Debre Birhan University

Institute Of Technology, Computing College

Department Of Computer Science

Selected Topics in Computer Science
Individual Assignment

Prepared By Dawit Terefe
DBUE/77511
Submitted to Instructor Girmachew G.

January 2023
Debre Birhan, Ethiopia

Contents

1.Explain MVC of Laravel

Introduction

Laravel is a free and open-source web PHP framework, which is based on MVC (Model-View-Controller) architecture. A Framework provides structure and starting point for creating our application. It helps to provide an amazing developer experience while providing powerful features through dependency.

Laravel Framework is used to develop complex Web Applications. Laravel can help a developer to develop a secure web application. It is a Scalable framework and it also has a wide Community across the world.

Laravel is a Full Stack Framework, which helps a developer create Full Stack Applications with the help of Laravel.

MVC of Laravel

MVC based framework mainly divides the whole application into three components:

- Model: It interacts with the database.
- View: User Interface. It contains everything which a user can see on the screen.
- Controller: It helps to connect Model and View and contains all the business logic. It is also known as the “Heart of the application in MVC”.

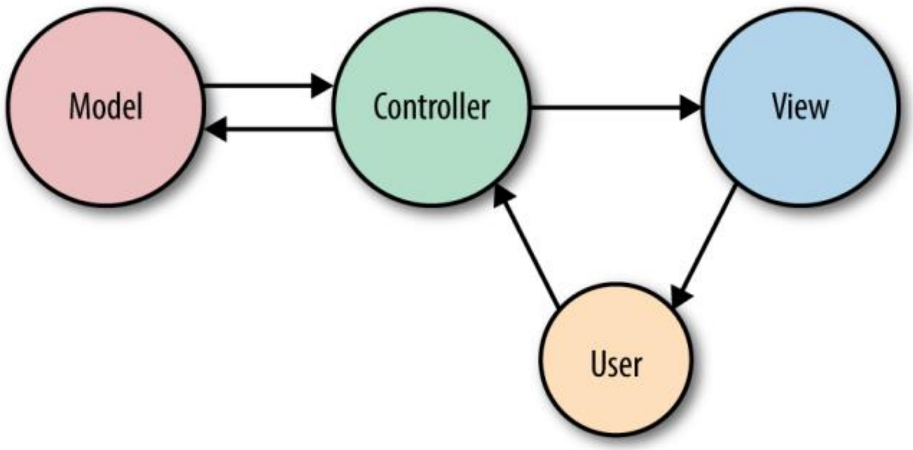

Fig 1.1 Basic Illustration of MVC

Model: This component of the MVC framework handles data used in our application. It helps to retrieve the data from the database and then perform some operation that our application is supposed to perform then it stores that data back in the database.

In simple words, we can say that Model is responsible for managing data that is passed between the database and the User Interface (View).

View: This component is User Interface, which defines the template which is sent as a response to the browser. This View components contain the part of code which helps to display data to the User Interface on the user's browser. For example, we can say the buttons, textbox, dropdown menu, and many more such widgets on the browser screen are the part of View Component.

Controller: This controller component helps to interact with the model component to fetch data from the database and then pass that data to the view component to get the desired output on the user's browser screen. Same way, when the user enters some data, the controller fetches that data and then performs some operation or just inserts that data into the database with the use of the model components.

Advantages of MVC

MVC is mainly used to separate Application code into user interfaces, data, and controlling logic. It can benefit the developer to easily maintain the code which can help to make a development process much smoother.

Advantages of Using MVC framework:

- Organizing large-scale web application projects.
- Easier to perform Modification.
- Modification in any part won't affect any other part of the code.
- Helps in a faster development process.
- Helps for Asynchronous Method Invocation.

2.Explain Routing

Introduction

The essential function of any web application framework is to take requests from a user and deliver responses, usually via HTTP(S). This means defining an application's routes, without routes, we have little to no ability to interact with the end user.

Routes are the web URLs that we can visit in our web application. For example, /home, /about, /dashboard etc. are all different routes that one can create in a Laravel Application.

