Selected computer science

Individual Assignment

INSTRUCTOR: Girmachew

PREPARED BY: HAILEYESUS MAMO

ID#DBUE/788/11

26/1/2023

1.Explain MVC of Laravel

What Is MVC

MVC stands for **Model View Controller structure**. This is a very useful development structure and also very popular in the market. Laravel has these features and this is why Laravel is one of the most used frameworks now.

What is Model in Laravel

Model is where we perform the database-related operations. Like, insert, delete, edit, update query, etc. The model folder exists in the App folder in Laravel 8. Here you have to create models corresponding to the database tables.

Like if your table name is users, where you want to do operations through a model then the model name will be User.

What is View

The view is where we store our front-end templates. These are called blade templates. What is Controller

The controller is the central part of the MVC model. The controller controls the model and views. Also, this is responsible for the Application logic. The controller takes the user input values from the view and passes the values to the model. Then also return the data from the model to view.

2. Explain Routing

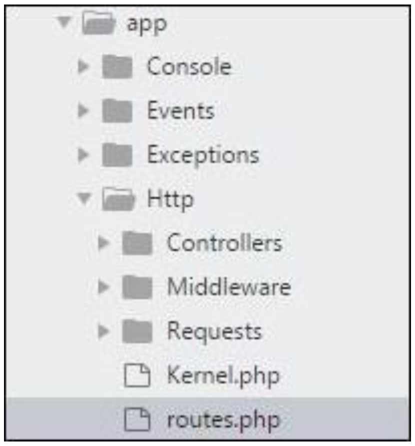
In Laravel, all requests are mapped with the help of routes. Basic routing routes the request to the associated controllers. This chapter discusses routing in Laravel.

Routing in Laravel includes the following categories −

- Basic Routing
- Route parameters
- Named Routes

Basic Routing

All the application routes are registered within the **app/routes.php** file. This file tells Laravel for the URIs it should respond to and the associated controller will give it a particular call. The sample route for the welcome page can be seen as shown in the screenshot given below −



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

Example

Observe the following example to understand more about Routing − **app/Http/routes.php**

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

resources/view/welcome.blade.php

```
<!DOCTYPE html>
<html>
   <head>
      <title>Laravel</title>
      <link href = "https://fonts.googleapis.com/css?family=Lato:100" rel = "stylesheet"          type =
"text/css">

      <style>         html, body {
            height: 100%;
         }          body {              margin: 0;
padding: 0;
```

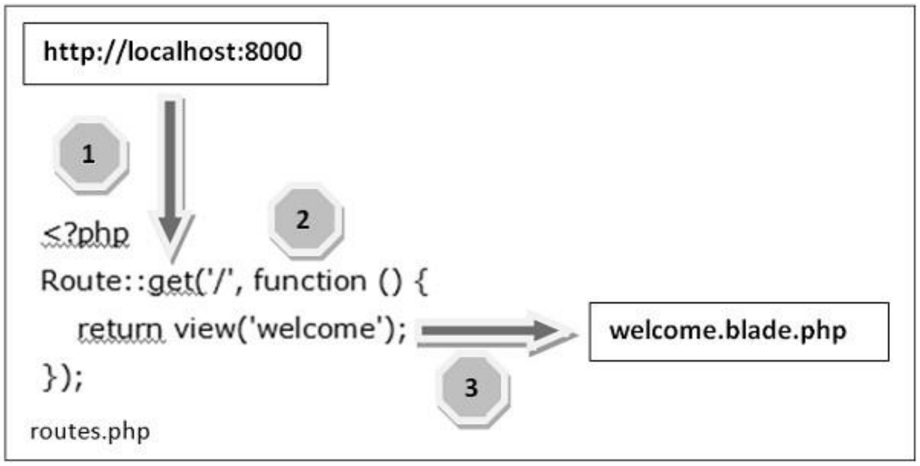
```
            width: 100%;
display: table;             font-
weight: 100;
            font-family: 'Lato';
         }
         .container {
text-align: center;
display: table-cell;
            vertical-align: middle;
         }
         .content {
            text-align: center;
display: inline-block;
         }
         .title {
            font-size: 96px;
         }
      </style>
   </head>

   <body>
      <div class = "container">

         <div class = "content">
            <div class = "title">Laravel 5.1</div>
         </div>

      </div>
   </body>
</html>
```

The routing mechanism is shown in the image given below −



3.Explain Migration and relationships

Laravel Relationships Migration Schema

- I’ve been having to look up relationship schemas too frequently and haven’t found a decent place that outlines them with migration and eloquent relationship declaration side by side. So if you’re looking for how to build a migration for (insert here) relationship, here it is.
- *Based on Laravel 7

