TOPS Technologies

Laravel HandBook

Table of Contents

What is Framework??

- A framework is an environment for creating the application using single programming language.
  Example: Codeignter, Hibernate
- There are also Framework which supports many Languages such .Net Framework

Laravel :

- A free, open-source cross platform PHP web framework intended for the development of web applications.
- Created by Taylor Otwell in June 2011.
  It is following the model-view-controller (MVC) architectural pattern.
  It is written in PHP 5.

- The source code of Laravel is hosted on GitHub and licensed under the terms of MIT License.

Laravel Features :

Routing system: A simple and easy roadmap to controllers, middleware and their methods
Contracts: A set of interfaces that defines the core services. Each contract has a corresponding implementation provided by the framework. For example, Laravel provides a queue implementation with a variety of drivers, and a mailer implementation that is powered by SwiftMailer.
Unit-Testing: Laravel itself contains unit tests that detect and prevent regressions in the framework. Unit tests can be run through the provided artisan command-line utility.
Laravel Views: Composers serve as customizable logical code units that can be executed when a view is loaded.
Automatic Pagination: Replacing the usual manual implementation approaches with automated methods integrated Authentication: A several well documented options for tweaking the behavior of the authentication services such as login, logout, access restrictions etc. Authorization: A simple way to authorize user actions against a given resource. Its somewhat like ACL (Access Control List) .Multiple file system: It third party package Flysystem to provide multiple file support. Local or Cloud based storage can also be used to provide simple configuration.

How secure is Laravel Framework?

Laravel uses hashed and salted passwords – meaning your users passwords are never saved in the DB in plain text.
Laravel uses prepared SQL statements which make injection attacks pretty much impossible
Laravel provides a convenient way to escape/unescape user input to prevent users injection of <script> tags and so on Laravel community has been very responsive to bug reports related to security
Prevents Cross-Site Request Forgery(CSRF)
Prevents Cross-Site Scripting

System Requirements

PHP >= 5.6.4
OpenSSL PHP Extension
PDO PHP Extension
Mbstring PHP Extension

Tokenizer PHP Extension
XML PHP Extension

Composer

Composer is a tool for dependency management in PHP. It allows you to declare the libraries your project depends on and it will manage (install/update) them for you.
Suppose: You have a project that depends on a number of libraries.Some of those libraries depend on other libraries.
Composer: Enables you to declare the libraries you depend on. Finds out which versions of which packages can and need to be installed, and installs them (meaning it downloads them into your project). Composer is multi-platform and we strive to make it run equally well on Windows, Linux and OSX.

System Requirements & Installation for composer

Composer requires PHP 5.3.2+ to run.
Download and run Composer-Setup.exe. It will install the latest Composer version and set up your PATH so that you can just call composer from any directory in your command line.

Installation of Laravel

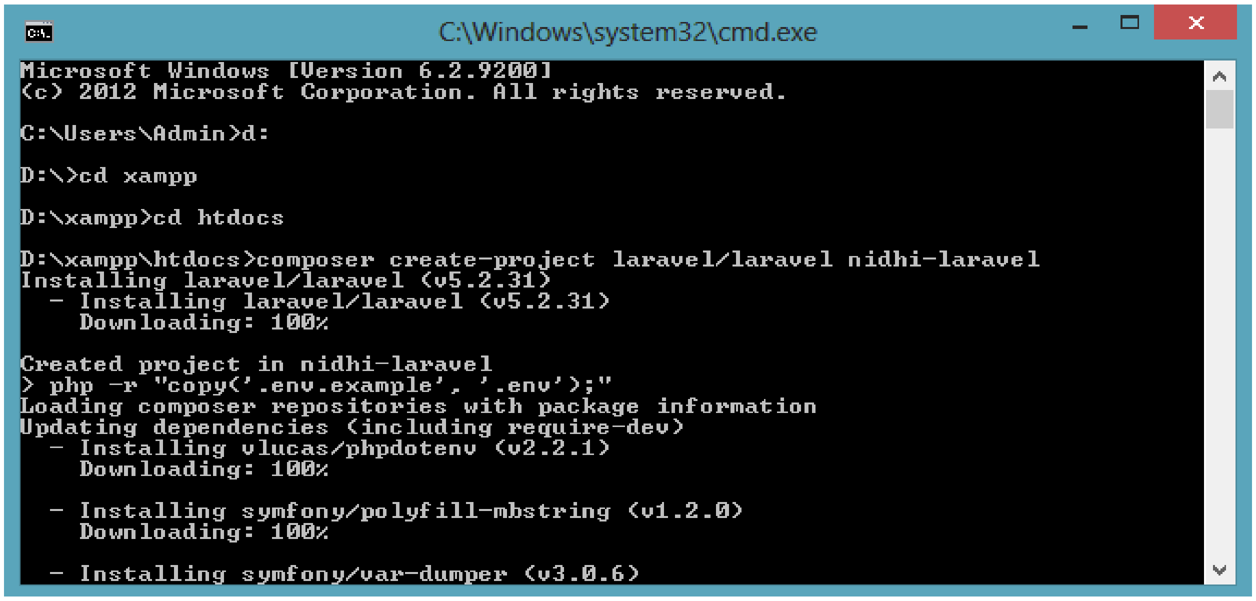
Laravel utilizes Composer to manage its dependencies. So, before using Laravel, make sure you have Composer installed on your machine. Two ways of Installing the laravel

1. Via Laravel Installer

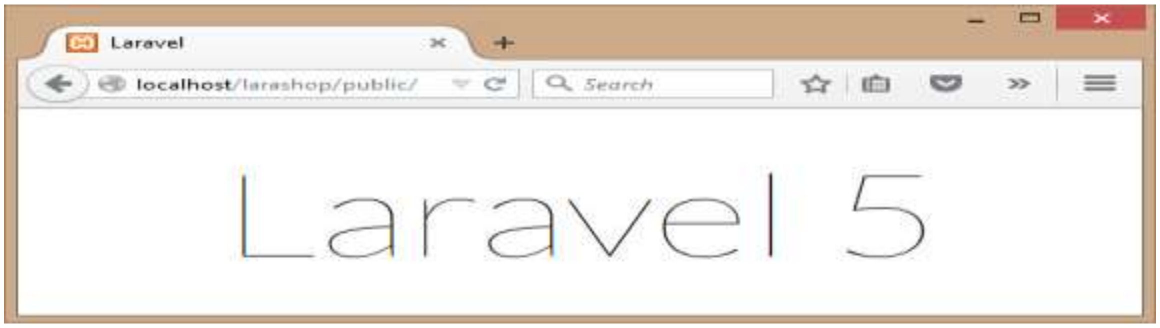
Download the Laravel installer using Composer: composer global require "laravel/installer
Once installed, the laravel new command will create a fresh
Laravel installation in the directory:
laravel new blog

2. Via Composer Create-Project

Alternatively, you may also install Laravel by issuing the Composer create-project command in your terminal: composer create-project --prefer-dist laravel/laravel blog

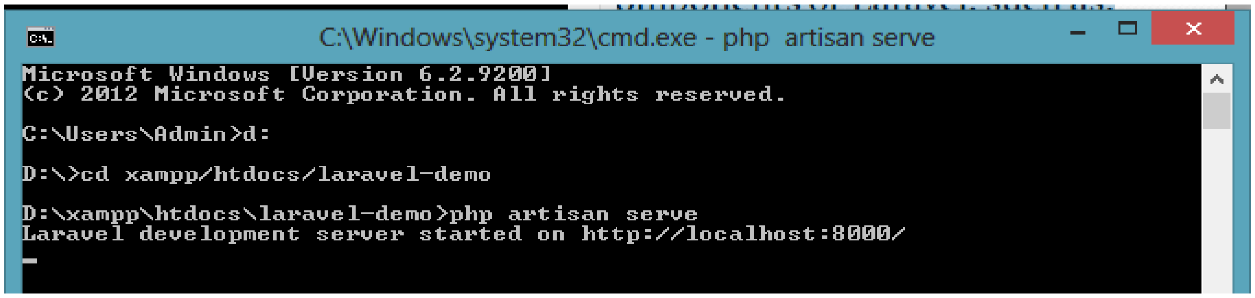


```
Microsoft Windows [Version 6.2.9200]
(c) 2012 Microsoft Corporation. All rights reserved.

C:\Users\Admin>d:

D:\>cd xampp

D:\xampp>cd htdocs

D:\xampp\htdocs>composer create-project laravel/laravel nidhi-laravel
Installing laravel/laravel (v5.2.31)
  - Installing laravel/laravel (v5.2.31)
    Downloading: 100%

Created project in nidhi-laravel
> php -r "copy('.env.example', '.env');"
Loading composer repositories with package information
Updating dependencies (including require-dev)
  - Installing vlucas/phpdotenv (v2.2.1)
    Downloading: 100%

  - Installing symfony/polyfill-mbstring (v1.2.0)
    Downloading: 100%

  - Installing symfony/var-dumper (v3.0.6)
```



