Roozbeh Kamalzadeh 2022/07/20

# MVC Design Pattern

MVC stands for Model, View, Controller.

The controller mediates between the models and views.

PHP MVC Frameworks simplify working with complex technologies by:

- hiding all the complex implementation details.
- Providing standard methods that we can use to build our applications.
- Increased developer productivity.
- Adherence to professional coding standards.

***- Model:***

This part concerned with the business logic and the application data. The data can come from flat file, database, xml document and other valid data sources.

***- Controller:***

The controller links the models and views together depending on the request resources.

***- Views:***

This part deals with presenting the data to the user.

# Composer

Composer is a tool for dependency management in PHP. It allows us to declare the libraries our project depends on and it will manage (install/update) them.

Composer is not a package manager in the same sense as pip. By default, it does not install anything globally.

# Laravel (PHP Back-End MVC Framework)

**Difference between Laravel and Django:**

- In terms of speed, Laravel doesn't compare with Django, unfortunately.
- Django takes security quite seriously and helps developers avoid the common mistakes of web development and implement some security best measures. While Laravel does also cover the basics of security it doesn't live up to Django's security level. That's why, for example, NASA uses Django for their website.
- Laravel comes with built-in support for API building, as the queries return JSON by default. Django doesn't come with this built-in feature.
- Django in GitHub has 43,384 stars, Laravel has 34,292 stars.

**Into the Laravel:**

***- routes:***

We can define different routes in web.php file when we want return HTML as response.

We can pass view as a method instead of get.

***- resource:***

*Example:*

```
Route::resource( url, controller);
```

This method can create some default routing for us and connect those routes to specific methods of the controller.

*- only :*

We can use this method to tell the resource to create some of the routes for us.

*Example:*

```
Route::resource( url, controller)→only(
                                    [ 'index', 'show' ]
                                    );
```

*- except:*

We can use this method to tell the resource to avoid create some of the routes for us.

***- apiResource :***

This method used for api routing so we should add it to the api.php file instead of web.php.

*Example:*

Route::apiResource( url, controller )

***- name:***

With this method we can specify a name for our route.

*Example:*

*Route::get( url, Controller )→name( someName )*

***- parameters:***

With this method we can change name of parameters.

```
Route::apiResource( url, controller )→parameters(
                                        [ "url"=> "newName" ]
                                                    );
```

***- redirect:***

This method provides a convenient shortcut so that we do not have to define a full route or controller for performing a simple redirect.

*Example:*

Route::redirect( from, to, status_code );

***- Route parameters:***

We can pass a parameter in our route and then get the parameter from the URL.

*Example:*

Route::get( '/user/{param}' , controller );

If we want that parameter to be optional we add ? Mark at the end of the parameter.

***- Regular Expression:***

*where:*

The where method accepts the name of the parameter and a regular expression defining how the parameter should be constrained.

*Example:*

Route::get( /{param}, controller )→where( 'param', [A-Z];

*whereNumber:*

→ whereNumber( 'param' );

*whereAlpha:*

→whereAlpha( 'param' );

*whereIn:*

→whereIn( 'param', list );

If the incoming request does not match the route pattern constraints, a 404 HTTP response will be returned.

*pattern:*

We can use *pattern* method for a route parameter to always be constrained by a given regular expression.

We should define these patterns in the boot method of our RouteServiceProvider class.

***- route helper function:***

We can generate URL with route helper function.

*Example:*

$url = route( 'nameOfRoute' );

If our named route has parameters we can add it to the route helper function.

*Example:*

$url = route( 'nameOfRoute', [param => value];

If we pass additional parameters to the route helper, those parameters automatically added to the end of generated URL.

***- redirect helper function:***

We can generate redirect with redirect heper function.

*Example:*

```
return redirect→route( 'nameOfRoute' );
```

***-group:***

We can use this method on route and group two or more routing together with same configuration.

For example we can link one controller class to two or more routes and the routes to the different methods of the controller:

```
Route::controller(controller::class)→group(function() {
    Route::get( 'url' , 'controllerMethod' );
    Route::post( 'url' , 'controllerMethod' );
}
```

***- Route::middleware:***

To assign middleware to all routes within a group, we may use the middleware method before defining the group.

***- views:***

I think this is like templates in django framework. Views can show data to client.

***- controller:***

Controller manage the logical flow and modify data before it goes to views or database(Models).

***- $_GET, $_POST:***

These variables can hold the form data and when we return these variables, they sends the data as JSON.

***- request():***

In laravel we don't use the above variables to get data from form. Instead of those variables, we use *request* method.

{ request()→all() } return all data from forms with any method.

***- CSRF (Cross-site request forgeries):***

CSRF are a type of malicious exploit whereby unauthorized commands are performed on behalf of an authenticated user.

***- CSRF token:***

This token is used to verify that the authenticated user is the person actually making the requests to the application. Since this token is stored in the user's session and changes each time the session is regenerated, a malicious application is unable to access it.

Anytime we define a "POST", "PUT", "PATCH", or "DELETE" HTML form in our application, we should include a hidden CSRF_token field in the form so that the CSRF protection middleware can validate the request.

