Laravel - Quick Guide

Laravel - Overview

Laravel is an open-source PHP framework, which is robust and easy to understand. It follows a model-view-controller design pattern. Laravel reuses the existing components of different frameworks which helps in creating a web application. The web application thus designed is more structured and pragmatic.

Laravel offers a rich set of functionalities which incorporates the basic features of PHP frameworks like CodeIgniter, Yii and other programming languages like Ruby on Rails. Laravel has a very rich set of features which will boost the speed of web development.

If you are familiar with Core PHP and Advanced PHP, Laravel will make your task easier. It saves a lot time if you are planning to develop a website from scratch. Moreover, a website built in Laravel is secure and prevents several web attacks.

Advantages of Laravel

Laravel offers you the following advantages, when you are designing a web application based on it –

- The web application becomes more scalable, owing to the Laravel framework.
- Considerable time is saved in designing the web application, since Laravel reuses the components from other framework in developing web application.
- It includes namespaces and interfaces, thus helps to organize and manage resources.

Composer

Composer is a tool which includes all the dependencies and libraries. It allows a user to create a project with respect to the mentioned framework (for example, those used in Laravel installation). Third party libraries can be installed easily with help of composer.

All the dependencies are noted in **composer.json** file which is placed in the source folder.

Artisan

Command line interface used in Laravel is called **Artisan**. It includes a set of commands which assists in building a web application. These commands are incorporated from Symphony framework, resulting in add-on features in Laravel 5.1 (latest version of Laravel).

Features of Laravel

Laravel offers the following key features which makes it an ideal choice for designing web applications –

Modularity

Laravel provides 20 built in libraries and modules which helps in enhancement of the application. Every module is integrated with Composer dependency manager which eases updates.

Testability

Laravel includes features and helpers which helps in testing through various test cases. This feature helps in maintaining the code as per the requirements.

Routing

Laravel provides a flexible approach to the user to define routes in the web application. Routing helps to scale the application in a better way and increases its performance.

Configuration Management

A web application designed in Laravel will be running on different environments, which means that there will be a constant change in its configuration. Laravel provides a consistent approach to handle the configuration in an efficient way.

Query Builder and ORM

Laravel incorporates a query builder which helps in querying databases using various simple chain methods. It provides **ORM** (Object Relational Mapper) and **ActiveRecord** implementation called Eloquent.

Schema Builder

Schema Builder maintains the database definitions and schema in PHP code. It also maintains a track of changes with respect to database migrations.

Template Engine

Laravel uses the **Blade Template** engine, a lightweight template language used to design hierarchical blocks and layouts with predefined blocks that include dynamic content.

E-mail

Laravel includes a **mail** class which helps in sending mail with rich content and attachments from the web application.

Authentication

User authentication is a common feature in web applications. Laravel eases designing authentication as it includes features such as **register, forgot password** and **send password reminders**.

Redis

Laravel uses **Redis** to connect to an existing session and general-purpose cache. Redis interacts with session directly.

Queues

Laravel includes queue services like emailing large number of users or a specified **Cron** job. These queues help in completing tasks in an easier manner without waiting for the previous task to be completed.

Event and Command Bus

Laravel 5.1 includes **Command Bus** which helps in executing commands and dispatch events in a simple way. The commands in Laravel act as per the application's lifecycle.

Laravel - Installation

For managing dependencies, Laravel uses **composer**. Make sure you have a Composer installed on your system before you install Laravel. In this chapter, you

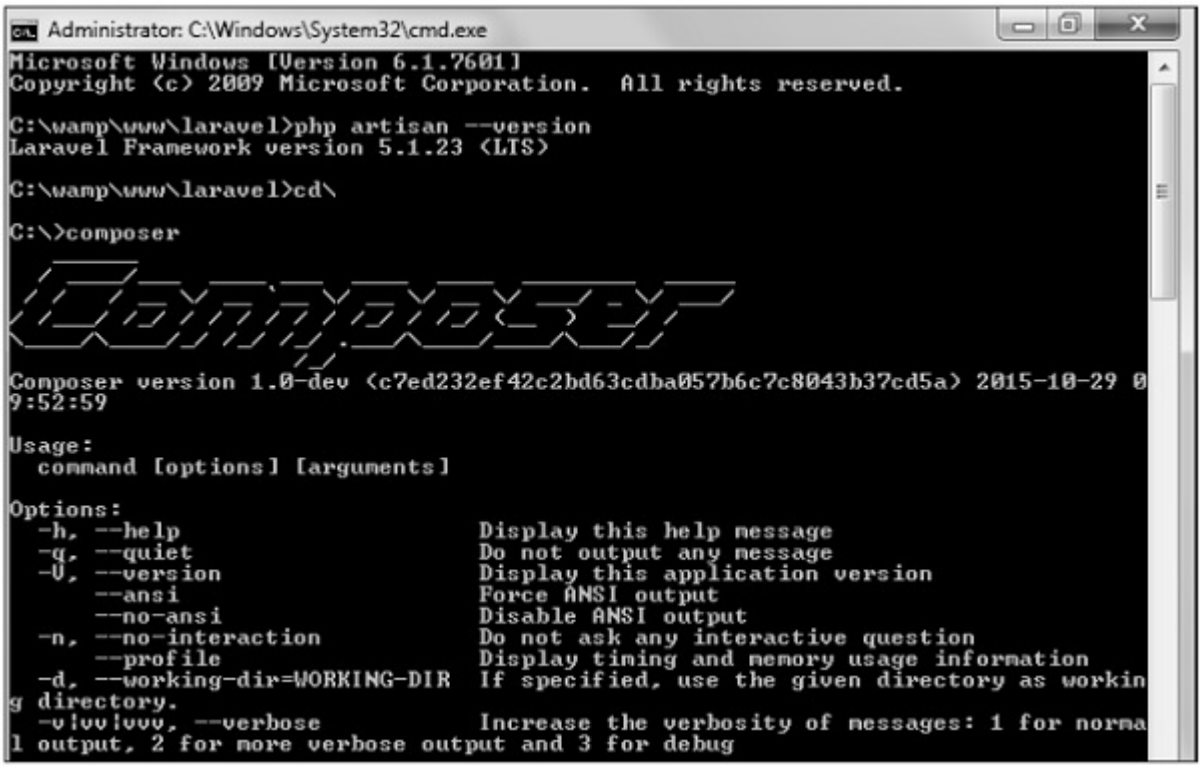
will see the installation process of Laravel.

You will have to follow the steps given below for installing Laravel onto your system –

Step 1 – Visit the following URL and download composer to install it on your system.

https://getcomposer.org/download/

Step 2 – After the Composer is installed, check the installation by typing the Composer command in the command prompt as shown in the following screenshot.



Step 3 – Create a new directory anywhere in your system for your new Laravel project. After that, move to path where you have created the new directory and type the following command there to install Laravel.

```
composer create-project laravel/laravel –-prefer-dist
```

Now, we will focus on installation of version 5.7. In Laravel version 5.7, you can install the complete framework by typing the following command –

```
composer create-project laravel/laravel test dev-develop
```

The output of the command is as shown below –

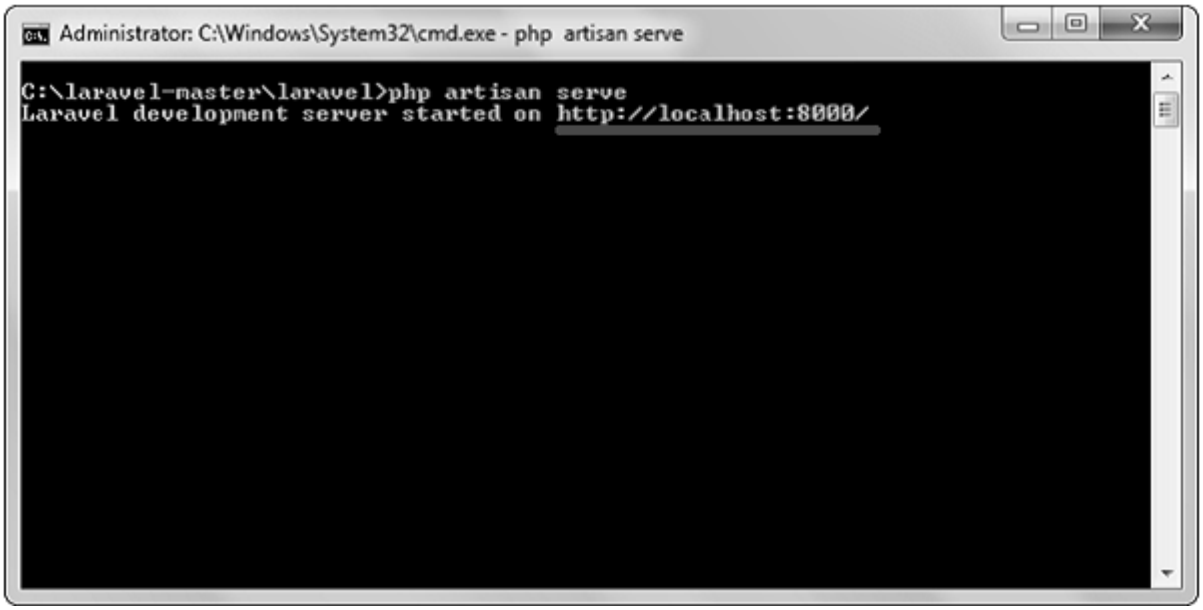
```
➜ code composer create-project laravel/laravel test dev-develop
Installing laravel/laravel (dev-develop d6acad21cb2288713d9c09a31f9b4ab86f116039)
  - Installing laravel/laravel (dev-develop develop): Cloning develop from cache
Created project in test
> @php -r "file_exists('.env') || copy('.env.example', '.env');"
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 71 installs, 0 updates, 0 removals
  - Installing vlucas/phpdotenv (v2.5.1): Loading from cache
  - Installing symfony/css-selector (v4.1.3): Loading from cache
  - Installing tijsverkoyen/css-to-inline-styles (2.2.1): Loading from cache
  - Installing symfony/polyfill-php72 (v1.9.0): Loading from cache
  - Installing symfony/polyfill-mbstring (v1.9.0): Loading from cache
  - Installing symfony/var-dumper (v4.1.3): Loading from cache
  - Installing symfony/routing (v4.1.3): Loading from cache
  - Installing symfony/process (v4.1.3): Loading from cache
  - Installing symfony/polyfill-ctype (v1.9.0): Loading from cache
  - Installing symfony/http-foundation (v4.1.3): Loading from cache
  - Installing symfony/event-dispatcher (v4.1.3): Loading from cache
  - Installing psr/log (1.0.2): Loading from cache
  - Installing symfony/debug (v4.1.3): Loading from cache
  - Installing symfony/http-kernel (v4.1.3): Loading from cache
  - Installing paragonie/random_compat (v9.99.99): Loading from cache
```

The Laravel framework can be directly installed with develop branch which includes the latest framework.

Step 4 – The above command will install Laravel in the current directory. Start the Laravel service by executing the following command.

```
php artisan serve
```

Step 5 – After executing the above command, you will see a screen as shown below –



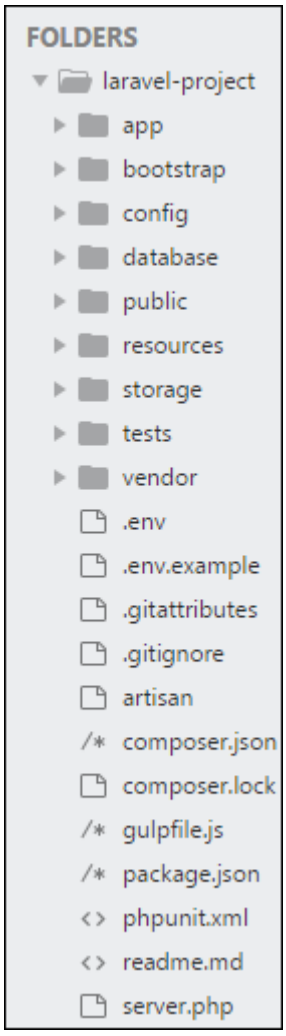
Step 6 – Copy the URL underlined in gray in the above screenshot and open that URL in the browser. If you see the following screen, it implies Laravel has been

installed successfully.

Laravel - Application Structure

The application structure in Laravel is basically the structure of folders, sub-folders and files included in a project. Once we create a project in Laravel, we get an overview of the application structure as shown in the image here.

The snapshot shown here refers to the root folder of Laravel namely **laravel-project**. It includes various sub-folders and files. The analysis of folders and files, along with their functional aspects is given below –

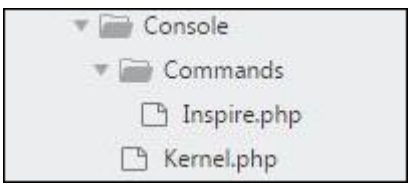


App

It is the application folder and includes the entire source code of the project. It contains events, exceptions and middleware declaration. The app folder comprises various sub folders as explained below –

Console

Console includes the artisan commands necessary for Laravel. It includes a directory named **Commands**, where all the commands are declared with the appropriate signature. The file **Kernal.php** calls the commands declared in **Inspire.php**.



If we need to call a specific command in Laravel, then we should make appropriate changes in this directory.

Events

This folder includes all the events for the project.

Events are used to trigger activities, raise errors or necessary validations and provide greater flexibility. Laravel keeps all the events under one directory. The default file included is **event.php** where all the basic events are declared.

Exceptions

This folder contains all the methods needed to handle exceptions. It also contains the file **handle.php** that handles all the exceptions.

Http

The **Http** folder has sub-folders for controllers, middleware and application requests. As Laravel follows the MVC design pattern, this folder includes model, controllers and views defined for the specific directories.

The **Middleware** sub-folder includes middleware mechanism, comprising the filter mechanism and communication between response and request.

The **Requests** sub-folder includes all the requests of the application.

Jobs

The **Jobs** directory maintains the activities queued for Laravel application. The base class is shared among all the Jobs and provides a central location to place them under one roof.

Listeners

Listeners are event-dependent and they include methods which are used to handle events and exceptions. For example, the **login** event declared includes a **LoginListener** event.

Policies

Policies are the PHP classes which includes the authorization logic. Laravel includes a feature to create all authorization logic within policy classes inside this sub folder.

Providers

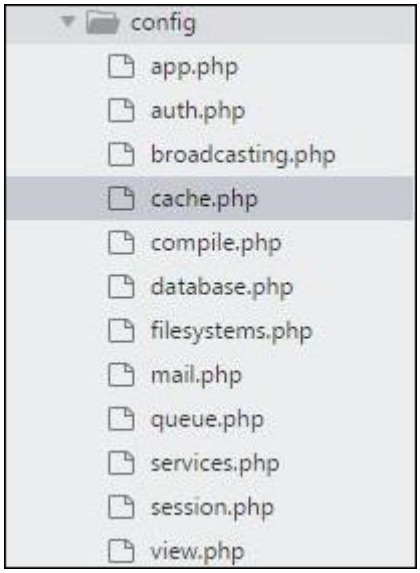
This folder includes all the service providers required to register events for core servers and to configure a Laravel application.

Bootstrap

This folder encloses all the application bootstrap scripts. It contains a sub-folder namely **cache**, which includes all the files associated for caching a web application. You can also find the file **app.php**, which initializes the scripts necessary for bootstrap.

Config

The **config** folder includes various configurations and associated parameters required for the smooth functioning of a Laravel application. Various files included within the config folder are as shown in the image here. The filenames work as per the functionality associated with them.



Database

As the name suggests, this directory includes various parameters for database functionalities. It includes three sub-directories as given below –

- **Seeds** – This contains the classes used for unit testing database.
- **Migrations** – This folder helps in queries for migrating the database used in the web application.
- **Factories** – This folder is used to generate large number of data records.

Public

It is the root folder which helps in initializing the Laravel application. It includes the following files and folders –

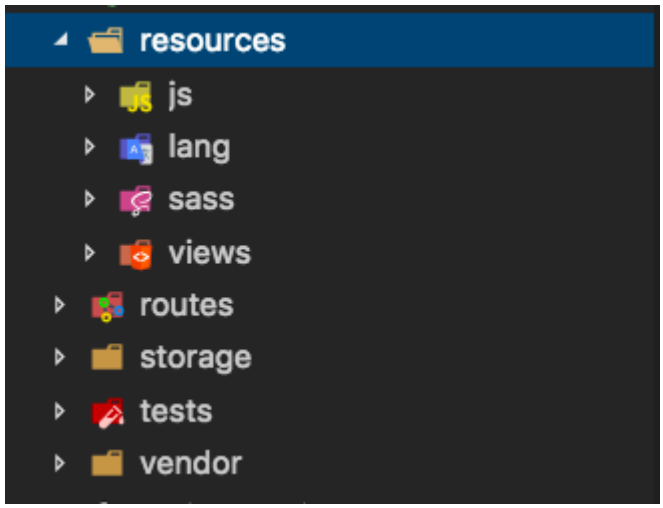
- **.htaccess** – This file gives the server configuration.
- **javascript and css** – These files are considered as assets.
- **index.php** – This file is required for the initialization of a web application.

Resources

Resources directory contains the files which enhances your web application. The sub-folders included in this directory and their purpose is explained below –

- **assets** – The assets folder include files such as LESS and SCSS, that are required for styling the web application.
- **lang** – This folder includes configuration for localization or internalization.
- **views** – Views are the HTML files or templates which interact with end users and play a primary role in MVC architecture.

Observe that the resources directory will be flattened instead of having an assets folder. The pictorial representation of same is shown below –



Storage

This is the folder that stores all the logs and necessary files which are needed frequently when a Laravel project is running. The sub-folders included in this directory and their purpose is given below –

- **app** – This folder contains the files that are called in succession.

- **framework** – It contains sessions, cache and views which are called frequently.
- **Logs** – All exceptions and error logs are tracked in this sub folder.

Tests

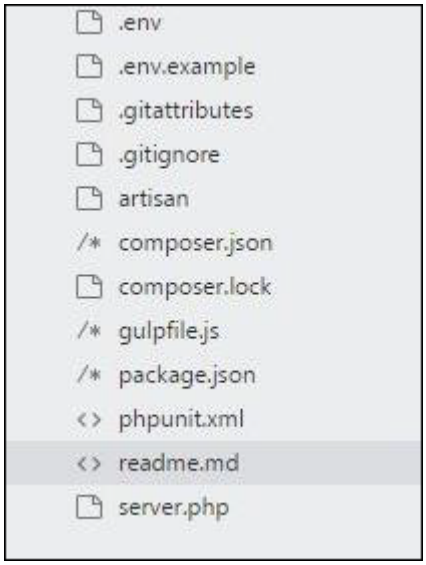
All the unit test cases are included in this directory. The naming convention for naming test case classes is **camel_case** and follows the convention as per the functionality of the class.

Vendor

Laravel is completely based on Composer dependencies, for example to install Laravel setup or to include third party libraries, etc. The Vendor folder includes all the composer dependencies.

In addition to the above mentioned files, Laravel also includes some other files which play a primary role in various functionalities such as GitHub configuration, packages and third party libraries.

The files included in the application structure are shown below –



Laravel - Configuration

In the previous chapter, we have seen that the basic configuration files of Laravel are included in the **config** directory. In this chapter, let us discuss the categories included in the configuration.

Environment Configuration

Environment variables are those which provide a list of web services to your web application. All the environment variables are declared in the **.env** file which includes the parameters required for initializing the configuration.

By default, the **.env** file includes following parameters −

```
APP_ENV = local
APP_DEBUG = true
APP_KEY = base64:ZPt2wmKE/X4eEhrzJU6XX4R93rCwYG8E2f8QUA7kGK8 =
APP_URL = http://localhost
DB_CONNECTION = mysql
DB_HOST = 127.0.0.1
DB_PORT = 3306
DB_DATABASE = homestead
DB_USERNAME = homestead
DB_PASSWORD = secret
CACHE_DRIVER = file
SESSION_DRIVER = file
QUEUE_DRIVER = sync
REDIS_HOST = 127.0.0.1
REDIS_PASSWORD = null
REDIS_PORT = 6379
MAIL_DRIVER = smtp
MAIL_HOST = mailtrap.ioMAIL_PORT = 2525
MAIL_USERNAME = null
MAIL_PASSWORD = null
MAIL_ENCRYPTION = null
```

Important Points

While working with basic configuration files of Laravel, the following points are to be noted −

- The **.env** file should not be committed to the application source control, since each developer or user has some predefined environment configuration for the web application.
- For backup options, the development team should include the **.env.example** file, which should contain the default configuration.

Retrieval of Environment Variables

All the environment variables declared in the **.env** file can be accessed by **env-helper** functions which will call the respective parameter. These variables are also listed into **$_ENV** global variable whenever application receives a request from the user end. You can access the environment variable as shown below −

```
'env' => env('APP_ENV', 'production'),
```

env-helper functions are called in the **app.php** file included in the **config** folder. The above given example is calling for the basic local parameter.

Accessing Configuration Values

You can easily access the configuration values anywhere in the application using the global config helper function. In case if the configuration values are not initialized, default values are returned.

For example, to set the default time zone, the following code is used −

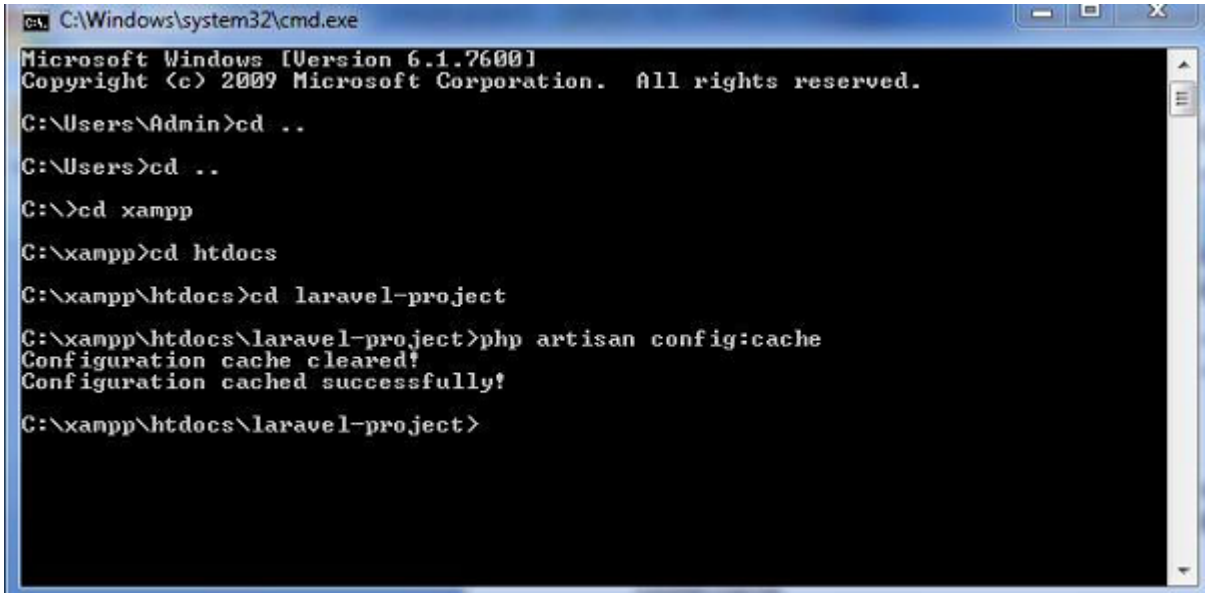
```
config(['app.timezone' => 'Asia/Kolkata']);
```

Caching of Configuration

To increase the performance and to boost the web application, it is important to cache all the configuration values. The command for caching the configuration values is −

```
config:cache
```

The following screenshot shows caching in a systematic approach −

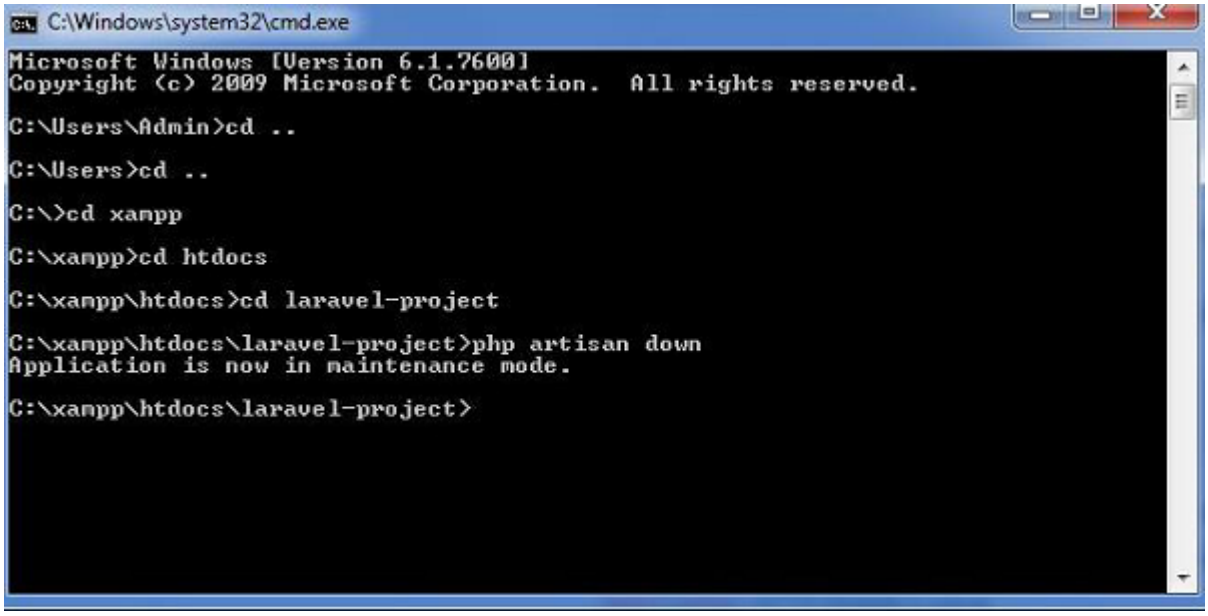


Maintenance Mode

Sometimes you may need to update some configuration values or perform maintenance on your website. In such cases, keeping it in **maintenance mode**, makes it easier for you. Such web applications which are kept in maintenance mode, throw an exception namely **MaintenanceModeException** with a status code of 503.