Creating a Route

In a Laravel application, you will define our web routes in routes/web.php and our API routes in routes/api.php. Web routes are those that will be visited by our end users; API routes are those for our API.

```
// Syntax of a route
Route::request_type('/url', 'function()');
```

The most basic Laravel routes accept a URI and a closure, providing a very simple and expressive method of defining routes and behavior without complicated routing configuration files.

```
// Creating a new route
Route::get('/Greetings', function() {
    return 'Hello world';
})
```

Breaking down the code given above Route::get means this is a route that will expect a GET request. The /Greetings is the name of the route and we can create a route with any name. Further, we have to specify what to do when we visit that route in the browser and we do so in the form of a callback function which returns a string saying Hello World.

Returning Web-page:

Instead of just returning strings, we can return webpages when someone visits a route. to do that. First of all, we create a file called index.blade.php in resources/views.

Index.blade.php

```
<!DOCTYPE html>
<html lang="en">
<body>
    <h1>Hello World.</h1>
</body>
</html>
```

Then we Add the following code to our web.php now.

```
// Creating a new route
Route::get('/viewhello', function() {
    return view('index');
});
```

The code given above we have used /viewhello as the name of route and in the callback function we have used a view () method which is an inbuild method provided by Laravel to serve webpage and it automatically picks the file matching from resources/views folder. For example, passing ‘index’ will serve index.blade.php which returns the ‘Hello World’ string.

Routes with controllers

Laravel provides us much more power than just writing a direct callback function. We can actually make our routes point to a function inside the controllers. To do so we can create our controller first manually in app/Http/Controllers or use the artisan command (php artisan make:controller MyController).

Code written below is a basic controller code where we are just using Controllers namespace to add the capability to use it, it’s just like importing the libraries.

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class Mycontroller extends Controller {
    public function index() {
        return view('index2');
    }
}
```

We have written controller and written frontend file below, then the last thing is registering the route.

```
<!DOCTYPE html>
<html lang="en">
<body>
    <h1>This is index 2.</h1>
</body>
</html>
```

Syntax of registering the routes (Older version of Laravel)

```
Route::request_type('/url', 'ControllerName@functionName');
Route::get('/viewindex2', 'Mycontroller@index');
```

Now this syntax is changed with the following syntax

```
Route::get('/viewindex2', [Mycontroller::class,'index']);
```

Here the first parameter defines ‘viewindex2’ which is the URL and we define the controller’s name and the function name in the second parameter within an array.

3.Explain Migration and Relationships

Introduction

In Laravel, Migration provides a way for easily sharing the schema of the database. It also makes the modification of the schema much easier. It is like creating a schema once and then sharing it many times. It gets very useful when we have multiple tables and columns as it would reduce the work over creating the tables manually.

To Create Migration: It can be created by using the artisan command as shown below:

```
php artisan make:migration create_articles_table
```

Here, articles are going to be the table and in place of that we can write any other table name which we want to create and is appropriate for the application but the table name, is to be in plural form. So, it is written as articles and not article. This is the naming scheme used by Laravel and it is important to specify create at the start and table at the end according to the naming scheme of a Migration file in Laravel.

All the migration file that we create using the artisan command are located at database/migrations directory. So, after we run the above command, it will generate a PHP file with the name we specified with the current date and time.

Basic Structure of a Migration

A migration file contains a class with the name of the file specified while creating the migration and it extends Migration. In that we have two functions, the first one is up () function and the second one is down () function. The up () is called when we run the migration to create a table and columns specified and the 'down()' function is called when we want to undo the creation of 'up()' function.

In the up () function, we have create method of the Schema facade (schema builder) and as we say before, the first argument in this function is the name of the table to be created. The second argument is a function with a Blueprint object as a parameter and for defining the table.

In the down () function, we have a dropIfExists method of the schema builder which when called will drop the table.

To Run Migration: Before running a migration, we first have to create a MySQL Database and Connect to it. After that is done, to Run a Migration, we can use an Artisan command as follows:

```
php artisan migrate
```

This command will run the up () function in the migration class file and will create the table with the all columns specified.

This command will run the up () function and create all the tables in the database for all the migration file in the database/migrations directory.

To rollback any last migration which is done, we can use the following Artisan command:

```
php artisan migrate:rollback
```

Relationships

A relationship, in the context of the database, is a relation or link between two tables via primary key and reference key. One table has a foreign key that references the primary key of another table. Relationships allow relational databases to split and store data in different tables.

Types of Eloquent Relationships

- One To One
- Many To Many
- HasMany Through

- One To Many
- HasOne Through

One to One Relationship

One to one relationship is one of basic relationships. For example, a Post model would be associated with a content model. To illustrate this relationship, we can create a post_content () method within the Post model and call the hasOne () method to relate the Content model.

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    public function post_content()
    {
        return $this->hasOne('App\Content');
    }
}
```

It is important to note that Eloquent establishes a foreign key based on the model's name and should have a matching value id. Here post_id is the foreign_key of Content.