***- view( nameOfView, [params] ):***

View method is used for set a view file. We can pass parameters in a list to the view method and then use those parameters in the view file.

***- blade:***

Blade is a powerful templating engine in laravel framework. The blade allows to use the templating engine easily, and it makes the syntax writing very simple.

***@csrf:***

This is a blade decorator, and with this decorator we can make an invisible input for CSRF_token.

***@extends( path ):***

We use this decorator for making a base template.

***@yield( str:name ):***

We use this decorator for putting up a template into another.

***@section( str:name ) ... @endsection:***

We use this decorator for separate a block of HTML and use that section in yield decorator. The HTML codes goes between the section and endsection.

***- @includeWhen( condition, path ):***

With this decorator we can make a condition for loading part of HTML code.

**- *@php:***

With this decorator we can parse php code.

**- *{ !! someText !! }:***

We can pass HTML code in it, and its parse the HTML code.

**-** @if( empty() ) shorthanded is @empty().

**-** @if( isset() ) shorthanded is @isset().

**-** @unless( condition ):

This decorator is opposite of @if decorator.

**- *$loop:***

This variable can gives us some information about the foreach loop.

**- *$loop→index:***

This gives us the index of current item of an array in the loop.

**- *$loop→iteration:***

This gives us the current item number of an array in the loop.

**- *$loop→remaining:***

This gives us the number of remaining items of an array in the loop.

***- $loop→parent:***

This gives us the parent loop of the current loop.

***- @each( pathToView, array, itemName, pathToDefaultView ):***

With this decorator we can loop through an array and generate every item with an specific view.

***- @push( str:name ):***

We can stack part of HTML code with this decorator.

***- @stack( str:name ):***

We can call the stacked code we add to stack with @push decorator, with This decorator.

***- @prepend( str:name ):***

We can add a part of HTML code to the beginning of our stack with this decorator.

***- @verbatim:***

This decorator can skip ( {{ ... }} ) and show ( { ) as plain text.

***- AppServiceProvider:***

In this file we can add new components and new directives.

We should define our new component or directives in the boot function.

*Example* ***:***

```
Blade::component( path, name );
Blade::directive( name, function() {
    ...some code...
});
```

***-resourceVerbs:***

We use this method to change name of a resource route.

*Example:*

Route:resourceVerbs( [ default => newName ] );

***- Artisan*:**

Artisan is the name of the command-line interface included with laravel.

***- artisan make:controller <nameOfController> :***

This command make a controller for us.

***--resource:***

We can use this option to make a controller with prefix methods.

*Example:*

php artisan make:controller --resource <name>

***--api:***

We can use this option to make a resource controller for api routing with prefix methods.

***- artisan route:list :***

This command shows us the list of our routes.

***- artisan route:cache :***

This command store our routes to cache and when we enter *artisan route:list* command artisan read our routes from cache instead of looking for our routs in routes files. With this command we can load our route list much faster.

***- artisan down:***

With this artisan command we can turn our application to maintenance mode.

***- artisan up:***

With this artisan command we can quit from maintenance mode.

***- artisan make:request <nameOfRequest>:***

With this artisan command we can create a custom request file.

***- Service Container* :**

The service container is responsible for managing our class dependencies and allows us to perform dependency injection.

By default, laravel allows us to use completely configuring-free dependency injection system. All we need to do is type hint the class that we want to inject.

By simply type-hinting the class we want to inject in our method, laravel is automatically looking up the injected class and its dependencies, tries to create a new instance of the class and pass it to the method.

In a class (controller), we might also need a dependency throughout all of the methods. In this case, you can simply inject the class in the class (controller) constructor.

***- Manual binding:***

If our dependency class takes an argument, we can't simply type-hinting the class. When we now try to inject this class, laravel will not know what to do with the constructor argument and throw an exception.

We will register all manual dependencies in a service provider. This should take palace in *register* method.

Each service provider has the container available as the *app* property.

Within a service provider, you always have access to the container via *$this→app* property.

***- Middleware:***

Middleware provide a convenient mechanism for inspecting and filtering HTTP requests entering our application.

All of these middlewares located in app/http/Middleware directory.

To create a new middleware we can use:

*php artisan make:middleware <nameOfMiddleware>*

***- Closure $next:***

To pass the request deeper into the application we should call the *$next* callback with the *$request:*

return $next( $request );

A middleware can perform tasks before or after the request deeper into the application.

For example if we set a condition before call next, it is a before middleware. Otherwise if we first store the response and then call next, it is a after middleware.

***- Global Middleware:***

We can add our middleware class in the *$middleware* property of our *app/Http/Kernel.php* class.

With this approach, our middleware can run during every HTTP request of our application.

***→middleware():***

We can add this method end of the route and add the middleware to the route.

We can pass a list as argument and add multiple middlewares to one route.

We can pass the key name we define in kernel.php for our middleware or we can pass the fully qualified class name.

***→withoutMiddleware():***

When we use group method to group some routes with the same middleware, with this method we can remove the middleware from on of the items of the group.