You can enable the maintenance mode on your Laravel web application using the following command –

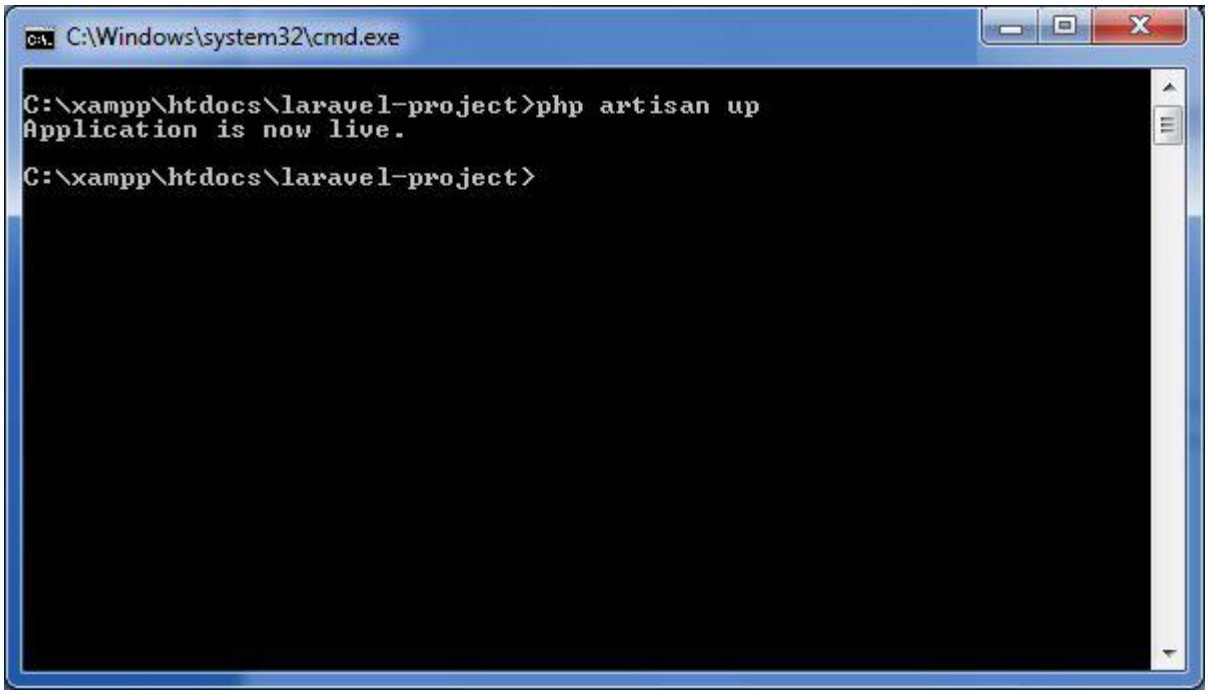
```
php artisan down
```



The following screenshot shows how the web application looks when it is down –

Once you finish working on updates and other maintenance, you can disable the maintenance mode on your web application using following command –

```
php artisan up
```



Now, you can find that the website shows the output with proper functioning and depicting that the maintenance mode is now removed as shown below –

Laravel - Routing

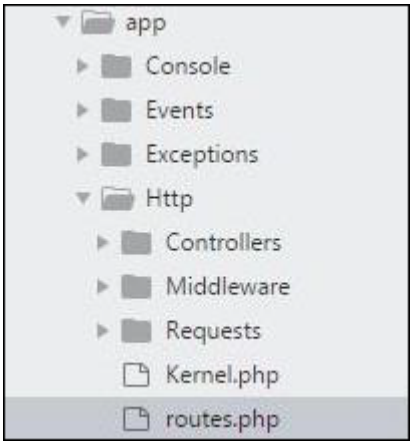
In Laravel, all requests are mapped with the help of routes. Basic routing routes the request to the associated controllers. This chapter discusses routing in Laravel.

Routing in Laravel includes the following categories –

- Basic Routing
- Route parameters
- Named Routes

Basic Routing

All the application routes are registered within the **app/routes.php** file. This file tells Laravel for the URIs it should respond to and the associated controller will give it a particular call. The sample route for the welcome page can be seen as shown in the screenshot given below –



```
Route::get ('/', function () {
   return view('welcome');});
```

Example

Observe the following example to understand more about Routing −

app/Http/routes.php

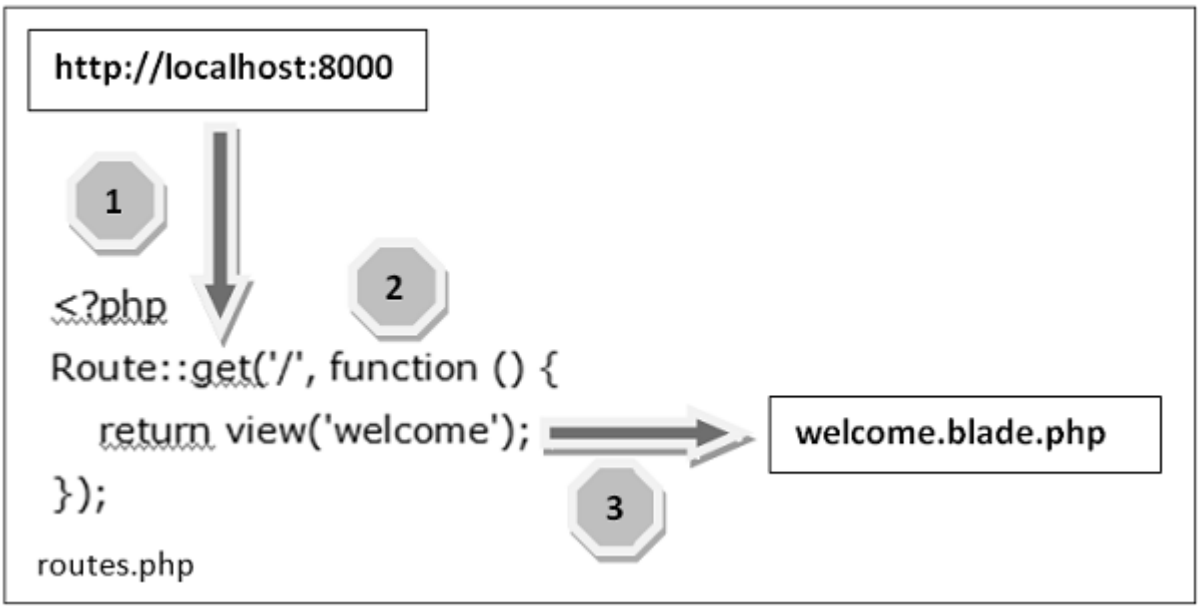
```
<?php
Route::get('/', function () {
   return view('welcome');
});
```

resources/view/welcome.blade.php

```
<!DOCTYPE html>
<html>
   <head>
      <title>Laravel</title>
      <link href = "https://fonts.googleapis.com/css?family=Lato:100"
         type = "text/css">

      <style>
         html, body {
            height: 100%;
         }
         body {
            margin: 0;
            padding: 0;
            width: 100%;
            display: table;
            font-weight: 100;
            font-family: 'Lato';
         }
         .container {
            text-align: center;
            display: table-cell;
            vertical-align: middle;
         }
         .content {
```

```
            text-align: center;
            display: inline-block;
         }
         .title {
            font-size: 96px;
         }
      </style>
   </head>

   <body>
      <div class = "container">

         <div class = "content">
            <div class = "title">Laravel 5.1</div>
         </div>

      </div>
   </body>
</html>
```

The routing mechanism is shown in the image given below −



Let us now understand the steps involved in routing mechanism in detail −

Step 1 − Initially, we should execute the root URL of the application.

Step 2 − Now, the executed URL should match with the appropriate method in the **route.php** file. In the present case, it should match the method and the root (‘/’) URL. This will execute the related function.

Step 3 – The function calls the template file **resources/views/welcome.blade.php.** Next, the function calls the **view()** function with argument **'welcome'** without using the **blade.php**.

This will produce the HTML output as shown in the image below –

Route Parameters

Sometimes in the web application, you may need to capture the parameters passed with the URL. For this, you should modify the code in **routes.php** file.

You can capture the parameters in **routes.php** file in two ways as discussed here –

Required Parameters

These parameters are those which should be mandatorily captured for routing the web application. For example, it is important to capture the user's identification number from the URL. This can be possible by defining route parameters as shown below –

```
Route::get('ID/{id}',function($id) {
   echo 'ID: '.$id;
});
```

Optional Parameters

Sometimes developers can produce parameters as optional and it is possible with the inclusion of **?** after the parameter name in URL. It is important to keep the default value mentioned as a parameter name. Look at the following example that shows how to define an optional parameter –

```
Route::get('user/{name?}', function ($name = 'TutorialsPoint') { return $name;});
```

The example above checks if the value matches to **TutorialsPoint** and accordingly routes to the defined URL.

Named Routes

Named routes allow a convenient way of creating routes. The chaining of routes can be specified using name method onto the route definition. The following code shows an example for creating named routes with controller −

```
Route::get('user/profile', 'UserController@showProfile')->name('profile');
```

The user controller will call for the function **showProfile** with parameter as **profile**. The parameters use **name** method onto the route definition.

Laravel - Middleware

Middleware acts as a bridge between a request and a response. It is a type of filtering mechanism. This chapter explains you the middleware mechanism in Laravel.

Laravel includes a middleware that verifies whether the user of the application is authenticated or not. If the user is authenticated, it redirects to the home page otherwise, if not, it redirects to the login page.

Middleware can be created by executing the following command −

```
php artisan make:middleware <middleware-name>
```

Replace the **<middleware-name>** with the name of your middleware. The middleware that you create can be seen at **app/Http/Middleware** directory.

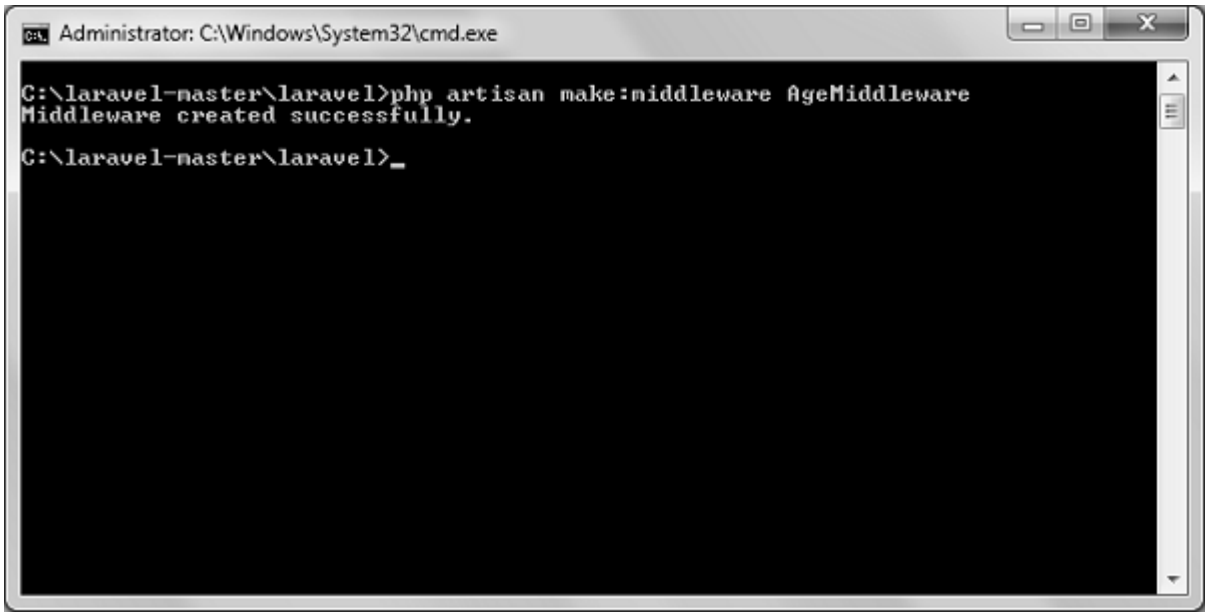
Example

Observe the following example to understand the middleware mechanism −

Step 1 − Let us now create AgeMiddleware. To create that, we need to execute the following command −

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

Step 2 – After successful execution of the command, you will receive the following output –



Step 3 – **AgeMiddleware** will be created at **app/Http/Middleware**. The newly created file will have the following code already created for you.

```
<?php

namespace App\Http\Middleware;
use Closure;

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

Registering Middleware

We need to register each and every middleware before using it. There are two types of Middleware in Laravel.

- Global Middleware
- Route Middleware

The **Global Middleware** will run on every HTTP request of the application, whereas the **Route Middleware** will be assigned to a specific route. The middleware can be registered at **app/Http/Kernel.php.** This file contains two properties

$middleware and **$routeMiddleware**. **$middleware** property is used to register Global Middleware and **$routeMiddleware** property is used to register route specific middleware.

To register the global middleware, list the class at the end of $middleware property.

```
protected $middleware = [
   \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::cla
   \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,
];
```

To register the route specific middleware, add the key and value to $routeMiddleware property.

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

Example

We have created **AgeMiddleware** in the previous example. We can now register it in route specific middleware property. The code for that registration is shown below.

The following is the code for **app/Http/Kernel.php** −

```
<?php

namespace App\Http;
use Illuminate\Foundation\Http\Kernel as HttpKernel;

class Kernel extends HttpKernel {
   protected $middleware = [
      \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::
```

```
        \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,
    ];

    protected $routeMiddleware = [
        'auth' => \App\Http\Middleware\Authenticate::class,
        'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasi
        'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
        'Age' => \App\Http\Middleware\AgeMiddleware::class,
    ];
}
```

Middleware Parameters

We can also pass parameters with the Middleware. For example, if your application has different roles like user, admin, super admin etc. and you want to authenticate the action based on role, this can be achieved by passing parameters with middleware. The middleware that we create contains the following function and we can pass our custom argument after the **$next** argument.

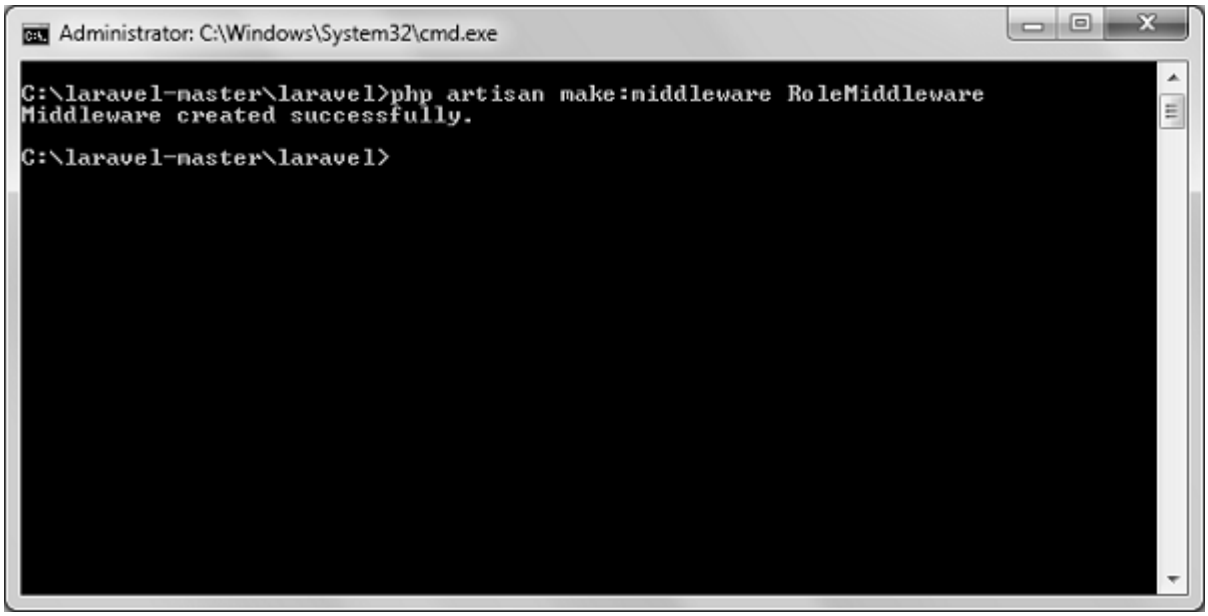
```
public function handle($request, Closure $next) {
   return $next($request);
}
```

Example

Step 1 – Create RoleMiddleware by executing the following command –

```
php artisan make:middleware RoleMiddleware
```

Step 2 – After successful execution, you will receive the following output –



Step 3 – Add the following code in the handle method of the newly created RoleMiddlewareat **app/Http/Middleware/RoleMiddleware.php.**

```
<?php

namespace App\Http\Middleware;
use Closure;

class RoleMiddleware {
   public function handle($request, Closure $next, $role) {
      echo "Role: ".$role;
      return $next($request);
   }
}
```

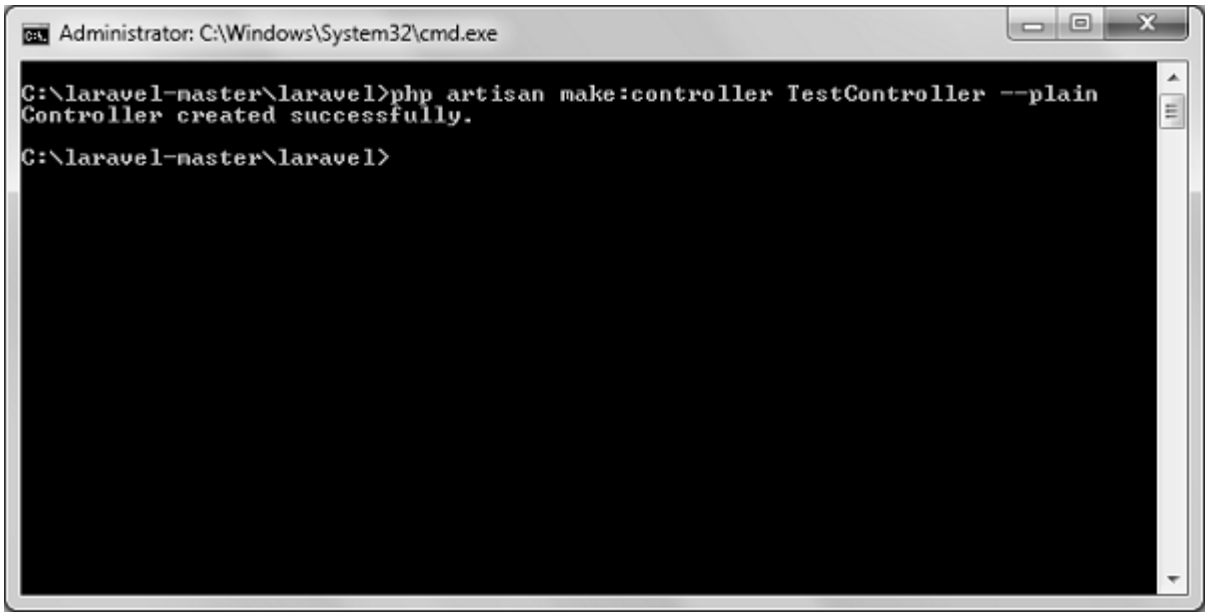
Step 4 – Register the RoleMiddleware in **app\Http\Kernel.php** file. Add the line highlighted in gray color in that file to register RoleMiddleware.

```
/**
 * The application's route middleware.
 *
 * @var array
 */
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'Age' => \App\Http\Middleware\AgeMiddleware::class,
    'After' => \App\Http\Middleware\AfterMiddleware::class,
    'Before' => \App\Http\Middleware\BeforeMiddleware::class,
    'First' => \App\Http\Middleware\FirstMiddleware::class,
    'Second' => \App\Http\Middleware\SecondMiddleware::class,
    'Role' => \App\Http\Middleware\RoleMiddleware::class,
];
```

Step 5 – Execute the following command to create **TestController** –

```
php artisan make:controller TestController --plain
```

Step 6 – After successful execution of the above step, you will receive the following output –



Step 7 – Copy the following lines of code to **app/Http/TestController.php** file.

app/Http/TestController.php

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests;
use App\Http\Controllers\Controller;

class TestController extends Controller {
   public function index() {
      echo "<br>Test Controller.";
   }
}
```

Step 8 – Add the following line of code in **app/Http/routes.php** file.

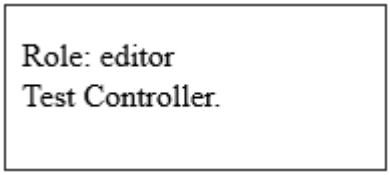
app/Http/routes.php

```
Route::get('role',[
   'middleware' => 'Role:editor',
   'uses' => 'TestController@index',
]);
```

Step 9 – Visit the following URL to test the Middleware with parameters

```
http://localhost:8000/role
```

Step 10 – The output will appear as shown in the following image.



Terminable Middleware

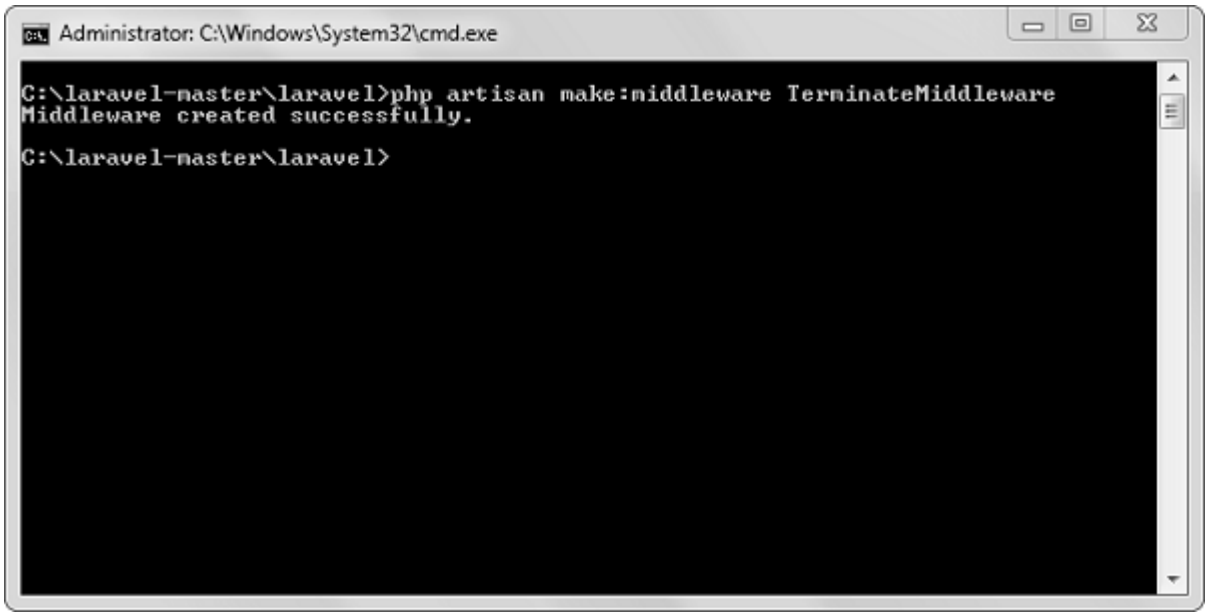
Terminable middleware performs some task after the response has been sent to the browser. This can be accomplished by creating a middleware with **terminate** method in the middleware. Terminable middleware should be registered with global middleware. The terminate method will receive two arguments **$request** and **$response.** Terminate method can be created as shown in the following code.

Example

Step 1 – Create **TerminateMiddleware** by executing the below command.

```
php artisan make:middleware TerminateMiddleware
```

Step 2 – The above step will produce the following output –



Step 3 – Copy the following code in the newly created **TerminateMiddleware** at **app/Http/Middleware/TerminateMiddleware.php.**

```
<?php

namespace App\Http\Middleware;
use Closure;

class TerminateMiddleware {
   public function handle($request, Closure $next) {
      echo "Executing statements of handle method of TerminateMiddlewa
      return $next($request);
   }

   public function terminate($request, $response) {
      echo "<br>Executing statements of terminate method of TerminateM
   }
}
```

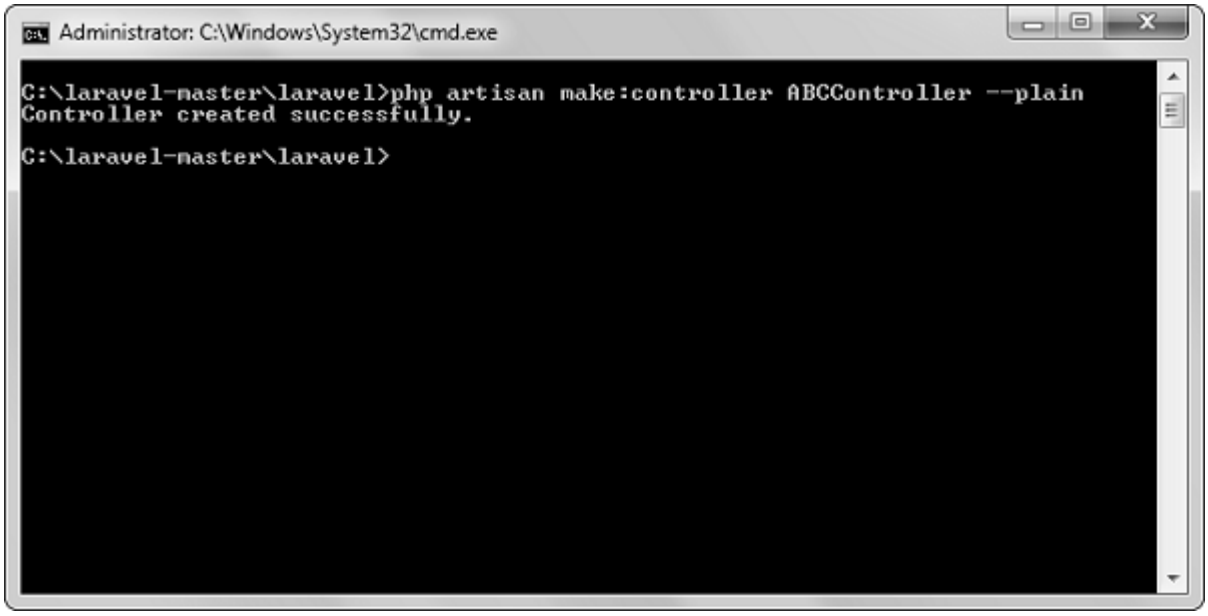
Step 4 – Register the **TerminateMiddleware** in **app\Http\Kernel.php** file. Add the line highlighted in gray color in that file to register TerminateMiddleware.

```
/**
 * The application's route middleware.
 *
 * @var array
 */
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'Age' => \App\Http\Middleware\AgeMiddleware::class,
    'After' => \App\Http\Middleware\AfterMiddleware::class,
    'Before' => \App\Http\Middleware\BeforeMiddleware::class,
    'First' => \App\Http\Middleware\FirstMiddleware::class,
    'Second' => \App\Http\Middleware\SecondMiddleware::class,
    'Role' => \App\Http\Middleware\RoleMiddleware::class,
    'terminate' => \App\Http\Middleware\TerminateMiddleware::class,
];
```

Step 5 – Execute the following command to create **ABCController**.

```
php artisan make:controller ABCController --plain
```

Step 6 – After the successful execution of the URL, you will receive the following output –



Step 7 – Copy the following code to **app/Http/ABCController.php** file.

app/Http/ABCController.php

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
```

```
use App\Http\Requests;
use App\Http\Controllers\Controller;

class ABCController extends Controller {
   public function index() {
      echo "<br>ABC Controller.";
   }
}
```

Step 8 − Add the following line of code in **app/Http/routes.php** file.

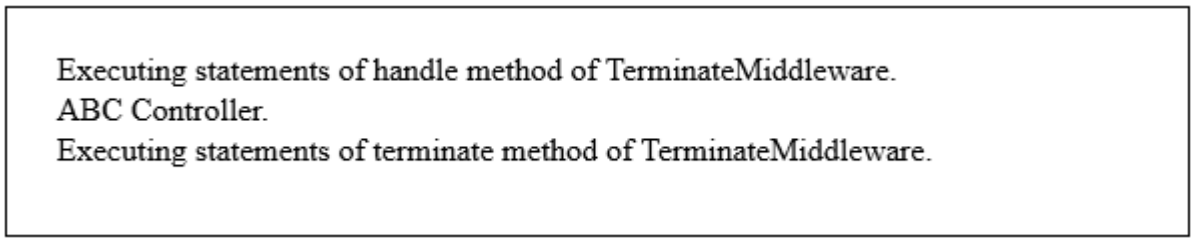
app/Http/routes.php

```
Route::get('terminate',[
  'middleware' => 'terminate',
  'uses' => 'ABCController@index',
]);
```

Step 9 − Visit the following URL to test the Terminable Middleware.

