HTTP Middleware

Introduction

HTTP middleware provide a convenient mechanism for filtering HTTP requests entering your application. For example, Laravel includes a middleware that verifies the user of your application is authenticated. If the user is not authenticated, the middleware will redirect the user to the login screen. However, if the user is authenticated, the middleware will allow the request to proceed further into the application.

Of course, additional middleware can be written to perform a variety of tasks besides authentication. A CORS middleware might be responsible for adding the proper headers to all responses leaving your application. A logging middleware might log all incoming requests to your application.

There are several middleware included in the Laravel framework, including middleware for maintenance, authentication, CSRF protection, and more. All of these middleware are located in the `app/Http/Middleware` directory.

Defining Middleware

To create a new middleware, use the `make:middleware` Artisan command:

```
php artisan make:middleware AgeMiddleware
```

This command will place a new `AgeMiddleware` class within your `app/Http/Middleware` directory. In this middleware, we will only allow access to the route if the supplied `age` is greater than 200. Otherwise, we will redirect the users back to the "home" URI.

```
<?php

namespace App\Http\Middleware;

use Closure;

class AgeMiddleware
{
    /**
     * Run the request filter.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        if ($request->input('age') <= 200) {
            return redirect('home');
        }

        return $next($request);
    }

}
```

As you can see, if the given `age` is less than or equal to `200`, the middleware will return an HTTP redirect to the client; otherwise, the request will be passed further into the application. To pass the request deeper into the application (allowing the middleware to "pass"), simply call the `$next` callback with the `$request`.

It's best to envision middleware as a series of "layers" HTTP requests must pass through before they hit your application. Each layer can examine the request and even reject it entirely.

Before / *After* Middleware

Whether a middleware runs before or after a request depends on the middleware itself. For example, the following middleware would perform some task **before** the request is handled by the application:

```
<?php

namespace App\Http\Middleware;

use Closure;

class BeforeMiddleware
{
    public function handle($request, Closure $next)
    {
        // Perform action

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

However, this middleware would perform its task **after** the request is handled by the application:

```
<?php

namespace App\Http\Middleware;

use Closure;

class AfterMiddleware
{
    public function handle($request, Closure $next)
    {
        $response = $next($request);

        // Perform action

        return $response;
    }
}
```

Registering Middleware

Global Middleware

If you want a middleware to be run during every HTTP request to your application, simply list the middleware class in the `$middleware` property of your `app/Http/Kernel.php` class.

Assigning Middleware To Routes

If you would like to assign middleware to specific routes, you should first assign the middleware a short-hand key in your `app/Http/Kernel.php` file. By default, the `$routeMiddleware` property of this class contains entries for the middleware included with Laravel. To add your own, simply append it to this list and assign it a key of your choosing. For example:

```
// Within App\Http\Kernel Class...

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

Once the middleware has been defined in the HTTP kernel, you may use the `middleware` key in the route options array:

```
Route::get('admin/profile', ['middleware' => 'auth', function () {
    //
}]);
```

Use an array to assign multiple middleware to the route:

```
Route::get('/', ['middleware' => ['first', 'second'], function () {
    //
}]);
```

Instead of using an array, you may also chain the `middleware` method onto the route definition:

```
Route::get('/', function () {
    //
})->middleware(['first', 'second']);
```

When assigning middleware, you may also pass the fully qualified class name:

```
use App\Http\Middleware\FooMiddleware;

Route::get('admin/profile', ['middleware' => FooMiddleware::class, function () {
    //
}]);
```

Middleware Groups

Sometimes you may want to group several middleware under a single key to make them easier to assign to routes. You may do this using the `$middlewareGroups` property of your HTTP kernel.

Out of the box, Laravel comes with `web` and `api` middleware groups that contains common middleware you may want to apply to web UI and your API routes:

```
/**
 * The application's route middleware groups.
 *
 * @var array
 */
protected $middlewareGroups = [
    'web' => [
        \App\Http\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
        \Illuminate\Session\Middleware\StartSession::class,
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,
        \App\Http\Middleware\VerifyCsrfToken::class,
    ],

    'api' => [
        'throttle:60,1',
        'auth:api',
    ],
];
```

Middleware groups may be assigned to routes and controller actions using the same syntax as individual middleware. Again, middleware groups simply make it more convenient to assign many middleware to a route at once:

```
Route::group(['middleware' => ['web']], function () {
    //
});
```

Keep in mind, the `web` middleware group is automatically applied to your default `routes.php` file by the `RouteServiceProvider`.

Middleware Parameters

Middleware can also receive additional custom parameters. For example, if your application needs to verify that the authenticated user has a given "role" before performing a given action, you could create a `RoleMiddleware` that receives a role name as an additional argument.

Additional middleware parameters will be passed to the middleware after the `$next` argument:

```
<?php

namespace App\Http\Middleware;

use Closure;

class RoleMiddleware
{
    /**
     * Run the request filter.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @param  string  $role
     * @return mixed
     */
    public function handle($request, Closure $next, $role)
    {
        if (! $request->user()->hasRole($role)) {
            // Redirect...
        }

        return $next($request);
    }

}
```

Middleware parameters may be specified when defining the route by separating the middleware name and parameters with a `:`. Multiple parameters should be delimited by commas:

```
Route::put('post/{id}', ['middleware' => 'role:editor', function ($id) {
    //
}]);
```

Terminable Middleware

Sometimes a middleware may need to do some work after the HTTP response has already been sent to the browser. For example, the "session" middleware included with Laravel writes the session data to storage *after* the response has been sent to the browser. To accomplish this, define the middleware as "terminable" by adding a `terminate` method to the middleware:

```php
<?php

namespace Illuminate\Session\Middleware;

use Closure;

class StartSession
{
    public function handle($request, Closure $next)
    {
        return $next($request);
    }

    public function terminate($request, $response)
    {
        // Store the session data...
    }
}
```

The `terminate` method should receive both the request and the response. Once you have defined a terminable middleware, you should add it to the list of global middlewares in your HTTP kernel.

When calling the `terminate` method on your middleware, Laravel will resolve a fresh instance of the middleware from the service container. If you would like to use the same middleware instance when the `handle` and `terminate` methods are called, register the middleware with the container using the container's `singleton` method.