***- Middleware Groups:***

We can group several middlewares under a single key to make them easier to assign to routes.

We can assign these groups using *$middlewareGroups* property of HTTP kernel.

We can add the middleware in the constructor function of a controller class:

```
$this->middleware( nameOfMiddleware );
```

This method add middleware to all functions of controller.

We can use Only or Except method for set middleware to an specific method of controller:

```
$this->middleware( nameOfmiddleware )->only( [someMethod] );
$this->middleware( nameOfmiddleware )->except( [someMethod] );
```

***- Middleware Parameter:***

We can pass parameter to middleware:

```
->middleware( 'nameOfMiddleware : parameter' );
```

We can access that parameter in handle method of the middleware class as a third parameter:

```
function handle( $request, Closure $next, $param ){...}
```

***Request():***

With the request class we can get the data. We usually inject the Request class as a dependency in a controller.

This class has several methods.

***->all():***

With this method we can get all of data. When we return that object, laravel automatically turns the object into a json.

***->get( key , default ) :***

With this method we can get an specific data associated to a key. If the key doesn't exist it sets the default value to that key.

We can use the key directly. This is a magic method:

```
$request->key;
```

***→has(key):***

With this method we can check if a key sets or not.

When we get a value with magic method, the priority of query string is higher than parameter.

*Example:*

**route: Route::get("/{id}", controller );**

**url: www.example.how/12?id=8**

**$request→id == 8**

***→route()→parameters:***

With this method we can access to the route parameters list.

***→route()→parameter( nameOfParameter ):***

With this method we can get the parameter.

***→only():***

With this method we can get specific data we want.

*Example:*

```
return $request→only( ['a' , 'b'] );
```

Its return only the data associated with a and b.

***→file():***

With this method we can get the files data.

***→url():***

With this method we can get the url without query strings.

***→fullUrl():***

With this method we can get the url with query strings.

***→fullUrlWithQuery( [ nameOfQuery => value ,] ) :***

With this method we can get the url with query strings and we can change the values of query strings.

***→cookies():***

We can get cookies with this method.

***Custom Request:***

We can create our custom request class in laravel. We can set authorization to our request and set validation to our incoming data.

We have two main method in our custom request class. One method is authorize and the other is rules.

In authorize method we can set in witch condition we can access the request.

In rules method we can validate our data before we send it to our controller.

We have some validation rules like; *required, array, between, integer* and so on.

We can separate every validation of an specific data with | sign.

*Example:*

```
public function rules( ){
    return[
        'name' => 'required | string',
        'age' => 'required | integer',
    ];
```

We can also use an array to separate validation. This approach has an advantage than using | sign. When we use an array to define our validation, we can add function and create our custom validation.

*Example:*

```
public function rules( ){
    return[
        'age' => [ 'required', 'integer', function($attr, $value, $fail){
                if ($value < 8 ){
                    $fail($attr . 'is not valid');
                }
            }],
    ];
```

We can set our custom validations here, but still it's not a good approach. If we want create a custom validation we should add it in a service provider and then use it in our rules method of our custom request class.

***- Response:***

When we return something to the user, we can say this is a response to the user's request.

We can return an string as response. We also can return an array as response and laravel automatically convert that array to JSON. We can return an object as response only when that object has ***__toString()*** mehtod.

***-response( str|view|array|null:content, int:statusCode, array:headers ):***

With this method we can send our response and we have a better control to customize our response.

We can set our status code and we can send our custom headers to the user.

***→header(key, value):***

We can specify the header outside the response method with chaining this method to the response method.

***→headers(array):***

We can set several headers with this method.

***→cookie(key, value):***

We can send our custom cookie in a response with this method. But there is a default middleware in laravel that encrypt our cookie's value and send the encrypted value to the user. If we don't want to encrypt that cookie, we can go to the middleware file and this middleware has a parameter named except. In this parameter we can pass our cookie name (key) and the middleware don't encrypt our cookie anymore.

***→download(fullPathToFile, str:name):***

With this method we can force the browser to download a file.

With the second parameter we can define the name of downloaded file.

***→file(fullPathToFile, array:headers):***

With this method we can force the browser to show the file to the user.

On this method we can't chain the header method and we should pass headers as second argument.

***- Custom response method:***

We can add custom methods to response in the boot method of appServiceProvider class.

*Example:*

```
Response::macro(nameOfMethod, function ($value) {
    ...modify $value...
    return $modifiedValue;
});
```

After we define our new method, we can use it like other methods of response and pass our value to it.

We can set custom headers for our custom response method too.

*Example:*

```
Response::macro(nameOfMethod, function ($value) {
    return Response::make($modifiedValue)
        →header(key, value);
});
```

***-redirect(innerUrl):***

With this method we can redirect to a new url. But it is better to redirect to the name of a route instead of hard-coding the url. For reach to this we can use ***→route().***

***→route( nameOfRoute, array:params ):***

*Example:*

```
return redirect()→route( 'foo', [ 'id' => 2 ] );
```

***→with(key, value):***

With this method we can set session and send it to the redirected route.

***-back():***

With this method we can redirect to the previous page.

*Example:*

```
return back();
```