```
http://localhost:8000/terminate
```

Step 10 − The output will appear as shown in the following image.



Laravel - Namespaces

Namespaces can be defined as a class of elements in which each element has a unique name to that associated class. It may be shared with elements in other classes.

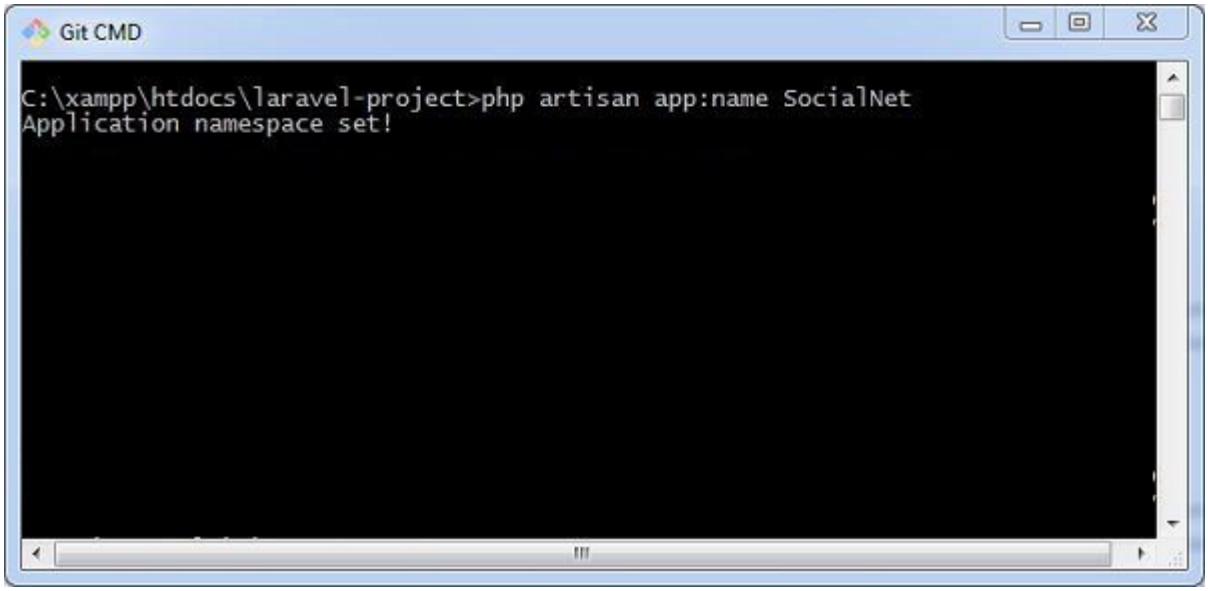
Declaration of namespace

The **use** keyword allows the developers to shorten the namespace.

```
use <namespace-name>;
```

The default namespace used in Laravel is App, however a user can change the namespace to match with web application. Creating user defined namespace with artisan command is mentioned as follows –

```
php artisan app:name SocialNet
```



The namespace once created can include various functionalities which can be used in controllers and various classes.

Laravel - Controllers

In the MVC framework, the letter 'C' stands for Controller. It acts as a directing traffic between Views and Models. In this chapter, you will learn about Controllers in Laravel.

Creating a Controller

Open the command prompt or terminal based on the operating system you are using and type the following command to create controller using the Artisan CLI (Command Line Interface).

```
php artisan make:controller <controller-name> --plain
```

Replace the <controller-name> with the name of your controller. This will create a plain constructor as we are passing the argument — **plain**. If you don't want to create a plain constructor, you can simply ignore the argument. The created constructor can be seen at **app/Http/Controllers**.

You will see that some basic coding has already been done for you and you can add your custom coding. The created controller can be called from routes.php by the following syntax.

Syntax

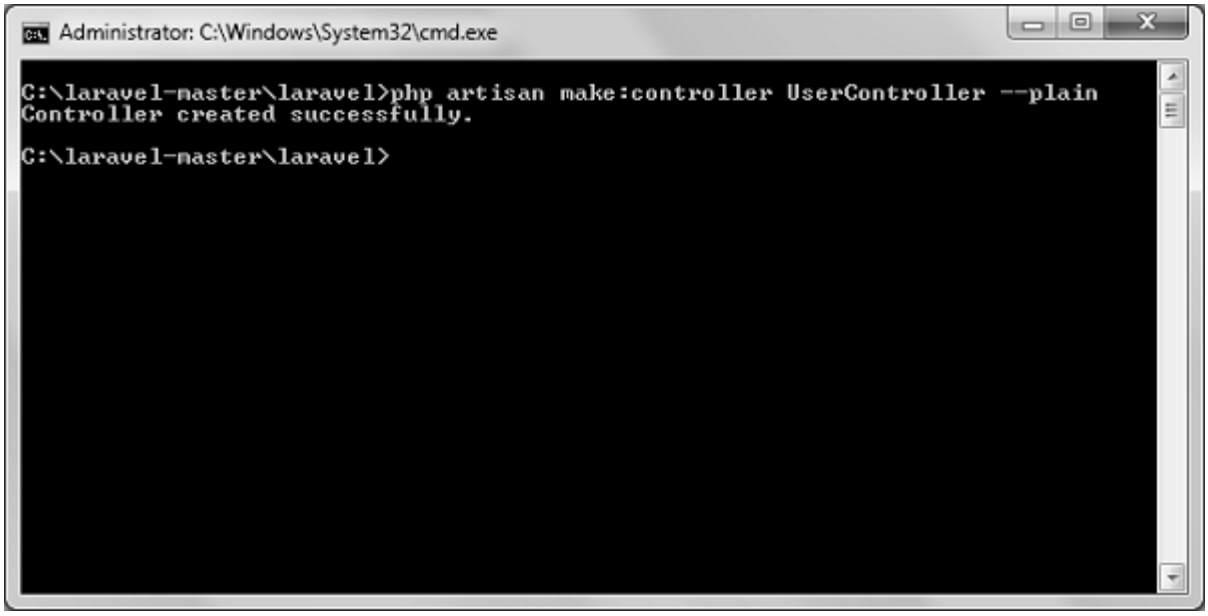
```
Route::get('base URI','controller@method');
```

Example

Step 1 – Execute the following command to create **UserController**.

```
php artisan make:controller UserController --plain
```

Step 2 – After successful execution, you will receive the following output.



Step 3 – You can see the created controller at **app/Http/Controller/UserController.php** with some basic coding already written for you and you can add your own coding based on your need.

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests;
use App\Http\Controllers\Controller;
```

```
class UserController extends Controller {
   //
}
```

Controller Middleware

We have seen middleware before and it can be used with controller also. Middleware can also be assigned to controller's route or within your controller's constructor. You can use the middleware method to assign middleware to the controller. The registered middleware can also be restricted to certain method of the controller.

Assigning Middleware to Route

```
Route::get('profile', [
  'middleware' => 'auth',
  'uses' => 'UserController@showProfile'
]);
```

Here we are assigning auth middleware to UserController in profile route.

Assigning Middleware within Controller's constructor

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests;
use App\Http\Controllers\Controller;

class UserController extends Controller {
   public function __construct() {
      $this->middleware('auth');
   }
}
```

Here we are assigning **auth** middleware using the middleware method in the **UserController** constructor.

Example

Step 1 – Add the following lines of code to the **app/Http/routes.php** file and save it.

routes.php

```
<?php
Route::get('/usercontroller/path',[
   'middleware' => 'First',
   'uses' => 'UserController@showPath'
]);
```

Step 2 – Create a middleware called **FirstMiddleware** by executing the following line of code.

```
php artisan make:middleware FirstMiddleware
```

Step 3 – Add the following code into the **handle** method of the newly created FirstMiddleware at **app/Http/Middleware**.

FirstMiddleware.php

```
<?php

namespace App\Http\Middleware;
use Closure;

class FirstMiddleware {
   public function handle($request, Closure $next) {
      echo '<br>First Middleware';
      return $next($request);
   }
}
```

Step 4 – Create a middleware called **SecondMiddleware** by executing the following command.

```
php artisan make:middleware SecondMiddleware
```

Step 5 – Add the following code in the handle method of the newly created SecondMiddleware at **app/Http/Middleware**.

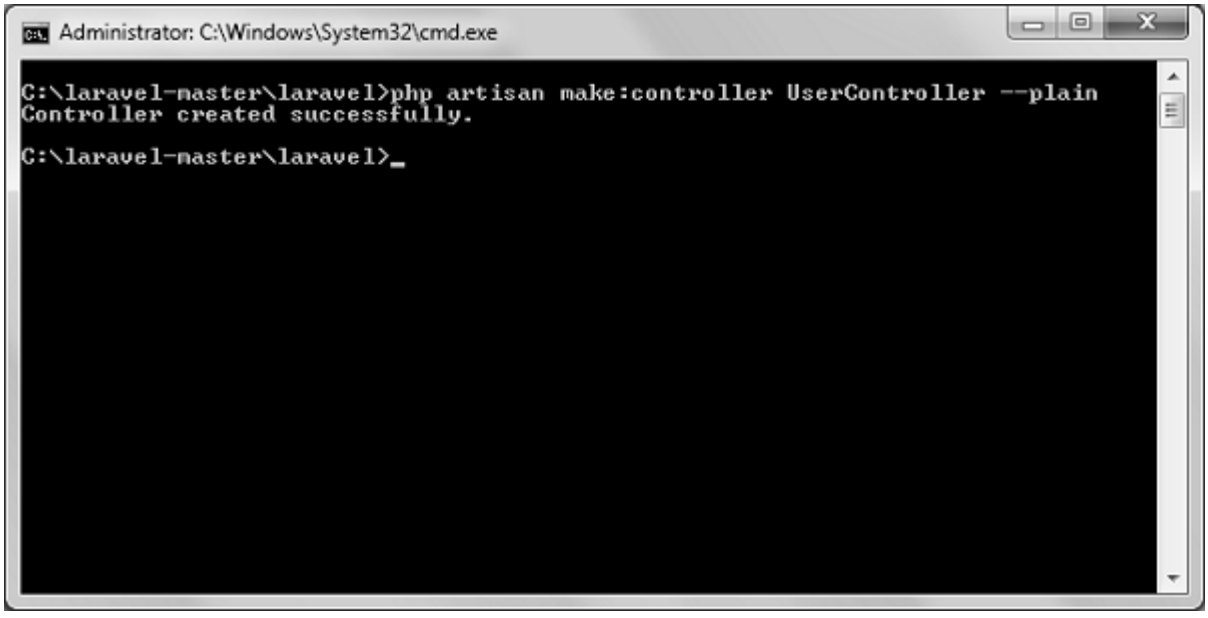
SecondMiddleware.php

```
<?php

namespace App\Http\Middleware;
use Closure;

class SecondMiddleware {
   public function handle($request, Closure $next) {
      echo '<br>Second Middleware';
      return $next($request);
   }
}
```

Step 6 – Create a controller called **UserController** by executing the following line.

```
php artisan make:controller UserController --plain
```

Step 7 – After successful execution of the URL, you will receive the following output –



Step 8 – Copy the following code to **app/Http/UserController.php** file.

app/Http/UserController.php

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests;
```

```
use App\Http\Controllers\Controller;

class UserController extends Controller {
   public function __construct() {
      $this->middleware('Second');
   }
   public function showPath(Request $request) {
      $uri = $request->path();
      echo '<br>URI: '.$uri;

      $url = $request->url();
      echo '<br>';

      echo 'URL: '.$url;
      $method = $request->method();
      echo '<br>';

      echo 'Method: '.$method;
   }
}
```

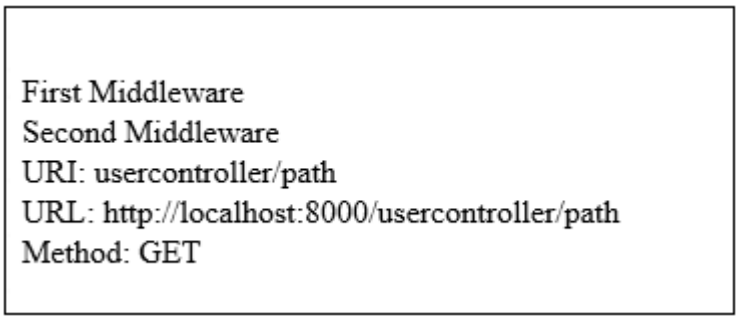
Step 9 – Now launch the php's internal web server by executing the following command, if you haven't executed it yet.

```
php artisan serve
```

Step 10 – Visit the following URL.

```
http://localhost:8000/usercontroller/path
```

Step 11 – The output will appear as shown in the following image.



Restful Resource Controllers

Often while making an application we need to perform **CRUD (Create, Read, Update, Delete)** operations. Laravel makes this job easy for us. Just create a controller and Laravel will automatically provide all the methods for the CRUD operations. You can also register a single route for all the methods in routes.php file.

Example

Step 1 – Create a controller called **MyController** by executing the following command.

```
php artisan make:controller MyController
```

Step 2 – Add the following code in

app/Http/Controllers/MyController.php file.

app/Http/Controllers/MyController.php

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests;
use App\Http\Controllers\Controller;

class MyController extends Controller {
   public function index() {
      echo 'index';
   }
   public function create() {
      echo 'create';
   }
   public function store(Request $request) {
      echo 'store';
   }
   public function show($id) {
      echo 'show';
   }
   public function edit($id) {
      echo 'edit';
   }
   public function update(Request $request, $id) {
```

```
      echo 'update';
   }
   public function destroy($id) {
      echo 'destroy';
   }
}
```

Step 3 – Add the following line of code in **app/Http/routes.php** file.

app/Http/routes.php

```
Route::resource('my','MyController');
```

Step 4 – We are now registering all the methods of MyController by registering a controller with resource. Below is the table of actions handled by resource controller.

Step 5 – Try executing the URLs shown in the following table.

Implicit Controllers

Implicit Controllers allow you to define a single route to handle every action in the controller. You can define it in route.php file with **Route:controller** method as shown below.

```
Route::controller(‘base URI’,’<class-name-of-the-controller>’);
```

Replace the <class-name-of-the-controller> with the class name that you have given to your controller.

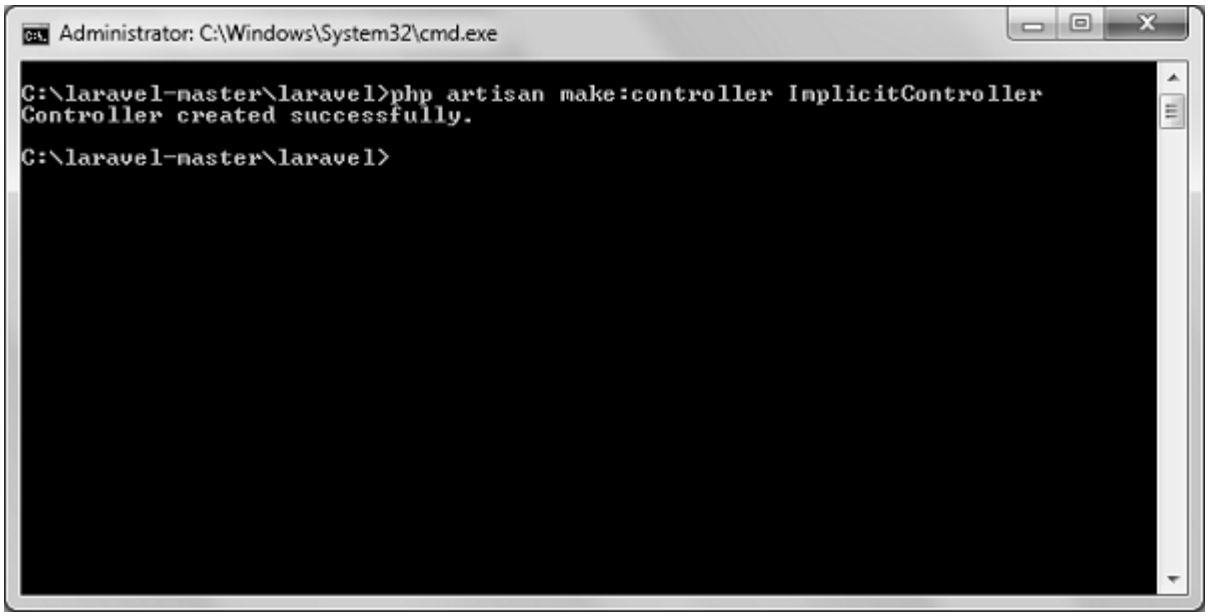
The method name of the controller should start with HTTP verb like get or post. If you start it with get, it will handle only get request and if it starts with post then it will handle the post request. After the HTTP verb you can, you can give any name to the method but it should follow the title case version of the URI.

Example

Step 1 – Execute the below command to create a controller. We have kept the class name **ImplicitController**. You can give any name of your choice to the class.

```
php artisan make:controller ImplicitController --plain
```

Step 2 – After successful execution of step 1, you will receive the following output –



Step 3 – Copy the following code to

app/Http/Controllers/ImplicitController.php file.

app/Http/Controllers/ImplicitController.php

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests;
use App\Http\Controllers\Controller;

class ImplicitController extends Controller {
   /**
      * Responds to requests to GET /test
   */
   public function getIndex() {
      echo 'index method';
   }

   /**
      * Responds to requests to GET /test/show/1
   */
   public function getShow($id) {
      echo 'show method';
   }

   /**
```

```
      * Responds to requests to GET /test/admin-profile
   */
   public function getAdminProfile() {
      echo 'admin profile method';
   }

   /**
      * Responds to requests to POST /test/profile
   */
   public function postProfile() {
      echo 'profile method';
   }
}
```

Step 4 – Add the following line to **app/Http/routes.php** file to route the requests to specified controller.

app/Http/routes.php

```
Route::controller('test','ImplicitController');
```

Constructor Injection

The Laravel service container is used to resolve all Laravel controllers. As a result, you are able to type-hint any dependencies your controller may need in its constructor. The dependencies will automatically be resolved and injected into the controller instance.

Example

Step 1 – Add the following code to **app/Http/routes.php** file.

app/Http/routes.php

```
class MyClass{
   public $foo = 'bar';
}
Route::get('/myclass','ImplicitController@index');
```

Step 2 – Add the following code to

app/Http/Controllers/ImplicitController.php file.

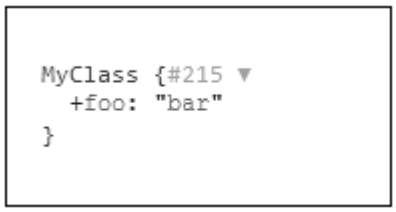
app/Http/Controllers/ImplicitController.php

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests;
use App\Http\Controllers\Controller;

class ImplicitController extends Controller {
   private $myclass;

   public function __construct(\MyClass $myclass) {
      $this->myclass = $myclass;
   }
   public function index() {
      dd($this->myclass);
   }
}
```

Step 3 – Visit the following URL to test the constructor injection.

```
http://localhost:8000/myclass
```

Step 4 – The output will appear as shown in the following image.



Method Injection

In addition to constructor injection, you may also type — hint dependencies on your controller's action methods.

Example

Step 1 – Add the following code to **app/Http/routes.php** file.

app/Http/routes.php

```
class MyClass{
   public $foo = 'bar';
}
Route::get('/myclass','ImplicitController@index');
```

Step 2 – Add the following code to

app/Http/Controllers/ImplicitController.php file.

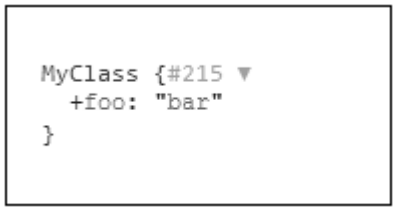
app/Http/Controllers/ImplicitController.php

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests;
use App\Http\Controllers\Controller;

class ImplicitController extends Controller {
   public function index(\MyClass $myclass) {
      dd($myclass);
   }
}
```

Step 3 – Visit the following URL to test the constructor injection.

```
http://localhost:8000/myclass
```

It will produce the following output –



Laravel - Request

In this chapter, you will learn in detail about Requests in Laravel.

Retrieving the Request URI

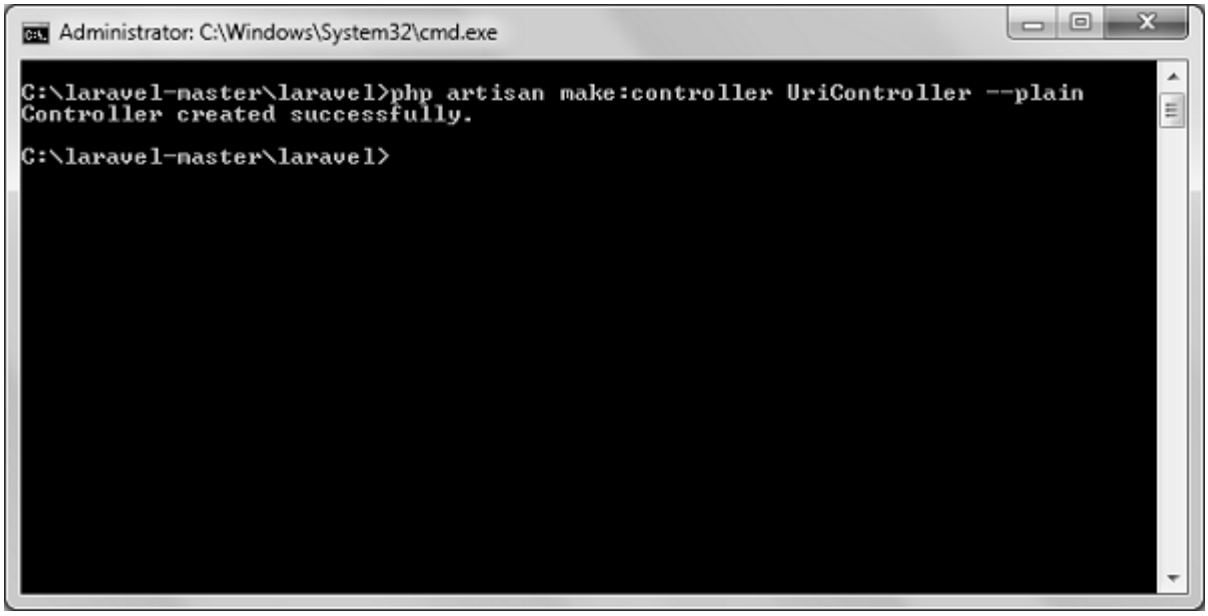
The **"path"** method is used to retrieve the requested URI. The **is** method is used to retrieve the requested URI which matches the particular pattern specified in the argument of the method. To get the full URL, we can use the **url** method.

Example

Step 1 – Execute the below command to create a new controller called **UriController**.

```
php artisan make:controller UriController –plain
```

Step 2 – After successful execution of the URL, you will receive the following output –



Step 3 – After creating a controller, add the following code in that file.

app/Http/Controllers/UriController.php

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests;
use App\Http\Controllers\Controller;

class UriController extends Controller {

   public function index(Request $request) {
      // Usage of path method
```

```
        $path = $request->path();
        echo 'Path Method: '.$path;
        echo '<br>';

        // Usage of is method
        $pattern = $request->is('foo/*');
        echo 'is Method: '.$pattern;
        echo '<br>';

        // Usage of url method
        $url = $request->url();
        echo 'URL method: '.$url;
    }
}
```

Step 4 – Add the following line in the **app/Http/route.php** file.

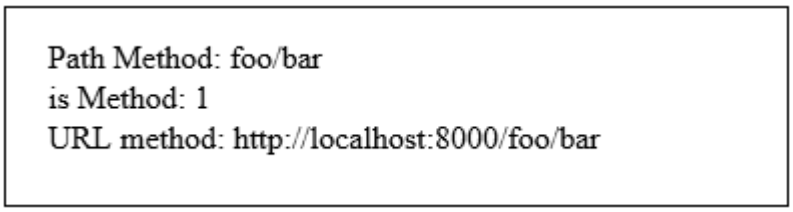
app/Http/route.php

```
Route::get('/foo/bar','UriController@index');
```

Step 5 – Visit the following URL.

```
http://localhost:8000/foo/bar
```

Step 6 – The output will appear as shown in the following image.



Retrieving Input

The input values can be easily retrieved in Laravel. No matter what method was used **"get"** or **"post"**, the Laravel method will retrieve input values for both the methods the same way. There are two ways we can retrieve the input values.

- Using the input() method
- Using the properties of Request instance

Using the input() method

The **input()** method takes one argument, the name of the field in form. For example, if the form contains username field then we can access it by the following way.

```
$name = $request->input('username');
```

Using the properties of Request instance

Like the **input()** method, we can get the username property directly from the request instance.