Local Development Server

If PHP installed locally and PHP's built-in development server can be used to serve application, Also the serve Artisan command can be used.



```
Microsoft Windows [Version 6.2.9200]
(c) 2012 Microsoft Corporation. All rights reserved.

C:\Users\Admin>d:

D:\>cd xampp/htdocs/laravel-demo

D:\xampp\htdocs\laravel-demo>php artisan serve
Laravel development server started on http://localhost:8000/
_
```

php artisan serve
php artisan serve --port=8000
This command will start a development server
at http://localhost:8000

Configuration of laravel

Public Directory: After installing Laravel, configure the web server's document / web root to be the public directory. The index.php in this directory serves as the front controller for all HTTP requests entering your application. Configuration Files: All of the configuration files for the Laravel framework are stored in the config directory. Directory Permissions: After installing Laravel, it is necessary to configure some permissions. Directories within the storage and the bootstrap/cache directories should be writable by your web server or Laravel will not run.

Configuration of laravel

Application Key: After installing Laravel set the application key to a random string. If installed Laravel via Composer or the Laravel installer, this key has already been set for you by the php artisan key:generate command. Typically, this string should be 32 characters long. The key can be set in the .env environment file. If not renamed the .env. example file to .env, do that now. If the application key is not set, user sessions and other encrypted data will not be secure!

Additional Configuration

The config/app.php file contains several options such as timezone and locale that you may wish to change according to your application. Can configure a few additional components of Laravel, such as:

- Cache
- Database
- Session

Directory Structure and Roles

App: Contains the core code of your application

- Bootstrap: Contains files that bootstrap the framework and configure auto loading. This directory also houses a cache directory which contains framework generated files for performance optimization such as the route and services cache files.
- Config: Contains all of your application's configuration files.
- Database: The database directory contains your database migration and seeds.
- Public: Contains the index.php file, which is the entry point for all requests entering the application. This directory also houses your assets such as images, JavaScript, and CSS.
- Resources: Contains your views as well as your raw, un-compiled assets such as LESS, SASS, or JavaScript. This directory also houses all of your language files.
- The Routes Directory: The routes directory contains all of the route definitions for your application. By default, three route files are included with Laravel:
  - web.php
  - api.php
  - console.php
- The web.php file contains routes that the RouteServiceProvider places in the web iddleware group, which provides session state, CSRF protection, and cookie encryption. If the application does not offer a stateless, RESTful API, all of the routes will most likely be defined in the web.php file.
- The api.php file contains routes that the RouteServiceProvider places in the api middleware group, which provides rate limiting.These routes are intended to be stateless, so requests entering the application through these routes are intended to be authenticated via tokens and will not have access to session state.
  - The console.php file is where you may define all of your Closure based console commands. Each Closure is bound to a command instance allowing a simple approach to interacting with each command's IO methods. Even though this file does not define HTTP routes, it defines console based entry points (routes) into your application.
  - Storage : Contains your compiled Blade templates, file based sessions, file caches, and other files generated by the framework. This directory is segregated into:
    1. app
    2. Framework
    3. Logs

The app directory may be used to store any files generated by your application. The framework directory is used to store framework generated files and caches. The logs directory contains your application's log files. The storage/app/public directory may be used to store usergenerated files, such as profile avatars, that should be publicly accessible.
A symbolic link at public/storage which points to this directory needs to be created . The link may be created using the php artisan storage:link command.
Tests: Contains your automated tests. Each test class should be suffixed with the word Test. You may run your tests using the phpunit or php vendor/bin/phpunit commands.
Vendor: The vendor directory contains your Composer dependencies.

App Directory Roles

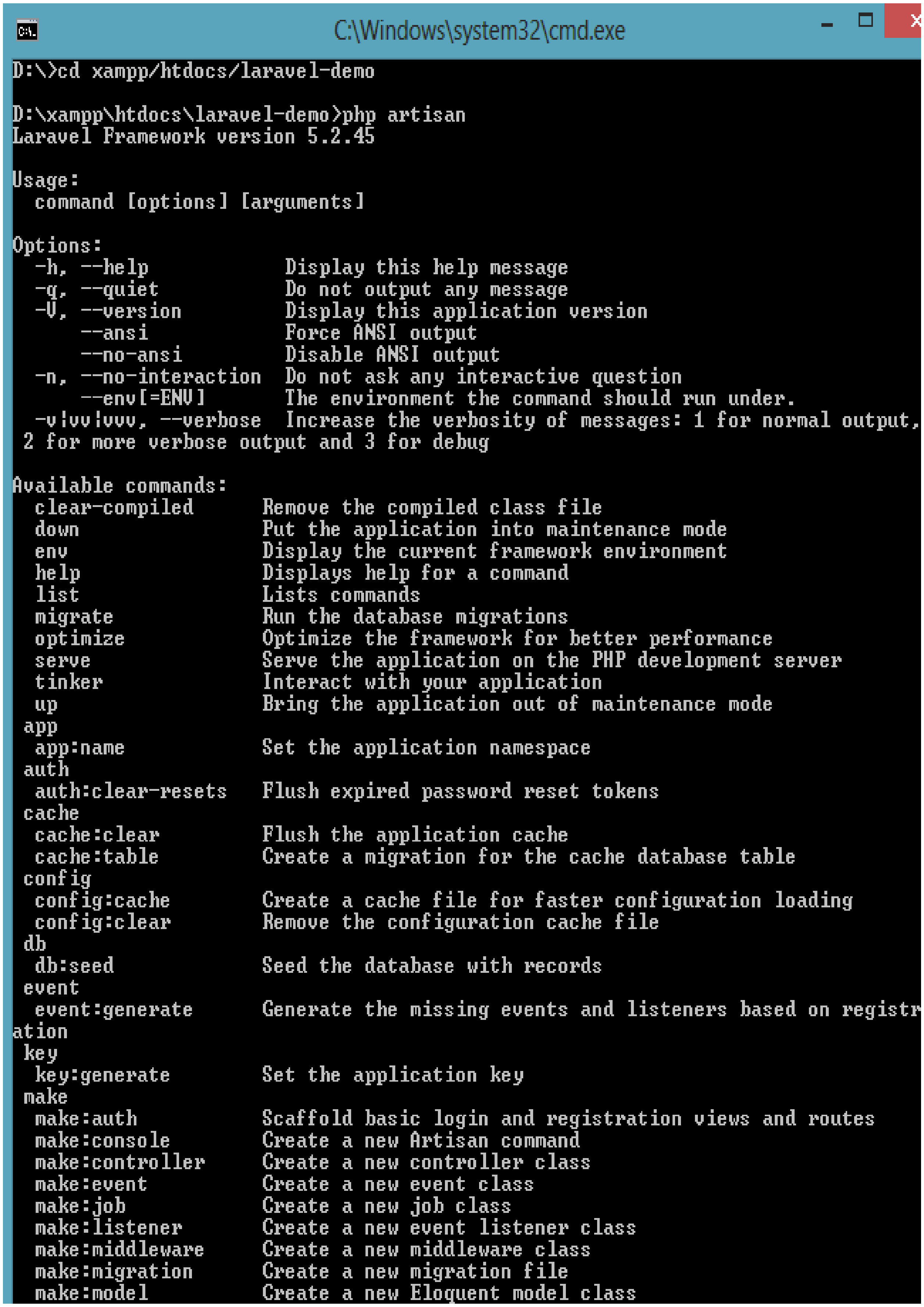
By default, this directory is namespaced under App and is auto loaded by Composer

Console: The Console directory contains all of the custom Artisan commands for your application. These commands may be generated using the make: command command. This directory also houses your console kernel, which is where your custom Artisan commands are registered and your scheduled tasks are defined.

