Understand the Basics of Laravel Middleware

by Sajal Soni 21 Jul 2017

Difficulty: Beginner Length: Medium Languages: English

Laravel 5 PHP Web Development

In this article, we'll dive deep into the Laravel framework to understand the concept of middleware. The first half of the article begins with an introduction to middleware and what it's actually used for.

As we move on, we'll cover how to create custom middleware in a Laravel application. After creation of your custom middleware, we'll explore the options available to register it with Laravel so that it could be actually invoked during the request processing flow.

I hope that you consider yourself familiar with basic Laravel concepts and the Artisan command-line tool to generate the scaffolding code. Of course, a working installation of the latest Laravel application allows you to run the examples provided in this article straight away.

What Is Middleware in Laravel?

We could think of middleware as a mechanism that allows you to hook into the typical request processing flow of a Laravel application. A typical Laravel route processing goes through certain stages of request processing, and the middleware is one of those layers an application has to pass through.

So what exactly is the point of hooking into the Laravel request processing flow? Think of something that requires an execution at the early stages of bootstrapping of an application. For example, it's necessary to authenticate users at the early stages to decide whether they're allowed to access the current route.

A few things I could think of that you could achieve through middleware are:

- logging of requests
- redirecting users
- altering/sanitizing the incoming parameters
- manipulating the response generated by the Laravel application

- and many more

In fact, the default Laravel application already ships a couple of important pieces of middleware with it. For example, there's middleware that checks if the site is in maintenance mode. On the other hand, there's middleware to sanitize the input request parameters. As I mentioned earlier, the user authentication is also achieved by the middleware itself.

I hope that the explanation so far helps you to feel more confident about the term middleware. If you're still confused, don't worry about it as we're going to build a piece of custom middleware from the next section onwards that should help you understand exactly how middleware could be used in the real world.

How to Create Custom Middleware

In this section, we'll create our custom middleware. But what exactly is our custom middleware going to accomplish?

Recently, I came across a custom requirement from my client that if users access the site from any mobile device, they should be redirected to the corresponding sub-domain URL with all the querystring parameters intact. I believe this is the perfect use case to demonstrate how Laravel middleware could be used in this particular scenario.

The reason why we would like to use middleware in this case is the need to hook into the request flow of the application. In our custom middleware, we'll inspect the user agent, and users are redirected to the corresponding mobile URL if they are using a mobile device.

Having discussed all that theory, let's jump into the actual development, and that's the best way to understand a new concept, isn't it?

As a Laravel developer, it's the Artisan tool that you'll end up using most of the time to create basic template code should you wish to create any custom functionality. Let's use it to create a basic template code for our custom middleware.

Head over to the command line and go the document root of your project. Run the following command to create the custom middleware template `MobileRedirect`.

```
1 php artisan make:middleware MobileRedirect
```

And that should create a file `app/Http/Middleware/MobileRedirect.php` with the following code.

```
01 <?php
02
03 namespace App\Http\Middleware;
04
05 use Closure;
06
07 class MobileRedirect
08 {
09     /**
10      * Handle an incoming request.
11      *
12      * @param  \Illuminate\Http\Request  $request
13      * @param  \Closure  $next
14      * @return mixed
```

```
     */
    public function handle($request, Closure $next)
    {
        return $next($request);
    }
}
```

More often than not, you'll notice the implementation of the `handle` method that acts as the backbone of the middleware, and the primary logic of the middleware you're looking to implement should go here.

Let me grab this opportunity to introduce the types of middleware that Laravel comes with. Mainly, the are two types—before middleware and after middleware.

As the name suggests, the before middleware is something that runs before the request is actually handled and the response is built. On the other hand, the after middleware runs after the request is handled by the application and the response is already built at this time.

In our case, we need to redirect the user before the request is handled, and hence it'll be developed as a before middleware.

Go ahead and modify the file `app/Http/Middleware/MobileRedirect.php` with the following contents.

```
<?php

namespace App\Http\Middleware;

use Closure;

class MobileRedirect
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        // check if the request is from mobile device
        if ($request->mobile == "1") {
            return redirect('mobile-site-url-goes-here');
        }

        return $next($request);
    }
}
```

For the sake of simplicity, we just check the existence of the `mobile` querystring parameter, and if it's set to `TRUE`, the user will be redirected to the corresponding mobile site URL. Of course, you would like to use the user agent detection library should you wish to detect it in real time.

Also, you would like to replace the `mobile-site-url-goes-here` route with the proper route or URL as it's just a placeholder for demonstration purposes.

Following our custom logic, there's a call to `$next($request)` that allows the request to be processed further in the application chain. The important thing to note in our case is that we've placed the mobile detection logic prior to the

`$next($request)` call, effectively making it a before middleware.

And with that, our custom middleware is almost ready to be tested. At the moment, there's no way Laravel knows about our middleware. To make that happen, you need to register your middleware with the Laravel application, and that's exactly the topic of our next section.

Before heading into the next section, I would like to demonstrate how the after middleware looks, just in case someone out there is curious about it.

```
<?php

namespace App\Http\Middleware;

use Closure;

class CustomMiddleWare
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        $response = $next($request);

        /* your custom logic goes here */

        return $response;
    }
}
```

As you would already have noticed, the custom logic of the middleware gets executed after the request is processed by the Laravel application. At this time, you have access to the `$response` object as well, which allows you to manipulate certain aspects of it if you wish to.

So that was the story of after middleware.