```
$request->username
```

Example

Observe the following example to understand more about Requests −

Step 1 − Create a Registration form, where user can register himself and store the form at **resources/views/register.php**

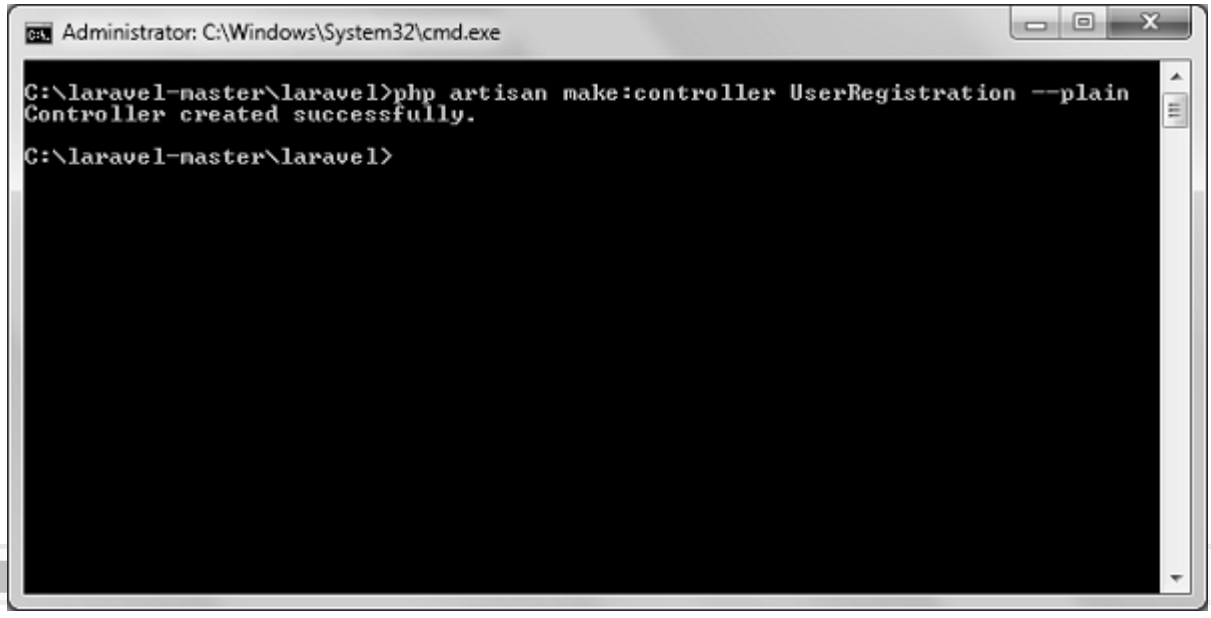
```
<html>

   <head>
      <title>Form Example</title>
   </head>

   <body>
      <form action = "/user/register" method = "post">
         <input type = "hidden" name = "_token" value = "<?php echo cs

         <table>
            <tr>
               <td>Name</td>
               <td><input type = "text" name = "name" /></td>
            </tr>
            <tr>
               <td>Username</td>
               <td><input type = "text" name = "username" /></td>
            </tr>
```

```
            <tr>
               <td>Password</td>
               <td><input type = "text" name = "password" /></td>
            </tr>
            <tr>
               <td colspan = "2" align = "center">
                  <input type = "submit" value = "Register" />
               </td>
            </tr>
         </table>

      </form>
   </body>
</html>
```

Step 2 – Execute the below command to create a **UserRegistration** controller.

```
php artisan make:controller UserRegistration --plain
```

Step 3 – After successful execution of the above step, you will receive the following output –



Step 4 – Copy the following code in

app/Http/Controllers/UserRegistration.php controller.

app/Http/Controllers/UserRegistration.php

```
<?php
```

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests;
use App\Http\Controllers\Controller;

class UserRegistration extends Controller {
   public function postRegister(Request $request) {
      //Retrieve the name input field
      $name = $request->input('name');
      echo 'Name: '.$name;
      echo '<br>';

      //Retrieve the username input field
      $username = $request->username;
      echo 'Username: '.$username;
      echo '<br>';

      //Retrieve the password input field
      $password = $request->password;
      echo 'Password: '.$password;
   }
}
```

Step 5 − Add the following line in **app/Http/routes.php** file.

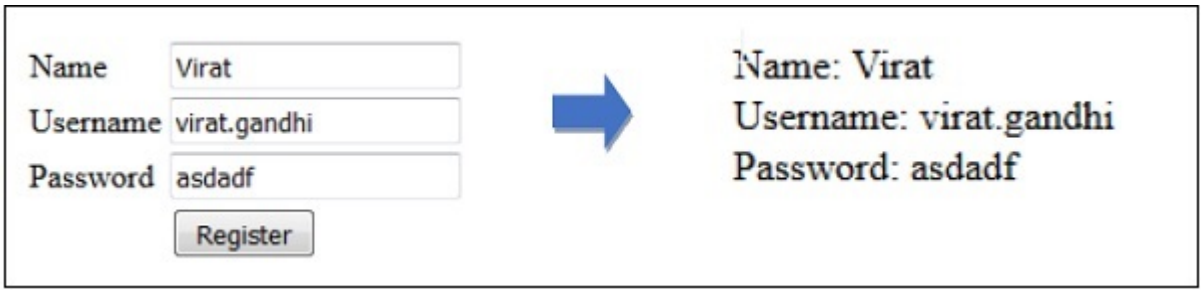
app/Http/routes.php

```
Route::get('/register',function() {
  return view('register');
});
Route::post('/user/register',array('uses'=>'UserRegistration@postRegister'));
```

Step 6 − Visit the following URL and you will see the registration form as shown in the below figure. Type the registration details and click Register and you will see on the second page that we have retrieved and displayed the user registration details.

```
http://localhost:8000/register
```

Step 7 − The output will look something like as shown in below the following images.



Laravel - Cookie

Cookies play an important role while dealing a user's session on a web application. In this chapter, you will learn about working with cookies in Laravel based web applications.

Creating a Cookie

Cookie can be created by global cookie helper of Laravel. It is an instance of **Symfony\Component\HttpFoundation\Cookie**. The cookie can be attached to the response using the withCookie() method. Create a response instance of **Illuminate\Http\Response** class to call the withCookie() method. Cookie generated by the Laravel are encrypted and signed and it can't be modified or read by the client.

Here is a sample code with explanation.

```
//Create a response instance
$response = new Illuminate\Http\Response('Hello World');

//Call the withCookie() method with the response method
$response->withCookie(cookie('name', 'value', $minutes));

//return the response
return $response;
```

Cookie() method will take 3 arguments. First argument is the name of the cookie, second argument is the value of the cookie and the third argument is the duration of the cookie after which the cookie will get deleted automatically.

Cookie can be set forever by using the forever method as shown in the below code.

```
$response->withCookie(cookie()->forever('name', 'value'));
```

Retrieving a Cookie

Once we set the cookie, we can retrieve the cookie by cookie() method. This cookie() method will take only one argument which will be the name of the cookie. The cookie method can be called by using the instance of **Illuminate\Http\Request**.

Here is a sample code.

```
//'name' is the name of the cookie to retrieve the value of
$value = $request->cookie('name');
```

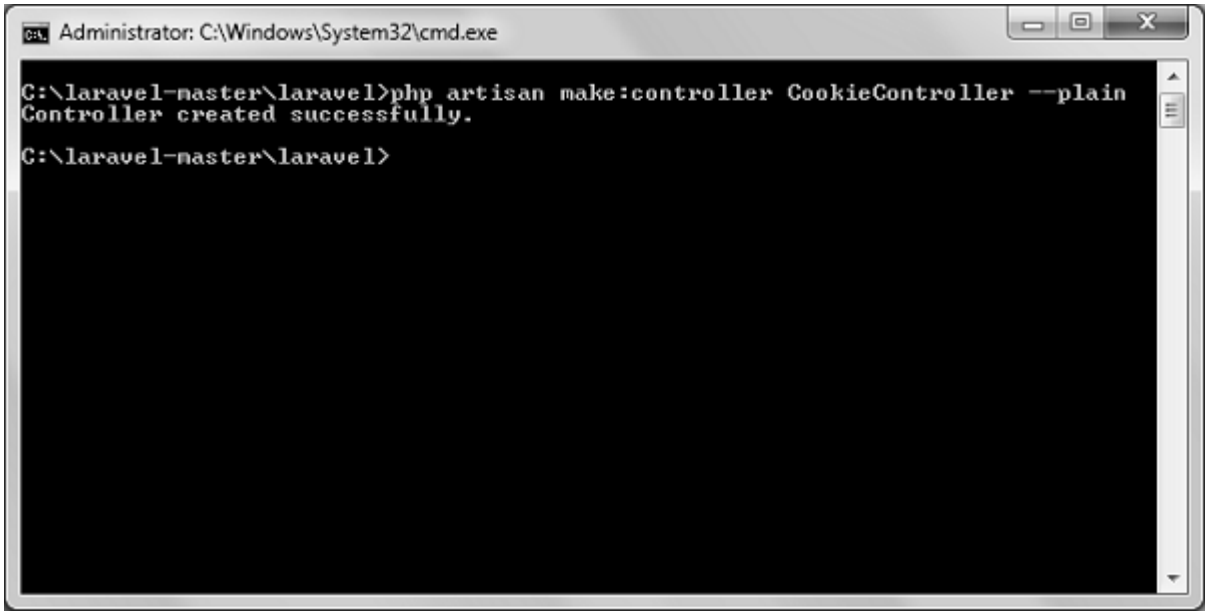
Example

Observe the following example to understand more about Cookies –

Step 1 – Execute the below command to create a controller in which we will manipulate the cookie.

```
php artisan make:controller CookieController --plain
```

Step 2 – After successful execution, you will receive the following output –



Step 3 – Copy the following code in

app/Http/Controllers/CookieController.php file.

app/Http/Controllers/CookieController.php

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Http\Response;
use App\Http\Requests;
use App\Http\Controllers\Controller;

class CookieController extends Controller {
   public function setCookie(Request $request) {
      $minutes = 1;
      $response = new Response('Hello World');
      $response->withCookie(cookie('name', 'virat', $minutes));
      return $response;
   }
   public function getCookie(Request $request) {
      $value = $request->cookie('name');
      echo $value;
   }
}
```

Step 4 – Add the following line in **app/Http/routes.php file**.

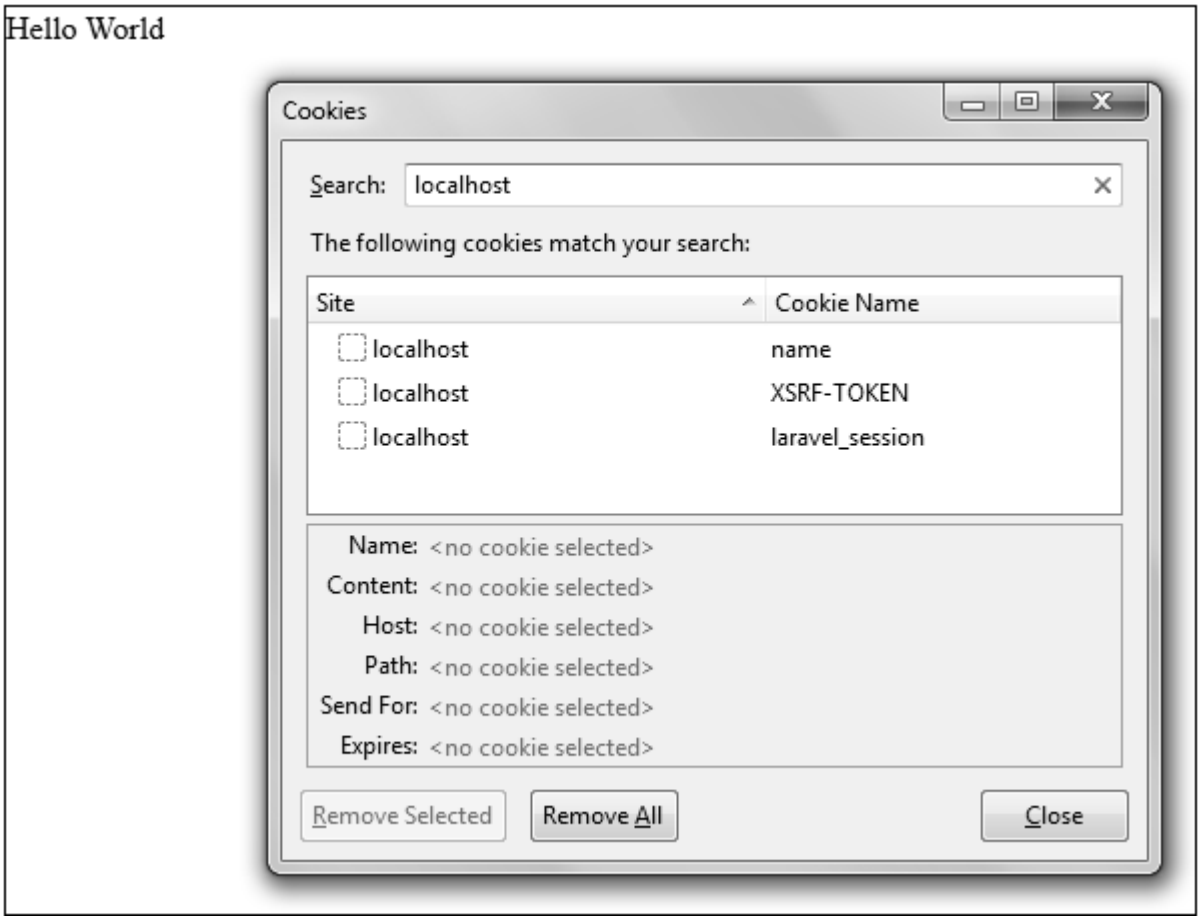
app/Http/routes.php

```
Route::get('/cookie/set','CookieController@setCookie');
Route::get('/cookie/get','CookieController@getCookie');
```

Step 5 – Visit the following URL to set the cookie.

```
http://localhost:8000/cookie/set
```

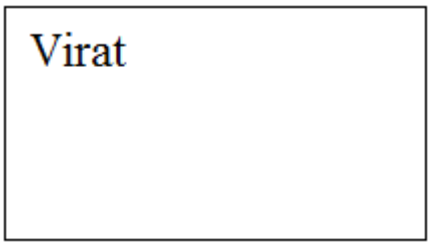
Step 6 – The output will appear as shown below. The window appearing in the screenshot is taken from firefox but depending on your browser, cookie can also be checked from the cookie option.



Step 7 – Visit the following URL to get the cookie from the above URL.

```
http://localhost:8000/cookie/get
```

Step 8 – The output will appear as shown in the following image.



Laravel - Response

A web application responds to a user's request in many ways depending on many parameters. This chapter explains you in detail about responses in Laravel web applications.

Basic Response

Laravel provides several different ways to return response. Response can be sent either from route or from controller. The basic response that can be sent is simple

string as shown in the below sample code. This string will be automatically converted to appropriate HTTP response.

Example

Step 1 – Add the following code to **app/Http/routes.php** file.

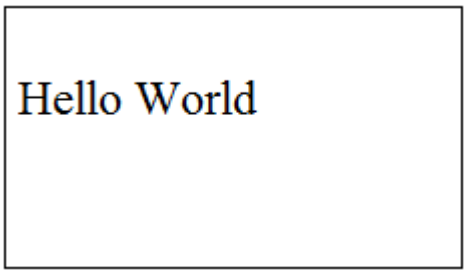
app/Http/routes.php

```
Route::get('/basic_response', function () {
   return 'Hello World';
});
```

Step 2 – **Visit** the following URL to test the basic response.

```
http://localhost:8000/basic_response
```

Step 3 – The output will appear as shown in the following image.



Attaching Headers

The response can be attached to headers using the header() method. We can also attach the series of headers as shown in the below sample code.

```
return response($content,$status)
   ->header('Content-Type', $type)
   ->header('X-Header-One', 'Header Value')
   ->header('X-Header-Two', 'Header Value');
```

Example

Observe the following example to understand more about Response –

Step 1 – Add the following code to **app/Http/routes.php** file.

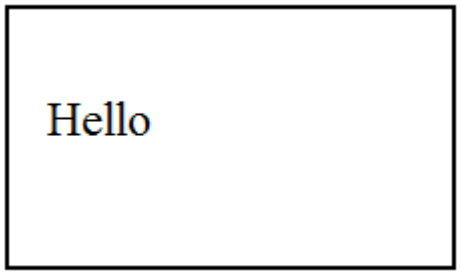
app/Http/routes.php

```
Route::get('/header',function() {
   return response("Hello", 200)->header('Content-Type', 'text/html');
});
```

Step 2 – Visit the following URL to test the basic response.

```
http://localhost:8000/header
```

Step 3 – The output will appear as shown in the following image.



Attaching Cookies

The **withcookie()** helper method is used to attach cookies. The cookie generated with this method can be attached by calling **withcookie()** method with response instance. By default, all cookies generated by Laravel are encrypted and signed so that they can't be modified or read by the client.

Example

Observe the following example to understand more about attaching cookies –

Step 1 – Add the following code to **app/Http/routes.php** file.

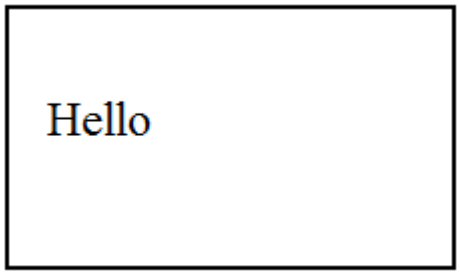
app/Http/routes.php

```
Route::get('/cookie',function() {
   return response("Hello", 200)->header('Content-Type', 'text/html')
      ->withcookie('name','Virat Gandhi');
});
```

Step 2 – **Visit** the following URL to test the basic response.

```
http://localhost:8000/cookie
```

Step 3 – The output will appear as shown in the following image.



JSON Response

JSON response can be sent using the json method. This method will automatically set the Content-Type header to **application/json**. The **json** method will automatically convert the array into appropriate **json** response.

Example

Observe the following example to understand more about JSON Response –

Step 1 – Add the following line in **app/Http/routes.php** file.

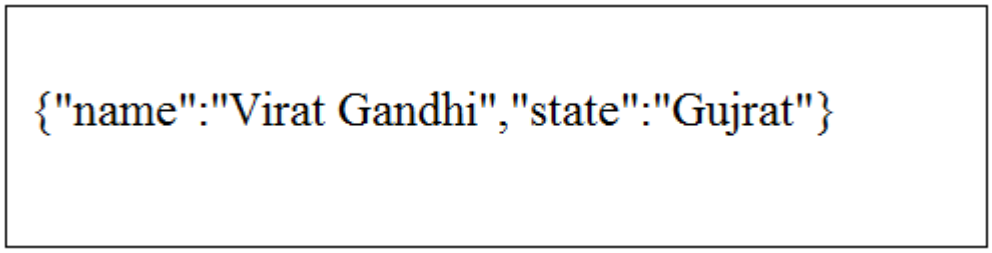
app/Http/routes.php

```
Route::get('json',function() {
   return response()->json(['name' => 'Virat Gandhi', 'state' => 'Guja
});
```

Step 2 – Visit the following URL to test the json response.

```
http://localhost:8000/json
```

Step 3 – The output will appear as shown in the following image.



Laravel - Views

In MVC framework, the letter **"V"** stands for **Views**. It separates the application logic and the presentation logic. Views are stored in **resources/views** directory. Generally, the view contains the HTML which will be served by the application.

Example

Observe the following example to understand more about Views –

Step 1 – Copy the following code and save it at **resources/views/test.php**

```
<html>
   <body>
      <h1>Hello, World</h1>
   </body>
</html>
```

Step 2 – Add the following line in **app/Http/routes.php** file to set the route for the above view.

app/Http/routes.php

```
Route::get('/test', function() {
  return view('test');
});
```

Step 3 – Visit the following URL to see the output of the view.

```
http://localhost:8000/test
```

Step 4 – The output will appear as shown in the following image.

Passing Data to Views

While building application it may be required to pass data to the views. Pass an array to view helper function. After passing an array, we can use the key to get the value of that key in the HTML file.

Example

Observe the following example to understand more about passing data to views –

Step 1 – Copy the following code and save it at **resources/views/test.php**

```
<html>
   <body>
      <h1><?php echo $name; ?></h1>
   </body>
</html>
```

Step 2 – Add the following line in **app/Http/routes.php** file to set the route for the above view.

app/Http/routes.php

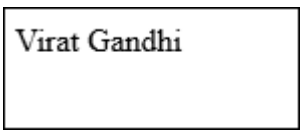
```
Route::get('/test', function() {
  return view('test',[‘name’=>’Virat Gandhi’]);
});
```

Step 3 – The value of the key name will be passed to test.php file and $name will be replaced by that value.

Step 4 – Visit the following URL to see the output of the view.

```
http://localhost:8000/test
```

Step 5 – The output will appear as shown in the following image.



Sharing Data with all Views

We have seen how we can pass data to views but at times, there is a need to pass data to all the views. Laravel makes this simpler. There is a method called **share()** which can be used for this purpose. The **share()** method will take two arguments, key and value. Typically **share()** method can be called from boot method of service provider. We can use any service provider, **AppServiceProvider** or our own service provider.

Example

Observe the following example to understand more about sharing data with all views –

Step 1 – Add the following line in **app/Http/routes.php** file.

app/Http/routes.php

```
Route::get('/test', function() {
  return view('test');
});

Route::get('/test2', function() {
  return view('test2');
});
```

Step 2 – Create two view files — **test.php** and **test2.php** with the same code. These are the two files which will share data. Copy the following code in both the files. **resources/views/test.php & resources/views/test2.php**

```
<html>
   <body>
      <h1><?php echo $name; ?></h1>
   </body>
</html>
```

Step 3 – Change the code of boot method in the file **app/Providers/AppServiceProvider.php** as shown below. (Here, we have used share method and the data that we have passed will be shared with all the views.)
app/Providers/AppServiceProvider.php

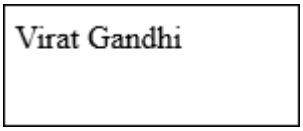
```
<?php

namespace App\Providers;
```

```
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider {

   /**
      * Bootstrap any application services.
      *
      * @return void
   */

   public function boot() {
      view()->share('name', 'Virat Gandhi');
   }

   /**
      * Register any application services.
      *
      * @return void
   */

   public function register() {
      //
   }
}
```

Step 4 – **Visit** the following URLs.

```
http://localhost:8000/test
http://localhost:8000/test2
```

Step 5 – The output will appear as shown in the following image.

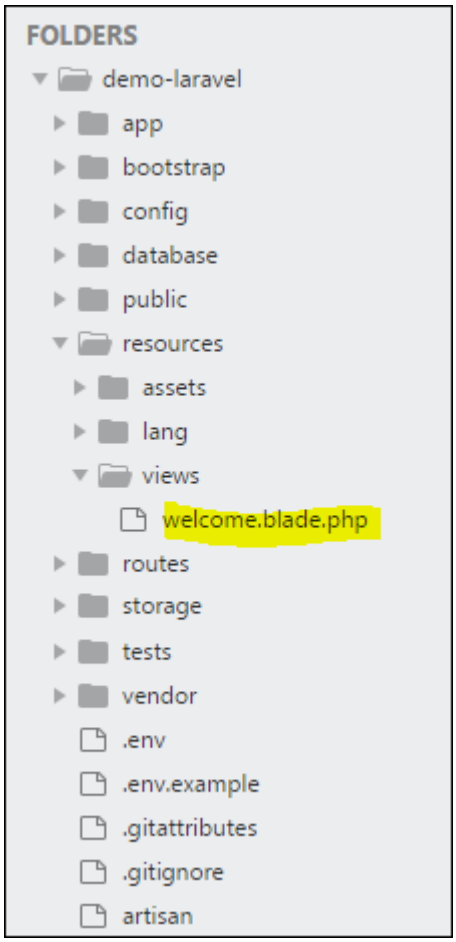


Laravel - Blade Templates

Laravel 5.1 introduces the concept of using **Blade**, a templating engine to design a unique layout. The layout thus designed can be used by other views, and includes a consistent design and structure.

When compared to other templating engines, Blade is unique in the following ways –

- It does not restrict the developer from using plain PHP code in views.
- The blade views thus designed, are compiled and cached until they are modified.



The complete directory structure of Laravel is shown in the screenshot given here.

You can observe that all views are stored in the **resources/views** directory and the default view for Laravel framework is **welcome.blade.php**.

Please note that other blade templates are also created similarly.

Steps for Creating a Blade Template Layout

You will have to use the following steps to create a blade template layout –

Step 1

- Create a layout folder inside the **resources/views** folder. We are going to use this folder to store all layouts together.
- Create a file name **master.blade.php** which will have the following code associated with it –

```
<html>
   <head>
      <title>DemoLaravel - @yield('title')</title>
   </head>
   <body>
      @yield('content')
   </body>
</html>
```

Step 2

In this step, you should extend the layout. Extending a layout involves defining the child elements. Laravel uses the **Blade @extends** directive for defining the child elements.

When you are extending a layout, please note the following points –

- Views defined in the Blade Layout injects the container in a unique way.
- Various sections of view are created as child elements.
- Child elements are stored in layouts folder as **child.blade.php**

An example that shows extending the layout created above is shown here –

```
@extends('layouts.app')
@section('title', 'Page Title')
@section('sidebar')
   @parent
<p>This refers to the master sidebar.</p>
@endsection
@section('content')
<p>This is my body content.</p>
@endsection
```

Step 3

To implement the child elements in views, you should define the layout in the way it is needed.

Observe the screenshot shown here. You can find that each of links mentioned in the landing page are hyperlinks. Please note that you can also create them as child elements with the help of blade templates by using the procedure given above.

Laravel - Redirections

Named route is used to give specific name to a route. The name can be assigned using the **"as"** array key.

```
Route::get('user/profile', ['as' => 'profile', function () {
  //
}]);
```

Note – Here, we have given the name **profile** to a route **user/profile**.

Redirecting to Named Routes

Example

Observe the following example to understand more about Redirecting to named routes –

Step 1 – Create a view called test.php and save it at

resources/views/test.php.

```
<html>
   <body>
      <h1>Example of Redirecting to Named Routes</h1>
   </body>
</html>
```

Step 2 – In **routes.php**, we have set up the route for **test.php** file. We have renamed it to **testing**. We have also set up another route **redirect** which will

redirect the request to the named route **testing**.

app/Http/routes.php

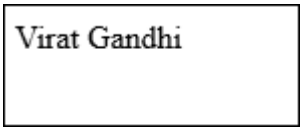
```
Route::get('/test', ['as'=>'testing',function() {
   return view('test2');
}]);

Route::get('redirect',function() {
   return redirect()->route('testing');
});
```

Step 3 – Visit the following URL to test the named route example.

```
http://localhost:8000/redirect
```

Step 4 – After execution of the above URL, you will be redirected to http://localhost:8000/test as we are redirecting to the named route **testing**.

Step 5 – After successful execution of the URL, you will receive the following output –



Redirecting to Controller Actions

Not only named route but we can also redirect to controller actions. We need to simply pass the controller and name of the **action** to the action method as shown in the following example. If you want to pass a parameter, you can pass it as the second argument of the action method.

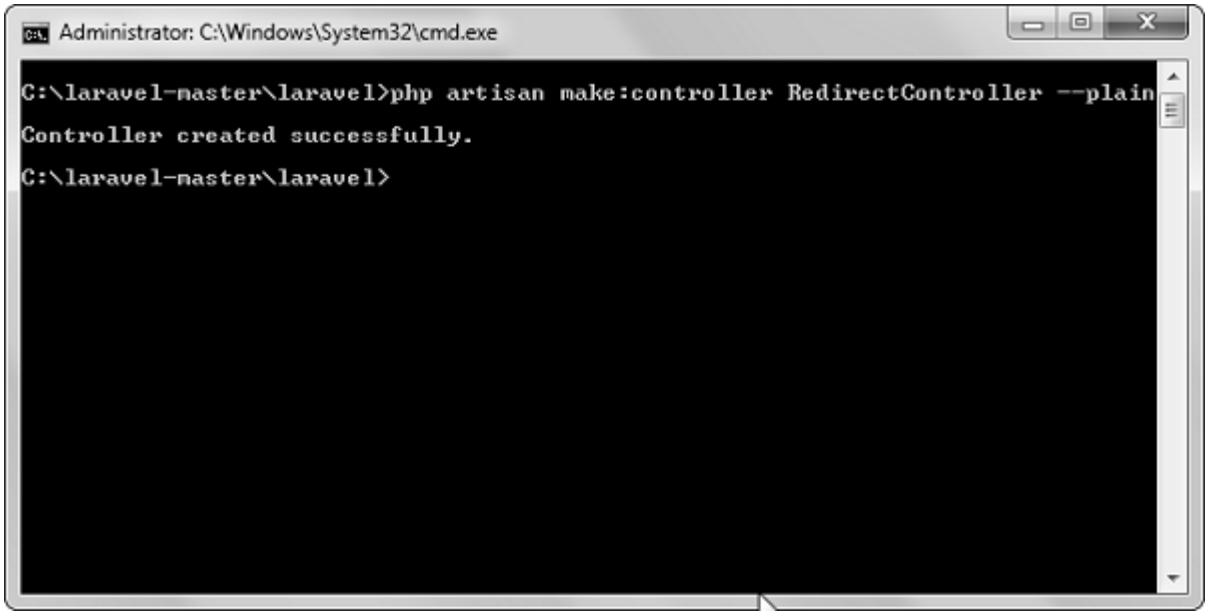
```
return redirect()->action('NameOfController@methodName',[parameters]);
```

Example

Step 1 – Execute the following command to create a controller called **RedirectController**.