- Events: This directory does not exist by default, but will be created by the event:generate and make:event Artisan commands.
  The Events directory, as you might expect, houses event classes. Events may be used to alert other parts of the application that a given action has occurred, providing a great deal of flexibility and decoupling.
- Exceptions: Contains your application's exception handler and is also a good place to place any exceptions thrown by your application. To customize how exceptions are logged or rendered, we need to modify the Handler class in this directory.
- Http: Contains your controllers, middleware, and form requests. Almost all of the logic to handle requests entering your application will be placed in this directory.
- Jobs: This directory does not exist by default, but will be created for you if you execute the make:job Artisan command.

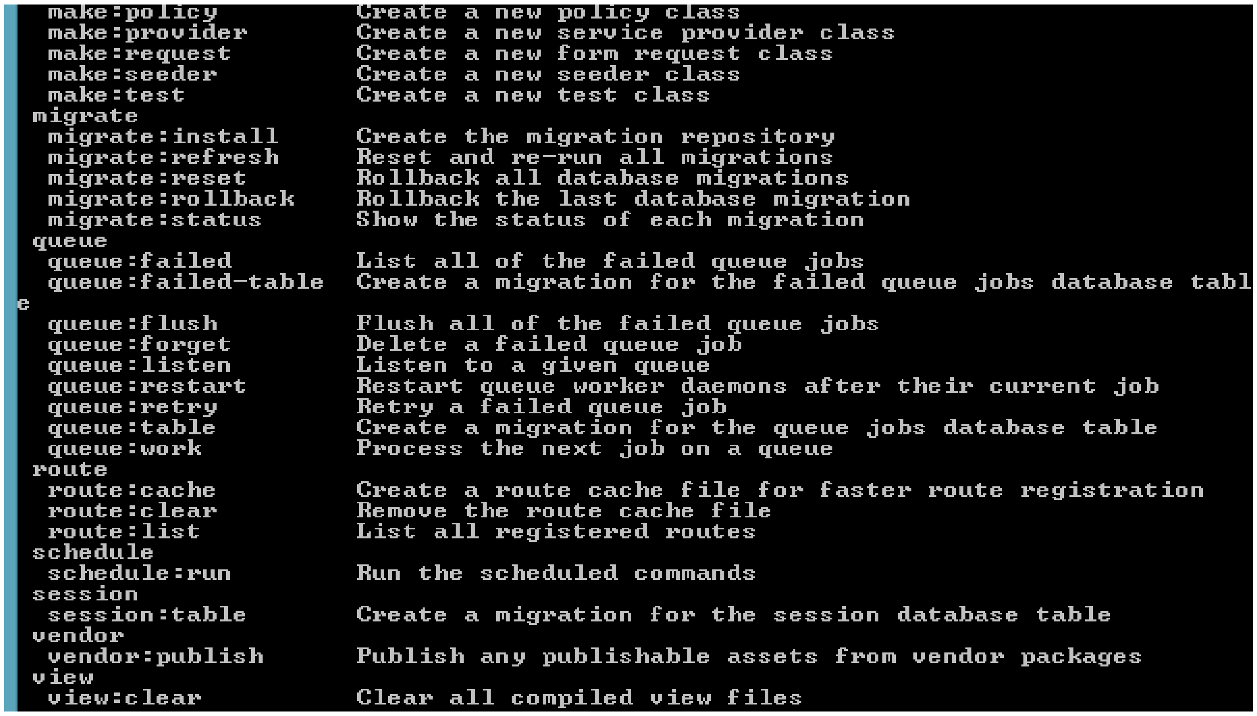
The Jobs directory houses the queue able jobs for your application. Jobs may be queued by your application or run synchronously within the current request lifecycle. Jobs that run synchronously during the current request are
sometimes referred to as "commands" since they are an implementation of the command pattern.

- Listeners :This directory does not exist by default, but will be created by executing the event:generate or make:listener Artisan commands.
  The Listeners directory contains the classes that handle your events. Event listeners receive an event instance and perform logic in response to the event being fired. For example, a UserRegistered event might be handled by a SendWelcomeEmail listener.
- Mail : This directory does not exist by default, but will be created for you if you execute the make:mail Artisan command.
  The Mail directory contains all of your classes that represent emails sent by your application.
  Mail objects allow you to encapsulate all of the logic of building an email in a single, simple class that may be sent using the Mail::send method
- Policies: This directory does not exist by default, but will be created for you if you execute the make:policy Artisan command.The Policies directory contains the authorization policy classes for the application. Policies are used to determine if a user can perform a given action against a resource.
  Providers: Contains all of the service providers for the `application. Service providers bootstrap your application by `binding services in the service container, registering events, or performing any other tasks to prepare your application for incoming requests.
  In a fresh Laravel application, this directory will already contain several providers. You are free to add your own providers to this
  directory as needed.



C:\Windows\system32\cmd.exe

```
D:\>cd xampp/htdocs/laravel-demo

D:\xampp\htdocs\laravel-demo>php artisan
Laravel Framework version 5.2.45

Usage:
  command [options] [arguments]

Options:
  -h, --help            Display this help message
  -q, --quiet           Do not output any message
  -V, --version         Display this application version
      --ansi            Force ANSI output
      --no-ansi         Disable ANSI output
  -n, --no-interaction  Do not ask any interactive question
      --env[=ENV]       The environment the command should run under.
  -v|vv|vvv, --verbose  Increase the verbosity of messages: 1 for normal output,
 2 for more verbose output and 3 for debug

Available commands:
  clear-compiled      Remove the compiled class file
  down                Put the application into maintenance mode
  env                 Display the current framework environment
  help                Displays help for a command
  list                Lists commands
  migrate             Run the database migrations
  optimize            Optimize the framework for better performance
  serve               Serve the application on the PHP development server
  tinker              Interact with your application
  up                  Bring the application out of maintenance mode
 app
  app:name            Set the application namespace
 auth
  auth:clear-resets   Flush expired password reset tokens
 cache
  cache:clear         Flush the application cache
  cache:table         Create a migration for the cache database table
 config
  config:cache        Create a cache file for faster configuration loading
  config:clear        Remove the configuration cache file
 db
  db:seed             Seed the database with records
 event
  event:generate      Generate the missing events and listeners based on registr
ation
 key
  key:generate        Set the application key
 make
  make:auth           Scaffold basic login and registration views and routes
  make:console        Create a new Artisan command
  make:controller     Create a new controller class
  make:event          Create a new event class
  make:job            Create a new job class
  make:listener       Create a new event listener class
  make:middleware     Create a new middleware class
  make:migration      Create a new migration file
  make:model          Create a new Eloquent model class
```



```
  make:policy           Create a new policy class
  make:provider         Create a new service provider class
  make:request          Create a new form request class
  make:seeder           Create a new seeder class
  make:test             Create a new test class
 migrate
  migrate:install       Create the migration repository
  migrate:refresh       Reset and re-run all migrations
  migrate:reset         Rollback all database migrations
  migrate:rollback      Rollback the last database migration
  migrate:status        Show the status of each migration
 queue
  queue:failed          List all of the failed queue jobs
  queue:failed-table    Create a migration for the failed queue jobs database tabl
e
  queue:flush           Flush all of the failed queue jobs
  queue:forget          Delete a failed queue job
  queue:listen          Listen to a given queue
  queue:restart         Restart queue worker daemons after their current job
  queue:retry           Retry a failed queue job
  queue:table           Create a migration for the queue jobs database table
  queue:work            Process the next job on a queue
 route
  route:cache           Create a route cache file for faster route registration
  route:clear           Remove the route cache file
  route:list            List all registered routes
 schedule
  schedule:run          Run the scheduled commands
 session
  session:table         Create a migration for the session database table
 vendor
  vendor:publish        Publish any publishable assets from vendor packages
 view
  view:clear            Clear all compiled view files
```

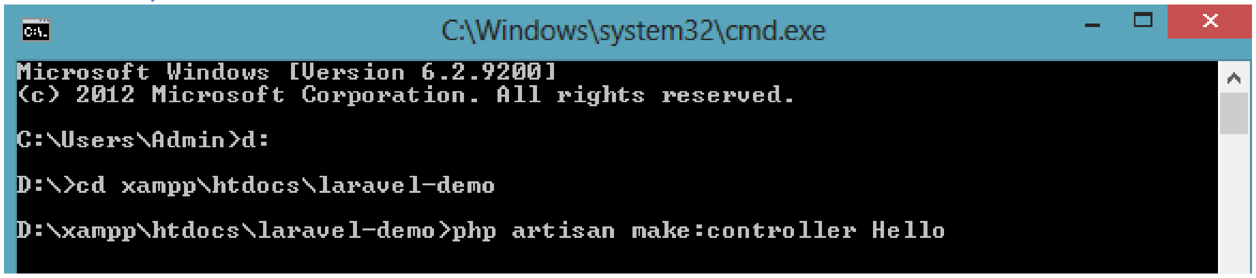
Artisan command line tool

Artisan is a command line tool that automates common tasks in Laravel. The tool is run from the command line. The artisan command line tool can be used to perform the following tasks and many more