Our Custom Middleware in Action

This section describes the process of registering the middleware with the Laravel application so that it could actually be invoked during the request processing flow.

Go ahead and open the file `app/Http/Kernel.php` and look for the following snippet.

```
/**
 * The application's global HTTP middleware stack.
 *
 * These middleware are run during every request to your application.
 *
 * @var array
 */
protected $middleware = [
    \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,
    \Illuminate\Foundation\Http\Middleware\ValidatePostSize::class,
```

```
    \App\Http\Middleware\TrimStrings::class,
    \Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull::class,
];
```

As you can see, the `$middleware` holds the array of middleware that comes with the default installation of Laravel. The middleware listed here will be executed upon every Laravel request, and thus it's an ideal candidate to place our own custom middleware.

Go ahead and include our custom middleware as shown in the following snippet.

```
protected $middleware = [
    \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,
    \Illuminate\Foundation\Http\Middleware\ValidatePostSize::class,
    \App\Http\Middleware\TrimStrings::class,
    \Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull::class,
    \App\Http\Middleware\MobileRedirect::class,
];
```

Now, try to access any of your Laravel routes with the querystring `mobile=1`, and that should trigger our middleware code!

So that's how you're supposed to register your middleware that needs to be run on every request. However, sometimes you wish to run your middleware for the specific routes only. Let's check how to achieve that by using the `$routeMiddleware`.

In the context of our current example, let's assume that the users will be redirected to a mobile site if they access any specific route on your site. In this scenario, you don't want to include your middleware in the `$middleware` list.

Instead, you would like to attach the middleware directly to the route definition, as shown below.

```
Route::get('/hello-world', 'HelloWorldController@index')->middleware(\App\Http\Middleware\MobileRedirect::class);
```

In fact, we could go one step further and create an alias for our middleware so that you don't have to use inline class names.

Open the file `app/Http/Kernel.php` and look for the `$routeMiddleware` that holds the mappings of aliases to middleware. Let's include our entry into that list, as shown in the following snippet.

```
protected $routeMiddleware = [
    'auth' => \Illuminate\Auth\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,
    'can' => \Illuminate\Auth\Middleware\Authorize::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
    'mobile.redirect' => \App\Http\Middleware\MobileRedirect::class
];
```

And the revised route definition looks like this.

```
Route::get('/hello-world', 'HelloWorldController@index')->middleware('mobile.redirect');
```

And that's the story of registering middleware with the Laravel application. That was pretty straightforward, wasn't it?

In fact, we've reached the end of this article, and I hope you've thoroughly enjoyed it.

Conclusion

Exploring the architectural concept in any framework is always exciting stuff, and that's what we did in this article as we explored middleware in the Laravel framework.

Starting with a basic introduction to middleware, we shifted our attention to the topic of creating custom middleware in a Laravel application. And it was the latter half of the article that discussed how to register your custom middleware with Laravel, and that was also the opportunity to explore the different ways you could attach your middleware.

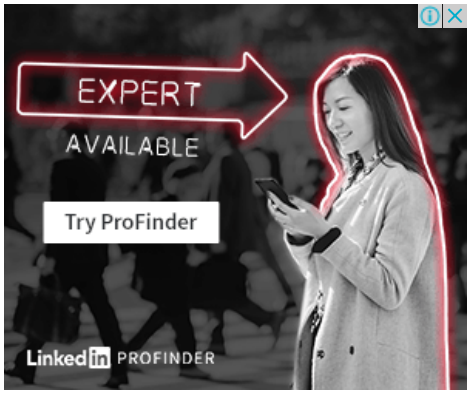
Hopefully the journey was fruitful and the article has helped you enrich your knowledge. Also, if you want me to come up with specific topics in the upcoming articles, you could always drop me a line about that.

That's it for today, and don't hesitate to shoot your queries, if any, using the feed below!

Sajal Soni
Software Engineer, INDIA

Sajal belongs to India and he loves to spend time creating websites based on open source frameworks. Apart from this, it's traveling and listening music which takes the rest of his time!

sajalsoni



Eyal Medina • 2 months ago

Thanks. Great article.

Reply • Share ›

Said Bakr • a year ago

I could not able to distinguish between the implementation using middleware and using the parent Controller class's constructor that all controllers extends.

Reply • Share ›

Sajal Soni ➚ Said Bakr • a year ago

Sorry I didn't get you Said. Can you please elaborate it more?

Reply • Share ›

achyutha ram ➚ Sajal Soni • 22 days ago

Sir , the task done by middleware can also be done using parent Controller as well, then y so middleware is one of the most important topic in laravel

Reply • Share ›

Sajal Soni ➚ achyutha ram • 21 days ago

I think the middleware is called early in the request execution flow of Laravel, so I guess before even the controller is called. That allows you to take early actions in the request.

Also, the middleware is something that's called globally, irrespective of the controller, so if you want to setup something that's called every time and don't want to tie it with controller, it's useful.

Further, it's a hook mechanism so developers can add their own middlewares easily without much hassle.

Reply • Share ›

achyutha ram ➚ Sajal Soni • 10 hours ago

tq for ur knowledge share

Reply • Share ›

QUICK LINKS - Explore popular categories

ENVATO TUTS+ +

JOIN OUR COMMUNITY +

HELP +

tuts+

26,494 Tutorials | 1,168 Courses | 28,701 Translations

Envato.com Our products Careers Sitemap

Follow Envato Tuts+