التكنولوجيا التطبيقية

APPLIED TECHNOLOGY

وزارة التربية والتعليم والتعليم الفني

Information Technology

تطوير المواقع و البرمجيات

الصف الثالث

2nd

Web Development and Programming

STUDENT GUIDE

2023 - 2024

التكنولوجيا التطبيقية
APPLIED TECHNOLOGY
وزارة التربية والتعليم والتعليم الفني

MINISTRY OF EDUCATION AND TECHNICAL EDUCATION
وزارة التربية والتعليم والتعليم الفني

تطوير المواقع و البرمجيات
الصف الثالث

Unit 22
Laravel

Short description	**By the end of the training unit, the student will be able to the following:** - Practice Web development	
Skills	**The learner is able to do:**	
	TPC4.7	Using controllers and routes for APIs and URLs
	TPC4.8	Creating and using composer packages
	TPC4.9	Create restful services, use an Effect Dependency Array and how to hand errors in data requests
Knowledge	**The learner must know and understand:**	
	TPK26	Analyze and solve common web applications tasks by writing PHP programs

Lesson One : Introduction to Laravel and Installation

1. Introduction

Laravel is an expressive and elegant web application framework. It offers a structured starting point for app development, letting developers focus on building remarkable applications with ease. Laravel excels in offering robust features like thorough dependency injection, a comprehensive database abstraction layer, queues, scheduled jobs, and extensive testing options. Suitable for both newcomers and experienced PHP framework users, Laravel supports your growth as a developer, aiding in both initial learning and advanced skill enhancement.

2. Why Use a Framework?

PHP packages, developed by experts, provide specific, well-maintained functionalities, saving time and effort. Frameworks like Laravel, Symfony, Lumen, and Slim integrate these packages with essential framework elements for efficient and cohesive component integration.

3. I'll Just Build It Myself

Starting a new web app from scratch involves several decisions. Firstly, you'll need to choose a library for handling HTTP requests and responses. Then, select a router and determine how to configure your routes, including their syntax and location. You'll also need to decide on the structure for your controllers and how they'll be loaded, likely requiring a dependency injection container. Each of these choices, from the routing syntax to the controllers' organization, impacts not only the initial development but also future maintenance and scalability, especially when managing multiple custom applications.

4. Consistency and Flexibility in Frameworks

Frameworks ensure consistency in component selection and interoperability. They provide conventions, like in Laravel's routing, reducing the need for developers to learn new code for each project. Frameworks should offer a solid foundation and customization flexibility, a key aspect of Laravel.

5. Evolution of Web and PHP Frameworks

Before Laravel, various PHP frameworks existed, influenced by Ruby on Rails (2004). Rails introduced MVC, RESTful APIs, and other rapid development tools. The first PHP framework, CakePHP, came in 2005, followed by others like Symfony and CodeIgniter. These frameworks ranged from rapid development-focused, like Rails, to more enterprise-oriented like Symfony. CodeIgniter, inspired by Rails, was simple and popular but lagged in technological updates. Dissatisfied with CodeIgniter, Taylor Otwell created Laravel in 2011.

6. Why Choose Laravel?

Laravel suits modern web application development, being scalable, community-driven, and versatile. It's a "progressive" framework, easy for beginners and robust for experts, supporting full-stack development. Laravel's scalability is evident from its performance in handling millions of requests. The framework's community contributes to its robustness.

7. Laravel's Versions and Development

Laravel started in 2011 with features like Eloquent ORM and routing. Rapid development led to versions 2 and 3, introducing new tools. Laravel 4, a rewrite, utilized Composer and Symfony components, adding queues and database seeding. Laravel 5 further evolved with a new structure and tools like Elixir and Scheduler. Laravel 6, introduced in 2019, adopted SemVer, focusing on frequent, less monumental releases.

8. What Makes Laravel Special?

Laravel stands out for its developer-centric approach, prioritizing speed and happiness. It's designed for simplicity and ease, with a shallow learning curve. Laravel offers a complete ecosystem for web development, from local development tools to advanced deployment solutions. The framework emphasizes convention over configuration and simplicity, enabling developers to build applications efficiently.

Lesson Two : Using Composer and Artisan

1. Composer

Composer is essential for modern PHP development. It's a PHP dependency manager, similar to NPM for Node or RubyGems for Ruby. Composer is integral for tasks like testing, script loading, and installations. You need Composer to install and update Laravel, as well as to manage external dependencies.

2. Local Development Environments

For Laravel projects, simple tools like MAMP, WAMP, or XAMPP are often sufficient for hosting a development environment. Alternatively, Laravel can run on PHP's built-in web server if the PHP version is compatible. To use this, run php -S localhost:8000 -t public from the Laravel site's root folder, and access your site at http://localhost:8000/. For more advanced needs like handling different local domains, MySQL, etc., a more robust tool than PHP's built-in server is recommended.

3. Artisan Serve

If you run php artisan serve after setting up your Laravel application, it'll serve it at http://localhost:8000, just like PHP's built-in web server. You're not getting anything else for free here, so its only meaningful benefit is that it's easier to remember**.**

3.1. Installing Laravel with the Laravel Installer Tool

If you have Composer installed globally, installing the Laravel installer tool is as simple as running the following command:

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

Once you have the Laravel installer tool installed, spinning up a new Laravel project is simple. Just run this command from your command line:

```
laravel new projectName
```

This will create a new subdirectory of your current directory named projectName and install a bare Laravel project in it.

3.2. Installing Laravel with Composer's create-project Feature

Before creating your first Laravel project, you should ensure that your local machine has PHP and Composer installed. If you are developing on macOS, PHP and Composer can be installed within minutes via Laravel Herd. In addition, we recommend installing Node and NPM.

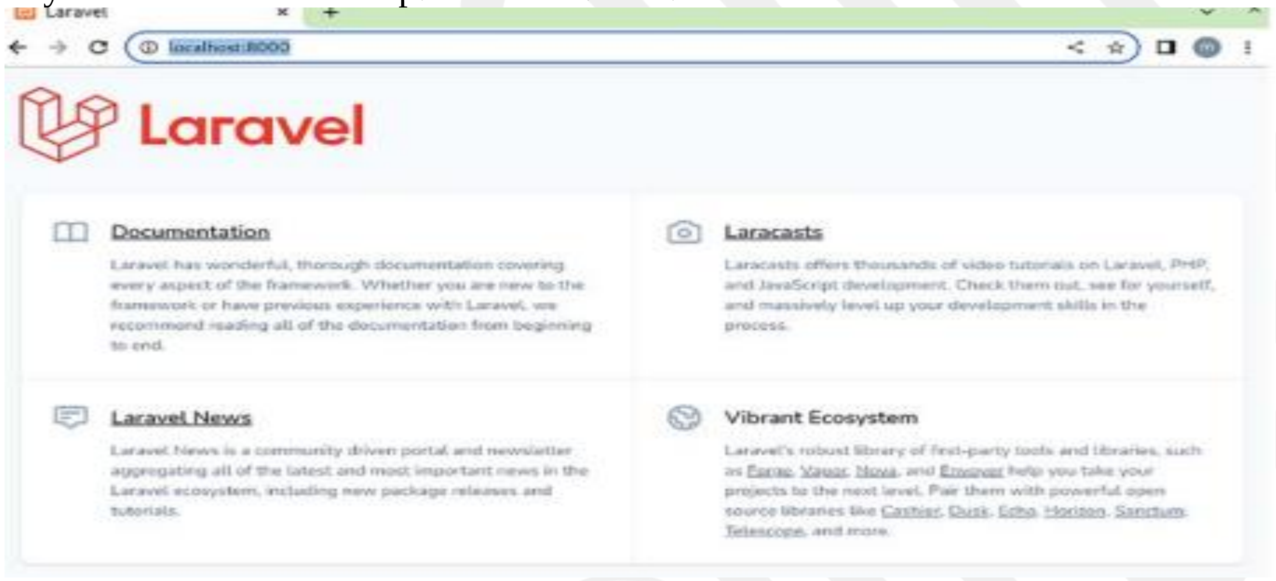
After you have installed PHP and Composer, you may create a new Laravel project via the Composer create-project command:

```
composer create-project laravel/laravel example-app
```

After the project has been created, start Laravel's local development server using the Laravel's Artisan CLI serve command:

```
cd example-app

php artisan serve
```

Once you have started the Artisan development server, your application will be accessible in your web browser at http://localhost:8000.



Documentation

Laravel has wonderful, thorough documentation covering every aspect of the framework. Whether you are new to the framework or have previous experience with Laravel, we recommend reading all of the documentation from beginning to end.

Laracasts

Laracasts offers thousands of video tutorials on Laravel, PHP, and JavaScript development. Check them out, see for yourself, and massively level up your development skills in the process.

Laravel News

Laravel News is a community driven portal and newsletter aggregating all of the latest and most important news in the Laravel ecosystem, including new package releases and tutorials.

Vibrant Ecosystem

Laravel's robust library of first-party tools and libraries, such as Forge, Vapor, Nova, and Envoyer help you take your projects to the next level. Pair them with powerful open source libraries like Cashier, Dusk, Echo, Horizon, Sanctum, Telescope, and more.

4. Laravel's Directory Structure

When you open up a directory that contains a skeleton Laravel application, you'll see the following files and directories:

```
app/
bootstrap/
config/
database/
public/
resources/
routes/
storage/
tests/
```

```
vendor/
.editorconfig
.env
.env.example
.gitattributes
.gitignore
artisan
composer.json
composer.lock
package.json
phpunit.xml
readme.md
vite.config.js
```

Let's walk through them one by one to get familiar.

