
{
    use Queueable;

    private $sender;
    private $reviewPeriod;
    private $name;

    /**
     * Create a new notification instance.
     *
     * @return void
     */
    public function __construct($sender, $reviewPeriod)
    {
        $this->sender = $sender;
        $this->reviewPeriod = $reviewPeriod;
        $this->name = $this->sender->first_name.' '.$this->sender->last_name;
    }

    /**
     * Get the notification's delivery channels.
     *
     * @param  mixed  $notifiable
     * @return array
     */
    public function via($notifiable)
    {
        return ['mail', 'database'];
    }

    /**
     * Get the mail representation of the notification.
     *
     * @param  mixed  $notifiable
     * @return \Illuminate\Notifications\Messages\MailMessage
     */
    public function toMail($notifiable)
    {
        return (new MailMessage)->view(
            'your.view.path', ['name' => $this->name, 'reviewPeriod' => $this->reviewPeriod]
        );
    }

    /**
     * Get the array representation of the notification.
     *
     * @param  mixed  $notifiable
     * @return array
     */
    public function toDatabase($notifiable)
    {
        return [
            'sender' => $this->sender->id,
            'receiver' => $notifiable->id,
            'message' => 'The employee with the code $userCode and name ' . $this->name . ' has published his/her goals for the revi
        ];
    }
}
```

- in controller file your can handle by following

```
$details = [

    'full_name' => $FirstName. ' ' . $LastName
];

$user->notify(new \App\Notifications\AppraisalGoalPublish($details));

# OR
 $user = User::where('role','admin')->get(); // to send to admin dashboard only.

Notification::send($user,new NewCommentNotification($details));

// Here , Custom Notification objects wil be enclosed of how many arguments is declared in the constructor   __construct($sender, $re
```

- Send The data in Blade file via the following method

```
    public function toArray($notifiable)  //  this mehod can be **toDatabase($notifiable)**
    {
        return [
            'message'=>"New Comment added on the News",
            // 'reviewText'=>$this->ReviewInfo,
        ];
    }
```

- Use in Blade file

```
# unReadNotifcation

     @php
$reviewcount = Auth::user()->unreadNotifications()->count()
     @endphp

# all UnReadNotification will  be looped here untill click or Read them.

but if want to delete from notice box after read of click the notification , must use the loop like this

   @forelse ($user->unreadNotifications as $item)
        @php
            $item->markAsRead();
        @endphp

@php
$user = Auth::user();
@endphp

         # all notification will  be looped here.

    @forelse($user->notifications as $notifiaction)
        <!-- item-->
        <a href="{{ route('pending.review') }}" class="dropdown-item notify-item">
            <div class="notify-icon bg-secondary">
                <i class="mdi mdi-heart"></i>
            </div>
            <p class="notify-details">{{ $notifiaction->data['message'] }}
                <b>Admin</b>
                <small class="text-muted"> {{ Carbon\Carbon::parse($notifiaction->created_at)->diffForHumans() }} </small>
            </p>
        </a>
     @empty

    @endforelse

    </div>

 **Note** :  {{ $notifiaction->data['message'] }}

  above, data is column name of notification table and message of json text of data column field that means the message key of this
    return [
            'message'=>"New Comment added on the News",
            // 'reviewText'=>$this->ReviewInfo,
        ];
```

all blade directives in laravel using with example

Blade is the templating engine used in Laravel, a popular PHP framework. Blade provides an elegant syntax for creating views and templates in Laravel applications. Here are some commonly used Blade directives with examples:

1. **{{ }}**: The double curly braces are used to escape and output content as HTML entities. This prevents cross-site scripting (XSS) attacks.

   Example:

   ```
   <p>Hello, {{ $name }}</p>
   ```

2. **{!! !!}**: The double curly braces with an exclamation mark allow you to output content as raw HTML. Use this directive with caution, as it doesn't automatically escape the content and may expose your application to XSS attacks.

   Example:

   ```
   <p>{!! $rawHtml !!}</p>
   ```

3. **@if, @else, @elseif, @endif**: These directives are used for conditional statements in Blade templates.

Example:

```
@if($isAdmin)
    <p>Welcome, Admin!</p>
@else
    <p>Welcome, Guest!</p>
@endif
```

4. **@unless**: This directive is used to perform the opposite of an `@if` condition. It executes the enclosed code block unless the condition is true.

   Example:

```
@unless($loggedIn)
    <p>Please log in to continue.</p>
@endunless
```

5. **@for, @foreach, @while**: These directives are used for looping through data.

   Example using `@foreach`:

```
<ul>
    @foreach($items as $item)
        <li>{{ $item }}</li>
    @endforeach
</ul>
```

6. **@forelse, @empty**: The `@forelse` directive works like `@foreach`, but it has a special `@empty` directive block that is executed if the loop is empty.

   Example:

```
<ul>
    @forelse($items as $item)
        <li>{{ $item }}</li>
    @empty
        <li>No items available.</li>
    @endforelse
</ul>
```

7. **@include**: This directive is used to include subviews or partials within your main views.

   Example:

```
@include('partials.header')
```

8. **@yield, @section, @show, @extends**: These directives are used for creating and extending layouts.

   Example using `@yield` and `@section`:

```
<!-- layout.blade.php -->
<html>
    <head>
        <title>@yield('title')</title>
    </head>
    <body>
        @yield('content')
    </body>
</html>

<!-- page.blade.php -->
@extends('layouts.layout')

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

@section('content')
    <p>This is the page content.</p>
@endsection
```

9. **@push, @stack**: These directives are used to push content onto a stack within a layout, allowing you to insert scripts or stylesheets at specific points.

   Example using `@push` and `@stack`:

```
<!-- layout.blade.php -->
<html>
    <head>
        @stack('styles')
    </head>
    <body>
        @yield('content')
        @stack('scripts')
    </body>
</html>

<!-- page.blade.php -->
@extends('layouts.layout')

@push('styles')
    <link rel="stylesheet" href="styles.css">
@endpush

@push('scripts')
    <script src="script.js"></script>
@endpush
```

10. **@guest and @auth** Blade contains authentication directives that could be used to determine if the current user is authenticated or not.

```
@guest
    // The user is not authenticated...
@endguest
@auth
    // The user is authenticated...
@endauth
```

These are some of the most commonly used Blade directives in Laravel. They allow you to build dynamic and interactive views for your application.

11. **@forelse**The forelse directive is a special kind of loop. It is a foreach directive combined with the empty directive. Take a look at the following example:

```
@if ($blogs->count())
  @foreach ($blogs as $blog)
    <li>{{ $blog->title }}</li>
  @endforeach
@else
  <p>There are no blogs.</p>
@endif
```

12.**@inject**, The @inject directive is one of my faviourite directive and used on most places in my applications. The @inject directive used to retrieve a service from the Laravel’s Service Container and inject into your view.

The first argument passed to this directive is a variable name the service will be placed into, while the second argument is the class or interface of the service you want to inject.

```
@inject('menu', 'App\Services\MenuService')

// then in your view

{!! $menu->render() !!}
```

Custom Blade Directive with Making Service Provider

Sure, I can provide you with an example of how to create a custom service provider to register custom Blade directives and then use those directives in a Blade template.

Step 1: Create the Custom Service Provider Let's create a custom service provider called `CustomBladeDirectivesServiceProvider` where we'll define our custom Blade directives.

1. Create the service provider: Run the following command to create a new service provider:

```
php artisan make:provider CustomBladeDirectivesServiceProvider
```

2. Open the generated `CustomBladeDirectivesServiceProvider.php` file located in the `app/Providers` directory. Inside the `boot` method, you can define your custom Blade directives:

```
namespace App\Providers;

use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;

class CustomBladeDirectivesServiceProvider extends ServiceProvider
{
    public function boot()
    {
        Blade::directive('datetime', function ($expression) {
            return "<?php echo \Carbon\Carbon::parse($expression)->format('Y-m-d H:i:s'); ?>";
        });


        Blade::directive('uppercase', function ($expression) {
            return "<?php echo strtoupper($expression); ?>";
        });


# @ifenv: Conditional check based on the environment.
        Blade::directive('ifenv', function ($expression) {
    return "<?php if(app()->environment($expression)): ?>";
});

Blade::directive('endifenv', function () {
    return "<?php endif; ?>";
});

# @ifenv: Conditional check based on the environment. End



# @repeat: Repeats the content within the directive a certain number of times.
Blade::directive('repeat', function ($expression) {
    list($count, $content) = explode(',', $expression, 2);
    return "<?php echo str_repeat($content, $count); ?>";
});

# @datetime is alternative of carbon datetitme
 Blade::directive('datetime', function ($expression) {
        return "<?php echo with($expression)->format('Y-m-d H:i:s'); ?>";
    });

# @isadmin directive Start
    Blade::directive('isadmin', function () {
        return "<?php if(auth()->user() && auth()->user()->isAdmin()): ?>";
    });

    Blade::directive('endisadmin', function () {
        return "<?php endif; ?>";
    });

    # @isadmin directive Start

    }

    public function register()
    {
        //
    }
}
```

Step 2: Register the Service Provider Add your custom service provider to the `providers` array in the `config/app.php` configuration file:

```
'providers' => [
    // ...
    App\Providers\CustomBladeDirectivesServiceProvider::class,
],
```

Step 3: Using the Custom Blade Directives in a View Now you can use the custom Blade directives in your Blade templates.

Create a new Blade view file, for example, `custom.blade.php`, and add the following content:

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

@section('content')
    <p>Current date and time: @datetime(now())</p>
    <p>Uppercase text: @uppercase('hello world')</p>
@endsection

# @ifenv: Conditional check based on the environment.
@ifenv('local')
    <p>This is a local environment.</p>
@endifenv



# @repeat: Repeats the content within the directive a certain number of times.
@repeat(3, '<p>Repeated content</p>')



<p>Current date and time: @datetime(now())</p>

@isadmin
    <p>You are an admin user.</p>
@endisadmin
```

In this example, the `@datetime` directive formats the current date and time, and the `@uppercase` directive converts the provided text to uppercase.