```
php artisan make:controller RedirectController --plain
```

Step 2 – After successful execution, you will receive the following output –



Step 3 – Copy the following code to file

app/Http/Controllers/RedirectController.php.

app/Http/Controllers/RedirectController.php

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests;
use App\Http\Controllers\Controller;

class RedirectController extends Controller {
   public function index() {
      echo "Redirecting to controller's action.";
   }
}
```

Step 4 – Add the following lines in **app/Http/routes.php**.

app/Http/routes.php

```
Route::get('rr','RedirectController@index');
Route::get('/redirectcontroller',function() {
   return redirect()->action('RedirectController@index');
});
```

Step 5 – Visit the following URL to test the example.

```
http://localhost:8000/redirectcontroller
```

Step 6 – The output will appear as shown in the following image.

Laravel - Working With Database

Laravel has made processing with database very easy. Laravel currently supports following 4 databases –

- MySQL
- Postgres
- SQLite
- SQL Server

The query to the database can be fired using raw SQL, the fluent query builder, and the Eloquent ORM. To understand the all CRUD (Create, Read, Update, Delete) operations with Laravel, we will use simple student management system.

Connecting to Database

Configure the database in **config/database.php** file and create the college database with structure in MySQL as shown in the following table.

Database: College

Table: student

We will see how to add, delete, update and retrieve records from database using Laravel in student table.

Laravel - Errors and Logging

This chapter deals with errors and logging in Laravel projects and how to work on them.

Errors

A project while underway, is borne to have a few errors. Errors and exception handling is already configured for you when you start a new Laravel project. Normally, in a local environment we need to see errors for debugging purposes. We need to hide these errors from users in production environment. This can be achieved with the variable **APP_DEBUG** set in the environment file **.env** stored at the root of the application.

For local environment the value of **APP_DEBUG** should be **true** but for production it needs to be set to **false** to hide errors.

Note – After changing the **APP_DEBUG** variable, you should restart the Laravel server.

Logging

Logging is an important mechanism by which system can log errors that are generated. It is useful to improve the reliability of the system. Laravel supports different logging modes like single, daily, syslog, and errorlog modes. You can set these modes in **config/app.php** file.

```
'log' => 'daily'
```

You can see the generated log entries in **storage/logs/laravel.log** file.

Laravel - Forms

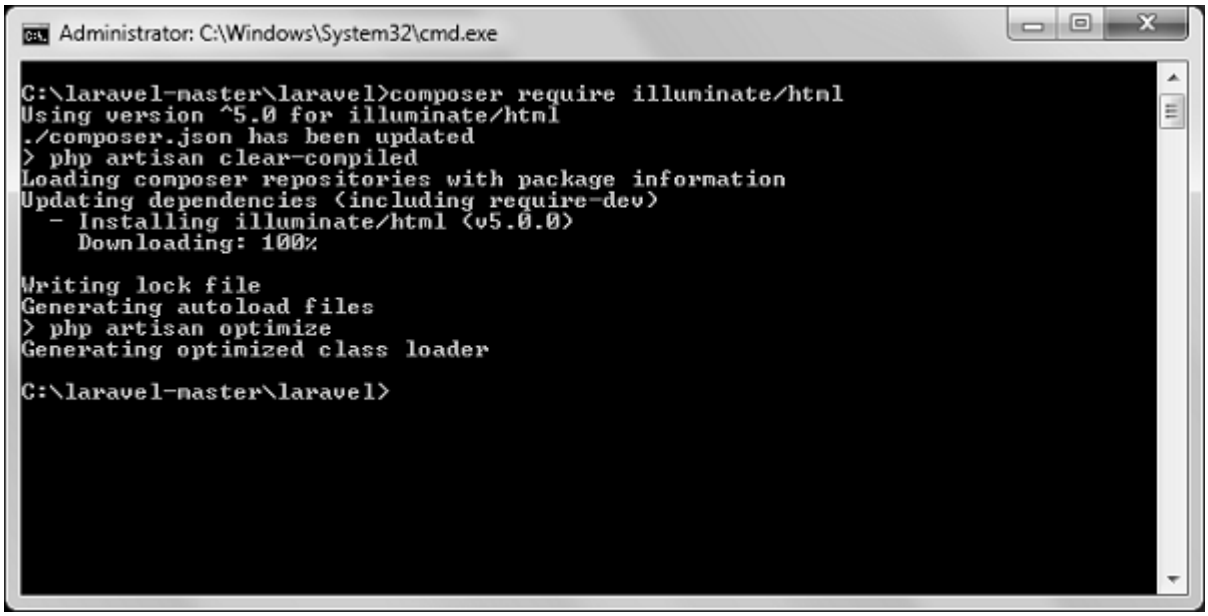
Laravel provides various in built tags to handle HTML forms easily and securely. All the major elements of HTML are generated using Laravel. To support this, we need to add HTML package to Laravel using composer.

Example 1

Step 1 – Execute the following command to proceed with the same.

```
composer require illuminate/html
```

Step 2 – This will add HTML package to Laravel as shown in the following image.



Step 3 – Now, we need to add the package shown above to Laravel configuration file which is stored at **config/app.php.** Open this file and you will see a list of Laravel service providers as shown in the following image. Add HTML service provider as indicated in the outlined box in the following image.

```
'providers' => [

    /*
     * Laravel Framework Service Providers...
     */
    Illuminate\Foundation\Providers\ArtisanServiceProvider::class,
    Illuminate\Auth\AuthServiceProvider::class,
    Illuminate\Broadcasting\BroadcastServiceProvider::class,
    Illuminate\Bus\BusServiceProvider::class,
    Illuminate\Cache\CacheServiceProvider::class,
    Illuminate\Foundation\Providers\ConsoleSupportServiceProvider::class,
    Illuminate\Routing\ControllerServiceProvider::class,
    Illuminate\Cookie\CookieServiceProvider::class,
    Illuminate\Database\DatabaseServiceProvider::class,
    Illuminate\Encryption\EncryptionServiceProvider::class,
    Illuminate\Filesystem\FilesystemServiceProvider::class,
    Illuminate\Foundation\Providers\FoundationServiceProvider::class,
    Illuminate\Hashing\HashServiceProvider::class,
    Illuminate\Mail\MailServiceProvider::class,
    Illuminate\Pagination\PaginationServiceProvider::class,
    Illuminate\Pipeline\PipelineServiceProvider::class,
    Illuminate\Queue\QueueServiceProvider::class,
    Illuminate\Redis\RedisServiceProvider::class,
    Illuminate\Auth\Passwords\PasswordResetServiceProvider::class,
    Illuminate\Session\SessionServiceProvider::class,
    Illuminate\Translation\TranslationServiceProvider::class,
    Illuminate\Validation\ValidationServiceProvider::class,
    Illuminate\View\ViewServiceProvider::class,
    Illuminate\Html\HtmlServiceProvider::class,
```

Step 4 – Add aliases in the same file for HTML and Form. Notice the two lines indicated in the outlined box in the following image and add those two lines.

```
'aliases' => [

    'App'       => Illuminate\Support\Facades\App::class,
    'Artisan'   => Illuminate\Support\Facades\Artisan::class,
    'Auth'      => Illuminate\Support\Facades\Auth::class,
    'Blade'     => Illuminate\Support\Facades\Blade::class,
    'Bus'       => Illuminate\Support\Facades\Bus::class,
    'Cache'     => Illuminate\Support\Facades\Cache::class,
    'Config'    => Illuminate\Support\Facades\Config::class,
    'Cookie'    => Illuminate\Support\Facades\Cookie::class,
    'Crypt'     => Illuminate\Support\Facades\Crypt::class,
    'DB'        => Illuminate\Support\Facades\DB::class,
    'Eloquent'  => Illuminate\Database\Eloquent\Model::class,
    'Event'     => Illuminate\Support\Facades\Event::class,
    'File'      => Illuminate\Support\Facades\File::class,
    'Gate'      => Illuminate\Support\Facades\Gate::class,
    'Hash'      => Illuminate\Support\Facades\Hash::class,
    'Input'     => Illuminate\Support\Facades\Input::class,
    'Inspiring' => Illuminate\Foundation\Inspiring::class,
    'Lang'      => Illuminate\Support\Facades\Lang::class,
    'Log'       => Illuminate\Support\Facades\Log::class,
    'Mail'      => Illuminate\Support\Facades\Mail::class,
    'Password'  => Illuminate\Support\Facades\Password::class,
    'Queue'     => Illuminate\Support\Facades\Queue::class,
    'Redirect'  => Illuminate\Support\Facades\Redirect::class,
    'Redis'     => Illuminate\Support\Facades\Redis::class,
    'Request'   => Illuminate\Support\Facades\Request::class,
    'Response'  => Illuminate\Support\Facades\Response::class,
    'Route'     => Illuminate\Support\Facades\Route::class,
    'Schema'    => Illuminate\Support\Facades\Schema::class,
    'Session'   => Illuminate\Support\Facades\Session::class,
    'Storage'   => Illuminate\Support\Facades\Storage::class,
    'URL'       => Illuminate\Support\Facades\URL::class,
    'Validator' => Illuminate\Support\Facades\Validator::class,
    'View'      => Illuminate\Support\Facades\View::class,
    'Form'      => Illuminate\Html\FormFacade::class,
    'Html'      => Illuminate\Html\HtmlFacade::class,
```

Step 5 – Now everything is setup. Let's see how we can use various HTML elements using Laravel tags.

Opening a Form

```
{{ Form::open(array('url' => 'foo/bar')) }}
  //
{{ Form::close() }}
```

Generating a Label Element

```
echo Form::label('email', 'E-Mail Address');
```

Generating a Text Input

```
echo Form::text('username');
```

Specifying a Default Value

```
echo Form::text('email', 'example@gmail.com');
```

Generating a Password Input

```
echo Form::password('password');
```

Generating a File Input

```
echo Form::file('image');
```

Generating a Checkbox Or Radio Input

```
echo Form::checkbox('name', 'value');
echo Form::radio('name', 'value');
```

Generating a Checkbox Or Radio Input That Is Checked

```
echo Form::checkbox('name', 'value', true);
echo Form::radio('name', 'value', true);
```

Generating a Drop-Down List

```
echo Form::select('size', array('L' => 'Large', 'S' => 'Small'));
```

Generating A Submit Button

```
echo Form::submit('Click Me!');
```

Example 2

Step 1 – Copy the following code to create a view called

resources/views/form.php.

resources/views/form.php

```
<html>
   <body>

      <?php
         echo Form::open(array('url' => 'foo/bar'));
            echo Form::text('username','Username');
            echo '<br/>';

            echo Form::text('email', 'example@gmail.com');
            echo '<br/>';

            echo Form::password('password');
            echo '<br/>';

            echo Form::checkbox('name', 'value');
            echo '<br/>';

            echo Form::radio('name', 'value');
            echo '<br/>';

            echo Form::file('image');
            echo '<br/>';

            echo Form::select('size', array('L' => 'Large', 'S' => 'Sm
            echo '<br/>';

            echo Form::submit('Click Me!');
         echo Form::close();
      ?>
```

```
    </body>
</html>
```

Step 2 – Add the following line in **app/Http/routes.php** to add a route for view form.php

app/Http/routes.php

```
Route::get('/form',function() {
   return view('form');
});
```

Step 3 – Visit the following URL to see the form.

```
http://localhost:8000/form
```

Step 4 – The output will appear as shown in the following image.

Laravel - Localization

Localization feature of Laravel supports different language to be used in application. You need to store all the strings of different language in a file and these files are stored at **resources/views** directory. You should create a separate directory for each supported language. All the language files should return an array of keyed strings as shown below.

```
<?php
return [
   'welcome' => 'Welcome to the application'
];
```

Example

Step 1 – Create 3 files for languages – **English, French**, and **German**. Save English file at **resources/lang/en/lang.php**

```
<?php
   return [
      'msg' => 'Laravel Internationalization example.'
   ];
?>
```

Step 2 – Save French file at **resources/lang/fr/lang.php**.

```
<?php
   return [
      'msg' => 'Exemple Laravel internationalisation.'
   ];
?>
```

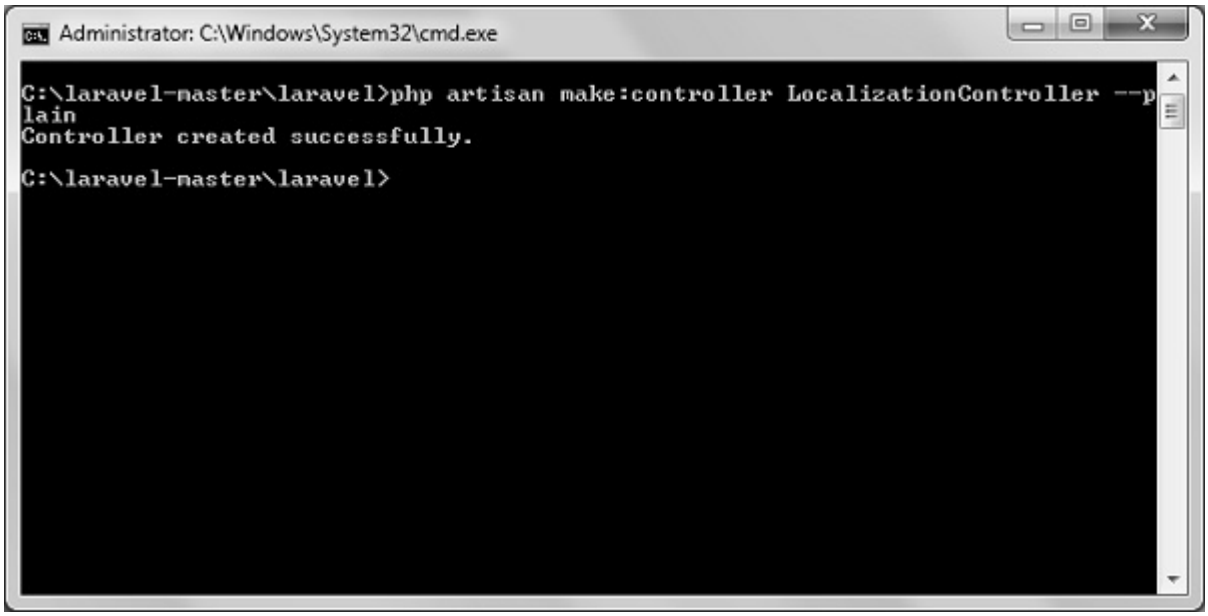
Step 3 – Save German file at **resources/lang/de/lang.php**.

```
<?php
   return [
      'msg' => 'Laravel Internationalisierung Beispiel.'
   ];
?>
```

Step 4 – Create a controller called **LocalizationController** by executing the following command.

```
php artisan make:controller LocalizationController --plain
```

Step 5 – After successful execution, you will receive the following output –



Step 6 – Copy the following code to file

app/Http/Controllers/LocalizationController.php

app/Http/Controllers/LocalizationController.php

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests;
use App\Http\Controllers\Controller;

class LocalizationController extends Controller {
   public function index(Request $request,$locale) {
      //set's application's locale
      app()->setLocale($locale);

      //Gets the translated message and displays it
      echo trans('lang.msg');
   }
}
```

Step 7 – Add a route for LocalizationController in **app/Http/routes.php** file. Notice that we are passing {locale} argument after localization/ which we will use to see output in different language.

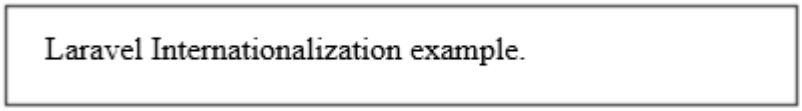
app/Http/routes.php

```
Route::get('localization/{locale}','LocalizationController@index');
```

Step 8 – Now, let us visit the different URLs to see all different languages. Execute the below URL to see output in English language.

```
http://localhost:8000/localization/en
```

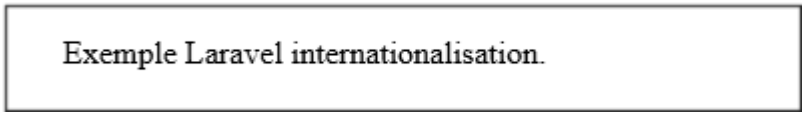
Step 9 – The output will appear as shown in the following image.



Step 10 – Execute the below URL to see output in French language.

```
http://localhost:8000/localization/fr
```

Step 11 – The output will appear as shown in the following image.



Step 12 – Execute the below URL to see output in German language

```
http://localhost:8000/localization/de
```

Step 13 – The output will appear as shown in the following image.

Laravel - Session

Sessions are used to store information about the user across the requests. Laravel provides various drivers like **file, cookie, apc, array, Memcached, Redis,** and **database** to handle session data. By default, file driver is used because it is lightweight. Session can be configured in the file stored at **config/session.php**.

Accessing Session Data

To access the session data, we need an instance of session which can be accessed via HTTP request. After getting the instance, we can use the **get()** method, which will take one argument, **"key"**, to get the session data.

```
$value = $request->session()->get('key');
```

You can use **all()** method to get all session data instead of **get()** method.

Storing Session Data

Data can be stored in session using the **put()** method. The **put()** method will take two arguments, the **"key"** and the **"value"**.

```
$request->session()->put('key', 'value');
```

Deleting Session Data

The **forget()** method is used to delete an item from the session. This method will take **"key"** as the argument.

```
$request->session()->forget('key');
```

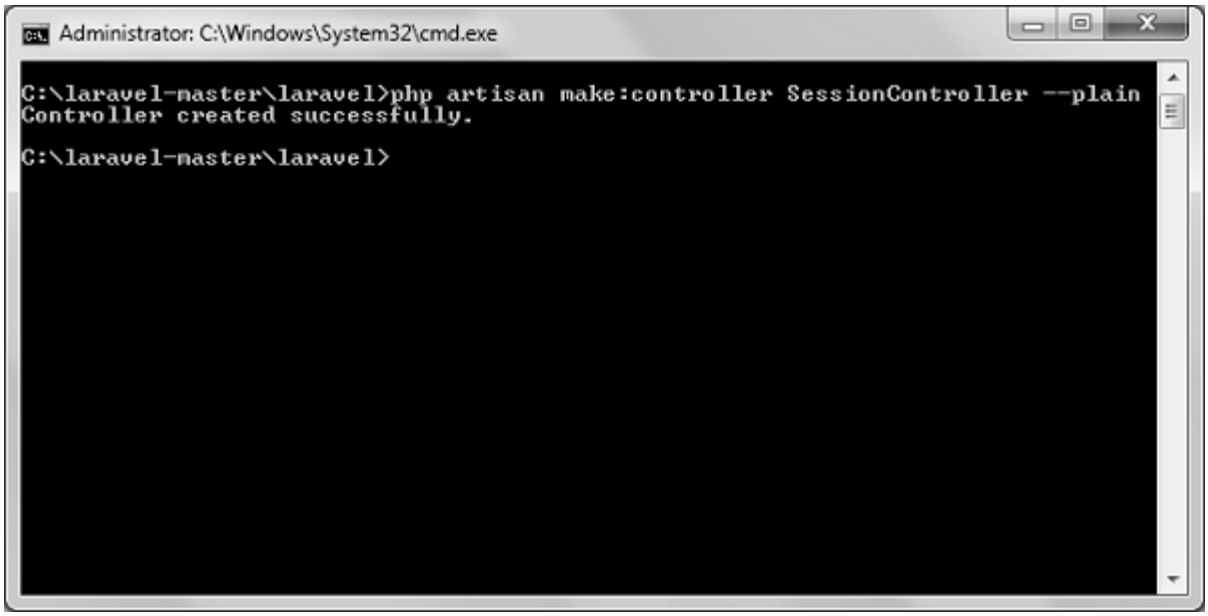
Use **flush()** method instead of **forget()** method to delete all session data. Use the **pull()** method to retrieve data from session and delete it afterwards. The pull() method will also take **key** as the argument. The difference between the **forget()** and the **pull()** method is that **forget()** method will not return the value of the session and **pull()** method will return it and delete that value from session.

Example

Step 1 – Create a controller called **SessionController** by executing the following command.

```
php artisan make:controller SessionController --plain
```

Step 2 – After successful execution, you will receive the following output –



Step 3 – Copy the following code in a file at

app/Http/Controllers/SessionController.php.

app/Http/Controllers/SessionController.php

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests;
use App\Http\Controllers\Controller;

class SessionController extends Controller {
   public function accessSessionData(Request $request) {
      if($request->session()->has('my_name'))
         echo $request->session()->get('my_name');
      else
         echo 'No data in the session';
   }
   public function storeSessionData(Request $request) {
      $request->session()->put('my_name','Virat Gandhi');
      echo "Data has been added to session";
   }
   public function deleteSessionData(Request $request) {
      $request->session()->forget('my_name');
      echo "Data has been removed from session.";
   }
}
```

Step 4 – Add the following lines at **app/Http/routes.php** file.

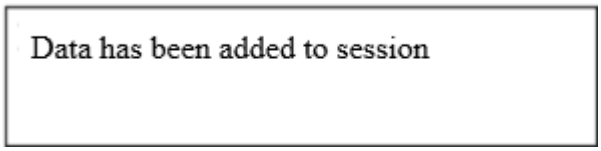
app/Http/routes.php

```
Route::get('session/get','SessionController@accessSessionData');
Route::get('session/set','SessionController@storeSessionData');
Route::get('session/remove','SessionController@deleteSessionData');
```

Step 5 – Visit the following URL to **set data in session**.

```
http://localhost:8000/session/set
```

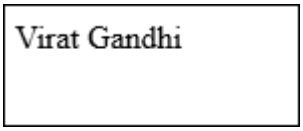
Step 6 – The output will appear as shown in the following image.



Step 7 – Visit the following URL to **get data from session**.

```
http://localhost:8000/session/get
```

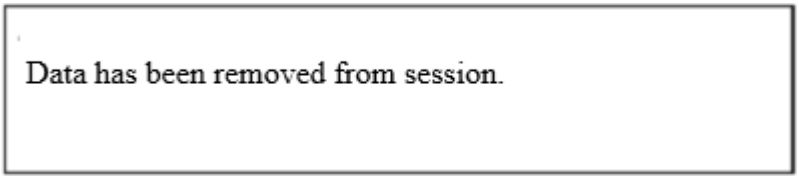
Step 8 – The output will appear as shown in the following image.



Step 9 – Visit the following URL to **remove session data**.

```
http://localhost:8000/session/remove
```

Step 10 – You will see a message as shown in the following image.



Laravel - Validation

Validation is the most important aspect while designing an application. It validates the incoming data. By default, base controller class uses a **ValidatesRequests** trait

which provides a convenient method to validate incoming HTTP requests with a variety of powerful validation rules.

Available Validation Rules in Laravel

Laravel will always check for errors in the session data, and automatically bind them to the view if they are available. So, it is important to note that a **$errors** variable will always be available in all of your views on every request, allowing you to conveniently assume the **$errors** variable is always defined and can be safely used. The following table shows all available validation rules in Laravel.

The **$errors** variable will be an instance of **Illuminate\Support\MessageBag**. Error message can be displayed in view file by adding the code as shown below.