The inverse of One to One

So far, we can access the content from the post. now we can create an inverse relationship on the content model so that the post can access the model from it. To do this, we can use the belongsTo method for getting the post data on the content model.

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Content extends Model
{
    public function post()
    {
        return $this->belongsTo('App\Post');
    }
}
```

We can now access the post content using the relation method like this:

```php
$content = Post::find(10)->post_content;
```

One to Many Relationship

One of many relationships determines a relationship where one single model owns the number of the other model. For example, a blog author may have many posts.

A single author can have written many post articles. Let's take an example of one-to-many relationships.

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Author extends Model
{
    ...
    public function post()
    {
        return $this->hasMany('App\Post');
    }
}
```

We can then access the collection of post article by using the post () method.

Using this, we get all the posts of a single author. Here we need to get the author details of a single post, for each post there a single author, within the Post model includes belongsTo relation.

```
$posts = App\Author::find(10)->post()->get();

 foreach ($posts as $post) {
     //do something
 }
```

Inverse One to Many Relationship

Since we can now access all the post articles of an author, it is time to allow a post article to access its parent model (Author model) or access author details by using the post model. The inverse relationship of both One to One and One to Many works the same way. Both use the belongsTo method to define the inverse relation. Thus, we can define one as:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
```

```
class Post extends Model
{
    public function author()
    {
        return $this->belongsTo('App\Author');
    }
}
```

Using this, here we need to get the author details by using posts, for each post there a single author, within the Post model includes belongsTo relation.

```
$post->author->author_name;
```

Many to Many Relationship

Each post has many tags and each tag can have many posts. To define many to many relationships, we use belongsToMany() method.

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
   ...
    public function tags()
    {
        return $this->belongsToMany('App\Tag');
    }
}
```

We can now access the tags in post model:

```
$post = App\Post::find(8);

 foreach ($post->tags as $tag) {
    //do something
 }
```

The inverse of Many to Many Relationships

The inverse of a many to many relationships can be defined by simply calling the similar belongsToMany method on the reverse model. We can illustrate that by defining posts () method in Tag model as:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Tag extends Model
{
    public function posts()
    {
        return $this->belongsToMany('App\Post');
    }
}
```

We can now access the post in tag model:

```php
$tag = App\Tag::find(8);

 foreach ($tag->posts as $post) {
     //do something
  }
```

HasMany Through

The “has-many-through” relationship provides a convenient shortcut for accessing distant relations via an intermediate relation. For example, a Country model might have many Post models through an intermediate User model. In this example, you could easily gather all blog posts for a given country. Let’s look at the tables required to define this relationship:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Country extends Model
{
    /**
     * Get all of the posts for the country.
     */
    public function posts()
    {
        return $this->hasManyThrough('App\Post', 'App\User');
    }
}
```

HasOne Through

The "has-one-through" relationship defines a one-to-one relationship with another model. However, this relationship indicates that the declaring model can be matched with one instance of another model by proceeding through a third model.

For example, in a vehicle repair shop application, each Mechanic model may be associated with one Car model, and each Car model may be associated with one Owner model. While the mechanic and the owner have no direct relationship within the database, the mechanic can access the owner through the Car model. Let's look at the tables necessary to define this relationship:

```
mechanics
    id - integer
    name - string

cars
    id - integer
    model - string
    mechanic_id - integer

owners
    id - integer
    name - string
    car_id - integer
```

Now that we have examined the table structure for the relationship, let's define the relationship on the Mechanic model:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Mechanic extends Model
{
    /**
     * Get the car's owner.
     */
    public function carOwner()
    {
        return $this->hasOneThrough(Owner::class, Car::class);
    }
}
```

The first argument passed to the hasOneThrough method is the name of the final model we wish to access, while the second argument is the name of the intermediate model.

Typical Eloquent foreign key conventions will be used when performing the relationship's queries. If you would like to customize the keys of the relationship, you may pass them as the third and fourth arguments to the hasOneThrough method. The third argument is the name of the foreign key on the intermediate model. The fourth argument is the name of the foreign key on the final model. The fifth argument is the local key, while the sixth argument is the local key of the intermediate model:

```
class Mechanic extends Model
{
    /**
     * Get the car's owner.
     */
    public function carOwner()
    {
        return $this->hasOneThrough(
            Owner::class,
            Car::class,
            'mechanic_id', // Foreign key on the cars table...
            'car_id', // Foreign key on the owners table...
            'id', // Local key on the mechanics table...
            'id' // Local key on the cars table...
        );
    }
}
```

4.Expalin Blade Template Engine

Introduction

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, we 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.

Purpose of Blade Template

Blade template is used because of the following reasons:

Displaying data

If we want to print the value of a variable, then we can do so by simply enclosing the variable within the curly brackets. See the syntax below:

```
{{$variable}};
```

In blade template, we do not need to write the code between.

```
<?php echo $variable; ?>. The above syntax is equivalent to <?= $variable ?>.
```

Ternary operator

In blade template, the syntax of ternary operator can be written as:

```
{{ $variable or 'default value'}}
```

The above syntax is equivalent to

```
<?= isset($variable) ? $variable : ?default value? ?>
```

HTML Entity Encoding

By default, Blade (and the Laravel e helper) will double encode HTML entities. If you would like to disable double encoding, call the Blade::withoutDoubleEncoding method from the boot method of your AppServiceProvider:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Blade::withoutDoubleEncoding();
    }
}
```

5.Directves

In addition to template inheritance and displaying data, Blade also provides convenient shortcuts for common PHP control structures, such as conditional statements and loops. These shortcuts provide a very clean, terse way of working with PHP control structures while also remaining familiar to their PHP counterparts.

Blade Template Control Statements

Blade templating engine also provides the control statements in Laravel as well as shortcuts for the control statements.

```
<html>
<body>
 <font size='5' face='Arial'>
@if(($id)==1)
student id is equal to 1.
@else
student id is not equal to 1
@endif
</font>
</body>
</html>
```

Blade template provides @unless directive as a conditional statement. The above code is equivalent to the following code:

```
<html>
 <body>
 <font size='5' face='Arial'>
@unless($id==1)
student id is not equal to 1.
@endunless
</font>
</body>
</html>
```

@hasSection directive

The blade templating engine also provides the @hasSection directive that determines whether the specified section has any content or not.

Let's understand through an example.

```
<html>   <body>  <title>
 @hasSection('title')
 @yield('title') - App Name
 @else
 Name
@endif
</title> </font>  </body>  </html>
```

Blade Loops

The blade templating engine provides loops such as @for, @endfor, @foreach, @endforeach, @while, and @endwhile directives. These directives are used to create the php loop equivalent statements.

@For loop

```
value of i :
@for($i=1;$i<11;$i++)
{{$i}}
@endfor
```

@Foreach loop

```
@foreach($students as $students)
{{$students}}<br>
@endforeach
```

@While loop

```
@while(($i)<5)
javatpoint
{{$i++}}
@endwhile
```

Authentication Directives

The @auth and @guest directives may be used to quickly determine if the current user is authenticated or is a guest:

```
@auth
    // The user is authenticated...
@endauth

@guest
    // The user is not authenticated...
@endguest
```

Environment Directives

We may check if the application is running in the production environment using the @production directive:

```
@production
    // Production specific content...
@endproduction
```

Or, you may determine if the application is running in a specific environment using the @env directive:

```
@env('staging')
    // The application is running in "staging"...
@endenv

@env(['staging', 'production'])
    // The application is running in "staging" or "production"...
@endenv
```

Section Directives

We may determine if a template inheritance section has content using the @hasSection directive:

```
@hasSection('navigation')
    <div class="pull-right">
        @yield('navigation')
    </div>

    <div class="clearfix"></div>
@endif
```

References

https://laravel-news.com/five-useful-laravel-blade-directives
https://masteringbackend.com/posts/complete-guide-on-laravel-relationships/
https://www.geeksforgeeks.org/laravel-migration-basics/
https://laravel.com/docs/9.x/controllers
https://laravel.com/docs/9.x/routing
Matt-Stauffer-Laravel_-Up-Running_-A-Framework-for-Building-Modern-PHP-Apps-O'Reilly-Media-2019
https://www.javatpoint.com/laravel-blade-template
https://www.geeksforgeeks.org/laravel-routing-basics/