Remember to adjust the layout and include the `@yield('content')` directive in your layout file (`layouts/app.blade.php` in this example).

After creating the view and layout, you can visit the route associated with the view to see the custom Blade directives in action.

That's it! You've created a custom service provider to register custom Blade directives and used those directives in a Blade template within the Laravel framework.

Page title setup dynamically

```
@yield('title');  # in home Page

@section('title', $news->news_title) # in different Pages
```

Validation In Laravel

Create Public Method for both of crating and updating data. in the following the concenated id ($requestForm->id)

```
 public function PermissionValidation($requestForm)
    {
      return   $requestForm->validate([
            'name'=>'required|unique:permissions,name,' . $requestForm->id,
            'group_name'=>'required',
        ],[
            'name.required'=> 'permission name must be filled up',
            'name.unique'=>'The permission Name already Exist, Try with another name'
        ],
    );
    }

    # Or separate ID parameter

     public function PermissionValidation($requestForm, $id == null)
    {
      return   $requestForm->validate([
            'name'=>'required|unique:permissions,name,' . $id,
            'group_name'=>'required',
        ],[
            'name.required'=> 'permission name must be filled up',
            'name.unique'=>'The permission Name already Exist, Try with another name'
        ],
    );
    }

    #or alternative system. We Know that one column validation rules can be declared in array ['required','email',Rule::unique('users

    $requestForm->validate(
        ['email' => [
                'required',
                'email',
                Rule::unique('users', 'email')->ignore($id),
            ],
            ]
            );
```

In Blade File to show the error

```
<div class="form-group col-md-6 mb-3">
                <label for="inputEmail4" class="form-label">Roles Name </label>
                <input type="text" value={{ old('name') }} name="name" class="form-control" id="inputEmail4" placeholder="Add Roles">
                @error('name')
                <div class="alert alert-danger w-100 text-center bg-white m-1">{{ $message }}</div>
            @enderror
            </div>
```

how to checked related checkbox by clicking a single checkbox and not checked all by one in foreach loop in laravel.

It seems like you want to achieve a "check all" functionality for the related checkboxes within a specific group, rather than checking all checkboxes across all groups. To implement this, you need to make sure that each group checkbox is responsible for toggling only the checkboxes within its own group. Here's how you can achieve this in Laravel using jQuery:

Assuming you have a foreach loop in your Laravel Blade view that generates group checkboxes and associated permission checkboxes:

```
@foreach($groups as $group)
    <input type="checkbox" class="group_name" data-group-id="{{ $group->id }}">
    <label>{{ $group->name }}</label>

    @foreach($group->permissions as $permission)
        <input type="checkbox" class="permission_checkbox group_{{ $group->id }}" name="permission[]" value="{{ $permission->id }}">
        <label>{{ $permission->name }}</label>
    @endforeach
@endforeach
```

In the above code, we've added the `data-group-id` attribute to the group checkbox and used a `group_{{ $group->id }}` class for the permission checkboxes associated with each group.

Now, you can update your jQuery script to handle the "check all" functionality within each group:

```
$('.group_name').on('click', function() {
    var groupId = $(this).data('group-id');
    var permissionCheckboxes = $('.permission_checkbox.group_' + groupId);

    permissionCheckboxes.prop('checked', $(this).is(':checked'));
});

# To select all combobox by one click , another example in the following

  $('#customckeck15').click(function(){
            if ($(this).is(':checked')) {
                $('input[type = checkbox]').prop('checked',true);
            }else{
                 $('input[type = checkbox]').prop('checked',false);
            }
         });
```

This script listens for the click event on each `.group_name` checkbox. When a group checkbox is clicked, it extracts the `data-group-id` attribute to determine the associated group's ID. Then, it selects all permission checkboxes with the class `.permission_checkbox` and the specific group class (e.g., `.group_1`, `.group_2`, etc.) and sets their checked status to match the checked status of the group checkbox.

With this setup, when you click a group checkbox, it will toggle the related permission checkboxes within that specific group, without affecting other groups' checkboxes.

Laravel Database data access from a table and related data based on parent data.

```
Suppose Permissions\'s Group Name and Permission name by Group Name.

      @php
          $idkey = 0;
          $forkey=0;
      @endphp

      @foreach ($role_has_permissions_group as $groupkey =>  $permissions_group_name)


         <div class="row">
            <div class="col-3">
                <div class="form-check mb-2 form-check-primary">
                    <input class="form-check-input group_name" data-group-id="{{ $groupkey }}" type="checkbox" value="" id="customck
                    <label class="form-check-label" for="customckeck{{ $forkey++ }}">
                        {{ $permissions_group_name->group_name }}
                    </label>
                </div>
            </div>

            @php
                $group_wise_permission_name = App\Models\User::GroupByPermissionName($permissions_group_name->group_name);
            @endphp

            <div class="col-9">
                @foreach ($group_wise_permission_name as $key => $permission_name)


                <div class="form-check mb-2 form-check-primary">
                    <input class="form-check-input permission_checkbox group_{{ $groupkey }}" name="permission[]" type="checkbox" va
                    <label class="form-check-label" for="customckeck{{ $permission_name->id }}">

                        {{ $permission_name->name }}

                    </label>
                </div>

                @endforeach

                <br>

            </div>

        </div>

        @endforeach
```

how to Redirect to role wise dashboard after a different login page and back to their own login page after the session close in the Laravel.