4.1. The Folders

The root directory contains the following folders by default:

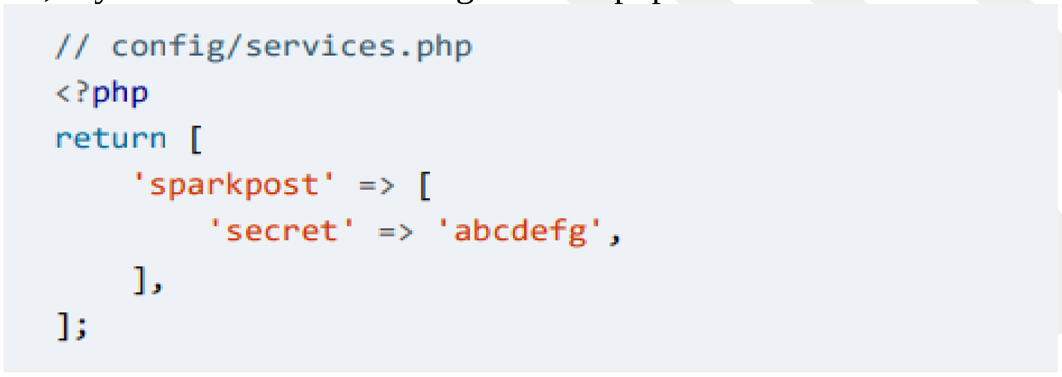
- App: Contains your application's core elements like models, controllers, and PHP domain code.
- Bootstrap: Holds files for Laravel's bootstrapping process during runtime.
- Config: Stores all configuration files.
- Database: Houses database migrations, seeds, and factories.
- Public: The server's point for serving the website, including index.php for bootstrapping and routing, and public files like images and scripts.
- Resources: Contains necessary files for scripts, views, and optionally, source CSS and JavaScript files.
- Routes: Hosts all HTTP and console route definitions.
- Storage: Stores caches, logs, and compiled system files.
- Tests: Dedicated to unit and integration tests.
- Vendor: Where Composer dependencies are installed and Git-ignored for version control exclusion, with Composer expected to run during remote server deployment.

4.2. The Loose Files

- .editorconfig: Sets coding standards for IDEs/text editors, like indent size and charset.
- .env and .env.example: Control environment variables. .env.example serves as a template for creating environment-specific .env files, which are not tracked by Git.
- .gitignore and .gitattributes: Manage Git configurations.
- Artisan: Enables running Artisan commands via the command line.
- composer.json and composer.lock: Composer configuration files, with composer.json being user-editable and composer.lock not. They define the project's PHP dependencies.
- package.json: Similar to composer.json, but for frontend assets and dependencies, guiding NPM on JavaScript dependencies.
- phpunit.xml: Configuration file for PHPUnit, used for Laravel testing.
- readme.md: Provides a basic introduction to Laravel (not included in Laravel installer).
- vite.config.js: (Optional) Configures Vite for compiling and processing frontend assets.

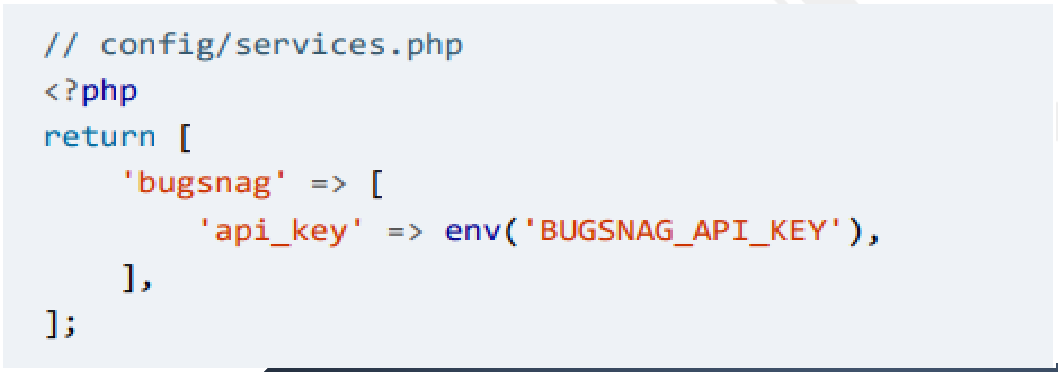
5. Configuration

The core settings of your Laravel application— database connection settings, queue and mail settings, etc.—live in files in the config folder. Each of these files returns a PHP array, and each value in the array is accessible by a config key that is comprised of the filename and all descendant keys, separated by dots (.).
So, if you create a file at config/services.php that looks like this:



```php
// config/services.php
<?php
return [
    'sparkpost' => [
        'secret' => 'abcdefg',
    ],
];
```

You can access that config variable using config('services.sparkpost.secret').
Any configuration variables that should be distinct for each environment (and therefore not committed to source control) will instead live in your .env files. Let's say you want to use a different Bugsnag API key for each environment. You'd set the config file to pull it from .env:



```php
// config/services.php
<?php
return [
    'bugsnag' => [
        'api_key' => env('BUGSNAG_API_KEY'),
    ],
];
```

This env() helper function pulls a value from your .env file with that same key. So now, add that key to your .env (settings for this environment) and .env.example (template for all environments) files:

```
# In .env
BUGSNAG_API_KEY=oinfp9813410942
```

```
# In .env.example
BUGSNAG_API_KEY=
```

Your .env file will already contain quite a few environment specific variables needed by the framework, like which mail driver you’ll be using and what your basic database settings are.

6. Introduction To Artisan

Modern PHP frameworks, including Laravel, rely heavily on command-line interactions post-installation. Laravel offers three key tools for this:

Artisan: A comprehensive set of command-line actions, with the capability to add custom commands.
Tinker: An interactive shell or REPL (Read-Eval-Print Loop) for Laravel applications.
The artisan tool is essentially a PHP file located in the root folder of your application. To use it, you execute php artisan from the command line, where artisan is the PHP file being parsed by PHP, and any subsequent text represents arguments for Artisan.

The available Artisan commands can vary based on the application's specific code or any packages it uses. Therefore, it's advisable to check the list of commands for each new Laravel application you work with. To view all available Artisan commands, run php artisan list in the project's root directory. Executing php artisan without any arguments will also display the list of commands.

6.1. Basic Artisan Commands

- clear-compiled: Removes Laravel’s compiled class file, a type of cache.
- down, up: Enables/disables maintenance mode for tasks like fixing errors or running migrations.
- dump-server: Starts a server to collect and display variables.
- env: Shows Laravel's current environment.
- help: Provides command help, e.g., php artisan help commandName.
- Migrate: Executes all database migrations.
- Optimize: Refreshes configuration and route files.
- Serve: Starts a PHP server at localhost:8000, with customizable host/port.
- tinker: Opens the Tinker REPL.
- stub:publish: Customizes available stubs.
- docs: Accesses Laravel docs; parameterized for specific topics.
- about: Displays an overview of your project environment and configurations.

6.2. Options

Artisan commands in Laravel have several useful options:

- -q: Suppresses all output.
- -v, -vv, -vvv: Controls output verbosity (normal, verbose, debug).
- --no-interaction: Prevents interruption from interactive prompts during automated processes.
- --env: Sets the environment (like local, production) for the command.
- --version: Displays the Laravel version in use.

Artisan commands can be run manually or as part of automated processes. For instance, in deployment, commands like php artisan config:cache can be automated. Options -q and --no-interaction ensure seamless execution in non-interactive environments, such as automated scripts.

6.3. The Grouped Commands

- Auth: Use auth:clear-resets to clear expired password reset tokens.
- Cache: Use cache:clear to clear the cache, cache:forget to remove specific items, and cache:table for database cache migration.
- Config: Use config:cache to cache configuration settings and config:clear to clear this cache.
- db: Use db:seed to seed the database using configured seeders.
- Event: Use event:list to show events/listeners, event:cache to cache this list, event:clear to clear the cache, and event:generate to create event files.
- Key: Use key:generate to create an application encryption key.

- Make: The make: commands create items from stubs with various parameters, e.g., make:migration.
- Migrate: Use migrate for database migrations, migrate:install to create migration tracking, migrate:reset to reset migrations, migrate:refresh to rerun all migrations, migrate:rollback for a single rollback, migrate:fresh to refresh all migrations, and migrate:status to check migration status.
- Route: Use route:list to view route definitions, route:cache to cache routes, and route:clear to clear the route cache.
- Session: Use session:table to create a database session migration.
- Storage: Use storage:link to create a symbolic link from public/storage to storage/app/public.
- View: Use view:clear to clear cached views when necessary.

Lesson Three : Routing , Rest and Views

1. Introduction to Routing

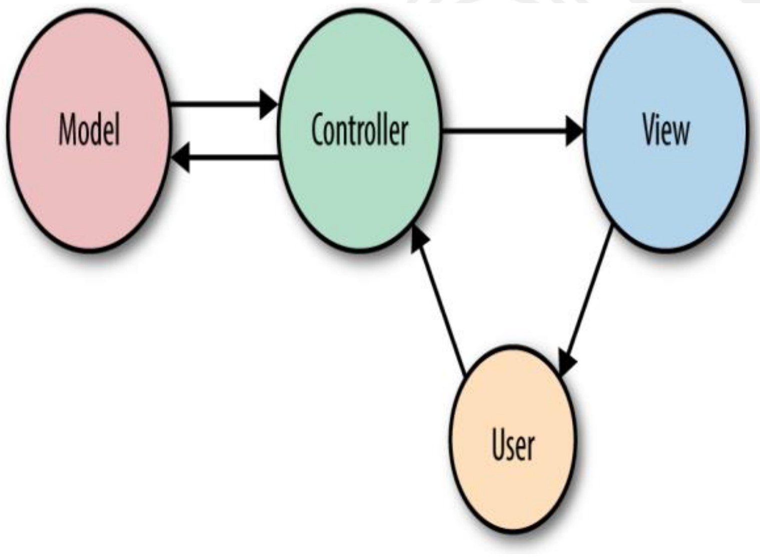
Web application frameworks primarily handle user requests and deliver responses, typically through HTTP(S). Defining an application's routes is a critical initial task in learning a web framework. Routes are essential for user interaction. In this chapter, we focus on routes in Laravel. We'll explore how to define routes, link them to their corresponding code, and utilize Laravel's routing tools for various routing requirements.

