

**DEBRE BIRHAN UNIVERSITY**

**COLLEGE OF COMPUTING**

**COMPUTER SCIENCE DEPARTMENT**

**INDIVIDUAL ASSIGNNMENT OF SELECTED TOPIC IN COMPUTER SCIENCE**

**BY KIRUBEL WONDWOSEN**

**ID DBUE/793/11**

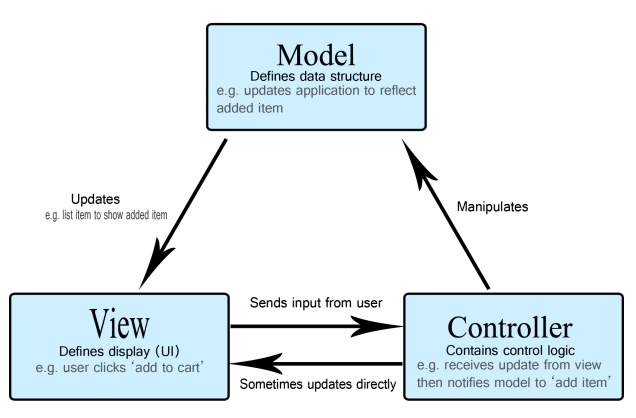
**JANUARY 2023**

**SUBMITTED TO INSTRUCTOR GIRMACHEW**

1. **Model-View-Controller (MVC) for laravel**

A **Model** is a representation of a real-life instance or object in our code base. The **View** represents the interface through which the user interacts with our application. When a user takes an action, the **Controller** handles the action and updates the **Model** if necessary.

Let’s look at a simple scenario.

If you go to an e-commerce website, the different pages you see are provided by the **View layer**. When you click on a particular product to view more, the **Controller layer** processes the user’s action. This may involve getting data from a data source using the **Model layer**. The data is then bundled up together and arranged in a **View layer** and displayed to the user. Rinse and repeat.

The three parts of the **MVC software-design pattern** can be described as follows:

* **Model**

The Model component corresponds to all the data-related logic that the user works with. This can represent either the data that is being transferred between the View and Controller components or any other business logic-related data. For example, a Customer object will retrieve the customer information from the database, manipulate it and update it data back to the database or use it to render data.

* **View**

The View component is used for all the UI logic of the application. For example, the Customer view will include all the UI components such as text boxes, dropdowns, etc. that the final user interacts with.

* **Controller**

Controllers act as an interface between Model and View components to process all the business logic and incoming requests, manipulate data using the Model component and interact with the Views to render the final output.

1. ***Laravel Routing***

**Routing** is one of the essential concepts in Laravel. Routing in Laravel allows us to route all your application requests to their appropriate controller. The main and primary routes in Laravel acknowledge and accept a URI (Uniform Resource Identifier) along with a closure, given that it should have to be a simple and expressive way of routing.

Laravel provides **two ways** of capturing the passed parameter:

* **Required Parameters**

We sometimes had to work with a segment(s) of your project's URL(Uniform Resource Locator). Route parameters are encapsulated within **{} (curly-braces)** with alphabets inside. Let us take an example where you have to capture the customer's ID or employee from the generated URL.

**Example:**

**Route :: get ('emp/{id}', function ($id) { echo 'Emp '.$id; });**

* **Optional Parameter**

Many parameters do not remain present within the URL, but the developers had to use them. So such parameters get indicated by a **" ?" (Question mark sign)** following the parameter's name.

**Example:**

**Route :: get ('emp/{desig?}', function ($desig = null) {echo $desig; });**

**Route :: get ('emp/{name?}', function ($name = 'Guest') {echo $name;});**

1. **Migration and relationships**

***Migration***

***Laravel Migration*** is an essential feature in Laravel that allows you to create a table in your database. It allows you to modify and share the application's database schema. You can modify the table by adding a new column or deleting an existing column.

Laravel migration is like version management tool of database. ***Laravel database migrations*** allow developers to quickly bootstrap, destroy, and recreate an application’s database, without the need to log into the database console or run any SQL queries.

To generate a migration you need run a command

***php artisan make:migration create\_contacts\_table***

This will generate a file in ***database\migrations*** folder. The file consist of new class extending the migration class of LARAVEL.

The new class consist of 2 major function **up()** & **down().**

* The **up()** function holds all information about migrating the file.

**public function up()**

**{**

**Schema::create('contacts', function (Blueprint $table)**

**{**

**$table->id();**

**$table->string('name');**

**$table->string('mobile\_no');**

**$table->boolean('status');**

**$table->timestamps();**

**});**

**}**

* Whereas **down()** function holds information about ***reversing*** the migration action.