- ### One to One

```
// Model A Migration
public function up()
    {
        Schema::table('A', function (Blueprint $table) {
            $table->unsignedBigInteger('id');
        });
    }

// Model B Migration
public function up()
    {
        Schema::table('B', function (Blueprint $table) {
            $table->unsignedBigInteger('id');
            $table->unsignedBigInteger('A_id');
            $table->foreign('A_id')->references('id')->on('A');
        });
    }

// A Model
public function b(){
    $this->hasOne(B::class);
}

// B Model
public function a(){
    $this->belongsTo(A::class);
}
```

- ### One to Many
- The migration looks the same as One to One, and similar in the Eloquent model. Just change ‘hasOne’ in the A class to ‘hasMany’

```
// Model A Migration
```

```
public function up()
    {
        Schema::table('A', function (Blueprint $table) {
            $table->unsignedBigInteger('id');
        });
    }

// Model B Migration
public function up()
    {
        Schema::table('B', function (Blueprint $table) {
            $table->unsignedBigInteger('id');
            $table->unsignedBigInteger('A_id');
            $table->foreign('A_id')->references('id')->on('A');
        });
    }

// A Model
public function b(){
    $this->hasMany(B::class);
}

// B Model
public function a(){
    $this->belongsTo(A::class);
}
```

- ## Many to Many
- Many to many relationships requires a third table called a pivot table that links the two.

```
// Model A Migration
public function up()
    {
        Schema::table('Amodel', function (Blueprint $table) {
            $table->unsignedBigInteger('id');
        });
    }

// Model B Migration
public function up()
    {
        Schema::table('Bmodel', function (Blueprint $table) {
            $table->unsignedBigInteger('id');
        });
    }

// Pivot Table migration named A_B_table
// naming structure is important in Laravel, the model names are singular
// and in alphabetical order
public function up()
    {
        Schema::table('A_B', function (Blueprint $table) {
            $table->unsignedBigInteger('A_id');
            $table->foreign('A_id')->references('id')->on('A');
            $table->unsignedBigInteger('B_id');
            $table->foreign('B_id')->references('id')->on('B');
        });
    }

// A Model
public function b(){
    $this->belongsToMany(B::class);
}

// B Model
public function a(){
    $this->belongsToMany(A::class);
}
```

4.Explain Blade template engine.

Laravel is one of the best PHP framework which is highly acknowledged for its **inbuilt lightweight templates** that help you **create amazing layouts using dynamic content seeding**.

Blade view files use the `.blade.php` file extension and are typically stored in the `resources/views` directory.

# Inheritance

Two of the primary benefits of using Blade are *template inheritance* and *sections*. We can define a blade page as a combination of layout and sections.

Since most of the general web applications will have the same layout across the web pages. In this example, we will examine the “master layout” with sections like “Top nav”, “Sidenav” and the “body content”.

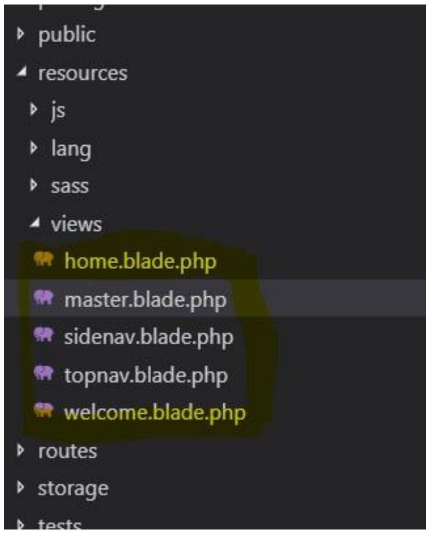
I'm using MaterializeCss in this example as it has some useful prebuilt components like sidenav.

Lets start:

Blade view files use the .blade.php file extension and are typically stored in the resources/views directory.

Let us create the following blade files in your project’s resource/views directory.

1. master.blade.php
2. topnav.blade.php
3. sidenav.blade.php
4. home.blade.php



In *master.blade.php*, let us initiate the html template with MaterializeCss CDN's. <!DOCTYPE html>

```
 <html>
  <head>

   <!--Import materialize.css-->
    <link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">
<link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/css/materialize.min.css">    <!--
style sheets section-->

   <meta name="viewport" content="width=device-width, initial-scale=1.0"/></head>
<body>
      @include('topnav')    <!-- include topnav.blade.php-->
      @include('sidenav')   <!-- include sidenav.blade.php-->      <div class="container">

        @yield('body-content')        </div>      <!--JavaScript of body for optimized loading-->
<!--
Jquery cdn--><script src="https://code.jquery.com/jquery-3.4.1.min.js"integrity="sha256-
CSXorXvZcTkaix6Yvo6HppcZGetbYMGWSFlBw8HfCJo="crossorigin="anonymous"></script
><s cript
src="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/js/materialize.min.js"></script></scr
ipt>< !-- also get script from the child view -->
<script>
@yield('page-script') <!-- to get script from page -->
</script>   </body>
</html>
```