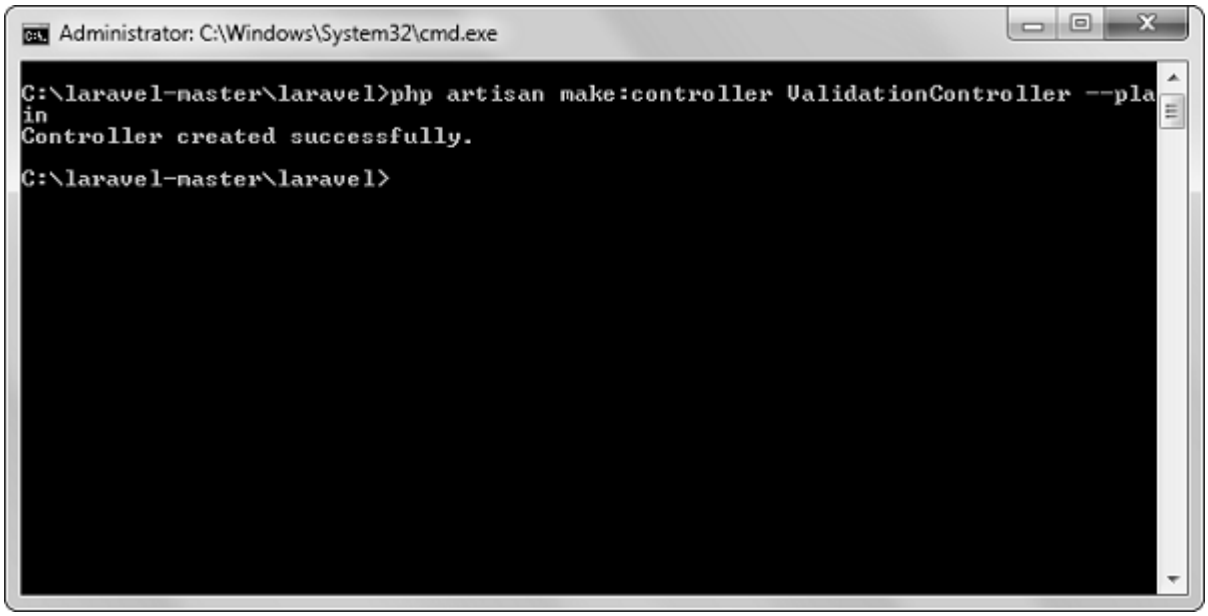
```
@if (count($errors) > 0)
   <div class = "alert alert-danger">
      <ul>
         @foreach ($errors->all() as $error)
            <li>{{ $error }}</li>
         @endforeach
      </ul>
   </div>
@endif
```

Example

Step 1 – Create a controller called **ValidationController** by executing the following command.

```
php artisan make:controller ValidationController --plain
```

Step 2 – After successful execution, you will receive the following output –



Step 3 – Copy the following code in

app/Http/Controllers/ValidationController.php file.

app/Http/Controllers/ValidationController.php

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests;
use App\Http\Controllers\Controller;

class ValidationController extends Controller {
   public function showform() {
      return view('login');
   }
   public function validateform(Request $request) {
      print_r($request->all());
      $this->validate($request,[
         'username'=>'required|max:8',
         'password'=>'required'
      ]);
   }
}
```

Step 4 – Create a view file called **resources/views/login.blade.php** and copy the following code in that file.

resources/views/login.blade.php

```
<html>

   <head>
      <title>Login Form</title>
   </head>

   <body>

      @if (count($errors) > 0)
         <div class = "alert alert-danger">
            <ul>
               @foreach ($errors->all() as $error)
                  <li>{{ $error }}</li>
               @endforeach
            </ul>
         </div>
      @endif

      <?php
         echo Form::open(array('url'=>'/validation'));
      ?>

      <table border = '1'>
         <tr>
            <td align = 'center' colspan = '2'>Login</td>
         </tr>
         <tr>
            <td>Username</td>
            <td><?php echo Form::text('username'); ?></td>
         </tr>
         <tr>
            <td>Password</td>
            <td><?php echo Form::password('password'); ?></td>
         </tr>
         <tr>
            <td align = 'center' colspan = '2'
               ><?php echo Form::submit('Login'); ?  ></td>
         </tr>
      </table>

      <?php
```

```
         echo Form::close();
      ?>

   </body>
</html>
```

Step 5 – Add the following lines in **app/Http/routes.php**.

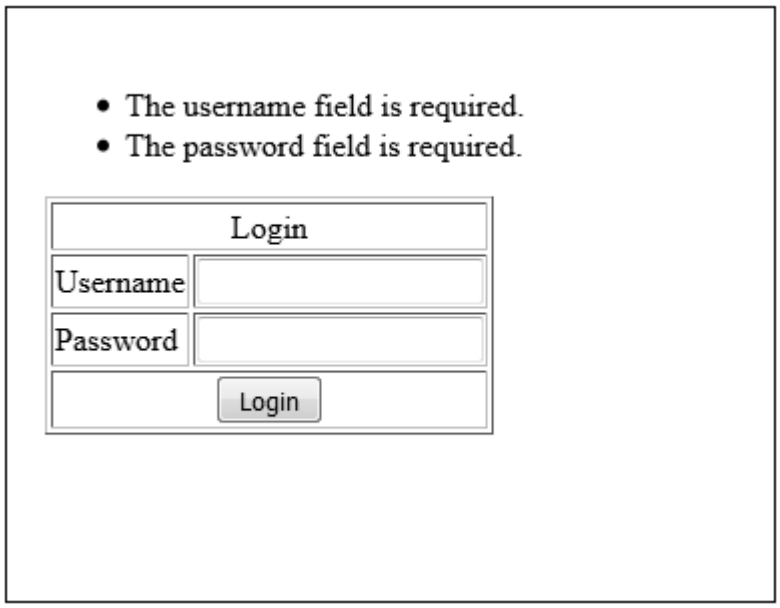
app/Http/routes.php

```
Route::get('/validation','ValidationController@showform');
Route::post('/validation','ValidationController@validateform');
```

Step 6 – Visit the following URL to test the validation.

```
http://localhost:8000/validation
```

Step 7 – Click the **"Login"** button without entering anything in the text field. The output will be as shown in the following image.



Laravel - File Uploading

Uploading Files in Laravel is very easy. All we need to do is to create a view file where a user can select a file to be uploaded and a controller where uploaded files will be processed.

In a view file, we need to generate a file input by adding the following line of code.

```
Form::file('file_name');
```

In Form::open(), we need to add **'files'=>'true'** as shown below. This facilitates the form to be uploaded in multiple parts.

```
Form::open(array('url' => '/uploadfile','files'=>'true'));
```

Example

Step 1 – Create a view file called **resources/views/uploadfile.php** and copy the following code in that file.

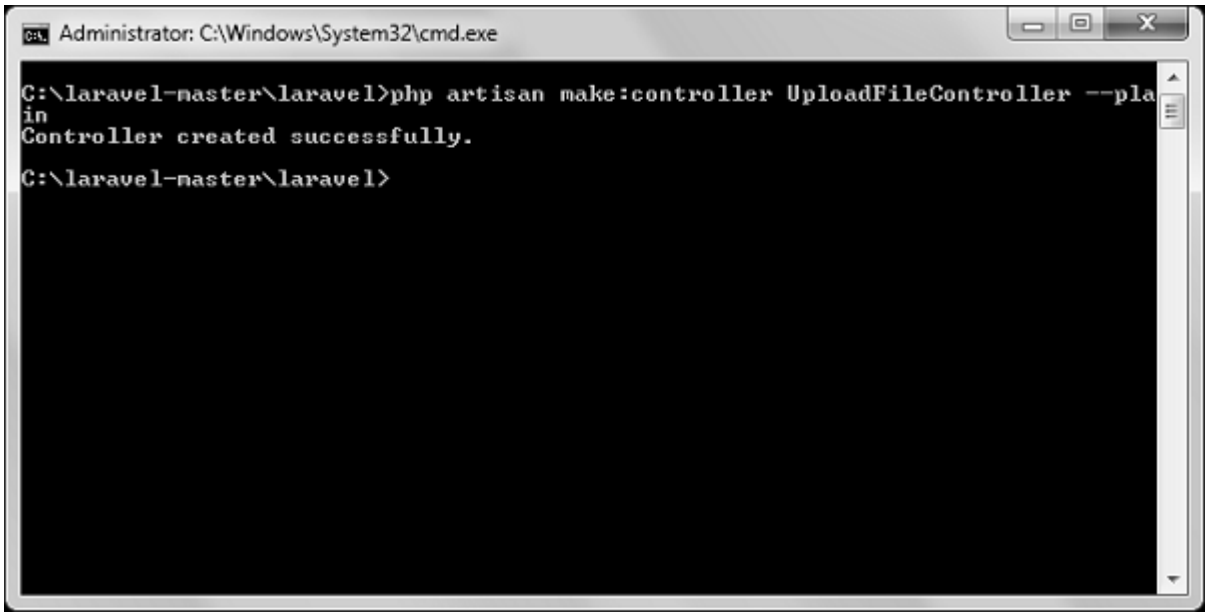
resources/views/uploadfile.php

```
<html>
   <body>
      <?php
         echo Form::open(array('url' => '/uploadfile','files'=>'true')
         echo 'Select the file to upload.';
         echo Form::file('image');
         echo Form::submit('Upload File');
         echo Form::close();
      ?>
   </body>
</html>
```

Step 2 – Create a controller called **UploadFileController** by executing the following command.

```
php artisan make:controller UploadFileController --plain
```

Step 3 – After successful execution, you will receive the following output –



Step 4 – Copy the following code in

app/Http/Controllers/UploadFileController.php file.

app/Http/Controllers/UploadFileController.php

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests;
use App\Http\Controllers\Controller;

class UploadFileController extends Controller {
   public function index() {
      return view('uploadfile');
   }
   public function showUploadFile(Request $request) {
      $file = $request->file('image');

      //Display File Name
      echo 'File Name: '.$file->getClientOriginalName();
      echo '<br>';

      //Display File Extension
      echo 'File Extension: '.$file->getClientOriginalExtension();
      echo '<br>';

      //Display File Real Path
```

```
        echo 'File Real Path: '.$file->getRealPath();
        echo '<br>';

        //Display File Size
        echo 'File Size: '.$file->getSize();
        echo '<br>';

        //Display File Mime Type
        echo 'File Mime Type: '.$file->getMimeType();

        //Move Uploaded File
        $destinationPath = 'uploads';
        $file->move($destinationPath,$file->getClientOriginalName());
    }
}
```

Step 5 – Add the following lines in **app/Http/routes.php**.

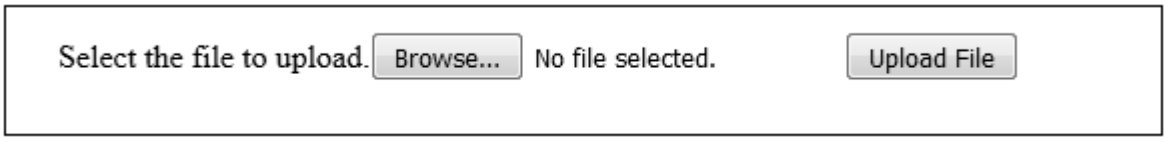
app/Http/routes.php

```
Route::get('/uploadfile','UploadFileController@index');
Route::post('/uploadfile','UploadFileController@showUploadFile');
```

Step 6 – Visit the following URL to test the upload file functionality.

```
http://localhost:8000/uploadfile
```

Step 7 – You will receive a prompt as shown in the following image.



Laravel - Sending Email

Laravel uses free feature-rich library **SwiftMailer** to send emails. Using the library function, we can easily send emails without too many hassles. The e-mail templates are loaded in the same way as views, which means you can use the Blade syntax and inject data into your templates.

The following table shows the syntax and attributes of **send** function –

In the third argument, the $callback closure received message instance and with that instance we can also call the following functions and alter the message as shown below.

- $message → subject('Welcome to the Tutorials Point');
- $message → from('email@example.com', 'Mr. Example');
- $message → to('email@example.com', 'Mr. Example');

Some of the less common methods include –

- $message → sender('email@example.com', 'Mr. Example');
- $message → returnPath('email@example.com');
- $message → cc('email@example.com', 'Mr. Example');
- $message → bcc('email@example.com', 'Mr. Example');
- $message → replyTo('email@example.com', 'Mr. Example');
- $message → priority(2);

To attach or embed files, you can use the following methods –

- $message → attach('path/to/attachment.txt');
- $message → embed('path/to/attachment.jpg');

Mail can be sent as HTML or text. You can indicate the type of mail that you want to send in the first argument by passing an array as shown below. The default type is HTML. If you want to send plain text mail then use the following syntax.

Syntax

```
Mail::send([‘text’=>’text.view’], $data, $callback);
```

In this syntax, the first argument takes an array. Use **text** as the key name of the view as value of the key.

Example

Step 1 – We will now send an email from Gmail account and for that you need to configure your Gmail account in Laravel environment file **- .env** file. Enable 2-step

verification in your Gmail account and create an application specific password followed by changing the .env parameters as shown below.

.env

```
MAIL_DRIVER = smtp
MAIL_HOST = smtp.gmail.com
MAIL_PORT = 587
MAIL_USERNAME = your-gmail-username
MAIL_PASSWORD = your-application-specific-password
MAIL_ENCRYPTION = tls
```

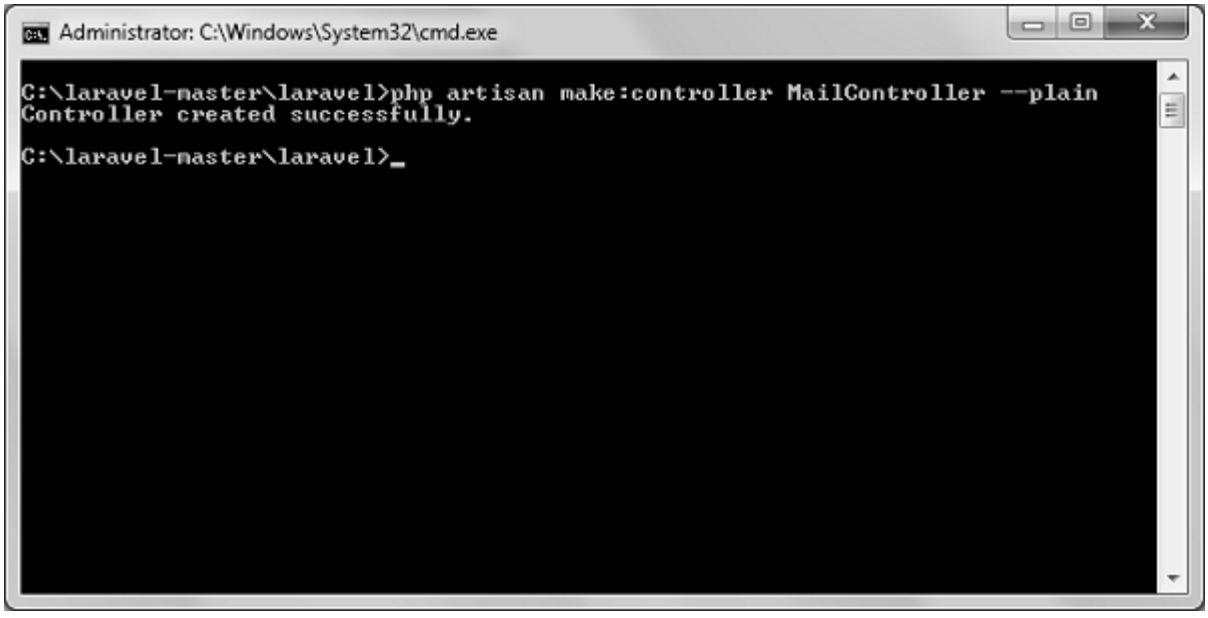
Step 2 – After changing the **.env** file execute the below two commands to clear the cache and restart the Laravel server.

```
php artisan config:cache
```

Step 3 – Create a controller called **MailController** by executing the following command.

```
php artisan make:controller MailController --plain
```

Step 4 – After successful execution, you will receive the following output –



Step 5 – Copy the following code in

app/Http/Controllers/MailController.php file.

app/Http/Controllers/MailController.php

```
<?php

namespace App\Http\Controllers;
use Illuminate\Http\Request;
use Mail;

use App\Http\Requests;
use App\Http\Controllers\Controller;

class MailController extends Controller {
   public function basic_email() {
      $data = array('name'=>"Virat Gandhi");

      Mail::send(['text'=>'mail'], $data, function($message) {
         $message->to('abc@gmail.com', 'Tutorials Point')->subject
            ('Laravel Basic Testing Mail');
         $message->from('xyz@gmail.com','Virat Gandhi');
      });
      echo "Basic Email Sent. Check your inbox.";
   }
   public function html_email() {
      $data = array('name'=>"Virat Gandhi");
      Mail::send('mail', $data, function($message) {
         $message->to('abc@gmail.com', 'Tutorials Point')->subject
            ('Laravel HTML Testing Mail');
         $message->from('xyz@gmail.com','Virat Gandhi');
      });
      echo "HTML Email Sent. Check your inbox.";
   }
   public function attachment_email() {
      $data = array('name'=>"Virat Gandhi");
      Mail::send('mail', $data, function($message) {
         $message->to('abc@gmail.com', 'Tutorials Point')->subject
            ('Laravel Testing Mail with Attachment');
         $message->attach('C:\laravel-master\laravel\public\uploads\im
         $message->attach('C:\laravel-master\laravel\public\uploads\te
         $message->from('xyz@gmail.com','Virat Gandhi');
      });
      echo "Email Sent with attachment. Check your inbox.";
   }
}
```

Step 6 – Copy the following code in **resources/views/mail.blade.php** file.

resources/views/mail.blade.php

```
<h1>Hi, {{ $name }}</h1>
l<p>Sending Mail from Laravel.</p>
```

Step 7 – Add the following lines in **app/Http/routes.php.**

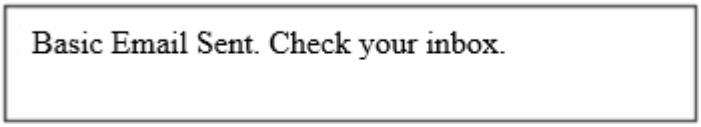
app/Http/routes.php

```
Route::get('sendbasicemail','MailController@basic_email');
Route::get('sendhtmlemail','MailController@html_email');
Route::get('sendattachmentemail','MailController@attachment_email');
```

Step 8 – Visit the following URL to test basic email.

```
http://localhost:8000/sendbasicemail
```

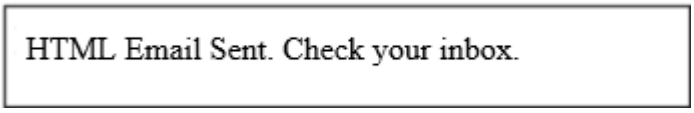
Step 9 – The output screen will look something like this. Check your inbox to see the basic email output.



Step 10 – Visit the following URL to test the HTML email.

```
http://localhost:8000/sendhtmlemail
```

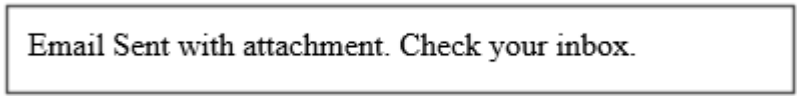
Step 11 – The output screen will look something like this. Check your inbox to see the html email output.



Step 12 – Visit the following URL to test the HTML email with attachment.

```
http://localhost:8000/sendattachmentemail
```

Step 13 – You can see the following output



Note – In the **MailController.php** file the email address in the from method should be the email address from which you can send email address. Generally, it should be the email address configured on your server.

Laravel - Ajax

Ajax (Asynchronous JavaScript and XML) is a set of web development techniques utilizing many web technologies used on the client-side to create asynchronous Web applications. Import jquery library in your view file to use ajax functions of jquery which will be used to send and receive data using ajax from the server. On the server side you can use the response() function to send response to client and to send response in JSON format you can chain the response function with json() function.

json() function syntax

```
json(string|array $data = array(), int $status = 200, array $headers = array(), int $optic
```

Example

Step 1 – Create a view file called **resources/views/message.php** and copy the following code in that file.

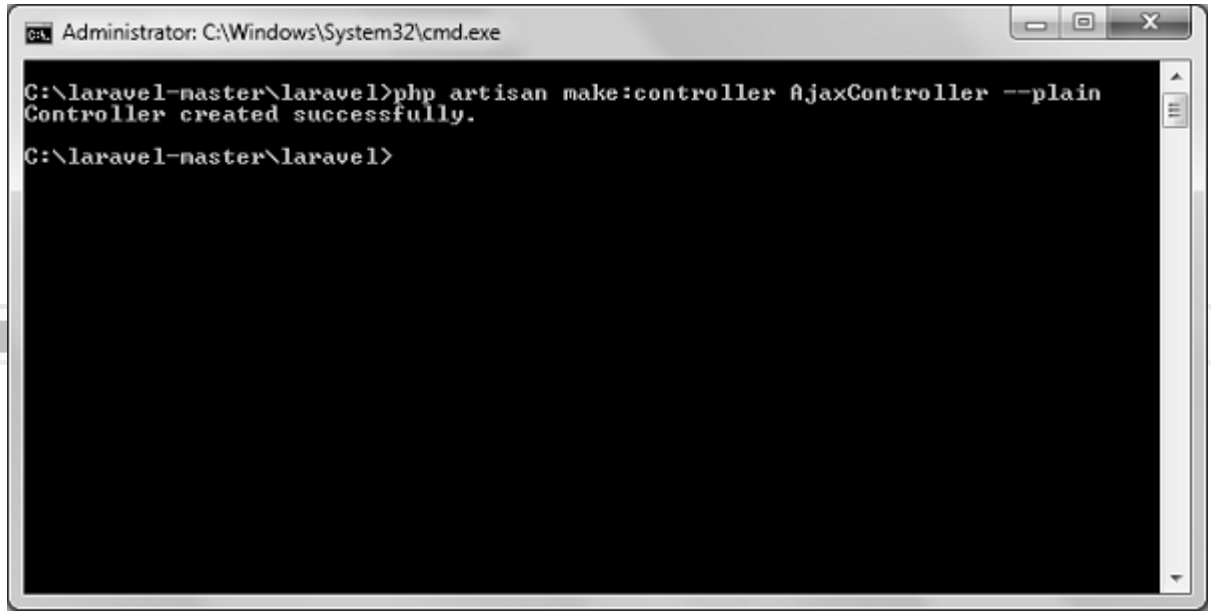
```
<html>
   <head>
      <title>Ajax Example</title>

      <script src = "https://ajax.googleapis.com/ajax/libs/jquery/2.1.
      </script>

      <script>
         function getMessage() {
            $.ajax({
               type:'POST',
               url:'/getmsg',
               data:'_token = <?php echo csrf_token() ?>',
               success:function(data) {
```

```
                $("#msg").html(data.msg);
            }
         });
      }
   </script>
</head>

<body>
   <div id = 'msg'>This message will be replaced using Ajax.
      Click the button to replace the message.</div>
   <?php
      echo Form::button('Replace Message',['onClick'=>'getMessage()
   ?>
</body>

</html>
```

Step 2 – Create a controller called **AjaxController** by executing the following command.

```
php artisan make:controller AjaxController --plain
```

Step 3 – After successful execution, you will receive the following output –



Step 4 – Copy the following code in

app/Http/Controllers/AjaxController.php file.

app/Http/Controllers/AjaxController.php

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests;
use App\Http\Controllers\Controller;

class AjaxController extends Controller {
   public function index() {
      $msg = "This is a simple message.";
      return response()->json(array('msg'=> $msg), 200);
   }
}
```

Step 5 – Add the following lines in **app/Http/routes.php**.

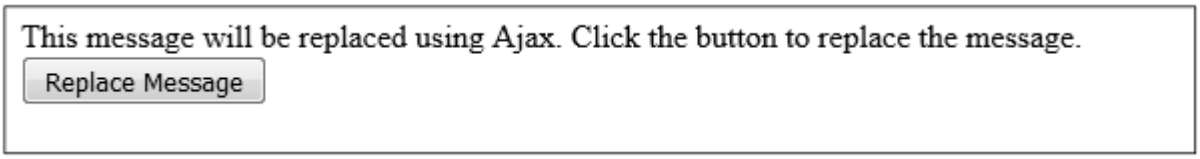
app/Http/routes.php

```
Route::get('ajax',function() {
  return view('message');
});
Route::post('/getmsg','AjaxController@index');
```

Step 6 – Visit the following URL to test the Ajax functionality.

```
http://localhost:8000/ajax
```

Step 7 – You will be redirected to a page where you will see a message as shown in the following image.



Step 8 – The output will appear as shown in the following image after clicking the button.

Laravel - Error Handling

Most web applications have specific mechanisms for error handling. Using these, they track errors and exceptions, and log them to analyze the performance. In this chapter, you will read about error handling in Laravel applications.

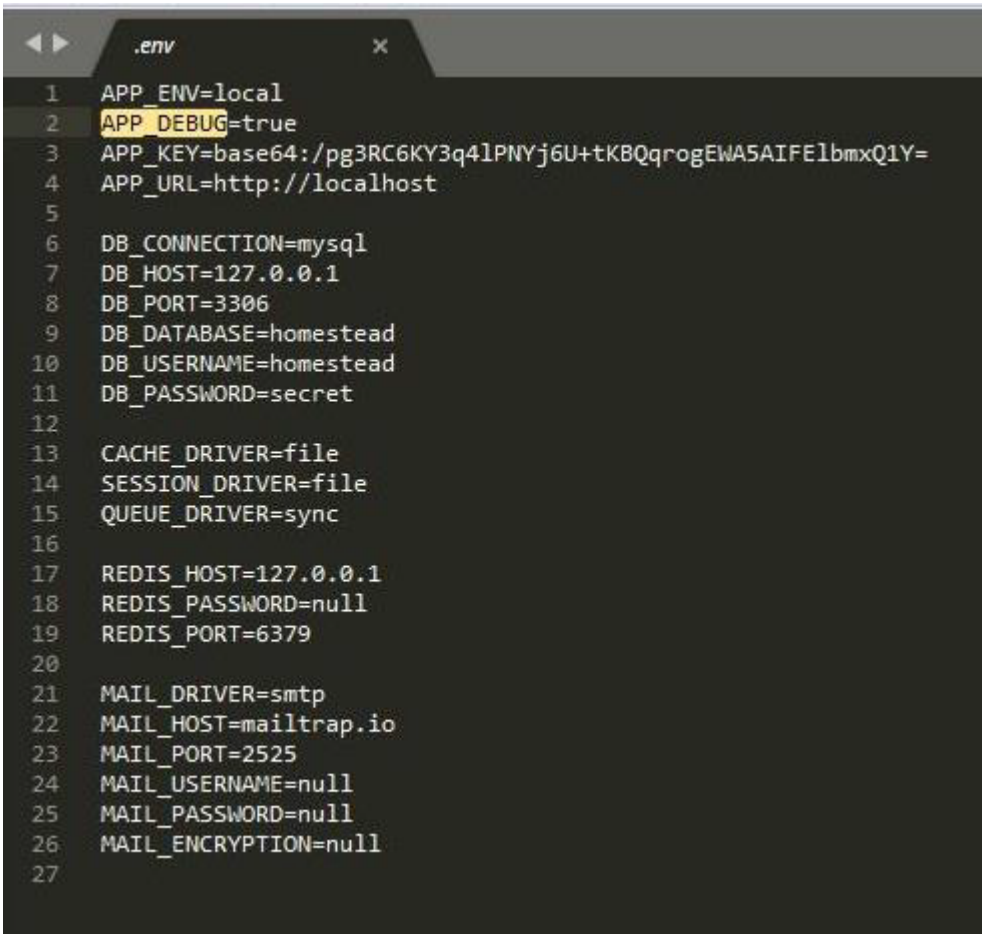
Important Points

Before proceeding further to learn in detail about error handling in Laravel, please note the following important points –

- For any new project, Laravel logs errors and exceptions in the **App\Exceptions\Handler** class, by default. They are then submitted back to the user for analysis.
- When your Laravel application is set in debug mode, detailed error messages with stack traces will be shown on every error that occurs within your web application.

```
/*
|--------------------------------------------------------------------------
| Application Debug Mode
|--------------------------------------------------------------------------
|
| When your application is in debug mode, detailed error messages with
| stack traces will be shown on every error that occurs within your
| application. If disabled, a simple generic error page is shown.
|
*/

'debug' => env('APP_DEBUG', false),
```

- By default, debug mode is set to **false** and you can change it to **true**. This enables the user to track all errors with stack traces.



- The configuration of Laravel project includes the **debug** option which determines how much information about an error is to be displayed to the user. By default in a web application, the option is set to the value defined in the environment variables of the **.env** file.

    - The value is set to **true** in a local development environment and is set to **false** in a production environment.
    - If the value is set to **true** in a production environment, the risk of sharing sensitive information with the end users is higher.

Error Log

Logging the errors in a web application helps to track them and in planning a strategy for removing them. The log information can be configured in the web application in **config/app.php** file. Please note the following points while dealing with Error Log in Laravel –

- Laravel uses monolog PHP logging library.
- The logging parameters used for error tracking are **single, daily, syslog** and **errorlog**.

- For example, if you wish to log the error messages in log files, you should set the log value in your app configuration to **daily** as shown in the command below −

```
'log' => env('APP_LOG',’daily’),
```

- If the **daily** log mode is taken as the parameter, Laravel takes error log for a period of **5 days**, by default. If you wish to change the maximum number of log files, you have to set the parameter of **log_max_files** in the configuration file to a desired value.