- Generate boilerplate code – easily create controllers, models etc.
- Database migrations – migrations are used to manipulate database objects. Migrations can be used to create and drop tables etc.
- Database seeding – seeding is a term used to add dummy records to the database.
- Routing
- Application configuration
- Run unit tests

Hello World Module creation

Step – 1



```
Microsoft Windows [Version 6.2.9200]
(c) 2012 Microsoft Corporation. All rights reserved.

C:\Users\Admin>d:

D:\>cd xampp\htdocs\laravel-demo

D:\xampp\htdocs\laravel-demo>php artisan make:controller Hello
```

Step – 2 Open /app/Http/Controllers/Hello.php

You will see the following code

```
<?php

namespace App\Http\Controllers;
```

```
use Illuminate\Http\Request;
use App\Http\Requests;
use App\Http\Controllers\Controller;
class Hello extends Controller
{   /**
    * Display a listing of the resource.
        * @return Response     */
   public function index()
   {
      //
   }
   /**
    * Show the form for creating a new resource.
        * @return Response
    */
   public function create()
   {
      //
   }
   /**
    * Store a newly created resource in storage.
     * @param  Request  $request
    * @return Response     */
   public function store(Request $request)
   {
      //
   }
   /**
    * Display the specified resource.
    * @param  int  $id
    * @return Response     */
   public function show($id)
   {
      //
   }
   /**
    * Show the form for editing the specified resource.
    * @param  int  $id
    * @return Response     */
   public function edit($id)
   {
      //
   }
   /**
    * Update the specified resource in storage.
```

```
  * @param  Request  $request
  * @param  int  $id
  * @return Response     */
 public function update(Request $request, $id)
 {
   //
 }
 /**
  * Remove the specified resource from storage.
  * @param  int  $id
  * @return Response
  */
 public function destroy($id)
 {
   //
 }
}
```

Step -3 Basic routing

In this section, we will create a new route that will display hello world in the web browser

1. Open routes.php file. The file is located in /app/Http/
2. Add the following code to routes.php

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

Open the following URL

```
http://localhost:8000/hello
```

Route to controller

For the sake of reference, we will comment the above code as follows

Step – 1

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

Add the following code to /app/Http/routes.php
Route::get('hello', 'Hello@index');

Step – 2 Open /app/Http/Controllers/Hello.php

Add the following code to the index function

```
public function index()
{
  return 'hello world from controller : )';
}
```

Save the changes
Open the following URL in your browser.

http://localhost:8000/hello

Loading a view from a controller

In a typical application, a controller will have to load a view. In this tutorial, we will

1. Create a view that will be loaded from the controller
2. Pass a variable to the view and modify
3. Add a new route to /app/Http/routes.php that will pass in the Hello world view
4. create a new file hello.blade.php in /resources/views/

Step -1 Add the following code to /resources/views/hello.blade.php

```
<!DOCTYPE html>
<html>    <head>      <title>Laravel</title>
<link href="//fonts.googleapis.com/css?family=Lato:100" rel="stylesheet" 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">Hello {{$name}}, welcome to Laraland! : )</div>
           </div>
        </div>
     </body>
</html>
```

- Hello {{$name}}, welcome to Laraland! :) is the text that is displayed in the browser. {{$name}} prints the value of the name variable. Open /app/Http/routes.php

Add the following code

```
Route::get('/hello/{name}', 'Hello@show');
```

HERE,

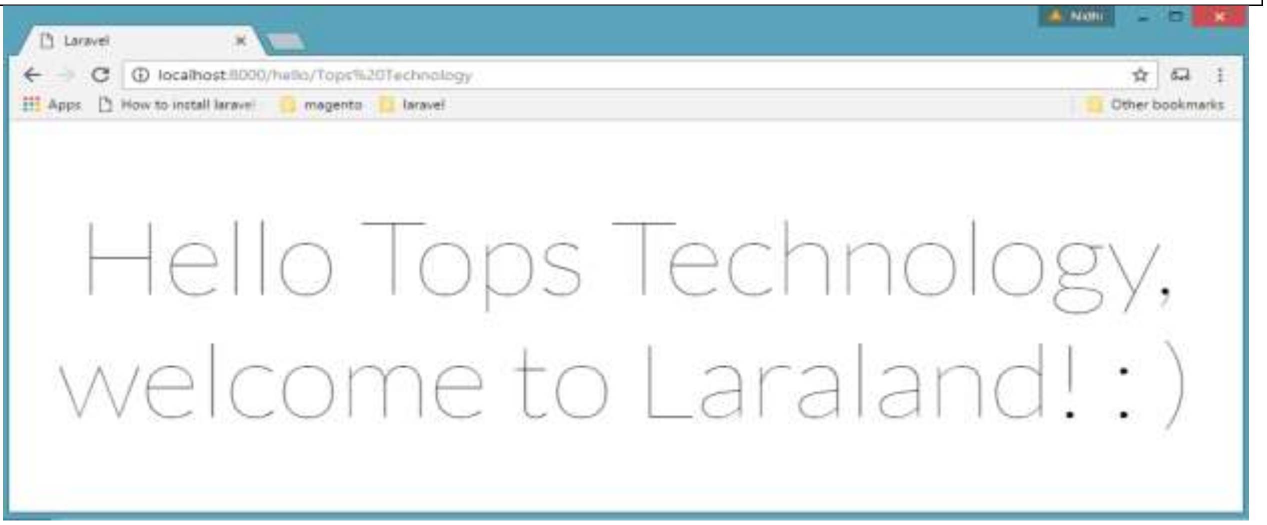
- Route::get('/hello/{name}', 'Hello@show'); directs the URI GET request for /hello/value to show() function in Hello controller.

Step -2 Open app/Http/Controllers/Hello.php

Modify show($id){...} to the following

```
public function show($name) {
   return view('hello',array('name' => $name));
}
```

http://localhost:8000/hello/tops Technology



Hello Tops Technology, welcome to Laraland! :)

Service Container:

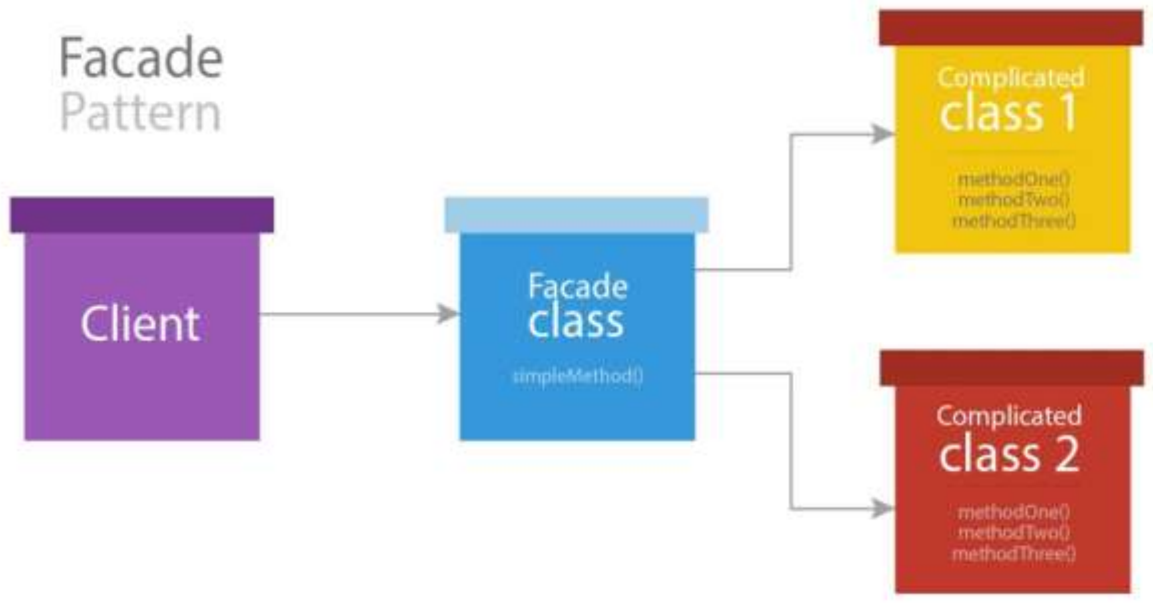
The Laravel service container is a powerful tool for managing class dependencies. Dependency injection is a fancy word that essentially means this: class dependencies are "injected" into the class via the constructor or, in some cases, "setter" methods.