2. A Quick Introduction to MVC, the HTTP Verbs, and REST

Model-View-Controller (MVC) structures MVC applications, commonly using REST-ish routes and verbs. Here's a concise explanation:

- Model: This represents a database table or a record from it, like "Company" or "Dog."
- View: This is the template for displaying data to the user, such as a login page with specific HTML, CSS, and JavaScript.
- Controller: Acting like a traffic cop, it handles HTTP requests from the browser. It retrieves and validates data from the database and other sources, then responds back to the user.

In this process, the user sends an HTTP request to the controller via their browser. The controller interacts with the model to write or retrieve data, then passes this data to the view. Finally, the view returns to the user's browser for display.



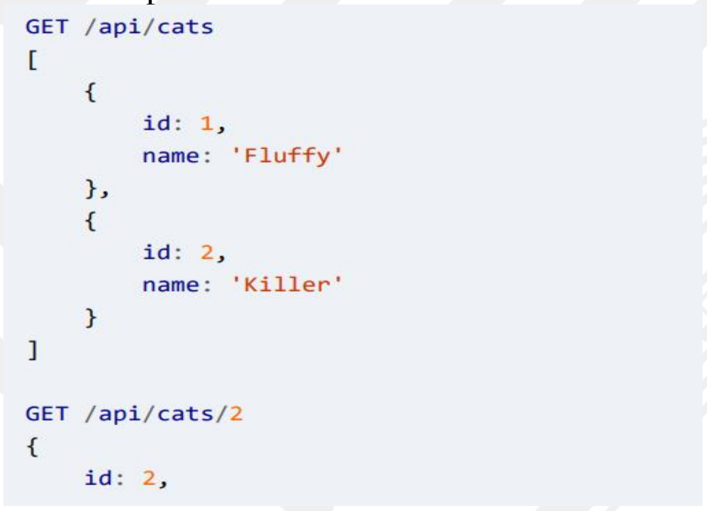
The HTTP Verbs include GET and POST, which are the most common, along with PUT, DELETE, HEAD, OPTIONS, and PATCH. Less commonly used are TRACE and CONNECT.

GET: Requests a resource or list of resources.
HEAD: Requests only the headers of the GET response.
POST: Creates a resource.
PUT: Overwrites a resource.
PATCH: Modifies a resource.
DELETE: Deletes a resource.
OPTIONS: Inquires about allowed verbs for a URL

2.1 What Is REST?

Representational State Transfer (REST) is an architectural style used for creating APIs. In the context of Laravel, RESTful APIs typically have these features:

They are organized around "resources" identifiable by URIs. For example, /cats for all cats, /cats/15 for a specific cat with ID 15.
They use HTTP verbs for interactions, like GET /cats/15 or DELETE /cats/15.
They are stateless: each request is authenticated independently without persistent sessions.
They are cacheable and consistent: requests generally return the same result for all users, with few exceptions.
They typically return data in JSON format.
A common API practice is to assign unique URL structures to each Eloquent model exposed as an API resource, allowing user interactions via specific verbs and returning JSON responses.



```
GET /api/cats
[
    {
        id: 1,
        name: 'Fluffy'
    },
    {
        id: 2,
        name: 'Killer'
    }
]

GET /api/cats/2
{
    id: 2,
    name: 'Killer'
}

POST /api/cats with body:
{
    name: 'Mr Bigglesworth'
}
(creates new cat)

PATCH /api/cats/3 with body:
{
    name: 'Mr. Bigglesworth'
}
(updates cat)

DELETE /api/cats/2
(deletes cat)
```

This gives you the idea of the basic set of interactions we are likely to have with our APIs. Let's dig into how to make them happen with Laravel.

3. Route Definitions

In a Laravel application, web routes are defined in routes/web.php, while API routes are in routes/api.php. Web routes are for user interactions, and API routes are for APIs. We'll focus on routes/web.php, where routes are defined by matching a path (like /) with a closure.

```
// routes/web.php
Route::get('/', function () {
    return 'Hello, World!';
});
```

Closures in PHP are anonymous functions that can be treated like objects, assigned to variables, passed as arguments, or even serialized. In Laravel, when visiting the root domain ("/"), a closure defined in the router is executed, and its result is returned. Instead of directly outputting content with 'echo' or 'print', returning the content is preferred. This approach allows the content to pass through Laravel's request and response cycle and middleware before reaching the user. A basic website in Laravel can be built within the web routes file using simple GET routes and templates, effectively serving a classic website.

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

Route::get('about', function () {
    return view('about');
});

Route::get('products', function () {
    return view('products');
```

4. Route Verbs

You might've noticed that we've been using Route::get() in our route definitions. This means we're telling Laravel to only match for these routes when the HTTP request uses the GET action. But what if it's a form POST , or maybe some JavaScript sending PUT or DELETE requests? There are a few other options for methods to call on a route definition, as illustrated in pervious Example .

```
Route::get('/', function () {
    return 'Hello, World!';
});

Route::post('/', function () {
    // Handle someone sending a POST request to this
});

Route::put('/', function () {
    // Handle someone sending a PUT request to this r
});

Route::delete('/', function () {
    // Handle someone sending a DELETE request to thi
});

Route::any('/', function () {
    // Handle any verb request to this route
});

Route::match(['get', 'post'], '/', function () {
    // Handle GET or POST requests to this route
});
```

5. Route Handling

Passing a closure to define a route is not the sole method; it's quick but not scalable for larger applications. Using route closures also prevents Laravel's route caching, impacting performance. Alternatively, specify a controller name and method as a string instead of a closure for better organization and caching benefits.

```
use App\Http\Controllers\WelcomeController;

Route::get('/', [WelcomeController::class, 'index']);
```

This is telling Laravel to pass requests to that path to the index() method of the App\Http\Controllers\WelcomeController controller. This method will be passed the same parameters and treated the same way as a closure you might've alternatively put in its place.

6. Route Parameters

If the route you're defining has parameters— segments in the URL structure that are variable— it's simple to define them in your route and pass them to your closure see The Next Example**.**

```
Route::get('users/{id}/friends', function ($id) {
    //
});
```

You can also make your route parameters optional by including a question mark (?) after the parameter name, as illustrated in coming Example . In this case, you should also provide a default value for the route's corresponding variable.

```
Route::get('users/{id?}', function ($id = 'fallbackId
    //
});
```

And you can use regular expressions (regexes) to define that a route should only match if a parameter meets particular requirements.

```
Route::get('users/{username}', function ($username) {
    //
})->where('username', '[A-Za-z]+');

Route::get('posts/{id}/{slug}', function ($id, $slug)
    //
})->where(['id' => '[0-9]+', 'slug' => '[A-Za-z]+']);
```

As you've probably guessed, if you visit a path that matches a route string but the regex doesn't match the parameter, it won't be matched. Since routes are matched top to bottomusers/abc would skip the first closure in pervious Example, but it would be matched by the second closure, so it would get routed there. On the other hand, posts/abc/123 wouldn't match any of the closures, so it would return a 404 (Not Found) error**.**

7. Route Names

The simplest way to refer to these routes elsewhere in your application is just by their path. There's a url() global helper to simplify that linking in your views, if you need it, The helper will prefix your route with the full domain of your site.

```
<a href="<?php echo url('/'); ?>">
// Outputs <a href="http://myapp.com/">
```

However, Laravel also allows you to name each route, which enables you to refer to it without explicitly referencing the URL. This is helpful because it means you can give simple nicknames to complex routes, and also because linking them by name means you don't have to rewrite your frontend links if the paths change**.**

```
// Defining a route with name() in routes/web.php:
Route::get('members/{id}', [\App\Http\Controller\Memb
    ->name('members.show');
```

