Middleware

Introduction

Middleware provide a convenient mechanism for inspecting and 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 your application's login screen. However, if the user is authenticated, the middleware will allow the request to proceed further into the application.

Defining Middleware

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

```
php artisan make:middleware EnsureTokenIsValid
```

This command will place a new EnsureTokenIsValid class within your app/Http/Middleware directory. In this middleware, we will only allow access to the route if the supplied token input matches a specified value. Otherwise, we will redirect the users back to the home URI:

```
<?php

namespace App\Http\Middleware;
use Closure;

class EnsureTokenIsValid
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        if ($request->input('token') !== 'my-secret-token') {
            return redirect('home');
        }

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

Middleware & Responses

Of course, a middleware can perform tasks before or after passing the request deeper into the application. 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);
    }
}
```

Registering Middleware

Global Middleware

If you want a middleware to run during every HTTP request to your application, 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 key in your application's app/Http/Kernel.php file. By default, the $routeMiddleware property of this class contains entries for the middleware included with Laravel. You may add your own middleware to this list and assign it a key of your choosing:

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

protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' =>
\Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,
    'cache.headers' => \Illuminate\Http\Middleware\SetCacheHeaders::class,
    'can' => \Illuminate\Auth\Middleware\Authorize::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
    'verified' => \Illuminate\Auth\Middleware\EnsureEmailIsVerified::class,
];
```

Once the middleware has been defined in the HTTP kernel, you may use the middleware method to assign middleware to a route:

```
Route::get('/profile', function () {
    //
})->middleware('auth');
```

You may assign multiple middleware to the route by passing an array of middleware names to the middleware method:

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

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

```
use App\Http\Middleware\EnsureTokenIsValid;

Route::get('/profile', function () {
    //
})->middleware(EnsureTokenIsValid::class);
```

Controllers

Introduction

Instead of defining all of your request handling logic as closures in your route files, you may wish to organize this behavior using "controller" classes. Controllers can group related request handling logic into a single class. For example, a UserController class might handle all incoming requests related to users, including showing, creating, updating, and deleting users. By default, controllers are stored in the app/Http/Controllers directory.

Writing Controllers

Basic Controllers

Let's take a look at an example of a basic controller. Note that the controller extends the base controller class included with Laravel: App\Http\Controllers\Controller:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\User;

class UserController extends Controller
{
    /**
     * Show the profile for a given user.
     *
     * @param  int  $id
     * @return \Illuminate\View\View
     */
    public function show($id)
    {
        return view('user.profile', [
            'user' => User::findOrFail($id)
        ]);
    }
}
```

You can define a route to this controller method like so:

```
use App\Http\Controllers\UserController;

Route::get('/user/{id}', [UserController::class, 'show']);
```

Database: Migrations

Introduction

Migrations are like version control for your database, allowing your team to define and share the application's database schema definition. If you have ever had to tell a teammate to manually add a column to their local database schema after pulling in your changes from source control, you've faced the problem that database migrations solve.

The Laravel Schema facade provides database agnostic support for creating and manipulating tables across all of Laravel's supported database systems. Typically, migrations will use this facade to create and modify database tables and columns.

Generating Migrations

You may use the make:migration Artisan command to generate a database migration. The new migration will be placed in your database/migrations directory. Each migration filename contains a timestamp that allows Laravel to determine the order of the migrations:

```
php artisan make:migration create_flights_table
```

Migration Structure

A migration class contains two methods: up and down. The up method is used to add new tables, columns, or indexes to your database, while the down method should reverse the operations performed by the up method.

Within both of these methods, you may use the Laravel schema builder to expressively create and modify tables. To learn about all of the methods available on the Schema builder, check out its documentation. For example, the following migration creates a flights table:

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateFlightsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('airline');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('flights');
    }
}
```