To achieve role-based redirection in Laravel after login and returning to the respective login page after session closure, you can follow these steps:

1. **Set Up Multiple Login Routes and Redirects:** In your `routes/web.php` file, define different login routes for each user role. Also, specify the routes to redirect users to after successful login.

```
Route::group(['middleware' => 'guest'], function () {
    // Admin login route
    Route::get('/admin/login', 'Auth\AdminLoginController@showLoginForm');
    Route::post('/admin/login', 'Auth\AdminLoginController@login');

    // User login route
    Route::get('/user/login', 'Auth\UserLoginController@showLoginForm');
    Route::post('/user/login', 'Auth\UserLoginController@login');
});
```

2. **Create Custom Login Controllers:** Create separate controller classes for each user role. Make sure these controllers extend Laravel's built-in `App\Http\Controllers\Auth\LoginController` and override the `$redirectTo` property.

```
namespace App\Http\Controllers\Auth;

use App\Http\Controllers\Controller;
use Illuminate\Foundation\Auth\AuthenticatesUsers;

class AdminLoginController extends Controller
{
    use AuthenticatesUsers;

    protected $redirectTo = '/admin/dashboard'; // Redirect after login

    public function __construct()
    {
        $this->middleware('guest')->except('logout');
    }
}

class UserLoginController extends Controller
{
    use AuthenticatesUsers;

    protected $redirectTo = '/user/dashboard'; // Redirect after login

    public function __construct()
    {
        $this->middleware('guest')->except('logout');
    }
}
```

3. **Implement Custom Logout Functionality:** Customize the logout functionality to redirect users back to their respective login pages after session closure. You can do this by overriding the `logout` method in your `LoginController`.

```
use Illuminate\Support\Facades\Auth;

class LoginController extends Controller
{
    // Existing code

    public function logout(Request $request)
    {
        $redirectTo = '/'; // Default redirect

        if (Auth::guard('admin')->check()) {
            $redirectTo = '/admin/login'; // Admin logout
        } elseif (Auth::guard('web')->check()) {
            $redirectTo = '/user/login'; // User logout
        }

        $this->guard()->logout();
        $request->session()->invalidate();

        return redirect($redirectTo);
    }
}
```

4. **Update Middleware for Dashboard Access:** Use middleware to ensure that only authenticated users with the appropriate roles can access their respective dashboards.

```
Route::middleware(['auth:admin', 'role:admin'])->group(function () {
    Route::get('/admin/dashboard', 'AdminController@dashboard');
});

Route::middleware(['auth', 'role:user'])->group(function () {
    Route::get('/user/dashboard', 'UserController@dashboard');
});
```

5. **Implement Role Middleware:** Create a custom middleware to check the user's role and grant or deny access accordingly. You can create a middleware using the `php artisan make:middleware RoleMiddleware` command.

```
public function handle($request, Closure $next, $role)
{
    if ($role === 'admin' && Auth::guard('admin')->check()) {
        return $next($request);
    } elseif ($role === 'user' && Auth::guard('web')->check()) {
        return $next($request);
    }

    abort(403, 'Unauthorized');
}
```

Remember to adjust namespaces and class names according to your application's structure. This example assumes you have separate guards for admin and user (you might need to define these guards in your `config/auth.php` file). Also, don't forget to implement the actual dashboard controller actions.

Always make sure to test thoroughly to ensure everything works as intended.

how to Redirect to role wise dashboard after a different login page and back to their own login page after the session close in the Laravel breeze package.

To achieve role-based redirection after a different login page and returning to their own login page after the session closes using the Laravel Breeze package, you can follow these steps:

1. **Set Up Role-based Redirection:**

   In Laravel Breeze, you can customize the redirection logic in the `Authenticate.php` middleware. This middleware handles the redirection after successful login. To redirect users based on their roles, you can modify the `handle` method in `app/Http/Middleware/Authenticate.php`:

```
public function handle($request, Closure $next, ...$guards)
{
    $guards = empty($guards) ? [null] : $guards;

    foreach ($guards as $guard) {
        if (Auth::guard($guard)->check()) {
            $user = Auth::guard($guard)->user();
            if ($user->hasRole('admin')) {
                return redirect('/admin/dashboard'); // Redirect to admin dashboard
            } elseif ($user->hasRole('user')) {
                return redirect('/user/dashboard'); // Redirect to user dashboard
            }
        }
    }

    return $next($request);
}
```

This code snippet assumes that you have a `hasRole` method in your User model to check the user's role.