```
// Linking the route in a view using the route() help
<a href="<?php echo route('members.show', ['id' => 14
```

This example illustrates a few new concepts. First, we're using fluent route definition to add the name, by chaining the name() method after the get() method. This method allows us to name the route, giving it a short alias to make it easier to reference elsewhere.

In our example, we've named this route members.show ; resourcePlural.action is a common convention within Laravel for route and view names.

You can name your route anything you'd like, but the common convention is to use the plural of the resource name, then a period, then the action. So, here are the routes most common for a resource named photo :

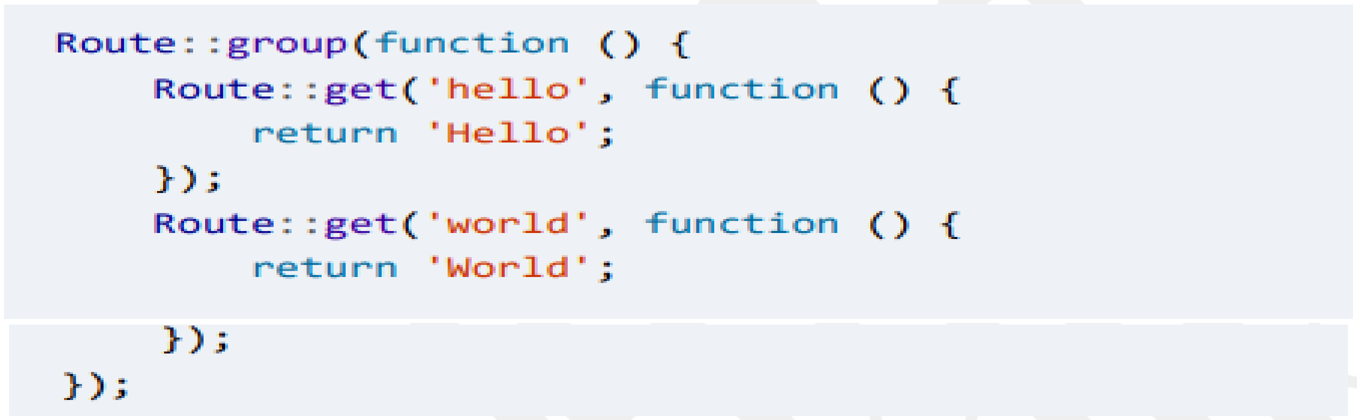
```
photos.index
photos.create
photos.store
photos.show
photos.edit
photos.update
photos.destroy
```

8. Route Groups

Often a group of routes share a particular characteristic— a certain authentication requirement, a path prefix, or perhaps a controller namespace. Defining these shared characteristics again and again on each route not only seems tedious but also can muddy up the shape of your routes file and obscure some of the structures of your application.

Route groups allow you to group several routes together and apply any shared configuration settings once to the entire group, to reduce this duplication. Additionally, route groups are visual cues to future developers (and to your own brain) that these routes are grouped together.

To group two or more routes together, you “surround” the route definitions with a route group, as shown in Example. In reality, you’re actually passing a closure to the group definition, and defining the grouped routes within that closure.

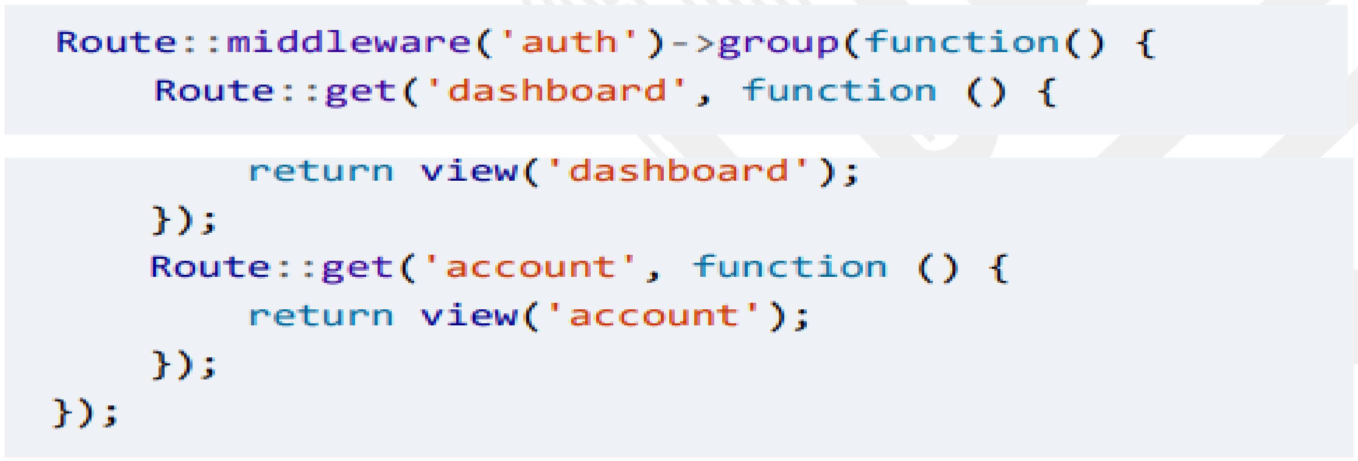


```
Route::group(function () {
    Route::get('hello', function () {
        return 'Hello';
    });
    Route::get('world', function () {
        return 'World';
    });
});
```

By default, a route group doesn’t actually do anything. There’s no difference between using the group in Example and separating a segment of your routes with code comments.

8.1 Middleware

Probably the most common use for route groups is to apply middleware to a group of routes , but, among other things, they’re what Laravel uses for authenticating users and restricting guest users from using certain parts of a site.



```
Route::middleware('auth')->group(function() {
    Route::get('dashboard', function () {
        return view('dashboard');
    });
    Route::get('account', function () {
        return view('account');
    });
});
```

Applying middleware in controllers Often it’s clearer and more direct to attach middleware to your routes in the controller instead of at the route definition. You can do this by calling the middleware() method in the constructor of your controller. The string you pass to the middleware() method is the name of the middleware, and you can optionally chain modifier methods (only() and except()) to define which methods will receive that middleware:

```
class DashboardController extends Controller
{
    public function __construct()
    {
        $this->middleware('auth');

        $this->middleware('admin-auth')
            ->only('editUsers');

        $this->middleware('team-member')
            ->except('editUsers');
    }
}
```

Note that if you're doing a lot of "only" and "except" customizations, that's often a sign that you should break out a new controller for the exceptional routes.

9. Views

In a few of the route closures we've looked at so far, we've seen something along the lines of return view('account') . What's going on here?

In the MVC pattern , views (or templates) are files that describe what some particular output should look like. You might have views for JSON or XML or emails, but the most common views in a web framework output HTML.

In Laravel, there are two formats of view you can use out of the box: plain PHP, or Blade templates,The difference is in the filename: about.php will be rendered with the PHP engine, and about.blade.php will be rendered with the Blade engine.

Once you've "loaded" a view with the view() helper, you have the option to simply return it (as in Example), which will work fine if the view doesn't rely on any variables from the controller.

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

This code looks for a view in resources/views/home.blade.php or resources/views/home.php, and loads its contents and parses any inline PHP or control structures until you have just the view's,output. Once you return it, it's passed on to the rest of the response stack and eventually returned to the user ,But what if you need to pass in variables? Take a look at that example.

```
Route::get('tasks', function () {
    return view('tasks.index')
        ->with('tasks', Task::all());
});
```

This closure loads the resources/views/tasks/index.blade.php or resources/views/tasks/index.php view and passes it a single variable named tasks , which contains the result of the Task::all() method. Task::all() is an Eloquent database query.

9.1 Returning Simple Routes Directly with Route::view()

Because it's so common for a route to just return a view with no custom data, Laravel allows you to define a route as a "view" route without even passing the route definition a closure or a controller/method reference, as you can see in Example**.**

```
// Returns resources/views/welcome.blade.php
Route::view('/', 'welcome');

// Passing simple data to Route::view()
Route::view('/', 'welcome', ['User' => 'Michael']);
```

9.2 Using View Composers to Share Variables with Every View

Sometimes it can become a hassle to pass the same variables over and over. There may be a variable that you want accessible to every view in the site, or to a certain class of views or a certain included subview— for example, all views related to tasks, or the header partial.

It's possible to share certain variables with every template or just certain templates, like in the following code:

```
view()->share('variableName', 'variableValue');
```

Lesson Four : Blade Templates , Controllers and Migrations

1.Introduction to Controllers

In the MVC pattern, controllers are classes that manage the logic for one or more routes. They often group similar routes, particularly in CRUD applications, which involve create, read, update, and delete operations. For instance, a controller might handle all actions related to a specific resource, like a blog post.

While it's tempting to put all logic in controllers, they should primarily act as traffic cops for HTTP requests in your application. Other request types, like cron jobs, Artisan command-line calls, or queue jobs, should be considered separately. Essentially, a controller's main role is to interpret an HTTP request and delegate it within the application.

To create a controller, you can use an Artisan command in the command line.

```
php artisan make:controller TaskController
```

Laravel includes Artisan, a command-line interface. It's used for tasks like running migrations, creating users, and more. Artisan's 'make' namespace helps generate system files, for example, using php artisan make:controller creates TaskController.php in app/Http/Controllers with predefined contents.

```
<?php

namespace App\Http\Controllers;

class TaskController extends Controller
{
    public function index()
    {
        return 'Hello, World!';
    }
}
```

Then, like we learned before, we'll hook up a route to it,

```
// routes/web.php
<?php

Route::get('/', [TaskController::class, 'index']);
```

That's it. Visit the / route and you'll see the words "Hello, World!"

2. Getting User Input

The second most common action to perform in a controller method is to take input from the user and act on it. That introduces a few new concepts, so let’s take a look at a bit of sample code and walk through the new pieces.
First, let’s bind our route; see Example:

```
// routes/web.php

Route::get('tasks/create', [TaskController::class, 'create']);

Route::post('tasks', [TaskController::class, 'store']);
```

Notice that we’re binding the GET action of tasks/create (which shows a form for creating a new task) and the POST action of tasks/ (which is where our form will POST to when we’re creating a new task). We can assume the create() method in our controller just shows a form, so let’s look at the store() method.

3. Form Method Spoofing

In HTML, forms are limited to GET and POST methods. To use other HTTP verbs like PUT, PATCH, or DELETE in Laravel, you need to specify them manually. While routing, you can define verbs with Route::get(), Route::post(), Route::patch(), Route::put(), Route::delete(), Route::any(), or Route::match().

For sending non-GET requests via a web browser, the form's 'method' attribute decides its HTTP verb. To submit forms with different verbs than GET or POST in Laravel, use form method spoofing. This involves adding a hidden input named '_method' to your form with a value of "PUT", "PATCH", or "DELETE". Laravel then treats the form submission as that specific request type. For instance, a form with a '_method' of "DELETE" will match routes defined with Route::delete(). JavaScript frameworks offer easier ways to send various requests like DELETE and PATCH.

```
<form action="/tasks/5" method="POST">

    <input type="hidden" name="_method" value="DELETE">

    <!-- or: -->

  @method ('DELETE')

</form>
```

4. CSRF Protection

In Laravel applications, you may encounter a TokenMismatchException when submitting forms due to CSRF (cross-site request forgery) protection. Laravel requires a token, named _token, for non-read-only routes (like POST, DELETE, etc.) to prevent CSRF attacks, where one website impersonates another to hijack user access. This token, generated at the start of each session, is checked against the session token for each submission. To avoid CSRF errors, you can either add the _token input to your form submissions, which is straightforward in HTML forms, or use a second method that bypasses this requirement for specific routes.