```
‘log_max_files’ => 25;
```

Severity Levels

As Laravel uses monolog PHP logging library, there are various parameters used for analyzing severity levels. Various severity levels that are available are **error, critical, alert** and **emergency messages**. You can set the severity level as shown in the command below −

```
'log_level' => env('APP_LOG_LEVEL', 'error')
```

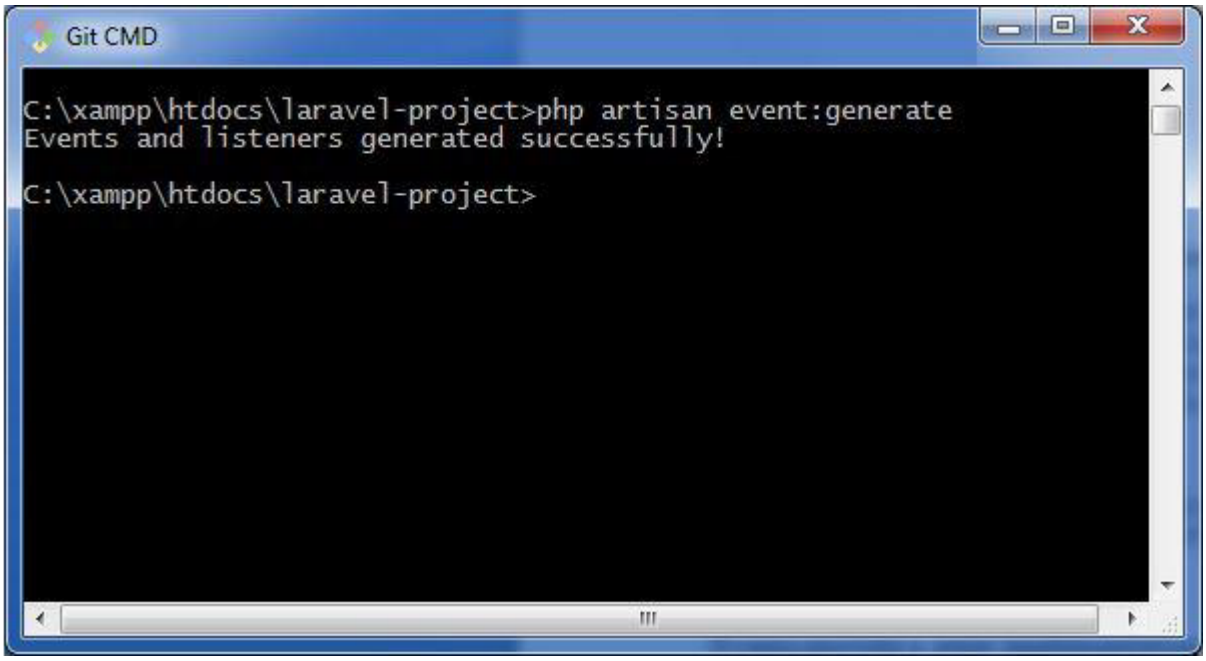
Laravel - Event Handling

Events provide a simple observer implementation which allows a user to subscribe and listen to various events triggered in the web application. All the event classes in Laravel are stored in the **app/Events** folder and the listeners are stored in the **app/Listeners** folder.

The artisan command for generating events and listeners in your web application is shown below −

```
php artisan event:generate
```

This command generates the events and listeners to the respective folders as discussed above.



Events and Listeners serve a great way to decouple a web application, since one event can have multiple listeners which are independent of each other. The events folder created by the artisan command includes the following two files: event.php and SomeEvent.php. They are shown here –

Event.php

```
<?php
namespace App\Events;
abstract class Event{
   //
}
```

As mentioned above, **event.php** includes the basic definition of class **Event** and calls for namespace **App\Events**. Please note that the user defined or custom events are created in this file.

SomeEvent.php

```
<?php

namespace App\Events;

use App\Events\Event;
use Illuminate\Queue\SerializesModels;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
```

```
class SomeEvent extends Event{
   use SerializesModels;
   /**
      * Create a new event instance.
      *
      * @return void
   */

   public function __construct() {
      //
   }

   /**
      * Get the channels the event should be broadcast on.
      *
      * @return array
   */

   public function broadcastOn() {
      return [];
   }
}
```

Observe that this file uses serialization for broadcasting events in a web application and that the necessary parameters are also initialized in this file.

For example, if we need to initialize order variable in the constructor for registering an event, we can do it in the following way −

```
public function __construct(Order $order) {
   $this->order = $order;
}
```

Listeners

Listeners handle all the activities mentioned in an event that is being registered. The artisan command **event:generate** creates all the **listeners** in the **app/listeners** directory. The Listeners folder includes a file **EventListener.php** which has all the methods required for handling listeners.

EventListener.php

```php
<?php

namespace App\Listeners;

use App\Events\SomeEvent;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Contracts\Queue\ShouldQueue;

class EventListener{
   /**
      * Create the event listener.
      *
      * @return void
   */

   public function __construct() {
      //
   }

   /**
      * Handle the event.
      *
      * @param SomeEvent $event
      * @return void
   */

   public function handle(SomeEvent $event) {
      //
   }
}
```

As mentioned in the code, it includes **handle** function for managing various events. We can create various independent listeners that target a single event.

Laravel - Facades

Facades provide a **static** interface to classes that are available in the application's service container. Laravel **facades** serve as **static proxies** to underlying classes in the service container, providing the benefit of a terse, expressive syntax while maintaining more testability and flexibility than traditional static methods.

How to create Facade

The following are the steps to create Facade in Laravel −

- **Step 1** − Create PHP Class File.
- **Step 2** − Bind that class to Service Provider.
- **Step 3** − Register that ServiceProvider to Config\app.php as providers.
- **Step 4** − Create Class which is this class extends to lluminate\Support\Facades\Facade.
- **Step 5** − Register point 4 to Config\app.php as aliases.

Facade Class Reference

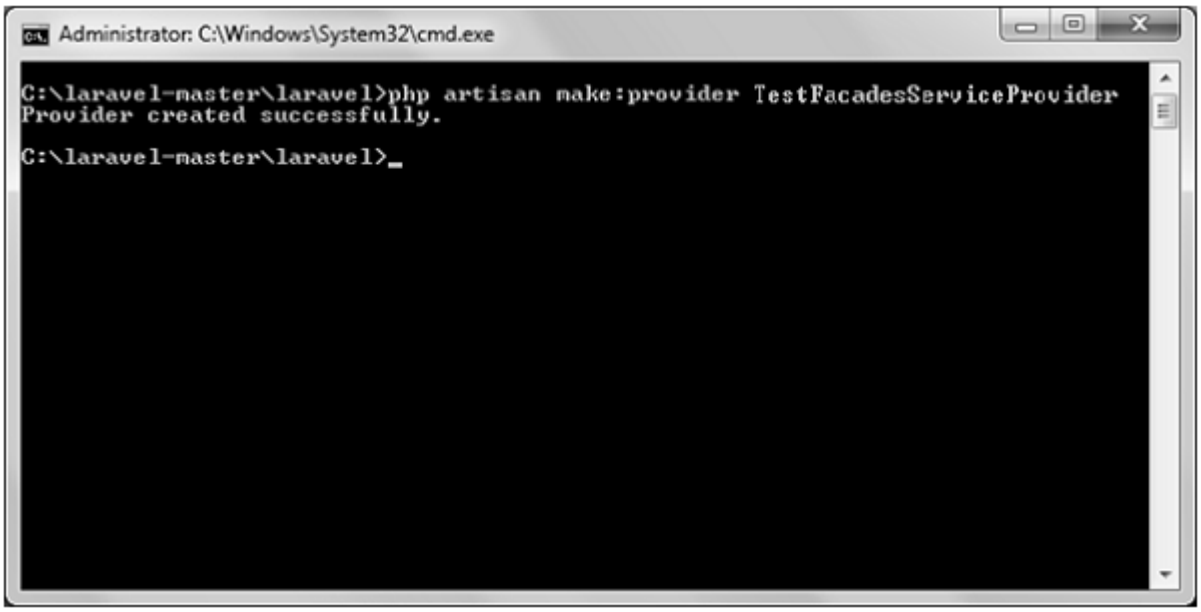
Laravel ships with many Facades. The following table show the in-built Facade class references −

Example

Step 1 − Create a service provider called **TestFacadesServiceProvider** by executing the following command.

```
php artisan make:provider TestFacadesServiceProvider
```

Step 2 − After successful execution, you will receive the following output −



Step 3 – Create a class called **TestFacades.php** at **App/Test**.

App/Test/TestFacades.php

```
<?php
   namespace App\Test;
   class TestFacades{
      public function testingFacades() {
         echo "Testing the Facades in Laravel.";
      }
   }
?>
```

Step 4 – Create a Facade class called **"TestFacades.php"** at **"App/Test/Facades"**.

App/Test/Facades/TestFacades.php

```
<?php

namespace app\Test\Facades;

use Illuminate\Support\Facades\Facade;

class TestFacades extends Facade {
   protected static function getFacadeAccessor() { return 'test'; }
}
```

Step 5 – Create a Facade class called **TestFacadesServiceProviders.php** at **App/Test/Facades.**

App/Providers/TestFacadesServiceProviders.php

```
<?php

namespace App\Providers;

use App;
use Illuminate\Support\ServiceProvider;

class TestFacadesServiceProvider extends ServiceProvider {
   public function boot() {
      //
```

```
    }
    public function register() {
       App::bind('test',function() {
          return new \App\Test\TestFacades;
       });
    }
}
```

Step 6 – Add a service provider in a file **config/app.php** as shown in the below figure.

config/app.php

```
/*
 * Application Service Providers...
 */
App\Providers\AppServiceProvider::class,
App\Providers\AuthServiceProvider::class,
App\Providers\EventServiceProvider::class,
App\Providers\RouteServiceProvider::class,
App\Providers\SomeclassServiceProvider::class,
App\Providers\TestFacadesServiceProvider::class,
```

Step 7 – Add an alias in a file **config/app.php** as shown in the below figure.

config/app.php

```
    'Schema'    => Illuminate\Support\Facades\Schema::class,
    'Session'   => Illuminate\Support\Facades\Session::class,
    'Storage'   => Illuminate\Support\Facades\Storage::class,
    'URL'       => Illuminate\Support\Facades\URL::class,
    'Validator' => Illuminate\Support\Facades\Validator::class,
    'View'      => Illuminate\Support\Facades\View::class,
    'Form'      => Illuminate\Html\FormFacade::class,
    'Html'      => Illuminate\Html\HtmlFacade::class,
    'Someclass' => App\Facades\Someclass::class,
    'TestFacades'=> App\Test\Facades\TestFacades::class,
],
```

Step 8 – Add the following lines in **app/Http/routes.php.**

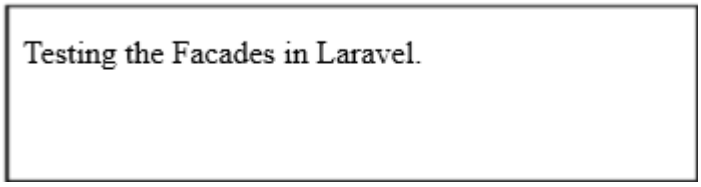
app/Http/routes.php

```
Route::get('/facadeex', function() {
   return TestFacades::testingFacades();
});
```

Step 9 – Visit the following URL to test the Facade.

```
http://localhost:8000/facadeex
```

Step 10 – After visiting the URL, you will receive the following output –



Laravel - Contracts

Laravel contracts are a set of interfaces with various functionalities and core services provided by the framework.

For example, **Illuminate\Contracts\Queue\Queue** contract uses a method which is needed for queuing jobs and **Illuminate\Contracts\Mail\Mailer** uses the method for sending emails.

Every contract defined includes corresponding implementation of the framework. All the Laravel contracts are available in the GitHub repository as mentioned below –

https://github.com/illuminate/contracts

This repository provides a variety of contracts available in the Laravel framework which can be downloaded and used accordingly.

Important Points

While working with Laravel contracts, please note the following important points –

- It is mandatory to define facades in the constructor of a class.
- Contracts are explicitly defined in the classes and you need not define the contracts in constructors.

Example

Consider the contract used for Authorization in Laravel which is mentioned below −

```
<?php

namespace Illuminate\Contracts\Auth\Access;

interface Authorizable{
   /**
      * Determine if the entity has a given ability.
      *
      * @param string $ability
      * @param array|mixed $arguments
      * @return bool
   */
   public function can($ability, $arguments = []);
}
```

The contract uses a function can which includes a **parameter** named **ability** and **arguments** which uses the user identification in the form of an **array**.

You will have to define a contract as shown in the syntax below −

```
interface <contract-name>
```

Contracts are used like facades for creating robust, well-tested Laravel applications. There are various **practical differences** with usage of contracts and facades.

The following code shows using a contract for caching a repository −

```
<?php

namespace App\Orders;
use Illuminate\Contracts\Cache\Repository as Cache;

class Repository{
   /**
      * The cache instance.
   */

   protected $cache;

   /**
```

```
      * Create a new repository instance.
      *
      * @param Cache $cache
      * @return void
   */

   public function __construct(Cache $cache) {
      $this->cache = $cache;
   }
}
```

Contract contains no implementation and new dependencies; it is easy to write an alternative implementation of a specified contract, thus a user can replace cache implementation without modifying any code base.

Laravel - CSRF Protection

CSRF refers to Cross Site Forgery attacks on web applications. CSRF attacks are the unauthorized activities which the authenticated users of the system perform. As such, many web applications are prone to these attacks.

Laravel offers CSRF protection in the following way −

Laravel includes an in built CSRF plug-in, that generates tokens for each active user session. These tokens verify that the operations or requests are sent by the concerned authenticated user.

Implementation

The implementation of CSRF protection in Laravel is discussed in detail in this section. The following points are notable before proceeding further on CSRF protection −

- CSRF is implemented within HTML forms declared inside the web applications. You have to include a hidden validated CSRF token in the form, so that the CSRF protection middleware of Laravel can validate the request. The syntax is shown below −

```
<form method = "POST" action="/profile">
  {{ csrf_field() }}
```

```
  ...
</form>
```

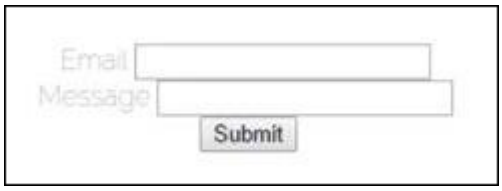
- You can conveniently build JavaScript driven applications using JavaScript HTTP library, as this includes CSRF token to every outgoing request.
- The file namely **resources/assets/js/bootstrap.js** registers all the tokens for Laravel applications and includes **meta** tag which stores **csrf-token** with **Axios HTTP library**.

Form without CSRF token

Consider the following lines of code. They show a form which takes two parameters as input: **email** and **message**.

```
<form>
   <label> Email </label>
      <input type = "text" name = "email"/>
      <br/>
   <label> Message </label> <input type="text" name = "message"/>
   <input type = ”submit” name = ”submitButton” value = ”submit”>
</form>
```

The result of the above code is the form shown below which the end user can view −



The form shown above will accept any input information from an authorized user. This may make the web application prone to various attacks.

Please note that the submit button includes functionality in the controller section. The **postContact** function is used in controllers for that associated views. It is shown below −

```
public function postContact(Request $request) {
  return $request-> all();
}
```

Observe that the form does not include any CSRF tokens so the sensitive information shared as input parameters are prone to various attacks.

Form with CSRF token

The following lines of code shows you the form re-designed using CSRF tokens −

```
<form method = ”post” >
   {{ csrf_field() }}
   <label> Email </label>
   <input type = "text" name = "email"/>
   <br/>
   <label> Message </label>
   <input type = "text" name = "message"/>
   <input type = ”submit” name = ”submitButton” value = ”submit”>
</form>
```

The output achieved will return JSON with a token as given below −

```
{
  "token": "ghfleifxDSUYEW9WE67877CXNVFJKL",
  "name": "TutorialsPoint",
  "email": "contact@tutorialspoint.com"
}
```

This is the CSRF token created on clicking the submit button.

Laravel - Authentication

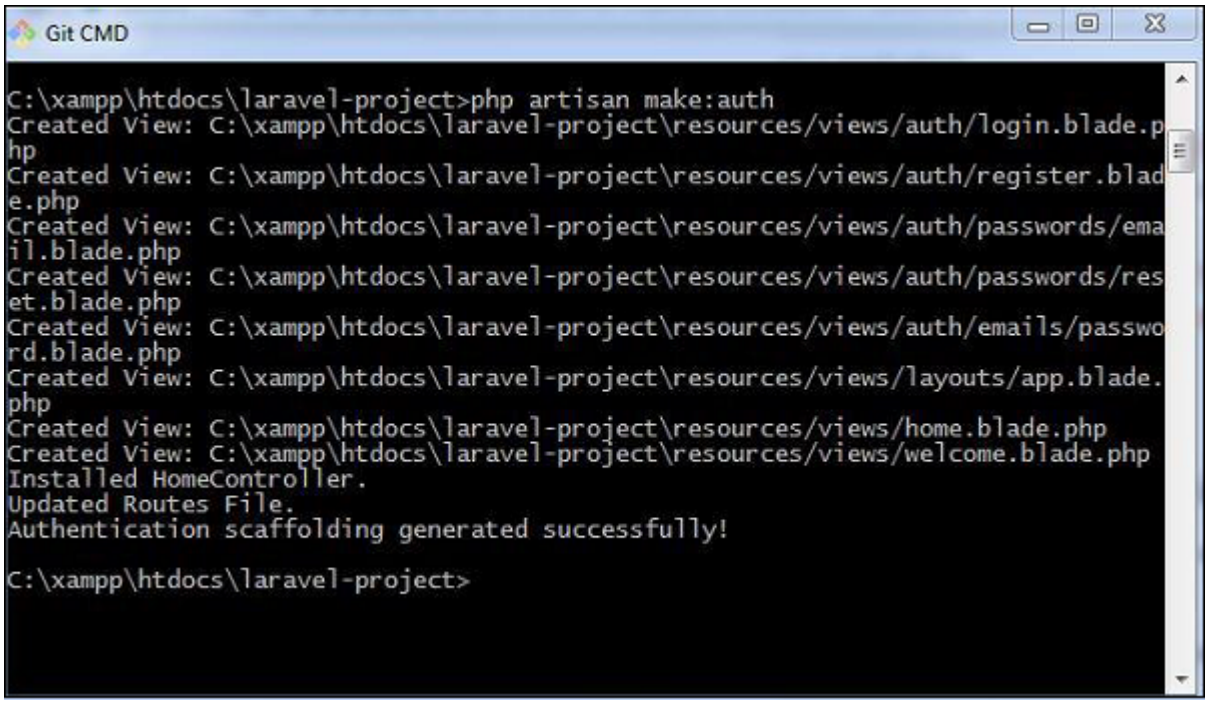
Authentication is the process of identifying the user credentials. In web applications, authentication is managed by sessions which take the input parameters such as email or username and password, for user identification. If these parameters match, the user is said to be authenticated.

Command

Laravel uses the following command to create forms and the associated controllers to perform authentication −

```
php artisan make:auth
```

This command helps in creating authentication scaffolding successfully, as shown in the following screenshot −



Controller

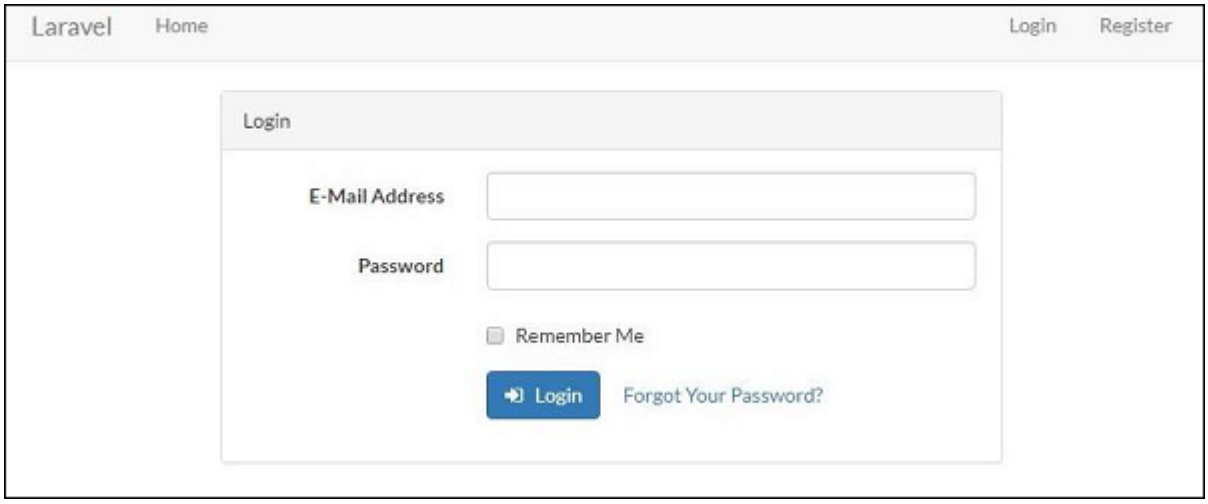
The controller which is used for the authentication process is **HomeController**.

```
<?php

namespace App\Http\Controllers;

use App\Http\Requests;
use Illuminate\Http\Request;

class HomeController extends Controller{
   /**
      * Create a new controller instance.
      *
      * @return void
   */

   public function __construct() {
      $this->middleware('auth');
   }

   /**
      * Show the application dashboard.
```

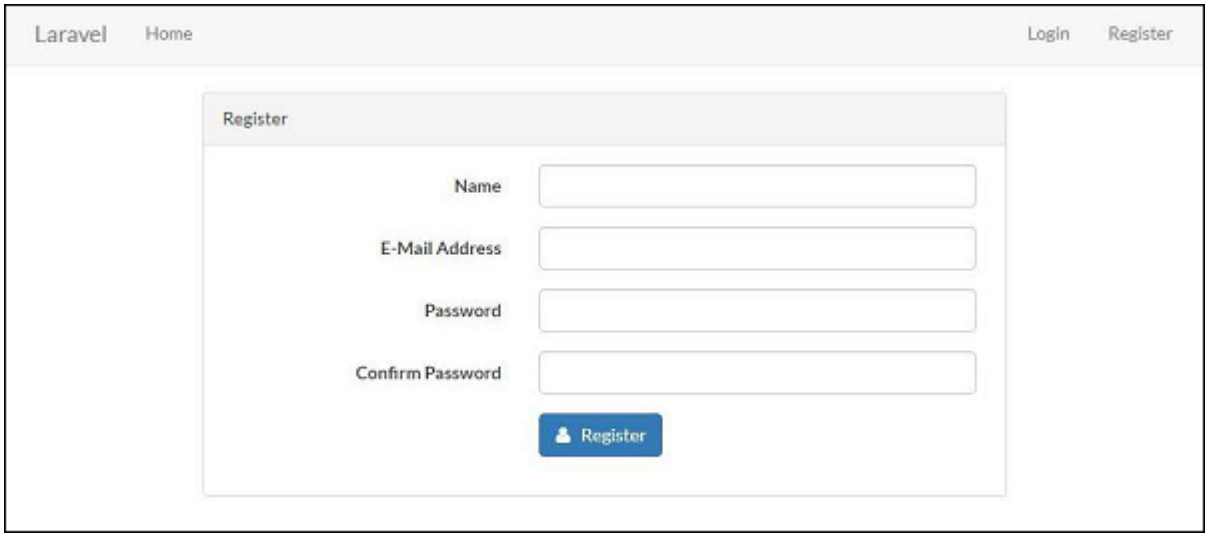
```
        *
        * @return \Illuminate\Http\Response
    */

    public function index() {
        return view('home');
    }
}
```

As a result, the scaffold application generated creates the login page and the registration page for performing authentication. They are as shown below −

Login



Registration



Manually Authenticating Users

Laravel uses the **Auth** façade which helps in manually authenticating the users. It includes the **attempt** method to verify their email and password.

Consider the following lines of code for **LoginController** which includes all the functions for authentication –

```
<?php

// Authentication mechanism
namespace App\Http\Controllers;

use Illuminate\Support\Facades\Auth;

class LoginController extends Controller{
   /**
      * Handling authentication request
      *
      * @return Response
   */

   public function authenticate() {
      if (Auth::attempt(['email' => $email, 'password' => $password]))

         // Authentication passed...
         return redirect()->intended('dashboard');
      }
   }
}
```

Laravel - Authorization

In the previous chapter, we have studied about authentication process in Laravel. This chapter explains you the authorization process in Laravel.

Difference between Authentication and Authorization

Before proceeding further into learning about the authorization process in Laravel, let us understand the difference between authentication and authorization.

In **authentication**, the system or the web application identifies its users through the credentials they provide. If it finds that the credentials are valid, they are

authenticated, or else they are not.

In **authorization**, the system or the web application checks if the authenticated users can access the resources that they are trying to access or make a request for. In other words, it checks their rights and permissions over the requested resources. If it finds that they can access the resources, it means that they are authorized.

Thus, **authentication** involves checking the validity of the user credentials, and **authorization** involves checking the rights and permissions over the resources that an authenticated user has.

Authorization Mechanism in Laravel

Laravel provides a simple mechanism for authorization that contains two primary ways, namely **Gates** and **Policies**.

Writing Gates and Policies

Gates are used to determine if a user is authorized to perform a specified action. They are typically defined in **App/Providers/AuthServiceProvider.php** using Gate facade. Gates are also functions which are declared for performing authorization mechanism.

Policies are declared within an array and are used within classes and methods which use authorization mechanism.

The following lines of code explain you how to use Gates and Policies for authorizing a user in a Laravel web application. Note that in this example, the **boot** function is used for authorizing the users.