2. **Return to Specific Login Page:**

   To redirect users back to their respective login pages after the session closes, you can customize the login path in the `LoginController.php` of Laravel Breeze:

```
protected function loggedOut(Request $request)
{
    return redirect()->route('login'); // Redirect to the general login page
}
```

In this example, it redirects users to the general login page. You can modify this method to redirect users to different login pages based on their roles.

Remember to adjust the route names, URLs, and role-checking logic to match your application's structure and requirements.

3. **Session Timeout Handling:**

   To handle the session timeout, you might need to customize the behavior in Laravel's built-in authentication system. Specifically, you can modify the `App\Http\Middleware\AuthenticateSession.php` middleware to achieve this.

```
protected function redirectTo($request)
{
    if (! $request->expectsJson()) {
        if (Auth::check()) {
            $user = Auth::user();
            if ($user->hasRole('admin')) {
                return route('admin.login'); // Redirect admin to admin login
            } elseif ($user->hasRole('user')) {
                return route('user.login'); // Redirect user to user login
            }
        }
    }
}
```

This code will redirect users to their respective login pages when their session times out.

Please make sure to adjust the code according to your specific routes, roles, and folder structures. It's also important to ensure that your authentication system and role management are properly set up before implementing these changes.

how to controll admin panel base on role and permission in laravel?

Controlling the admin panel based on roles and permissions in Laravel involves a multi-step process. You can utilize packages like Spatie Laravel Permissions and create middleware to handle role and permission checks. Here's a step-by-step guide:

1. **Install and Configure Spatie Laravel Permissions:** Install the package using Composer:

```
composer require spatie/laravel-permission
```

After installation, run the migration to create the necessary tables:

```
php artisan migrate
```

In your `config/auth.php` file, set the user provider to use the `eloquent` provider.

```
'providers' => [
    'users' => [
        'driver' => 'eloquent',
        'model' => App\Models\User::class,
    ],
],
```

2. **Define Roles and Permissions:** In your `User` model, use the `HasRoles` and `HasPermissions` traits provided by the package:

```
use Spatie\Permission\Traits\HasRoles;
use Spatie\Permission\Traits\HasPermissions;

class User extends Authenticatable
{
    use HasRoles, HasPermissions;

    // ...
}
```

Define roles and permissions in your application. You can do this in a seeder or directly in a migration:

```
use Spatie\Permission\Models\Role;
use Spatie\Permission\Models\Permission;

// Create roles
$adminRole = Role::create(['name' => 'admin']);
$userRole = Role::create(['name' => 'user']);

// Create permissions
$manageUsers = Permission::create(['name' => 'manage users']);
$managePosts = Permission::create(['name' => 'manage posts']);

// Assign permissions to roles
$adminRole->givePermissionTo([$manageUsers, $managePosts]);
```

3. **Create Middleware for Role and Permission Checks:** Create a middleware to check for roles and permissions. This middleware can be used to protect routes and actions that require specific roles or permissions.

```
php artisan make:middleware CheckRolePermission
```

In the `handle` method of the middleware, perform the role and permission checks:

```
public function handle($request, Closure $next, ...$rolesAndPermissions)
{
    $user = Auth::user();

    if (! $user->hasAnyRole($rolesAndPermissions) && ! $user->hasAnyPermission($rolesAndPermissions)) {
        abort(403, 'Unauthorized action.');
    }

    return $next($request);
}
```

4. **Apply Middleware to Routes:** Apply the custom middleware to routes that require specific roles or permissions.

```
Route::middleware(['auth', 'role-permission:admin,manage users'])->group(function () {
    // Your admin panel routes here
});
```

5. **Use Middleware in Controllers:** In your controller methods, use the middleware to protect specific actions:

```
public function manageUsers()
{
    // This action is only accessible to users with the "admin" role or the "manage users" permission
}
```

By following these steps, you can control access to your admin panel based on roles and permissions using the Spatie Laravel Permissions package. Remember to adjust the role and permission names, middleware, and routes according to your application's requirements.

Error Handling In Laravel

As of my last knowledge update in September 2021, Laravel 8 was the latest version available. However, I can still provide you with information about error handling and custom error pages in Laravel, although the specifics might change if Laravel 10 is released after my last update.

In Laravel, you can handle various types of errors and create custom error pages to provide a better user experience. Common HTTP errors like 404 (Not Found) and 500 (Internal Server Error) can be customized to display user-friendly messages or views.

Here's how you can handle and customize these errors in Laravel:

1. **Customizing Error Views:** Laravel allows you to create custom error views for different HTTP error codes. These views are located in the `resources/views/errors` directory. You can create blade view files for specific error codes, such as `404.blade.php` for the 404 error and `500.blade.php` for the 500 error. Customize these views according to your design and messaging preferences.