```
<form  action="/tasks/5"  method="POST">

      <?php echo csrf_field(); ?>

      <!-- or: -->

      <input  type="hidden"  name="_token"  value="<?php echo csr-

f_token(); ?>">

      <!-- or: -->

  @csrf

</form>
```

In JavaScript applications, it takes a bit more work, but not much. The most common solution for sites using JavaScript frameworks is to store the token on every page in a tag like this one:

```
<meta  name="csrf-token"  content="<?php echo csrf_token(); ?>"

  id="token">
```

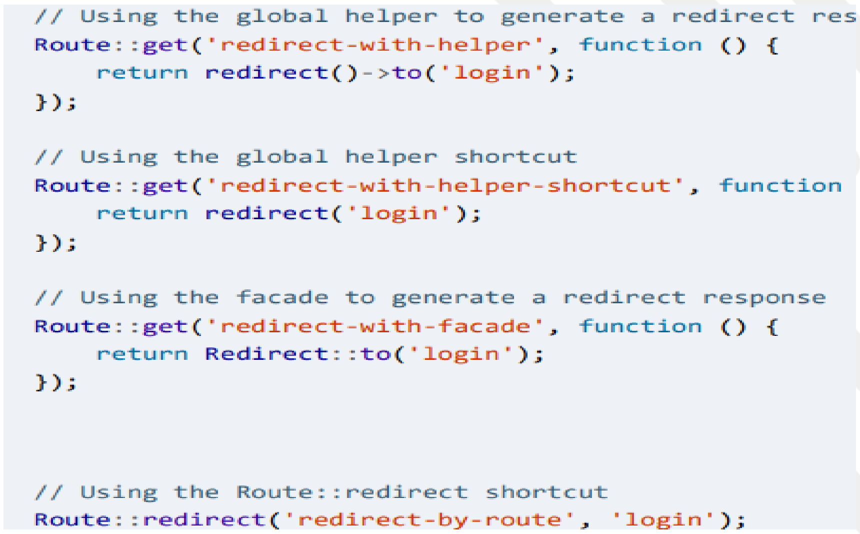
Storing the token in a tag makes it easy to bind it to the correct HTTP header, which you can do once globally for all requests from your JavaScript framework.

```
$.ajaxSetup({

      headers:  {

            'X-CSRF-TOKEN':  $('meta[name="csrf-token"]').attr(

'content')

      }

});
```

Laravel will check the X-CSRF-TOKEN (and X-XSRF-TOKEN , which Axios uses) on every request, and valid tokens passed there will mark the CSRF protection as satisfied.

5. Redirects

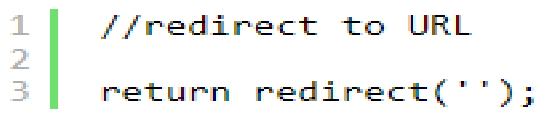
In Laravel, controllers and route definitions can return more than just views. One such return type is a redirect. Commonly, redirects are generated using the redirect() global helper or the facade, both creating an instance of Illuminate\Http\RedirectResponse. These approaches include convenience methods for easier use. Alternatively, you can manually create a redirect response, but this requires extra work. The example demonstrates various methods to return a redirect.



```
// Using the global helper to generate a redirect res
Route::get('redirect-with-helper', function () {
    return redirect()->to('login');
});

// Using the global helper shortcut
Route::get('redirect-with-helper-shortcut', function
    return redirect('login');
});

// Using the facade to generate a redirect response
Route::get('redirect-with-facade', function () {
    return Redirect::to('login');
});



// Using the Route::redirect shortcut
Route::redirect('redirect-by-route', 'login');
```

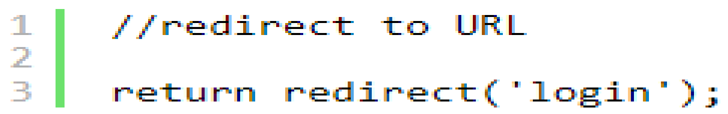
5.1Redirect to URL

This is the simplest and most straightforward redirect usage in Laravel. We can simply give the URL where we want to redirect. This will redirect to any URL given as a parameter inside the redirect function. A well-explained example is given below.



```
//redirect to URL

return redirect('');
```

This will simply redirect to the homepage. It's redirect output will be http://www.website.com/



```
//redirect to URL

return redirect('login');
```

This will redirect to the login page. Its redirect URL will be http://website.com/login

5.2Redirect Back

Laravel redirect helper function has a back() chain method. As the name suggests, it is used to redirect to the previous page. It comes in handy in multiple situations such as validation and operations. The below example will redirect back to the previous page.

```
//Redirect Back

return redirect()->back()
```

We can also redirect back with inputs as in the below example.

```
//Redirect back with inputs

return redirect()->back()->withInput($request);
```

In addition, we can also redirect back with a message as in the below example.

```
//Redirect back with message

return redirect()->with('success', 'Article successfully created.');
OR
return redirect()->withSuccess('Article successfully created.');
```

5.3 Redirect to Route

In the first example, we redirected to the unnamed route URL. But, we can also redirect to the named route. This is a more common practice as we can use a named route without any changes even if there are changes in the URL.

```
// Redirect to route

return redirect()->route('article');
```

We can also pass parameters when redirecting using a named route.

```
// Redirect to route with parameter

return redirect()->route('article', $id); //single parameter

return redirect()->route('article', [1,2,3,4]); //multiple parameter
```

5.4 Redirect to Controller Action

We can also redirect to a controller action. For this, we simply need to give a controller and action name. Using this redirect method, it doesn’t care what its URL is.

```
//Redirect to controller action

return redirect()->action('ArticleController@index');
```

In addition to this, we can also pass parameters to the action using the following method.

```
//Redirect to controller with parameter

return redirect()->action('ArticleController@index', [1,2,3,4]);
```

6. Blade Templating

PHP is effective as a templating language but can be cumbersome with its inline <?php syntax. Modern frameworks often provide their own templating languages to address this. Laravel, for instance, offers Blade, a templating engine influenced by .NET's Razor. Blade features a succinct syntax, is easy to learn, has a robust inheritance model, and is simple to extend. This makes writing in Blade more streamlined compared to traditional PHP.

```
<h1>{{ $group->title }}</h1>
{!! $group->heroImageHtml() !!}

@forelse ($users as $user)
    • {{ $user->first_name }} {{ $user->last_name }}
@empty
    No users in this group.
@endforelse
```

Blade uses curly braces for echoing and prefixes its custom tags, known as "directives," with an @ symbol. These directives are used for control structures, inheritance, and custom features. Blade's syntax is straightforward, making it user-friendly for simple tasks. However, its true potential is evident in more complex scenarios like nested inheritance, intricate conditionals, or recursion. Being part of Laravel, Blade simplifies complex requirements. As Blade syntax converts to PHP code and is cached, it's efficient. While you can use native PHP in Blade files, it's generally better to stick to Blade or custom directives for template-related tasks.

6.1. Echoing Data

In Blade, the {{ }} syntax is used for echoing PHP variables, similar to echo $variable in plain PHP. By default, Blade safely escapes outputs with PHP's htmlentities() function. This means {{ $variable }} in Blade is equivalent to echo htmlentities($variable); in PHP. To echo content without escaping, use {!! !!} instead. Blade's echo syntax resembles that of many frontend frameworks. To differentiate from other templating languages like Handlebars, Blade ignores {{ }} when it is preceded by an @. For instance, Blade will process {{ $example }}, but @{{ $example }} will output as-is without parsing.

```
// Parsed as Blade; the value of $bladeVariable
{{ $bladeVariable }}
```

```
// @ is removed and "{{ handlebarsVariable }}"
@{{ handlebarsVariable }}
```

6.2 Control Structures

Most of the control structures in Blade will be very familiar. Many directly echo the name and structure of the same tag in PHP, There are a few convenience helpers, but in general, the control structures just look cleaner than they would in PHP.

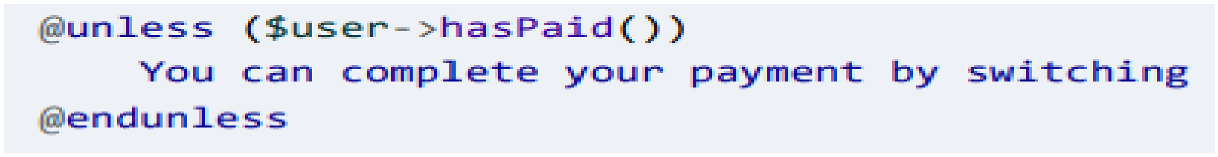
6.2.1 Conditionals

First, let's take a look at the control structures that allow for logic.
@if, @else, @elseif, and @endif

Blade's @if ($condition) compiles to . @else , @elseif , and @endif also compile to the exact same style of syntax in PHP. Take a look at Example .

```
@if (count($talks) === 1)
    There is one talk at this time period.
@elseif (count($talks) === 0)
    There are no talks at this time period.