Service Providers:

Service providers are the central place of all Laravel application bootstrapping. Your own application, as well as all of Laravel's core services are bootstrapped via service providers. But, what do we mean by "bootstrapped"? In general, we mean registering things, including registering service container bindings, event listeners, filters, and even routes. Service providers are the central place to configure your application. If you open the config/app.php file included with Laravel, you will see a providers array. These are all of the service provider classes that will be loaded for your application. Of course, many of them are "deferred" providers, meaning they will not be loaded on every request, but only when the services they provide are actually needed. we need to add this package 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.

Facades :

The Facade pattern is a software design pattern which is often used in object oriented programming. A facade is, in fact, a class wrapping a complex library to provide a simpler and more readable interface to it. The Facade pattern can also be used to provide a unified and welldesigned API to a group of complex and poorly designed APIs.



How Facades Are implemented in Laravel

As you probably know, every service inside the container has a unique name. In a Laravel application, to access a service directly from the container, we can use the App::make() method or the app() helper function.

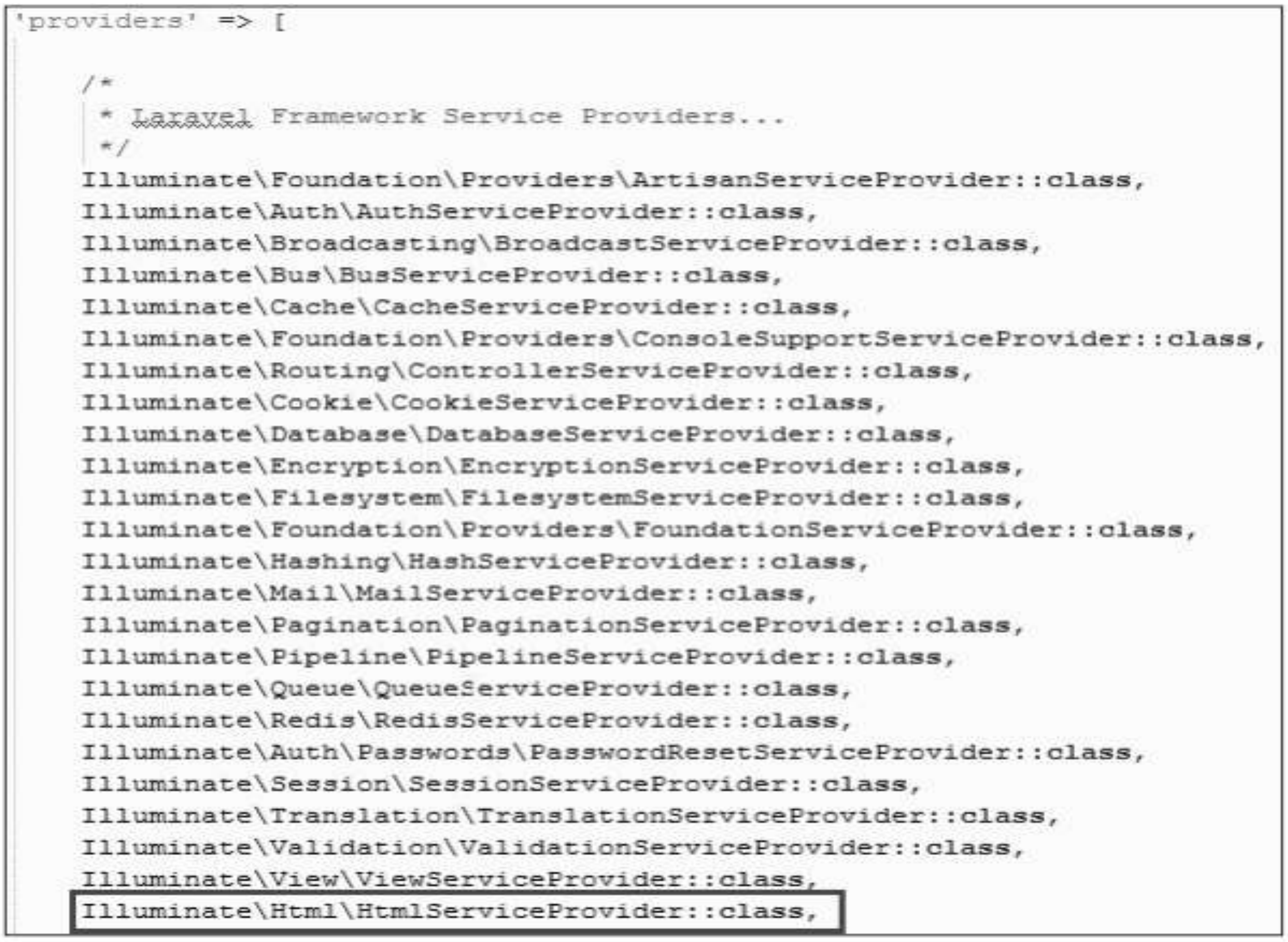
```
<?php
App::make('some_service')->methodName();
```

Aliases

Since Laravel facades are PHP classes, we need to import them before we can use them. Thanks to the namespaces and auto loading support in PHP, all classes are automatically loaded when we access them by the fully-qualified name. PHP also supports aliasing of classes by using the use directive:

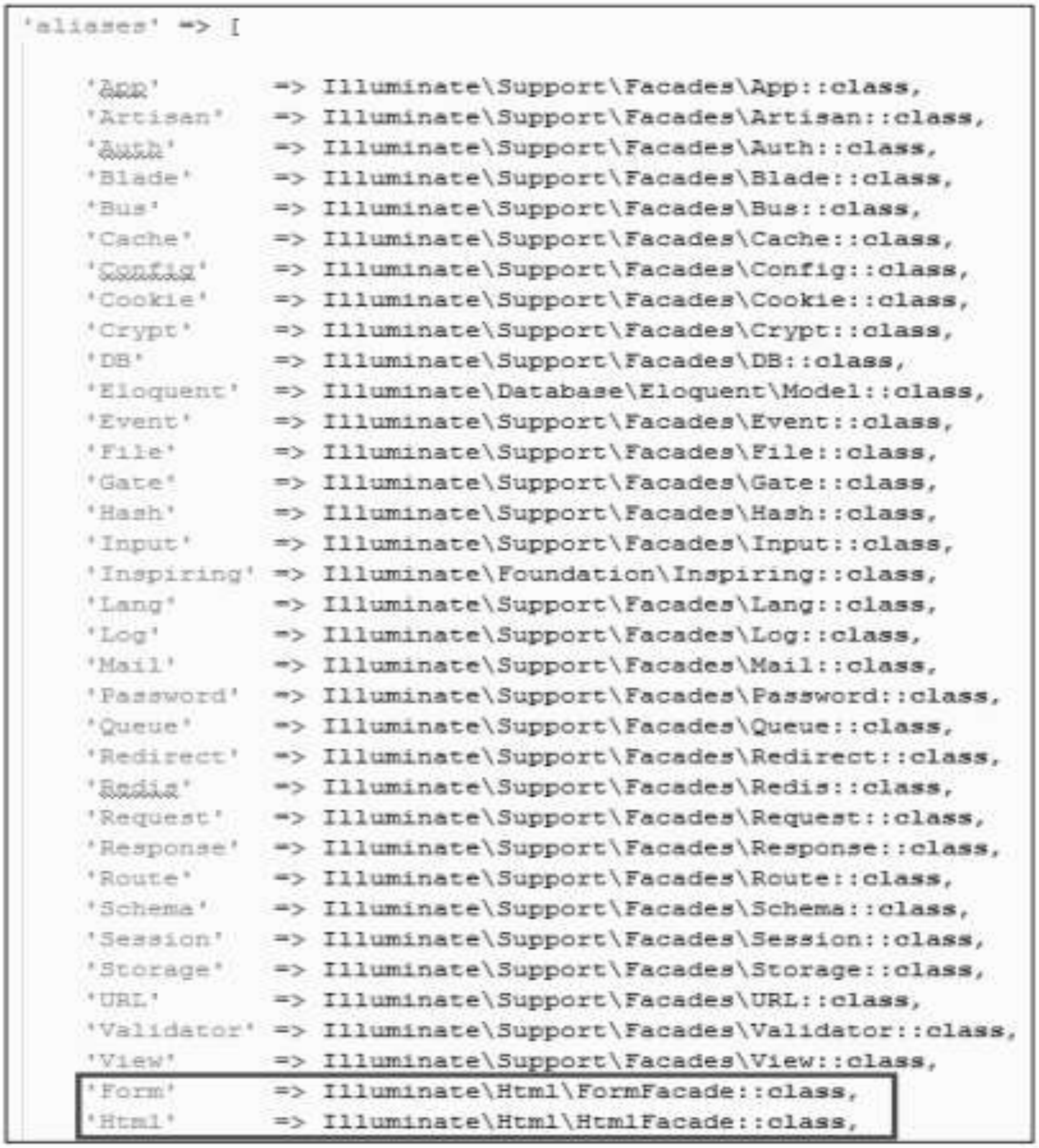
```
use App\Facades\SomeServiceFacade
SomeServiceFacade:SomeMethod();
```

app/ composer.json ->>>



```
'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,
```