**public function down()**

**{**

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

**}**

**Run & Rollback The Migration**

To run a migration we need to use the command

***php artisan migrate***

For rolling back latest migration we have command

***php artisan migrate:rollback***

When we have to rollback till specific steps we can pass steps in rollback command like

***php artisan migrate:rollback --step=3***

*(This will rollback the migration up to 3 step starting from latest.)*

***Relationships***

An **Eloquent relationship** is a very important feature in Laravel that allows you to relate the tables in a very easy format.

***There are three different types of relationships:***

* **One to one**

The one to one relationship is the most basic relation that exists. This type of relationship means that a model of type A may only be linked to one model of type B, and vice versa. Let’s take a look at how we would define this relationship in code.

**<?php**

**namespace App;**

**use Illuminate\Database\Eloquent\Model;**

**class User extends Model**

**{**

**public function passport() {**

**return $this->hasOne(App\Passport::class);**

**}}**

**?>**

In the User model we create a method called passport. We tell Laravel that the User model has one Passport by using the **hasOne** method.

* **One to many**

The next relationship that you can define in Laravel is a one to many relationship. This type of relationship means that one model of type A may be linked to multiple models of type B. But the model of type B belongs to only one model of type A.

In code this would look like this:

**<?php**

**namespace App;**

**use Illuminate\Database\Eloquent\Model;**

**class User extends Model**

**{**

**public function invoices() {**

**return $this->hasMany(App\Invoice::class);**

**}**

**}**

**?>**

It looks just like the code that we saw earlier to define the one to one relationship, right?

All we have to do now is let the Invoice model know that it belongs to a User model. Let’s define the inverse of the one to many relationship.

* **Many to many**

The last relationship is the many to many relationship. This type of relationship means that one model of type A may be linked to multiple models of type B, and vice versa.

**<?php**

**namespace App;**

**use Illuminate\Database\Eloquent\Model;**

**class Invoice extends Model**

**{**

**public function products() {**

**return $this->belongsToMany(App\Product::class);**

**}**

**}**

**?>**

***And you can define the inverse of this relationship like this:***

**<?php**

**namespace App;**

**use Illuminate\Database\Eloquent\Model;**

**class Product extends Model**

**{**

**public function invoices() {**

**return $this->belongsToMany(App\Invoice::class);**

**}}**

**?>**

**Linking a model through another model**

* **Has one through**

The ***“has one through”*** relationship links models through a single intermediate relation. Imagine the situation where each product has one supplier and each product has one product history record. Then the supplier model can access the product history record through the product.

**<?php**

**namespace App;**

**use Illuminate\Database\Eloquent\Model;**

**class Supplier extends Model**

**{**

**public function productHistory() {**

**return $this->hasOneThrough(App\History::class, App\Product::class);**

**}**

**}**

**?>**

The first argument that is passed to the hasOneThrough method is the name of the final model. The second argument is the name of the intermediate model.

* **Has many through**

The “has many through” relationship is the equivalent of the “has one through” relationship, but for multiple records.

**<?php**

**namespace App;**

**use Illuminate\Database\Eloquent\Model;**

**class Supplier extends Model**

**{**

**public function productHistory() {**

**return $this->hasManyThrough(App\History::class, App\Product::class);**

**}**

**}**

**?>**

This way the supplier model can get access to the ***history entries of the product***.

1. **Blade Template Engine**

The ***Blade*** is a powerful templating engine in a Laravel framework. The blade allows to use the templating engine easily, and it makes the syntax writing very simple. The blade templating engine provides its own structure such as conditional statements and loops. To create a blade template, you just need to create a view file and save it with a ***“.blade.php”*** extension instead of ***“.php”*** extension. The blade templates are stored in the ***“/resources/view”*** directory. The main advantage of using the blade template is that we can create the ***master template***, which can be extended by other files.

When compared to other templating engines, Blade is unique in the following ways:-

* It does not restrict the developer from using plain PHP code in views.
* The blade views thus designed, are compiled and cached until they are modified.

The complete directory structure of Laravel is shown in the screenshot given here.

We can observe that all views are stored in the resources/views directory and the default view for Laravel framework is ***welcome.blade.php***.

***Steps for Creating a Blade Template Layout***

You will have to use the following steps to create a blade template layout −

* ***Step 1***

Create a layout folder inside the resources/views folder. We are going to use this folder to store all layouts together.

Create a file name master.blade.php which will have the following code associated with it −

**<html>**

**<head>**

**<title>DemoLaravel - @yield('title')</title>**

**</head>**

**<body>**

**@yield('content')**

**</body>**

**</html>**

* ***Step 2***

In this step, you should extend the layout. Extending a layout involves defining the child elements. Laravel uses the Blade @extends directive for defining the child elements.