@else
    There are {{ count($talks) }} talks at this time
```

Just like with the native PHP conditionals, you can mix and match these how you want. They don't have any special logic; there's literally a parser looking for something with the shape of @if ($condition) and replacing it with the appropriate PHP code.
@unless and @endunless
@unless , on the other hand, is a new syntax that doesn't have a direct equivalent in PHP. It's the direct inverse of @if . @unless ($condition) is the same as
<?php if(! $condition) You can see it in use in Example.

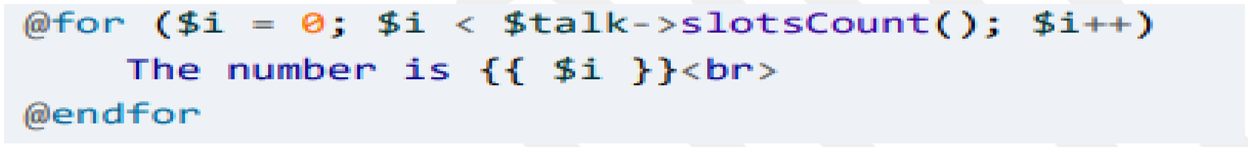


```
@unless ($user->hasPaid())
    You can complete your payment by switching
@endunless
```

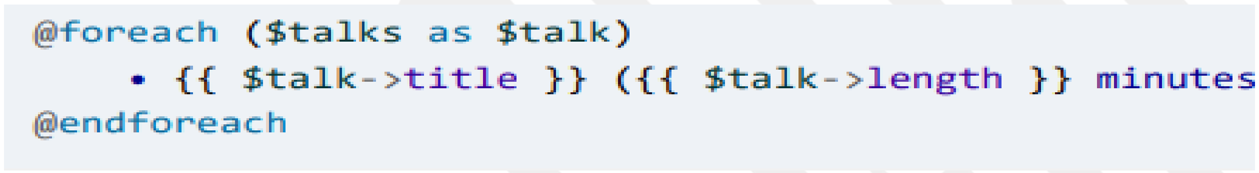
6.2.2 Loops

- **@for, @foreach, and @while**

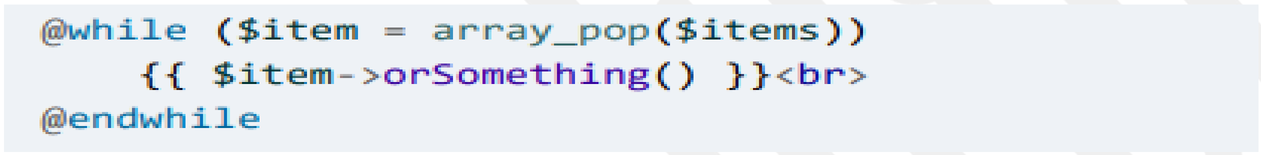
@for , @foreach , and @while work the same in Blade as they do in PHP;



```
@for ($i = 0; $i < $talk->slotsCount(); $i++)
    The number is {{ $i }}<br>
@endfor
```



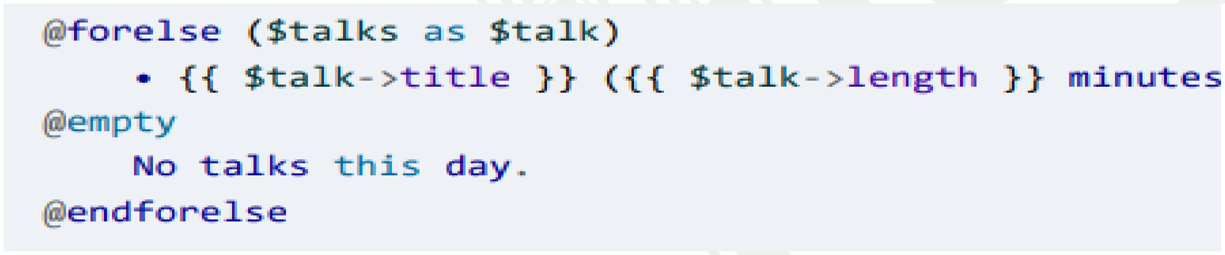
```
@foreach ($talks as $talk)
    • {{ $talk->title }} ({{ $talk->length }} minutes
@endforeach
```



```
@while ($item = array_pop($items))
    {{ $item->orSomething() }}<br>
@endwhile
```

- **@forelse and @endforelse**

@forelse is a @foreach that also allows you to program in a fallback if the object you’re iterating over is empty.



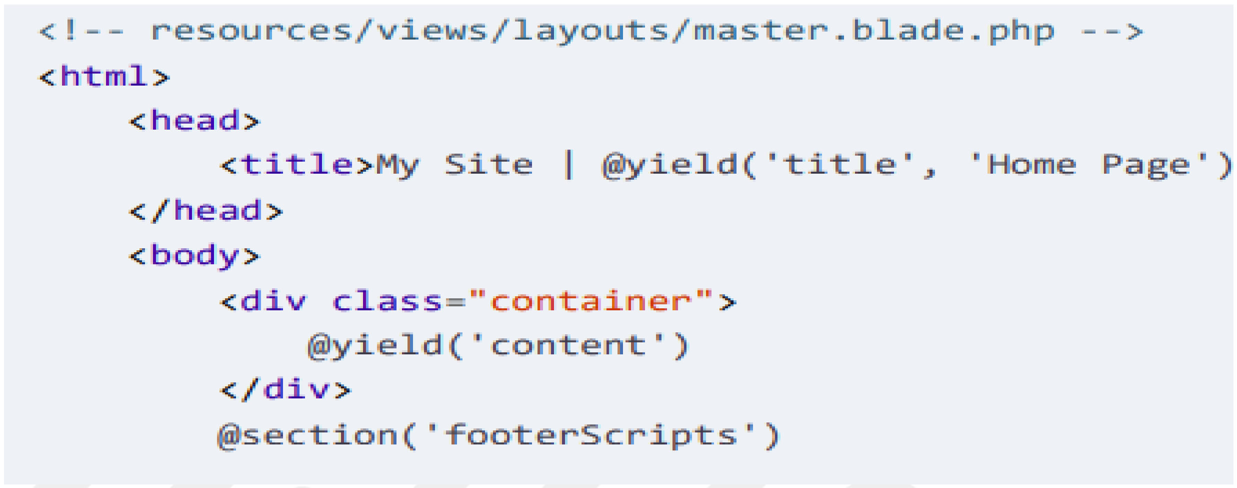
```
@forelse ($talks as $talk)
    • {{ $talk->title }} ({{ $talk->length }} minutes
@empty
    No talks this day.
@endforelse
```

7. Template Inheritance

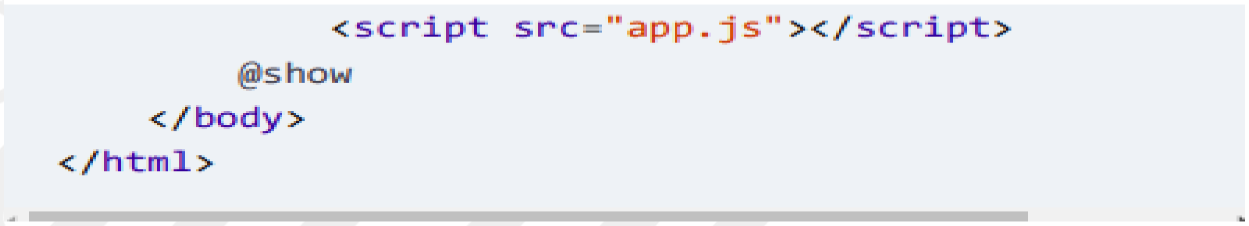
Blade provides a structure for template inheritance that allows views to extend, modify, and include other views. Let’s take a look at how inheritance is structured with Blade.

7.1. Defining Sections with @section/@show and @yield

Let’s start with a top-level Blade layout, like in Example . This is the definition of a generic page wrapper that we’ll later place page-specific content into.



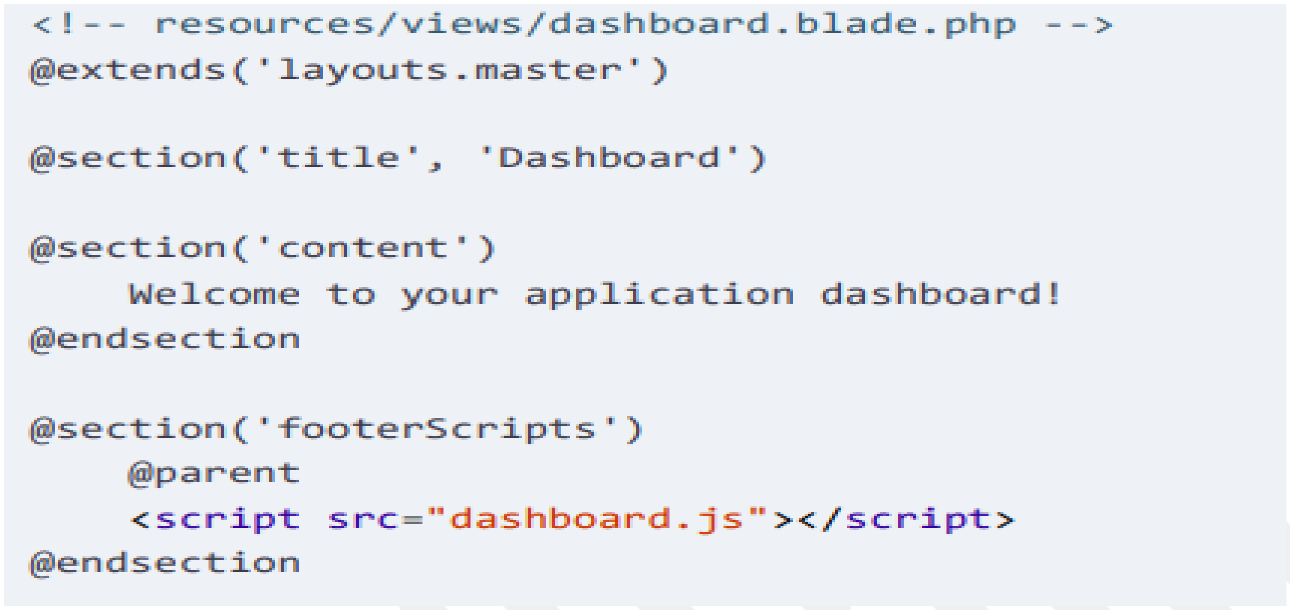
```
<!-- resources/views/layouts/master.blade.php -->
<html>
    <head>
        <title>My Site | @yield('title', 'Home Page')
    </head>
    <body>
        <div class="container">
            @yield('content')
        </div>
        @section('footerScripts')
```



```
            <script src="app.js"></script>
        @show
    </body>
</html>
```

In this HTML-like Blade template, we use three directives: @yield('content'), @yield('title', 'Home Page'), and @section/@show with @section/@show containing <script src="app.js"></script>. Each directive marks a section for later extension, handling unextended sections differently. @yield('content') provides no default, so it displays nothing if unextended. @yield('title', 'Home Page') displays 'Home Page' only if not extended, with child sections unable to access this default. @section/@show, however, sets a default that child sections can use via @parent, making its default content accessible to them.

Once you have a parent layout like this, you can extend it in a new template file like in Example:



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

@section('title', 'Dashboard')

@section('content')
    Welcome to your application dashboard!
@endsection

@section('footerScripts')
    @parent
    <script src="dashboard.js"></script>
@endsection
```

This child view allows us to cover a few new concepts in Blade inheritance.