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,
```

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.

Basic Routing

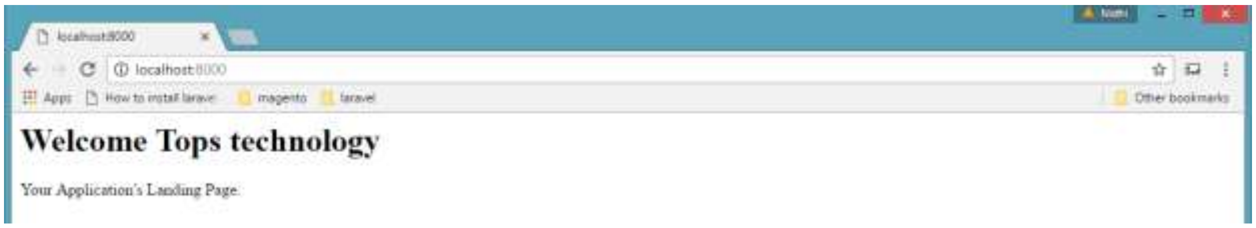
The most basic Laravel routes simply accept a URI and a Closure or using controller, providing a very simple and expressive method of defining routes.

Step -1 :app/ http/routes.php

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

Step -2 : resourcse /views / welcome.balde.php

```
<!DOCTYPE html>
<html> <head> <title>Laravel</title>
<link href = "https://fonts.googleapis.com/css?family=Lato:100" rel =
"stylesheet" 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</div> </div>
</div>
<div class="container">
  <div class="row">
    <div class="col-md-10 col-md-offset-1">
      <div class="panel panel-default">
        <div class="panel-heading">Welcome</div>
        <div class="panel-body">
          Your Application's Landing Page.
        </div>
      </div>
    </div>
  </div>
</div>