When defining a child view, use the Blade @extends directive to specify which layout the child view should "inherit". Views which extend a Blade layout may inject content into the layout's sections using @section directives. Remember, as seen in the example above, the contents of these sections will be displayed in the layout using @yield:

@include is the *directive* used to include one blade file into another. Here we are including both the topnav.blade.php and sidenav.php inside the master.blade.php with the below lines of code.

```
@include('topnav')   <!-- include topnav.blade.php-->
@include('sidenav')  <!-- include sidenav.blade.php-->
```

@yield directive is used to yield the content from one blade to another blade. We are getting the content from *home.blade.php into master.blade.php with the code.*

Every section should have unique section name and it can be yielded with that name. Example, ‘body-content’ is the section name.

```
@yield('body-content')
```

Now, let add some awesome code to “topnav.blade.php”,

“sidenav.blade.php” and “home.blade.php. *topnav.blade.php*

```
<nav>
  <div class="nav-wrapper">
```

```
   <a href="#" class="brand-logo right">Logo</a>
    <ul id="nav-mobile" class="left hide-on-med-and-down">
      <li><a href="sass.html">Sass</a></li>
      <li><a href="badges.html">Components</a></li>
      <li><a href="collapsible.html">JavaScript</a></li>
</ul>
   </div>
</nav>
```

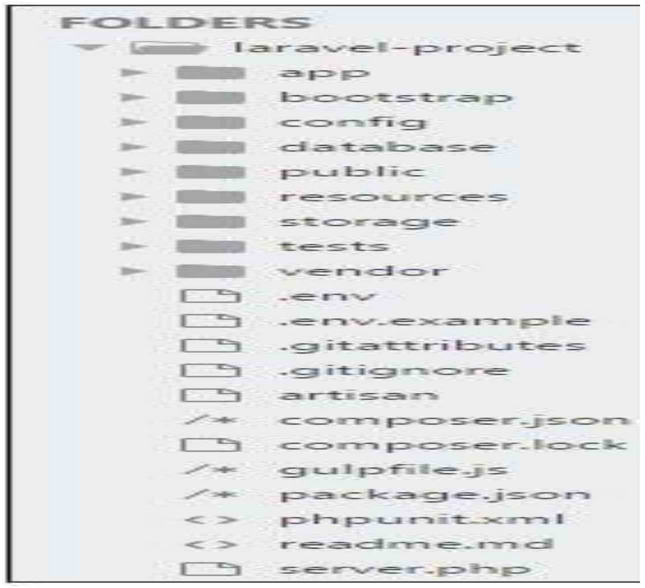
sidenav.blade.php

```
<ul id="slide-out" class="sidenav">
  <li><a href="#!">First Sidebar Link</a></li>
  <li><a href="#!">Second Sidebar Link</a></li>
  <li class="no-padding"><ul class="collapsible collapsible-accordion">
<li>
   <a class="collapsible-header">Dropdown<i
class="materialicons">arrow_drop_down</i></a><div class="collapsible-body">   <ul>
    <li><a href="#!">First</a></li>
    <li><a href="#!">Second</a></li>
    <li><a href="#!">Third</a></li>
    <li><a href="#!">Fourth</a></li>
  </ul>
</div></li>
</ul>
</li>
</ul><ul class="right hide-on-med-and-down">
  <li><a href="#!">First Sidebar Link</a></li>
  <li><a href="#!">Second Sidebar Link</a></li>
  <li><a class="dropdown-trigger" href="#!" data-target="dropdown1">Dropdown<i
class="materialicons right">arrow_drop_down</i></a></li><ul id='dropdown1'
class='dropdown-content'>
  <li><a href="#!">First</a></li>
  <li><a href="#!">Second</a></li>
  <li><a href="#!">Third</a></li>
  <li><a href="#!">Fourth</a></li>
</ul></ul><a href="#" data-target="slide-out" class="sidenav-trigger"><i
class="materialicons">menu</i></a>
```

5.Directive laravel.

The application structure in Laravel is basically the structure of folders, sub-folders and files included in a project. Once we create a project in Laravel, we get an overview of the application structure as shown in the image here.

The snapshot shown here refers to the root folder of Laravel namely laravelproject. It includes various sub-folders and files. The analysis of folders and files, along with their functional aspects is given below –



```
FOLDERS
  laravel-project
    app
    bootstrap
    config
    database
    public
    resources
    storage
    tests
    vendor
    .env
    .env.example
    .gitattributes
    .gitignore
    artisan
    composer.json
    composer.lock
    gulpfile.js
    package.json
    phpunit.xml
    readme.md
    server.php
```

Sometimes in laravel views, you have to write certain logic: checking for an admin, or a regular user, checking their rights. However, thanks to the built-in directive creation capabilities, this process can be made more convenient by taking all the validation logic into directives.

If you are new to learning Laravel, luckily enough, there are already quite a few built-in blade templating directives out there. The creators of the framework took care of solving the most demanded tasks and programmed them into the core of the framework in the form of directives.