2. **Exception Handling:** Laravel's `App\Exceptions\Handler` class is responsible for handling exceptions and converting them into HTTP responses. You can customize how exceptions are handled in the `render` method of this class. To handle specific exceptions, you can check the type of exception and return a custom response. For example:

```
public function render($request, Throwable $exception)
{
    if ($exception instanceof \Symfony\Component\HttpKernel\Exception\NotFoundHttpException) {
        return response()->view('errors.404', [], 404);
    }

    // Handle other exceptions...

    return parent::render($request, $exception);
}
```

3. **Registering Custom Error Pages:** You can also register custom error pages in your `app/Providers/AppServiceProvider.php` file's `boot` method using the `View::share` method:

```
public function boot()
{
    $this->registerErrorViews();

    // Other boot logic...
}

protected function registerErrorViews()
{
    view()->share('errorCode', ''); // Set a default value

    view()->composer('errors.404', function ($view) {
        $view->with('errorCode', 404);
    });

    // Add similar composers for other error views...
}
```

Remember to customize the handling of each exception based on your application's requirements.

4. **Maintenance Mode:**

You can put your application into maintenance mode by running the `php artisan down` command. This will display a maintenance mode view when users access your application.

```
php artisan down --message="We're doing some maintenance. Please check back later."
```

5. **Customizing Maintenance Mode Page:**

You can customize the maintenance mode page by modifying the `resources/views/errors/503.blade.php` view.

As of my last knowledge update in September 2021, Laravel 10 had not been released yet. The latest version available at that time was Laravel 8. However, I can provide you with information on how error handling and custom error pages are typically implemented in Laravel. Keep in mind that some details might have changed in newer versions, so it's always a good idea to consult the official Laravel documentation for the most up-to-date information.

In Laravel, you can handle various types of errors and customize error pages using the following steps:

1. **Handling Exceptions:**

Laravel provides a powerful exception handling mechanism through the `App\Exceptions\Handler` class. You can customize this class to handle different types of exceptions. Here's how you can define custom exception handling:

First, create a new exception class if needed:

```
php artisan make:exception CustomException
```

Then, open the `app\Exceptions\Handler.php` file and update the `report` and `render` methods to handle your custom exceptions and other built-in exceptions:

```
use Exception;
use Illuminate\Foundation\Exceptions\Handler as ExceptionHandler;

class Handler extends ExceptionHandler
{
    // ...

    public function render($request, Exception $exception)
    {
        if ($exception instanceof CustomException) {
            return response()->view('errors.custom', [], 500);
        }

        return parent::render($request, $exception);
    }

    // ...


 # OR alternative


    public function render($request, Exception $exception)
{
    if ($this->isHttpException($exception)) {
        if ($exception->getStatusCode() == 404) {
            return response()->view('errors.' . '404', [], 404);
        }
        if ($exception->getStatusCode() == 500) {
            return response()->view('errors.' . '500', [], 500);
        }
    }
    return parent::render($request, $exception);
}



    # OR alternative

    public function register(): void
{

   $this->renderable(function (NotFoundHttpException $e, Request $request) {




        if ($request->is('api/*')) {
            return response()->json([
                'message' => 'Record not found.'
            ], 404);
        }



    });



}
}
# OR ALTERNATIVE

public function render($request, Throwable $exception)
{
    if ($exception instanceof NotFoundHttpException) {
        return response()->view('errors.404', [], 404);
    }
    if ($exception instanceof \PDOException) {
        return response()->view('errors.500', [], 500);
    }

    return parent::render($request, $exception);
}
```

Inside the `errors` directory, create Blade view files for different error codes. For example:

- `resources/views/errors/404.blade.php` for the 404 error page.
- `resources/views/errors/500.blade.php` for the 500 error page.

Customize these views as needed, adding your own HTML, CSS, and design elements.

3. **Error Logging:**

Laravel also supports error logging out of the box. You can configure error logging in the `.env` file using the `LOG_CHANNEL` and related settings.

For example, to log errors to the `daily` log channel, add or update the following in your `.env` file:

```
LOG_CHANNEL=daily
```

4. **Using Route Closures for Error Pages (Optional):**

In some cases, you might want to return custom error pages directly from your routes. You can do this using route closures:

```
Route::get('/404', function () {
    return view('errors.404');
});

Route::get('/500', function () {
    return view('errors.500');
});
```

5. **HTTP Exceptions (404, 500, etc.):**

Laravel provides the abort() function to throw HTTP exceptions. You can use this function to return a specific HTTP response with a status code and a message.

```
// For a 404 Not Found error
if (!$user) {
    abort(404, 'User not found');
}

// For a 500 Internal Server Error
if ($errorCondition) {
    abort(500, 'Something went wrong');
}
```

how to get a database html data in laravel with various Format

In this example, strip_tags($post->content) removes all HTML tags from the post's content, and then Strlimit is applied to the stripped content. The use of {!! !!} around the Strlimit output ensures that the HTML is not escaped, allowing it to be rendered as HTML.

```
@foreach($posts as $post)
    <div class="post">
        <h2>{{ $post->title }}</h2>
        <p>{!! Str::limit(strip_tags($post->content), 150,'...') !!}</p> // '...' is optional parameter . it can be used like ''
         or
          <p>{!! Str::words(strip_tags($post->content), 150,'') !!}</p>
    </div>
@endforeach
```

polymorphic relationship tags in news , video and photo gallery table in laravel

Setting up a polymorphic relationship in Laravel for a news portal project where tags can be associated with news articles, videos, and photo galleries involves a few steps. Polymorphic relationships allow a single relationship to be used for multiple types of models. Here's how you can achieve this:

Let's assume you have three tables/models: `News`, `Video`, and `PhotoGallery`, and you want to associate tags with each of these models.