</body> </html>
```

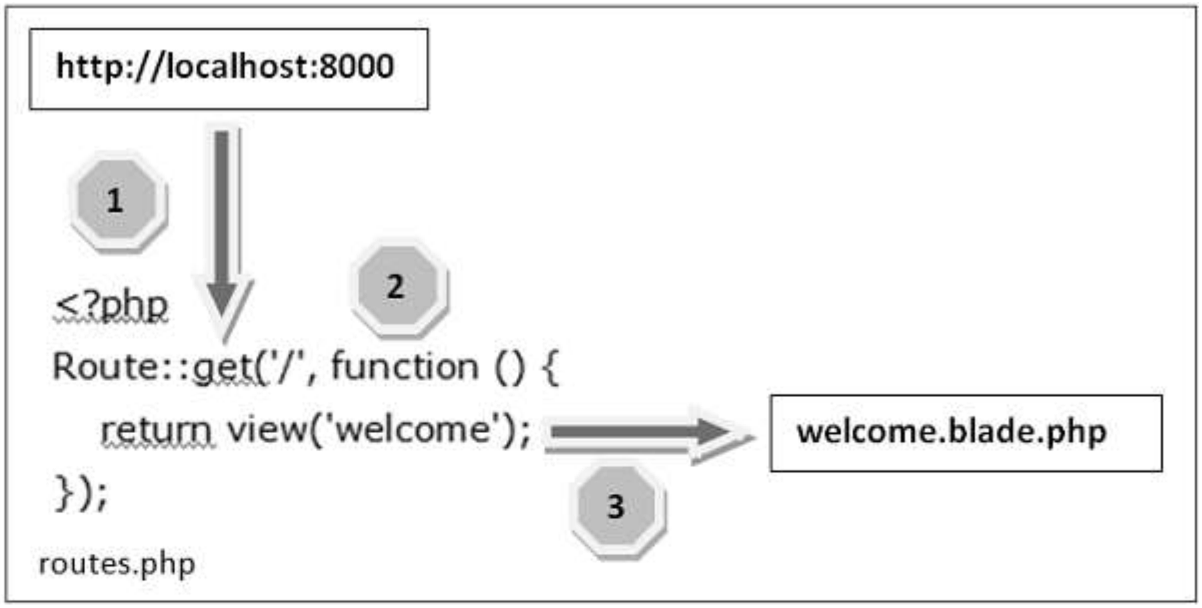


Welcome Tops technology

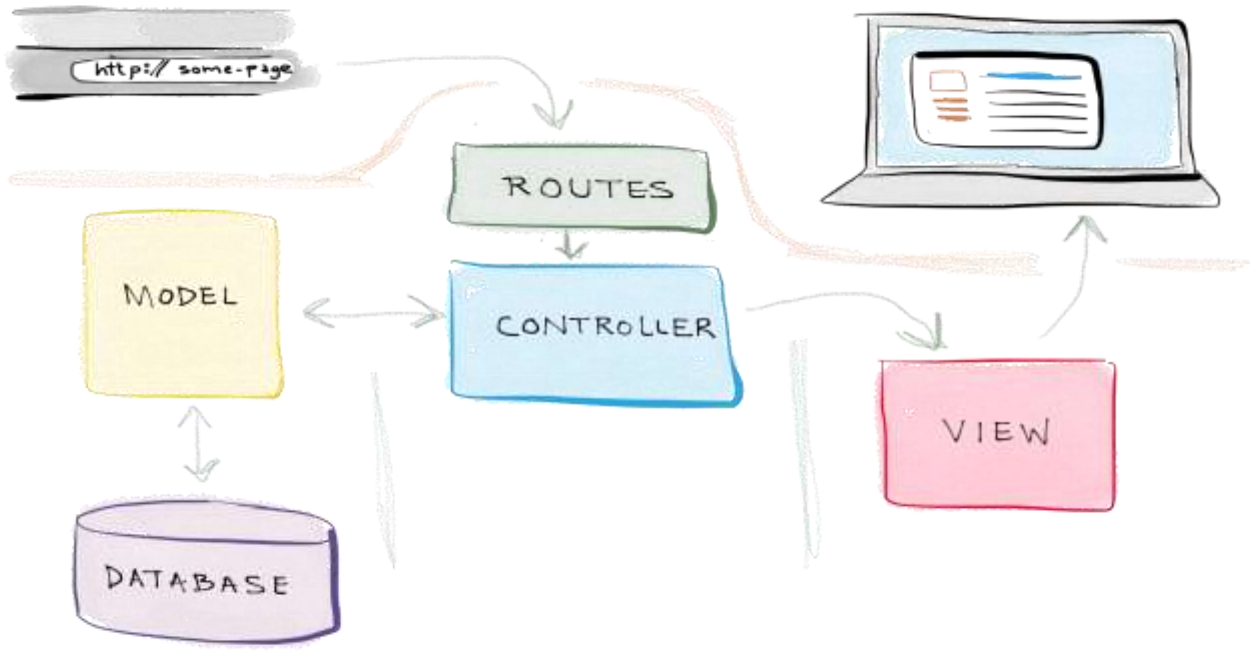
Your Application's Landing Page.

The Default Route Files

All Laravel routes are defined in your route files, which are located in the routes directory. These files are automatically loaded by the framework. The routes/web.php file defines routes that are for your web interface. These routes are assigned the web middleware group, which provides features like session state and CSRF(cross-site request forgery) protection .



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



Step By Step Explanation

Let us now understand the steps in detail −

Step 1 − First, we need to execute the root URL of the application.

Step 2 − The executed URL will match with the appropriate method in the route.php file. In our case, it will match to get the method and the root (‘/’) URL. This will execute the related function.

Step 3 − The function calls the template fileresources/views/welcome.blade.php. The function later calls the view() function with argument ‘welcome’ without using the blade.php. It will produce the following HTML output.

The Default Route Files

The routes in routes/api.php are stateless and are assigned the api middleware group. For most applications, you will begin by defining routes in your routes/web.php file.

Available Router Methods :

The router allows you to register routes that respond to any HTTP verb

- Route::get($uri, $callback); Route::post($uri, $callback);
- Route::put($uri, $callback);
- Route::patch($uri, $callback);
- Route::delete($uri, $callback);
- Route::options($uri, $callback);

Routing Parameters

• Often in the application, we intend to capture the parameters passed with the URL. To do this, we need to modify the code in routes.php file accordingly.
• There are two ways by which we can capture the parameters passed with the URL.

- Required Parameters
- Optional Parameters

➢ Required Parameters

These parameters must be present in the URL. For example, you may intend to capture the ID from the URL to do something with that ID.
Here is the sample coding for routes.php file for that purpose.
Route::get('ID/{id}',function($id){ echo 'ID: '.$id; });
Description:Whatever argument that we pass after the root RL(http://localhost:8000/ID/5), it will be stored in $id and we can use that parameter for further processing but here we are simply displaying it. We can pass it onto view or controller for further processing.
You may define as many route parameters as required by your route:
Example:
Route::get('posts/{post}/comments/{comment}',
function ($postId, $commentId) { // });
• Route parameters are always encased within {} braces and should consist of alphabetic characters.
• Route parameters may not contain a - character. Use an underscore (_) instead.

Optional Parameters

There are some parameters which may or may not be present in the URL and in such cases we can use the optional parameters. The presence of these parameters is not necessary in the URL.These parameters are indicated by “?” sign after the name of the parameters.

Here is the sample coding for routes.php file for that purpose.

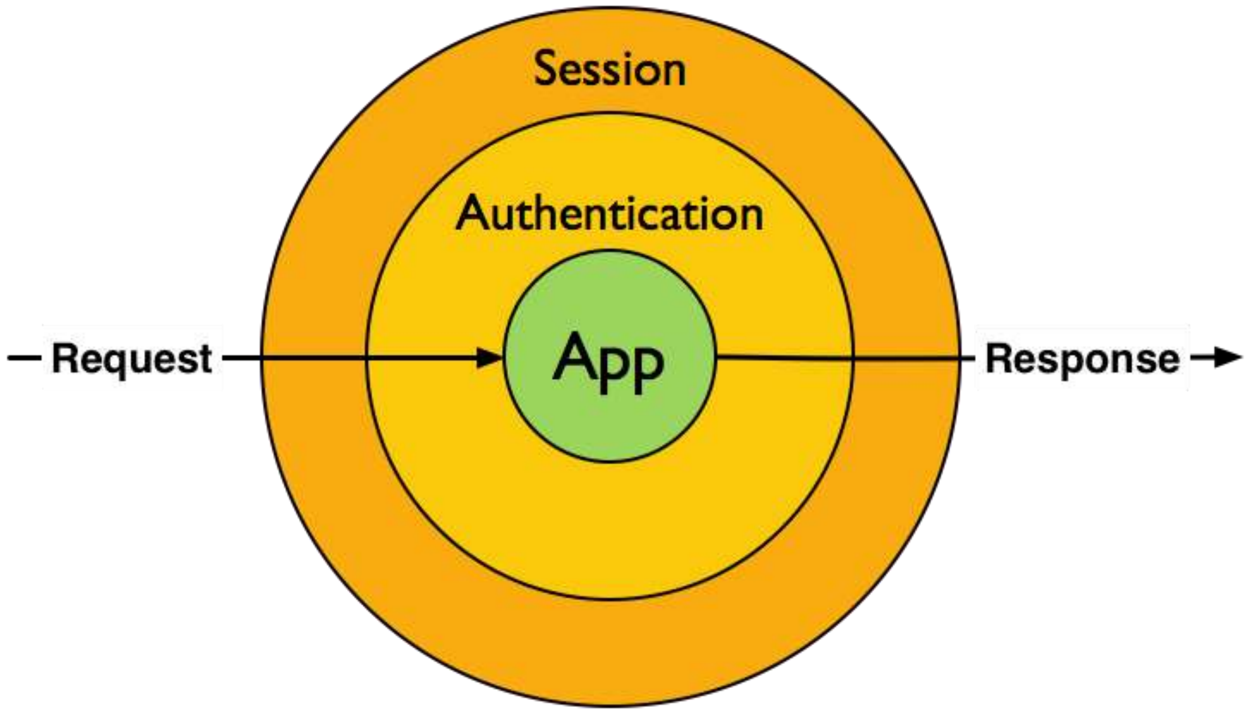
```
Route::get('/user/{name?}',function($name = 'Tops')
{
echo "Name: ".$name; });
```

Middleware

● As the name suggest, Middleware acts as a middle man between request and response.
● Middleware provide a convenient mechanism for filtering HTTP requests ntering your application.
● It is a type of filtering mechanism. For example, Laravel includes a middleware that verifies whether user of the application is authenticated or not.
● If the user is authenticated, he will be redirected to the home page otherwise, he will be redirected to the login page.
● Of course, additional middleware can be written to perform a variety of tasks besides authentication.
● A CORS middleware might be responsible for adding the proper headers to all responses leaving your application.
● All of these middleware are located in the app/Http/Middlewaredirectory.
● **middleware is a series of wrappers around your application that decorate the requests and the responses in a way that isn't a part of your application logic.**

Defining Middleware

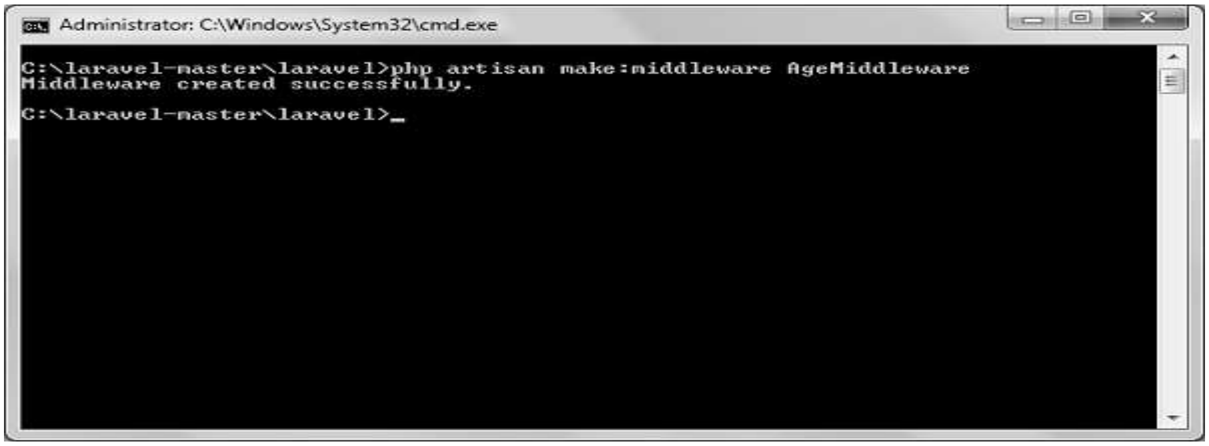
●To create a new middleware, use the make:middleware Artisan command
php artisan make:middleware <middleware-name>



Example :
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



```
C:\laravel-master\laravel>php artisan make:middleware AgeMiddleware
Middleware created successfully.

C:\laravel-master\laravel>_
```

Step 3 − AgeMiddlware will be created atapp/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);
}
}
 ?>
```

Register Middleware

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

- Global Middleware.
- Route Middleware

Global Middleware:

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

Assigning Middleware To Routes

- If you would like to assign middleware to specific routes, you should first assign the

middleware a key in your app/Http/Kernel.php file.
This file contains two properties

- $middleware
- $routeMiddleware.

Assigning Middleware To Routes

- $middleware property is used to register Global Middleware
- $routeMiddleware property is used to register route specific middleware.
- To register the global middleware, list the class at the end of $middleware property.
- By default, the $routeMiddleware property of this class contains entries for the middleware included with Laravel.