8. Migations

Modern frameworks, such as Laravel, streamline database structuring through code-driven migrations. They allow for the coding of new tables, columns, indexes, and keys, enabling rapid transformation of a bare database into the app's precise schema.

8.1. Defining Migrations

A migration contains 'up' (apply changes) and optional 'down' (revert changes) methods. Migrations run in chronological order based on their filenames, like 2018_10_12_000000_create_users_table.php. The system executes the 'up()' method to migrate and 'down()' to roll back, undoing the previous changes.

8.2. Creating a migration

Laravel's command-line tools include a command for creating migration files, executed as php artisan make:migration [name]. For instance, php artisan make:migration create_users_table generates a migration for a User table. Optional flags include --create=table_name for creating a new table and --table=table_name for modifying an existing one..

8.3. Creating tables

our migrations depend on the Schema facade and its methods. Everything we can do in these migrations will rely on the methods of Schema .

To create a new table in a migration, use the create() method —the first parameter is the table name, and the second is a closure that defines its columns:

```
Schema::create('users', function (Blueprint $table) {
    // Create columns here
});
```

8.4. Creating columns

To create new columns in a table, whether in a create table call or a modify table call, use the instance of Blueprint that’s passed into your closure:

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

Blueprint instances in Laravel offer various methods for creating columns. These methods work in MySQL, and Laravel adapts them for other databases. Here's a concise overview:

- id(): Creates a big incrementing ID column (bigIncrements('id')).
- Integer Types:
- integer(colName), tinyInteger(colName), smallInteger(colName), mediumInteger(colName), bigInteger(colName): Standard integer columns.
- unsignedTinyInteger(colName), unsignedSmallInteger(colName), unsignedMediumInteger(colName), unsignedBigInteger(colName): Unsigned integer columns.
- string(colName, length): VARCHAR column with optional length.
- binary(colName): BLOB column.
- boolean(colName): BOOLEAN column (TINYINT(1) in MySQL).
- char(colName, length): CHAR column with optional length.
- Date and Time:
- date(colName), datetime(colName), dateTimeTz(colName): DATE or DATETIME columns, with dateTimeTz for timezone support.
- Decimal:
- decimal(colName, precision, scale), unsignedDecimal(colName, precision, scale): DECIMAL columns with precision and scale.
- double(colName, total, decimal): DOUBLE column with total digits and decimals.
- enum(colName, [choices]): ENUM column with specified choices.
- float(colName, precision, scale): FLOAT column (like DOUBLE in MySQL).
- Foreign Keys:
- foreignId(colName), foreignUuid(colName): UNSIGNED BIGINT or UUID columns.
- foreignIdFor(colName): UNSIGNED BIG INT column named colName_id.

8.5. Dropping tables

If you want to drop a table, there's a dropIfExists() method on Schema that takes one parameter, the table name:

```
Schema::dropIfExists('contacts');
```

8.6. Modifying columns

To modify a column, just write the code you would write to create the column as if it were new, and then append a call to the change() method after it.

So, if we have a string column named name that has a length of 255 and we want to change its length to 100 , this is how we would write it:

```
Schema::table('users', function (Blueprint $table) {
    $table->string('name', 100)->change();
});
```

Here's how we rename a column:

```
Schema::table('contacts', function (Blueprint $table)
{
    $table->renameColumn('promoted', 'is_promoted');
});
```

And this is how we drop a column:

```
Schema::table('contacts', function (Blueprint $table)
{
    $table->dropColumn('votes');
});
```

8.7. Squashing Migrations

If you have too many migrations to reason with, you can merge them all into a single SQL file that Laravel will run before it runs any future migrations. This is called "squashing" your migrations.

```
// Squash the schema but keep your existing migrat
php artisan schema:dump

// Dump the current database schema and delete all
php artisan schema:dump --prune
```

8.8. Indexes and foreign keys

We've covered how to create, modify, and delete columns. Let's move on to indexing and relating them. If you're not familiar with indexes, your databases can survive if you just never use them, but they're pretty important for performance optimization and for some data integrity controls with regard to related tables. I'd recommend reading up on them, but if you absolutely must.

To Adding indexes:

```
// After columns are created...
$table->primary('primary_id'); // Primary key; unnece
$table->primary(['first_name', 'last_name']); // Comp
$table->unique('email'); // Unique index
$table->unique('email', 'optional_custom_index_name')
$table->index('amount'); // Basic index
$table->index('amount', 'optional_custom_index_name')
```

Note that the first example, primary() , is not necessary if you're using the increments() or bigIncrements() methods to create your index; this will automatically add a primary key index for you, We can remove indexes as shown in Example:

```
$table->dropPrimary('contacts_id_primary');
$table->dropUnique('contacts_email_unique');
```

To add a foreign key that defines that a particular column references a column on another table, Laravel's syntax is simple and clear:

```
$table->foreign('user_id')
     ->references('id')
     ->on('users')
     ->cascadeOnDelete();
```

Here we're adding a foreign index on the user_id column, showing that it references the id column on the users table. Couldn't get much simpler. If we want to specify foreign key constraints, we can do that too, with cascadeOnUpdate() , restrictOnUpdate() , cascadeOnDelete() , restrictOnDelete() , and nullOnDelete() .

to drop a foreign key, we can either delete it by referencing its index name (which is automatically generated by combining the names of the columns and tables being referenced):

```
$table->dropForeign('contacts_user_id_foreign');
```

8.9. Running Migrations

Once you have your migrations defined, how do you run them? There's an Artisan command for that:

```
php artisan migrate
```

The migrate command in Laravel runs any pending migrations, executing their up() methods. It tracks which migrations have been executed and only runs those that are outstanding.
Other useful migration commands include:

- migrate:install: Initializes the database table to track migrations. Typically, it's automatically executed with migrations, so it's rarely needed separately.
- migrate:reset: Reverts all migrations applied to the database.
- migrate:refresh: Reverts and then re-applies all migrations.
- migrate:fresh: Deletes all tables and re-applies all migrations, skipping the down() method of migrations.
- migrate:rollback: Reverts the most recent batch of migrations, or a specified number using --step=n.
- migrate:status: Displays the status of each migration, indicating whether it's been run or not.

For database inspection, Laravel offers Artisan commands:

- db:show: Provides an overview of the entire database, including tables and connection details.
- db:table: When given a table name, shows details like size and columns.

db:monitor: Not specified, but likely used for monitoring database performance or status.

8.10. Seeding

Seeding with Laravel is so simple, it has gained widespread adoption as a part of normal development workflows in a way it hasn't in previous PHP frameworks. There's a _database/seeders folder that comes with a DatabaseSeeder class, which has a run() method that is called when you call the seeder. There are two primary ways to run the seeders: along with a migration, or separately.

To run a seeder along with a migration, just add --seed to any migration call:

```
php artisan migrate --seed
php artisan migrate:refresh --seed
```

And to run it independently:

```
php artisan db:seed
php artisan db:seed VotesTableSeeder
```

This will call the run() method of the DatabaseSeeder by default, or the seeder class specified when you pass in a class name.

To create a seeder, use the make:seeder Artisan command:

```
php artisan make:seeder ContactsTableSeeder
```

You'll now see a ContactsTableSeeder class show up in the database/seeders directory. Before we edit it, let's add it to the DatabaseSeeder class, as shown in Example, so it will run when we run our seeders.

```
// database/seeders/DatabaseSeeder.php
...
public function run(): void
{
    $this->call(ContactsTableSeeder::class);
}
```

Now let's edit the seeder itself. The simplest thing we can do there is manually insert a record using the DB facade, as illustrated in next example.

```php
<?php

namespace Database\Seeders;

use Illuminate\Database\Seeder;

class ContactsTableSeeder extends Seeder
{
    public function run(): void
    {
        DB::table('contacts')->insert([
            'name' => 'Lupita Smith',
            'email' => 'lupita@gmail.com',
        ]);
    }
}
```

This will get us a single record, which is a good start. But for truly functional seeds, you'll likely want to loop over some sort of random generator and run this insert() many times, right? Laravel has a feature for that.

Lesson Five : Eloquent and Practical Exercise

1. Introduction to Eloquent

Eloquent is an ActiveRecord ORM that simplifies database interactions by offering a unified interface for various database types. It enables operations on both table-level (like retrieving all users with User::all()) and row-level (like managing a single user instance, $sharon). Instances in Eloquent can handle their own data persistence, allowing operations like $sharon->save() or $sharon->delete(). Emphasizing simplicity, Eloquent follows Laravel's "convention over configuration" principle, enabling powerful model creation with minimal coding.