1. **Database Setup:**

You'll need a table to store tags and a pivot table to manage the relationships. Create the necessary migration files:

```
php artisan make:migration create_tags_table
php artisan make:migration create_taggables_table
```

In the `create_tags_table` migration:

```
public function up()
{
    Schema::create('tags', function (Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->timestamps();
    });
}
```

In the `create_taggables_table` migration:

```
public function up()
{
    Schema::create('taggables', function (Blueprint $table) {
        $table->id();
        $table->unsignedBigInteger('tag_id');
        $table->unsignedBigInteger('taggable_id');
        $table->string('taggable_type');
        $table->timestamps();

        $table->unique(['tag_id', 'taggable_id', 'taggable_type']);

        $table->foreign('tag_id')->references('id')->on('tags')->onDelete('cascade');
    });

}

 # OR alternative

    public function up()
{
    Schema::create('taggables', function (Blueprint $table) {
        $table->id();
        $table->unsignedBigInteger('tag_id');
        $table->unsignedBigInteger('taggable_id');
        $table->string('taggable_type');
        $table->timestamps();
    });
}
```

Run the migrations:

```
php artisan migrate
```

2. **Models Setup:**

In each of your models (`News`, `Video`, `PhotoGallery`), you'll define the polymorphic relationship with the `Tag` model.

```
use Illuminate\Database\Eloquent\Model;

# In your Tag model, define the morphedByMany relationship:

class Tag extends Model
{
    // ...

   public function taggable()
{
    return $this->morphTo();
}

# or alternative

public function news()
{
    return $this->morphedByMany('App\News', 'taggable');
}

public function videos()
{
    return $this->morphedByMany('App\Video', 'taggable');
}

public function photoGalleries()
{
    return $this->morphedByMany('App\PhotoGallery', 'taggable');
}



}



class News extends Model
{
    // ...

    public function tags()
    {
        return $this->morphToMany(Tag::class, 'taggable');
    }
}

class Video extends Model
{
    // ...

    public function tags()
    {
        return $this->morphToMany(Tag::class, 'taggable');
    }
}

class PhotoGallery extends Model
{
    // ...

    public function tags()
    {
        return $this->morphToMany(Tag::class, 'taggable');
    }
}
```

3. **Creating and Retrieving Tags:**

You can now use these relationships to attach and retrieve tags for each model.

```
$news = News::find(1);
$news->tags()->attach([1, 2, 3]); // Attach tags to the news article

$tags = $news->tags; // Retrieve tags associated with the news article

// Similarly, for Video and PhotoGallery
$video = Video::find(1);
$video->tags()->attach([2, 3, 4]);

$photoGallery = PhotoGallery::find(1);
$photoGallery->tags()->attach([1, 4, 5]);

# or alternative
// Attaching tags to a model
$news = News::find(1);
$news->tags()->attach($tagIds);

// Retrieving tags for a model
$news = News::find(1);
$tags = $news->tags;
```

4. **Retrieving Models by Tag:**

You can also retrieve models associated with a specific tag.

```
$tag = Tag::find(1);

$newsWithTag = $tag->news; // Retrieve news articles with this tag
$videosWithTag = $tag->videos; // Retrieve videos with this tag
$photoGalleriesWithTag = $tag->photoGalleries; // Retrieve photo galleries with this tag
```

That's the basic setup for a polymorphic relationship involving tags and multiple types of models in a Laravel news portal project. You can customize and expand upon this foundation to suit your specific project requirements.

public function tags() **: MorphToMany** why used that type of class with : colon in laravel ?

The : MorphToMany after the method declaration indicates the return type of the method, which is a hint for developers and IDEs to understand the expected return type.

```
public function tags() : MorphToMany
{
    return $this->morphToMany(Tag::class,'taggable');
};
```

Regarding the : MorphToMany in the method declaration, it's optional and not strictly necessary for the code to work correctly. It's a type hint that helps you and your IDE understand the expected return type of the method. Laravel will still establish the relationship correctly even if you omit the : MorphToMany type hint. However, including it can improve code readability and help catch potential type-related errors during development.

In Wrap up : The : MorphToMany type hint is optional but can be helpful for code clarity and type safety.