```
<?php

namespace App\Providers;

use Illuminate\Contracts\Auth\Access\Gate as GateContract;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as Ser

class AuthServiceProvider extends ServiceProvider{
   /**
      * The policy mappings for the application.
      *
      * @var array
   */
```

```
    protected $policies = [
        'App\Model' => 'App\Policies\ModelPolicy',
    ];

    /**
        * Register any application authentication / authorization servic
        *
        * @param \Illuminate\Contracts\Auth\Access\Gate $gate
        * @return void
    */

    public function boot(GateContract $gate) {
        $this->registerPolicies($gate);
        //
    }
}
```

Laravel - Artisan Console

Laravel framework provides three primary tools for interaction through command-line namely: **Artisan, Ticker** and **REPL**. This chapter explains about Artisan in detail.

Introduction to Artisan

Artisan is the command line interface frequently used in Laravel and it includes a set of helpful commands for developing a web application.

Example

Here is a list of few commands in Artisan along with their respective functionalities −

To start Laravel project

```
php artisan serve
```

To enable caching mechanism

```
php artisan route:cache
```

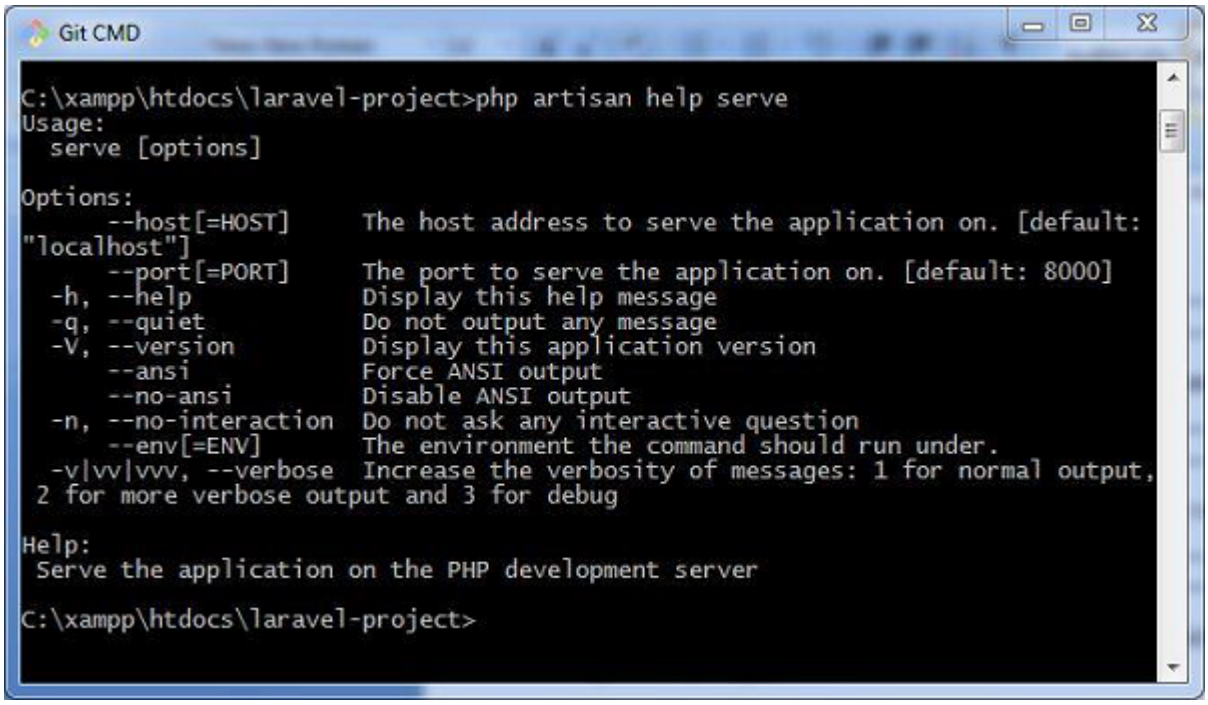
To view the list of available commands supported by Artisan

```
php artisan list
```

To view help about any command and view the available options and arguments

```
php artisan help serve
```

The following screenshot shows the output of the commands given above –



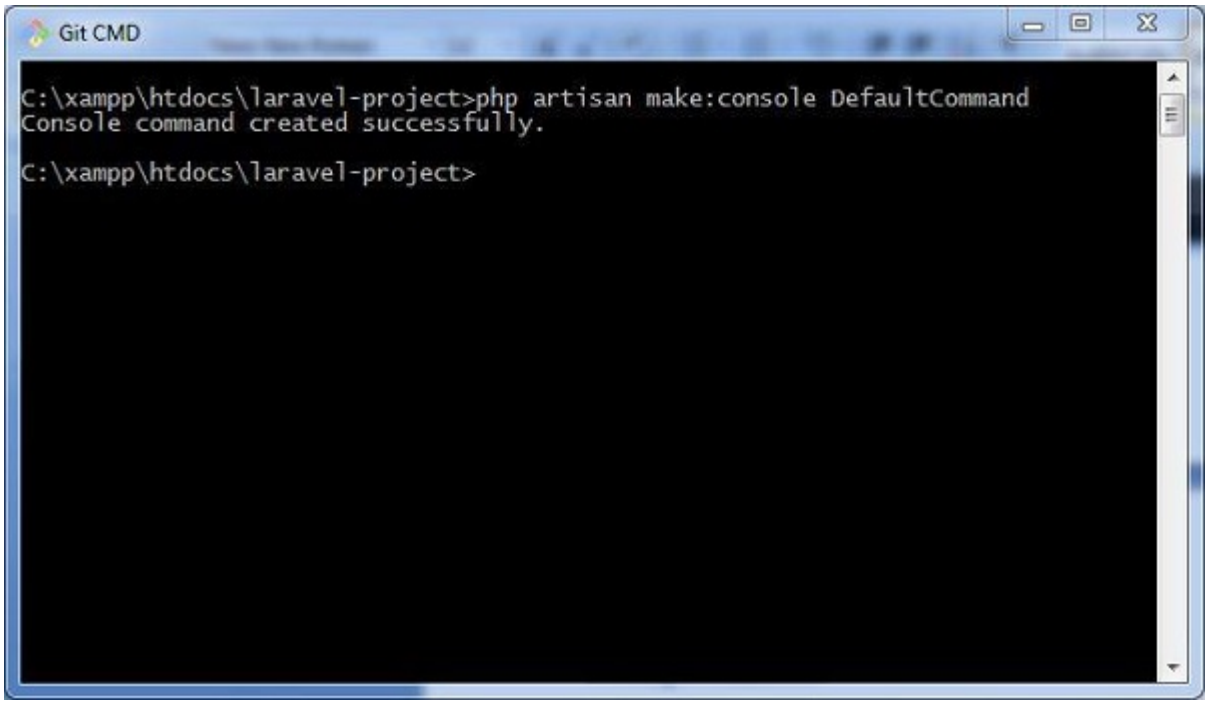
Writing Commands

In addition to the commands listed in Artisan, a user can also create a custom command which can be used in the web application. Please note that commands are stored in **app/console/commands directory**.

The default command for creating user defined command is shown below –

```
php artisan make:console <name-of-command>
```

Once you type the above given command, you can see the output as shown in the screenshot given below –



The file created for **DefaultCommand** is named as **DefaultCommand.php** and is shown below −

```
<?php

namespace App\Console\Commands;
use Illuminate\Console\Command;

class DefaultCommand extends Command{
   /**
      * The name and signature of the console command.
      *
      * @var string
   */

   protected $signature = 'command:name';

   /**
      * The console command description.
      *
      * @var string
   */

   protected $description = 'Command description';

   /**
      * Create a new command instance.
      *
```

```
      * @return void
   */

   public function __construct() {
      parent::__construct();
   }

   /**
      * Execute the console command.
      *
      * @return mixed
   */

   public function handle() {
      //
   }
}
```

This file includes the signature and description for the command that user defined. The public function named **handle** executes the functionalities when the command is executed. These commands are registered in the file **Kernel.php** in the same directory.

You can also create the schedule of tasks for the user defined command as shown in the following code −

```
<?php

namespace App\Console;

use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;

class Kernel extends ConsoleKernel {
   /**
      * The Artisan commands provided by your application.
      *
      * @var array
   */

   protected $commands = [
      // Commands\Inspire::class,
      Commands\DefaultCommand::class
```

```
    ];

    /**
       * Define the application's command schedule.
       *
       * @param \Illuminate\Console\Scheduling\Schedule $schedule
       * @return void
    */

    protected function schedule(Schedule $schedule) {
       // $schedule->command('inspire')
       // ->hourly();
    }
}
```

Note that the schedule of tasks for the given command is defined in the function named **schedule**, which includes a parameter for scheduling the tasks which takes **hourly** parameter.

The commands are registered in the array of commands, which includes the path and name of the commands.

Once the command is registered, it is listed in Artisan commands. The values included in the signature and description section will be displayed when you call for the help attribute of the specified command.

Let us see how to view the attributes of our command **DefaultCommand**. You should use the command as shown below −

```
php artisan help DefaultCommand
```

Laravel - Encryption

Encryption is a process of converting a plain text to a message using some algorithms such that any third user cannot read the information. This is helpful for transmitting sensitive information because there are fewer chances for an intruder to target the information transferred.

Encryption is performed using a process called **Cryptography**. The text which is to be encrypted is termed as **Plain Text** and the text or the message obtained after the encryption is called **Cipher Text**. The process of converting cipher text to plain text is called **Decryption**.

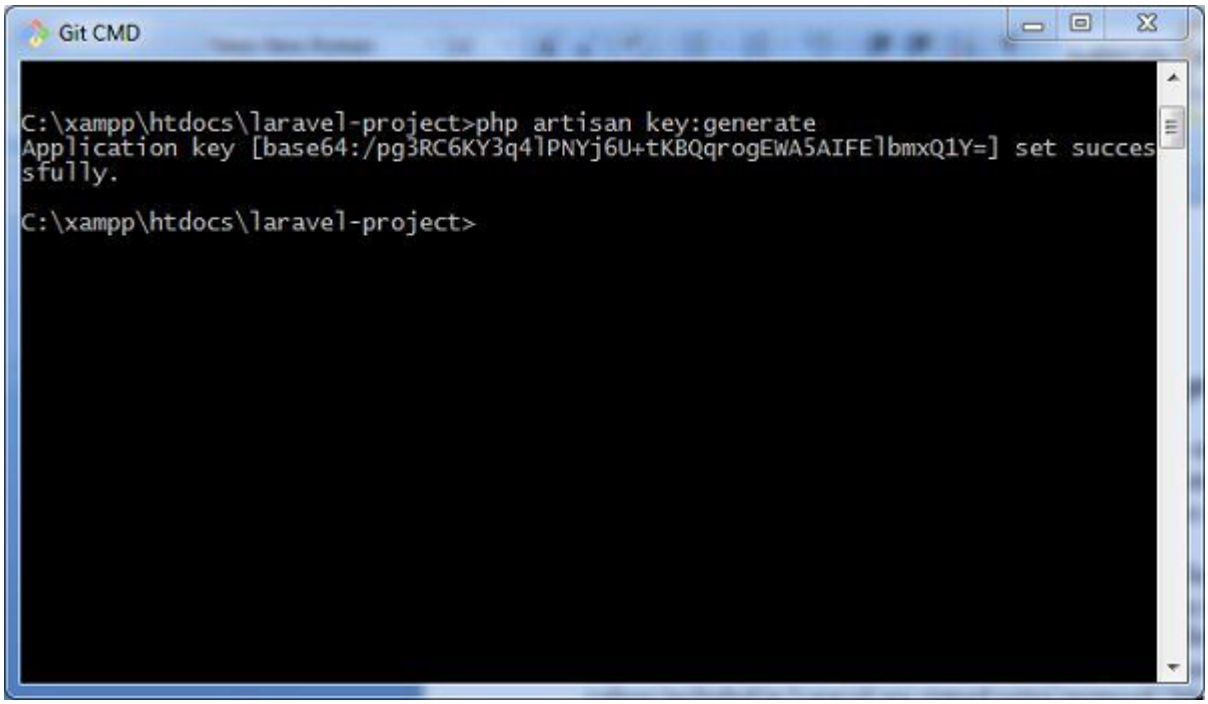
Laravel uses **AES-256** and **AES-128** encrypter, which uses Open SSL for encryption. All the values included in Laravel are signed using the protocol **Message Authentication Code** so that the underlying value cannot be tampered with once it is encrypted.

Configuration

The command used to generate the **key** in Laravel is shown below –

```
php artisan key:generate
```

Please note that this command uses the PHP secure random bytes' generator and you can see the output as shown in the screenshot given below –



The command given above helps in generating the key which can be used in web application. Observe the screenshot shown below –

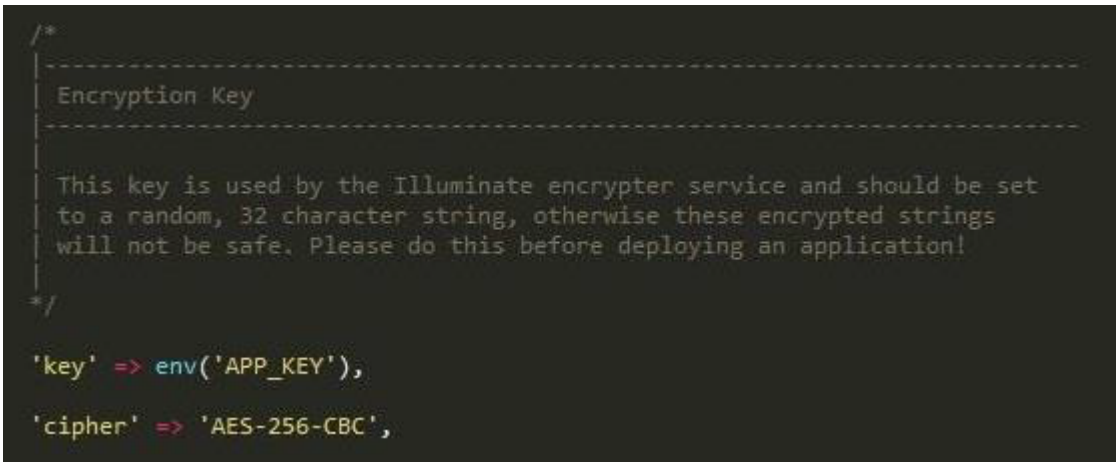
Note

The values for encryption are properly aligned in the **config/app.php** file, which includes two parameters for encryption namely **key** and **cipher**. If the value using this key is not properly aligned, all the values encrypted in Laravel will be insecure.

Encryption Process

Encryption of a value can be done by using the **encrypt helper** in the controllers of Laravel class. These values are encrypted using OpenSSL and AES-256 cipher. All the

encrypted values are signed with Message Authentication code (MAC) to check for any modifications of the encrypted string.



```
/*
|--------------------------------------------------------------------------
| Encryption Key
|--------------------------------------------------------------------------
|
| This key is used by the Illuminate encrypter service and should be set
| to a random, 32 character string, otherwise these encrypted strings
| will not be safe. Please do this before deploying an application!
|
*/

'key' => env('APP_KEY'),

'cipher' => 'AES-256-CBC',
```

The code shown below is mentioned in a controller and is used to store a secret or a sensitive message.

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class DemoController extends Controller{
   **
      * Store a secret message for the user.
      *
      * @param Request $request
      * @param int $id
      * @return Response
   */

   public function storeSecret(Request $request, $id) {
      $user = User::findOrFail($id);
      $user->fill([
         'secret' => encrypt($request->secret)
      ])->save();
   }
}
```

Decryption Process

Decryption of the values is done with the **decrypt helper**. Observe the following lines of code −

```
use Illuminate\Contracts\Encryption\DecryptException;

// Exception for decryption thrown in facade
try {
   $decrypted = decrypt($encryptedValue);
} catch (DecryptException $e) {
   //
}
```

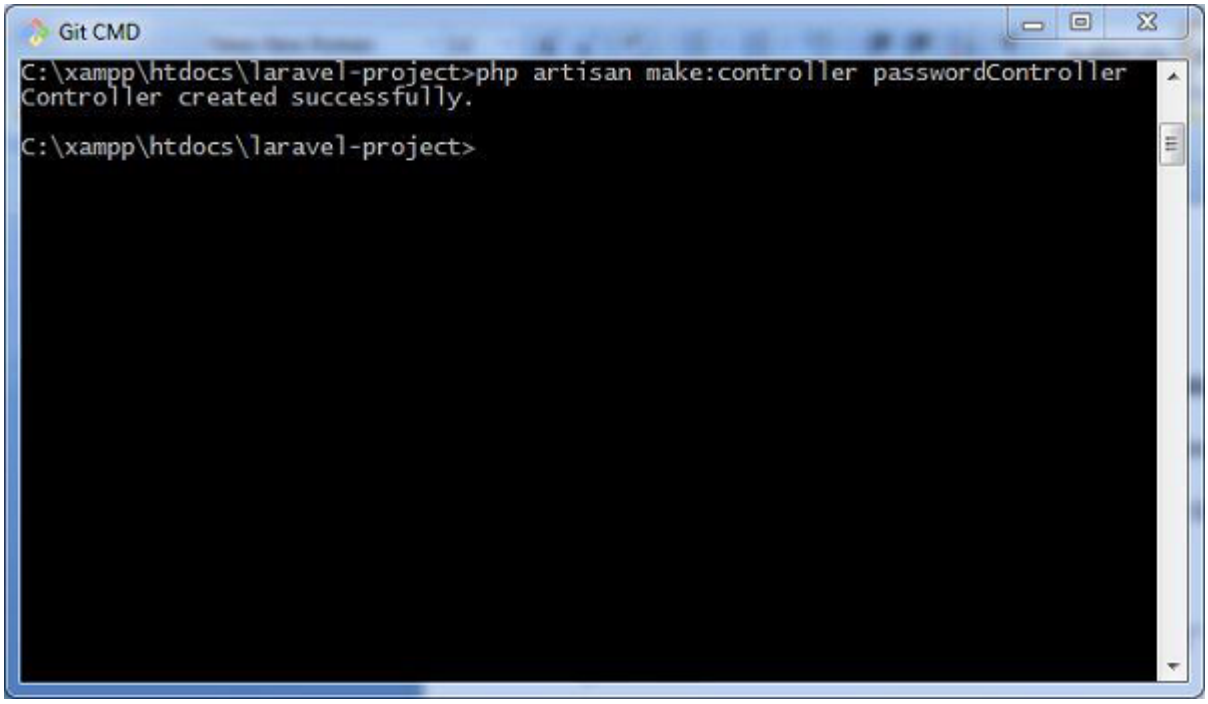
Please note that if the process of decryption is not successful because of invalid MAC being used, then an appropriate exception is thrown.

Laravel - Hashing

Hashing is the process of transforming a string of characters into a shorter fixed value or a key that represents the original string. Laravel uses the **Hash** facade which provides a secure way for storing passwords in a hashed manner.

Basic Usage

The following screenshot shows how to create a controller named **passwordController** which is used for storing and updating passwords −



The following lines of code explain the functionality and usage of the **passwordController** −

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use App\Http\Controllers\Controller

class passwordController extends Controller{
   /**
      * Updating the password for the user.
      *
      * @param Request $request
      * @return Response
   */

   public function update(Request $request) {
      // Validate the new password length...
      $request->user()->fill([
         'password' => Hash::make($request->newPassword) // Hashing pa
      ])->save();
   }
}
```

The hashed passwords are stored using **make** method. This method allows managing the work factor of the **bcrypt** hashing algorithm, which is popularly used in Laravel.

Verification of Password against Hash

You should verify the password against hash to check the string which was used for conversion. For this you can use the **check** method. This is shown in the code given below −

```
if (Hash::check('plain-text', $hashedPassword)) {
  // The passwords match...
}
```

Note that the **check** method compares the plain-text with the **hashedPassword** variable and if the result is true, it returns a true value.

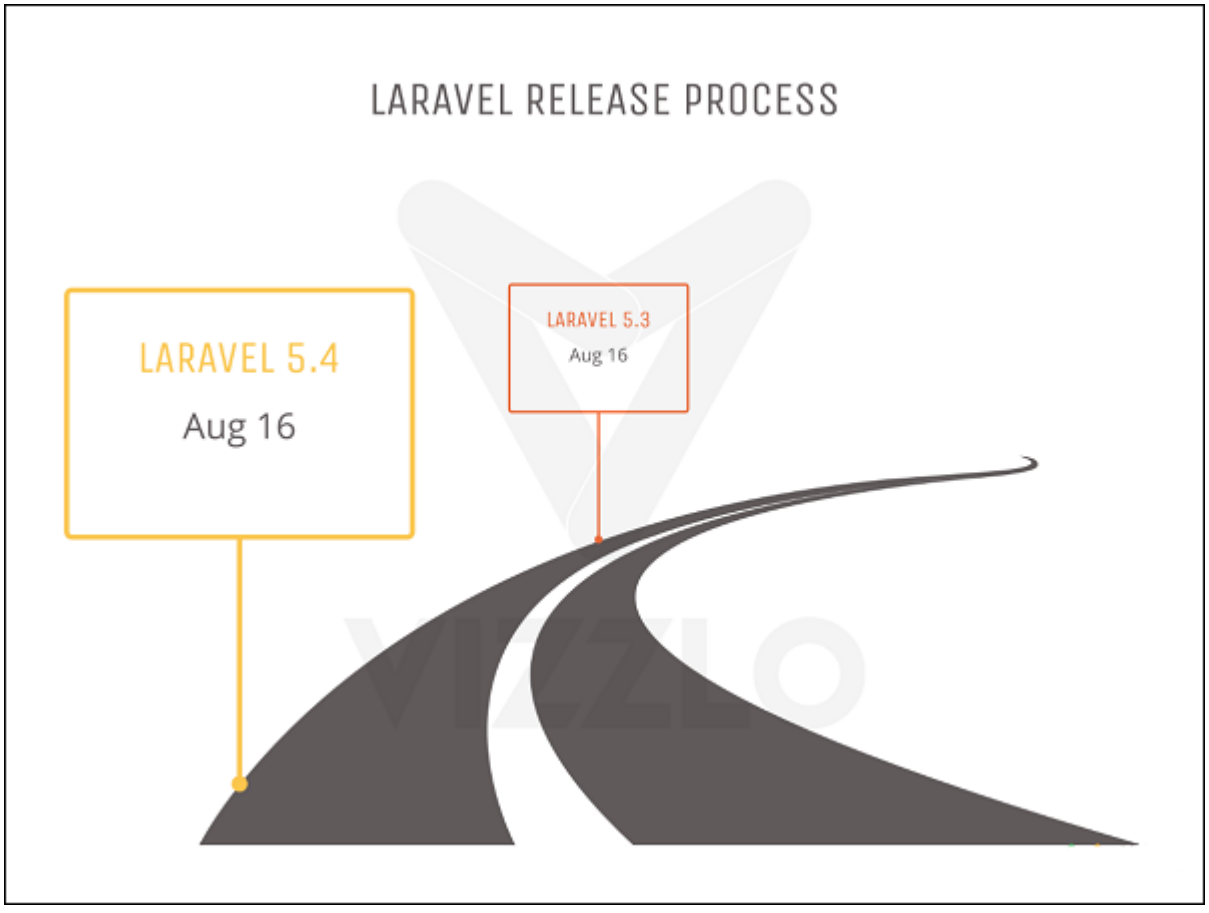
Laravel - Understanding Release Process

Every web application framework has its own version history and it is always being updated and maintained. Every latest version brings new functionality and functions which are either changed or deprecated, so it is important that you know which version will be suitable for your projects.

When it comes to Laravel, there are two active versions as given below –

- Laravel 4- released in May 2013
- Laravel 5.1- released in February 2015

Laravel 5.1 also includes various releases with the latest version of Laravel 5.1.5 which includes all the robust features for web development. The roadmap of Laravel or the version release is shown in the image below –



The following points are worth notable in the context of understanding the release process of Laravel –

- The old directory of **app/models** is removed in Laravel 5.1.

- All the controllers, middleware and requests are grouped within a directory under the app/Http folder.
- A new folder namely **Providers** directory is replaced with the **app/start** files in the previous versions of Laravel 4.x.
- All the language files and views are moved to the **resources** directory.
- New artisan command **route:cache** is used for registration of new routes and is included with the release of Laravel 5.1 and further versions.
- Laravel supports **HTTP middleware** and also includes **CSRF tokens** and authentication model.
- All the authentication models are located under one directory namely **resources/views/auth**. It includes user registration, authentication and password controllers.

Laravel Releases

Note that the highlighted version marks the latest release.

Laravel - Guest User Gates

The Guest User Gates feature is an add-on to the latest 5.7 version released in September 2018. This feature is used to initiate the authorization process for specific users.

In Laravel 5.6, there was a procedure where it used to return **false** for unauthenticated users. In Laravel 5.7, we can allow guests to go authorization checks by using the specific **nullable** type hint within the specified controller as given below –

```
<?php
Gate::define('view-post', function (?User $user) {
  // Guests
});
```

Explanation of the Code

By using a **nullable** type hint the $user variable will be null when a guest user is passed to the gate. You can then make decisions about authorizing the action. If you allow nullable types and return true, then the guest will have authorization. If you

don’t use a nullable type hint, guests will automatically get the 403 response for Laravel 5.7, which is displayed below −

The difference between 403 and 404 error is that 404 is displayed when user tries to access the unknown resource or URL and 403 error as mentioned in the snapshot above is displayed if unauthorized user accesses the website.

Laravel - Artisan Commands

Laravel 5.7 comes with new way of treating and testing new commands. It includes a new feature of testing artisan commands and the demonstration is mentioned below −

```
class ArtisanCommandTest extends TestCase{
   public function testBasicTest() {
      $this->artisan('nova:create', [
         'name' => 'My New Admin panel'
      ])
      ->expectsQuestion('Please enter your API key', 'apiKeySecret')
      ->expectsOutput('Authenticating...')
      ->expectsQuestion('Please select a version', 'v1.0')
      ->expectsOutput('Installing...')
      ->expectsQuestion('Do you want to compile the assets?', 'yes')
      ->expectsOutput('Compiling assets...')
      ->assertExitCode(0);
   }
}
```

Explanation of Code

Here a new class named “ArtisanCommandTest” is created under test cases module. It includes a basic function **testBasicTest** which includes various functionalities of assertions.

The artisan command **expectsQuestion** includes two attributes. One with question and other with an **apiKeySecret**. Here, the artisan validates the apiKeySecret and verifies the input sent by user.

The same scenario applies for the question “Please select a version” where a user is expected to mention a specific version.

Laravel - Pagination Customizations

Laravel includes a feature of pagination which helps a user or a developer to include a pagination feature. Laravel paginator is integrated with the query builder and Eloquent ORM. The paginate method automatically takes care of setting the required limit and the defined offset. It accepts only one parameter to paginate i.e. the number of items to be displayed in one page.

Laravel 5.7 includes a new pagination method to customize the number of pages on each side of the paginator. The new method no longer needs a custom pagination view.

The custom pagination view code demonstration is mentioned below −

```
<?php
namespace App\Http\Controllers;
use Illuminate\Support\Facades\DB;
use App\Http\Controllers\Controller;
class UserController extends Controller{
   /**
   * Show all of the users for the application.
   *
   * @return Response
   */
   public function index() {
      $users = DB::table('users')->paginate(15);
      return view('user.index', ['users' => $users]);
   }
}
```

The new pagination customization as per Laravel standards is mentioned below −

```
<?php
User::paginate(10)->onEachSide(5);
```

Note that **onEachSide** refers to the subdivision of each pagination records with 10 and subdivision of 5.

Laravel - Dump Server

Laravel dump server comes with the version of Laravel 5.7. The previous versions do not include any dump server. Dump server will be a development dependency in

laravel/laravel composer file.

With release of version 5.7, you'll get this command which includes a concept out-of-thebox which allows user to dump data to the console or an HTML file instead of to the browser. The command execution is mentioned below −

```
php artisan dump-server
# Or send the output to an HTML file
php artisan dump-server --format=html > dump.html
```

Explanation

The command runs a server in the background which helps in collection of data sent from the application, that sends the output through the console. When the command is not running in the foreground, the dump() function is expected to work by default.

Laravel - Action URL

Laravel 5.7 introduces a new feature called "callable action URL". This feature is similar to the one in Laravel 5.6 which accepts string in action method. The main purpose of the new syntax introduced Laravel 5.7 is to directly enable you access the controller.

The syntax used in Laravel 5.6 version is as shown −

```
<?php
$url = action('UserController@profile', ['id' => 1]);
```

The similar action called in Laravel 5.7 is mentioned below −

```
<?php
$url = action([PostsController::class, 'index']);
```

One advantage with the new callable array syntax format is the feature of ability to navigate to the controller directly if a developer uses a text editor or IDE that supports code navigation.