When you are extending a layout, please note the following points − Views defined in the Blade Layout injects the container in a unique way. Various sections of view are created as child elements. Child elements are stored in layouts folder as child.blade.php

An example that shows extending the layout created above is shown here −

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

**@section('title', 'Page Title')**

**@section('sidebar')**

**@parent**

**<p>This refers to the master sidebar.</p>**

**@endsection**

**@section('content')**

**<p>This is my body content.</p>**

**@endsection**

* **Step 3**

To implement the child elements in views, you should define the layout in the way it is needed. We can also create them as child elements with the help of blade templates by using the procedure given above. Named route is used to give specific name to a route. The name can be assigned using the “as” array key.

**Route::get('user/profile', ['as' => 'profile', function () {**

**//**

**}]);**

1. **Directives**

**The App Directory**

The app directory contains the core code of your application. We'll explore this directory in more detail soon; however, almost all of the classes in our application will be in this directory.

**The Bootstrap Directory**

The bootstrap directory contains the app.php file which bootstraps the framework. This directory also houses a cache directory which contains framework generated files for performance optimization such as the route and services cache files. We should not typically need to modify any files within this directory.

**The Database Directory**

The database directory contains your database migrations, model factories, and seeds. If you wish, you may also use this directory to hold an SQLite database.

**The Public Directory**

The public directory contains the index.php file, which is the entry point for all requests entering your application and configures autoloading. This directory also houses your assets such as images, JavaScript, and CSS.

**The Resources Directory**

The resources directory contains your views as well as your raw, un-compiled assets such as CSS or JavaScript.

**The Routes Directory**

The routes directory contains all of the route definitions for your application. By default, several route files are included with Laravel: web.php, api.php, console.php, and channels.php.

The web.php file contains routes that the RouteServiceProvider places in the web middleware group, which provides session state, CSRF protection, and cookie encryption. If your application does not offer a stateless, RESTful API then all your routes will most likely be defined in the web.php file.

The api.php file contains routes that the RouteServiceProvider places in the api middleware group. These routes are intended to be stateless, so requests entering the application through these routes are intended to be authenticated via tokens and will not have access to session state.

The console.php file is where you may define all of your closure based console commands. Each closure is bound to a command instance allowing a simple approach to interacting with each command's IO methods. Even though this file does not define HTTP routes, it defines console based entry points (routes) into your application.

The channels.php file is where you may register all of the event broadcasting channels that your application supports.

**The Tests Directory**

The tests directory contains your automated tests. Example PHPUnit unit tests and feature tests are provided out of the box. Each test class should be suffixed with the word Test. You may run your tests using the phpunit or php vendor/bin/phpunit commands. Or, if you would like a more detailed and beautiful representation of your test results, you may run your tests using the php artisan test Artisan command.

**The Vendor Directory**

The vendor directory contains your Composer dependencies.

**The Blade Directives**

***Blade directives*** are shortcut codes for the implementation of basic PHP structure control, such as loop and conditional statements. It makes your code snippets clean and easy to understand.

Note: These directives are basically only used in the blade template, so don’t try to use them on our controllers.

**The if blade directive**

**@if()**

**@elseif()**

**@else**

**@endif**

The if(){} statement is similar to the endif statement, except that the endif statement will serve as the closing {} curly braces.

The if(){} statement can be used like so:

**@if(1=1)**

**{{ one will always be one.lol }}**

**@endif**

**The Auth blade directive**

**@auth**

**@endauth**

This blade directive is used to check if a particular user has been authenticated. Refer to this shot for more information on this.

* ***Reference***
* [***https://www.geeksforgeeks.org/introduction-to-laravel-and-mvc-framework/***](https://www.geeksforgeeks.org/introduction-to-laravel-and-mvc-framework/)
* [***https://blog.pusher.com/laravel-mvc-use/***](https://blog.pusher.com/laravel-mvc-use/)
* [***https://www.cloudways.com/blog/routing-in laravel/#:~:text=Routing%20in%20Laravel%20allows%20users,and%20expressive%20way%20of%20routing***](https://www.cloudways.com/blog/routing-in%20laravel/#:~:text=Routing%20in%20Laravel%20allows%20users,and%20expressive%20way%20of%20routing)
* [***https://dev.to/techtoolindia/how-to-create-migration-in-laravel--2532***](https://dev.to/techtoolindia/how-to-create-migration-in-laravel--2532)
* [***https://www.javatpoint.com/laravel-migration***](https://www.javatpoint.com/laravel-migration)
* [***https://www.javatpoint.com/laravel-blade-template***](https://www.javatpoint.com/laravel-blade-template)
* [***https://www.tutorialspoint.com/laravel/laravel\_redirections.htm***](https://www.tutorialspoint.com/laravel/laravel_redirections.htm)
* [***https://www.educative.io/answers/what-are-blade-directives***](https://www.educative.io/answers/what-are-blade-directives)
* [***https://laravel.com/docs/9.x/structure#the-database-directory***](https://laravel.com/docs/9.x/structure#the-database-directory)
* [***https://www.tutorialspoint.com/laravel/laravel\_application\_structure.htm***](https://www.tutorialspoint.com/laravel/laravel_application_structure.htm)