Using Polymorphic In Blade File (CRUD)

If you want to display the previously associated tags as a comma-separated list in a single input field for editing news, you can follow these steps:

1. Store or Create File.

```
 public function AllPhotoGallery(){

        $photo = PhotoGallery::latest()->get();
        return view('backend.photo.all_photo',compact('photo'));

    } // End Method

    # In above blade file set the code like the following
                       <td>
                        @php
                            $allTags = $item->tags->pluck('name')->toArray();
                        @endphp
                        @foreach ($allTags as $tag)
                            <span class="badge fill-round bg-primary">{{ $tag }}</span>
                        @endforeach
                    </td>


public function store(Request $request)
{
 $gallery = PhotoGallery::create([
                    'photo_gallery'=>$save_url,
                    'post_date'=>Carbon::now()->format('D F Y'),
                ]);

    // Attach tags
     // data set in input like international,national,business . it is used if the input tag is
     <input type="text" name="tags"  class="selectize-close-btn" value="{{ old('tags') }}"> tag name will be store as string.

     if stored input value in  array  name="tags[]" ,
     we get the tags ,

     $tags = $request->tags.

     Otherwise

     $tags = explode(',', $request->input('tags'));

    foreach ($tags as $tagName) {
        $tag = Tag::insertGetId(['name' => $tagName]);
        $gallery ->tags()->attach($tag);
    }

    return redirect()->route('news.index')->with('success', 'News created successfully');
}
```

1. Retrieve the existing tags associated with the news item in your controller:

```
$news = News::findOrFail($id);
$existingTags = $news->tags->pluck('name')->implode(',');
```

2. Pass the `$existingTags` variable to your Blade view:

```
return view('news.edit', compact('news', 'existingTags'));
```

3. In your Blade view's edit form, populate the input field with the existing tags:

```
<form method="POST" action="{{ route('news.update', $news->id) }}">
    <!-- ... other fields ... -->
        <input type="text" name="tags" value="{{ $existingTags }}" class="form-control">
    <!-- ... submit button ... -->
</form>
```

4. In your update method of the controller, you can handle updating the tags as follows:

```
 public function AllVideoGallery(){

     $video = VideoGallery::latest()->get();
     return view('backend.video.all_video',compact('video'));

 } // End Method


 public function AddVideoGallery(){
     return view('backend.video.add_video');
 } // End Method


public function StoreVideoGallery(Request $request){


   $videoGallery =  VideoGallery::create([

         'video_title' => $request->video_title,
         'video_url' => $request->video_url,
     ]);

     $tags = explode(',',$request->input('tags'));
     $tagIds = [];
     foreach ($tags as  $tag) {
         $storeTag = Tag::firstOrCreate(['name'=>trim($tag)]);
         $tagIds[] = $storeTag->id;
     }
     $videoGallery->tags()->attach($tagIds);

      $notification = array(
         'message' => 'Video Inserted Successfully',
         'alert-type' => 'success'

     );
     return redirect()->route('all.video.gallery')->with($notification);


 }// End Method



 public function EditVideoGallery($id){

     $video = VideoGallery::findOrFail($id);
     $existing_tags = $video->tags->pluck('name')->implode(',');
     // dd($existing_tags);
     return view('backend.video.edit_video',compact('video','existing_tags'));

 }// End Method


 public function UpdateVideoGallery(Request $request){

     $video_id = $request->id;
     $videoImage = VideoGallery::findOrFail($video_id);
     $prev_tags = $videoImage->tags->pluck('id')->toArray();

     VideoGallery::findOrFail($video_id)->update([

         'video_title' => $request->video_title,
         'video_url' => $request->video_url,

     ]);
```

```
        return redirect()->route('all.video.gallery')->with($notification);



        //    $videoImage->tags()->detach();
          $tags = explode(',', $request->input('tags'));
          $tagID=[];
          foreach ($tags as $tagName) {
              $tag = Tag::firstOrCreate(['name' => $tagName]);
              $tagID[]=$tag->id;
          };

          $videoImage->tags()->sync($tagID);
          // Delete the tag that is associated with video image
          Tag::whereIn('id',$prev_tags)->has('videoGallary','=',0)->delete();

        return redirect()->route('all.video.gallery')->with($notification);

        }

     public function DeleteVideoGallery($id){

        $photo = VideoGallery::findOrFail($id);
        $tagIds = $photo->tags->pluck('id')->toArray();

        $photo->tags()->detach();
        VideoGallery::findOrFail($id)->delete();

        Tag::whereIn('id',$tagIds)->has('videoGallary','=',0)->delete();

        return redirect()->back()->with($notification);

    }// End Method
```

Note : Apologies for any confusion. In the context of `Tag::whereIn('id', $tagIds)->has('news', '=', 0)->delete();`, the `0` is not related to an index. It represents the count of related news articles.

The line of code `->has('news', '=', 0)` is checking for tags that have zero related news articles. It's not referring to an index but rather specifying a condition for filtering tags based on the count of related news articles.

In Laravel's Eloquent ORM, the `has` method is used to filter results based on the existence of related records. In this case, it's being used to filter tags that have no related news articles (i.e., their count is zero).

So, the entire line of code is essentially saying "delete tags that are in the given `$tagIds` array and have no related news articles (count equals 0)." It's part of the process of cleaning up unused tags from the `tags` table.