2. Creating and Defining Eloquent Models

First, let's create a model. There's an Artisan command for that:

```
php artisan make:model Contact
```

This is what we'll get, in app/Models/Contact.php:

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Contact extends Model
{
    //
}
```

The default behavior for table names is that Laravel "snake cases" and pluralizes your class name, so SecondaryContact would access a table named secondary_contacts . If you'd like to customize the name, set the $table property explicitly on the model:

```
    protected $table = 'contacts_secondary';
```

Laravel assumes, by default, that each table will have an autoincrementing integer primary key, and it will be named id . If you want to change the name of your primary key, change the $primaryKey property:

```
    protected $primaryKey = 'contact_id';
```

And if you want to set it to be nonincrementing, use:

```
    public $incrementing = false;
```

To manage growing project complexity, use the model:show command for a model summary. This includes database and table names, attribute details (with SQL column modifiers, type, and size), mutators, model relations, and observers. Note: Eloquent assumes tables have created_at and updated_at timestamps. If not used, disable $timestamps.

```
    public $timestamps = false;
```

You can customize the format Eloquent uses to store your timestamps to the database by setting the $dateFormat class property to a custom string. The string will be parsed using PHP's date() syntax, so the following example will store the date as seconds since the Unix epoch:

```
    protected $dateFormat = 'U';
```

3. Retrieving Data with Eloquent

Most of the time you pull data from your database with Eloquent, you'll use static calls on your Eloquent model. Let's start by getting everything:

```
$allContacts = Contact::all();
```

That was easy. Let's filter it a bit:

```
$vipContacts = Contact::where('vip', true)->get();
```

We can see that the Eloquent facade gives us the ability to chain constraints, and from there the constraints get very familiar:

```
$newestContacts = Contact::orderBy('created_at',
    ->take(10)
    ->get();
```

It turns out that once you move past the initial facade name, you're just working with Laravel's query builder. You can do a lot more— we'll cover that soon— but everything you can do with the query builder on the DB facade you can do on your Eloquent objects.

3.1 Get One vs Get Many

You can use first() to get the first query record, or find() for a specific ID record. Adding "OrFail" to these methods (as in firstOrFail() or findOrFail()) throws an exception if no match is found. findOrFail() is often used for URL segment entity lookups, throwing an exception if the entity isn't found. For example: $newestContacts = Contact::orderBy('created_at')->take(10)->get();.

```
public function show($contactId)

{

    return view('contacts.show')

        ->with('contact', Contact::findOrFail($contactId));

}
```

Any method intended to return a single record (first() , firstOrFail() , find() , or findOrFail()) will return an instance of the Eloquent class. So, Contact::first() will return an instance of the class Contact with the data from the first row in the table filling it out. To get many rows use get() which works with Eloquent just like it does in normal query builder calls—build a query and call get() at the end to get the results:

```
$vipContacts = Contact::where('vip', true)->get();
```

However, there is an Eloquent-only method, all() , which you'll often see people use when they want to get an unfiltered list of all data in the table:

```
$contacts = Contact::all();
```

4. Inserts and Updates with Eloquent

Inserting and updating values is one of the places where Eloquent starts to diverge from normal query builder syntax.

4.1 Inserts

There are two primary ways to insert a new record using Eloquent.

First, you can create a new instance of your Eloquent class, set your properties manually, and call save() on that instance, like in Example**.**

```
$contact = new Contact;
$contact->name = 'Ken Hirata';
$contact->email = 'ken@hirata.com';
$contact->save();

// or

$contact = new Contact([
    'name' => 'Ken Hirata',
    'email' => 'ken@hirata.com',
]);
$contact->save();

// or

$contact = Contact::make([
    'name' => 'Ken Hirata',
    'email' => 'ken@hirata.com',
]);
$contact->save();
```

Until saved using save(), a Contact instance exists without being stored in the database. It lacks an id, won't persist if the app closes, and misses created_at and updated_at timestamps. Alternatively, Model::create() directly saves a new instance to the database, in contrast to make(), which only creates a model instance without saving it.

```
$contact = Contact::create([
    'name' => 'Keahi Hale',
    'email' => 'halek481@yahoo.com',
]);
```

4.2 Updates

Updating records looks very similar to inserting. You can get a specific instance, change its properties, and then save, or you can make a single call and pass an array of updated properties. Example illustrates the first approach.

```
$contact = Contact::find(1);
$contact->email = 'natalie@parkfamily.com';
$contact->save();
```

Since this record already exists, it will already have a created_at timestamp and an id , which will stay the same, but the updated_at field will be changed to the current date and time , example illustrates the second approach.

passing an array to the update() method

```
Contact::where('created_at', '<', now()->subYear())
    ->update(['longevity' => 'ancient']);

// or

$contact = Contact::find(1);
$contact->update(['longevity' => 'ancient']);
```

This method expects an array where each key is the column name and each value is the column value.
We've looked at a few examples of how to pass arrays of values into Eloquent class methods. However, none of these will actually work until you define which fields are "fillable" on the model. The goal of this is to protect you from (possibly malicious) user input accidentally setting new values on fields you don't want changed. Consider the common scenario in Example.

```
// ContactController
public function update(Contact $contact, Request $request)
{
    $contact->update($request->all());
}
```

Thankfully, that won't actually work until you define your model's fillable fields. You can either whitelist the fillable fields, or blacklist the "guarded" fields to determine which fields can or cannot be edited via "mass assignment"—that is, by passing an array of values into either create() or update() . Note that nonfillable properties can still be changed by direct assignment (e.g., $contact->password = 'abc';). Example shows both approaches.

```
class Contact extends Model
{
        protected $fillable = ['name', 'email'];

        // or

        protected $guarded = ['id', 'created_at', 'updated_at',
  'owner_id'];
}
```

5. Deleting with Eloquent

Deleting with Eloquent is very similar to updating with Eloquent, but with (optional) soft deletes, you can archive your deleted items for later inspection or even recovery.

5.1 Normal Deletes

The simplest way to delete a model record is to call the delete() method on the instance itself:

```
$contact = Contact::find(5);
$contact->delete();
```

However, if you only have the ID, there's no reason to look up an instance just to delete it; you can pass an ID or an array of IDs to the model's destroy() method to delete them directly:

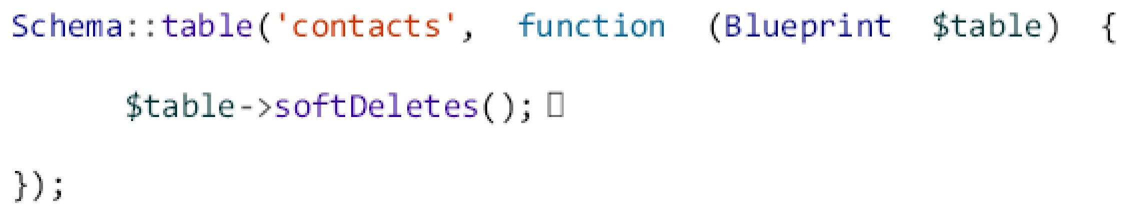
```
Contact::destroy(1);
// or
Contact::destroy([1, 5, 7]);
```

Finally, you can delete all of the results of a query:

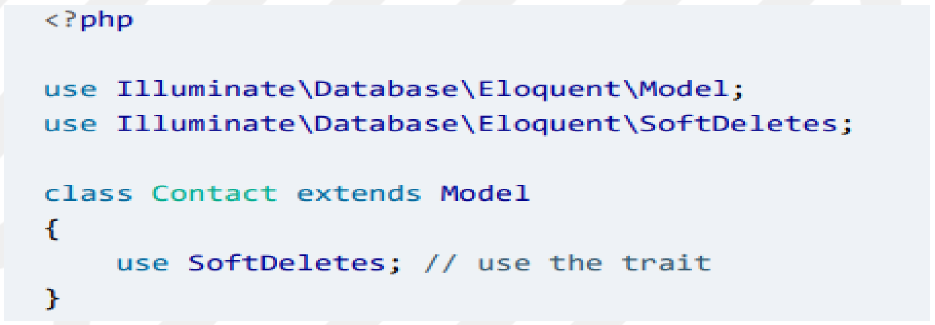
```
Contact::where('updated_at', '<', now()->subYear())->delete();
```

5.2 Soft Deletes

Soft deletes in databases mark rows as deleted without actually removing them. This allows for later inspection and potential restoration of data. In applications using Eloquent's soft deletes, queries automatically exclude soft-deleted data, unless explicitly included. To implement soft deletes, you need a deleted_at column in your table. While using this feature, remember it can lead to larger databases over time if not managed properly. To enable soft deletes, add the deleted_at column via migration and use the SoftDeletes trait in your model. Eloquent provides a softDeletes() method for schema building to easily add this column.

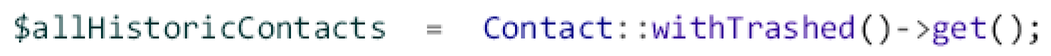


```
Schema::table('contacts', function (Blueprint $table) {

    $table->softDeletes();

});
```



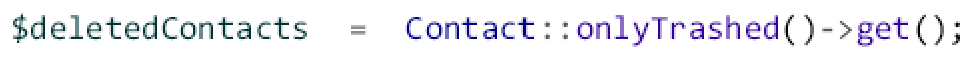
```php
<?php

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Contact extends Model
{
    use SoftDeletes; // use the trait
}
```

Once you make these changes, every delete() and destroy() call will now set the deleted_at column on your row to be the current date and time instead of deleting that row. And all future queries will exclude that row as a result.
So, how do we get soft-deleted items? First, you can add soft-deleted items to a query:



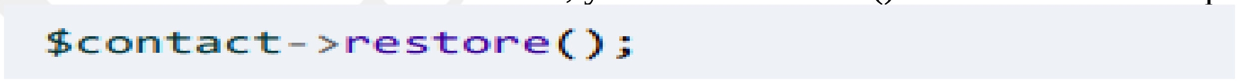
```
$allHistoricContacts = Contact::withTrashed()->get();
```

Next, you can use the trashed() method to see if a particular instance has been soft-deleted:



```
$deletedContacts = Contact::onlyTrashed()->get();
```

If you want to restore a soft-deleted item, you can run restore() on an instance or a query:



```
$contact->restore();
```