Step -4 app/Http/Kernel.php

```
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,

  ];
```

Middleware Parameters

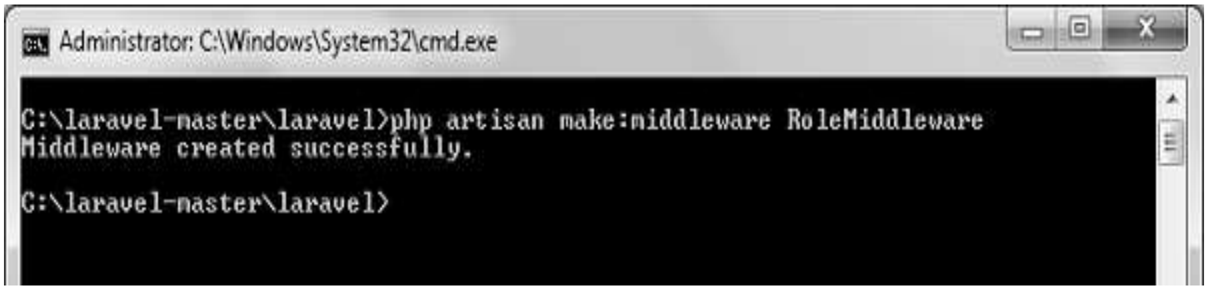
- Middleware can also receive additional parameters.
- For example, if your application needs to verify that the authenticated user has a given "role" before performing a given action, you could create a CheckRole middleware that receives a role name as an additional argument.
- The middleware that we create contains the following function and we can pass our custom argument after the $nextargument.

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

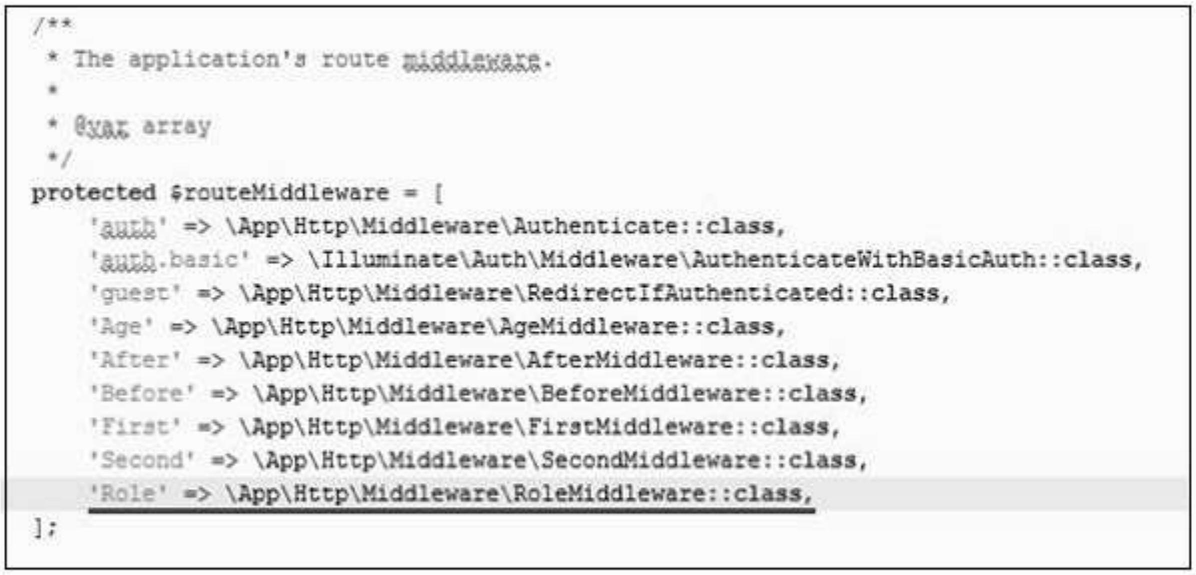


```
C:\laravel-master\laravel>php artisan make:middleware RoleMiddleware
Middleware created successfully.

C:\laravel-master\laravel>
```

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
```

Step 6– Copy the following code to app/Http/TestController.php file.
app/Http/TestController.php

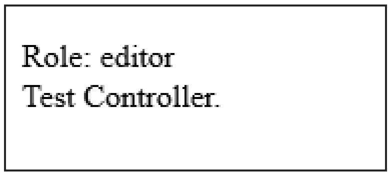
```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Http\Requests;
use App\Http\Controllers\Controller;
```

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

Step 7 – 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 8 – Visit the following URL to test the Middleware with parameters
http://localhost:8000/role
Step 9– The output will appear as shown in the following image.



```
Role: editor
Test Controller.
```

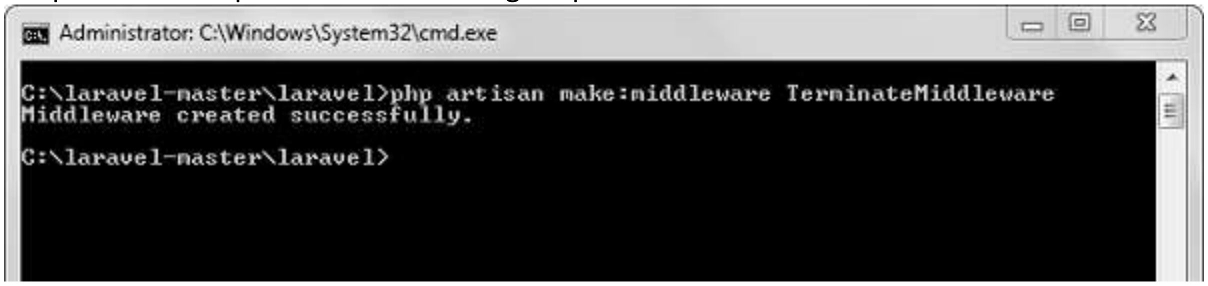
Terminable Middleware

- Terminable middleware performs some task after the response has been sent to the rowser.
- 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.

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

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

Step 2 – This will produce the following output –

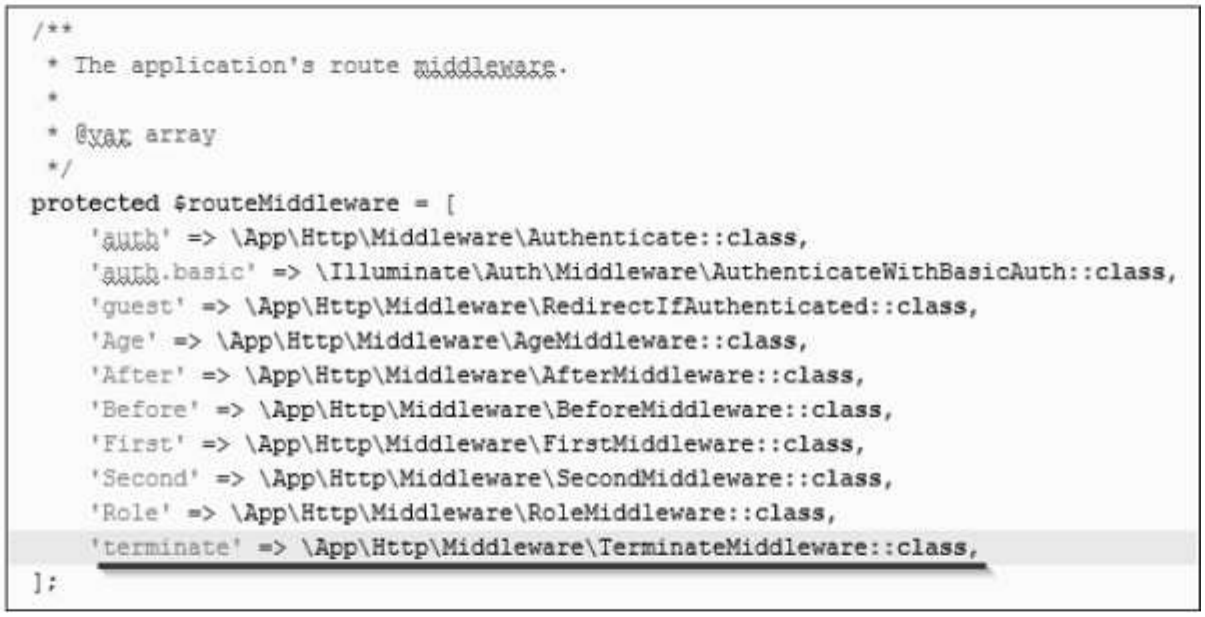


```
C:\laravel-master\laravel>php artisan make:middleware TerminateMiddleware
Middleware created successfully.

C:\laravel-master\laravel>
```

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 TerminateMiddleware.";
    return $next($request);
  }
  public function terminate($request, $response){
    echo "<br>Executing statements of terminate method of TerminateMiddleware.";
  }
}
```

Step4 –Registerthe 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
```

Step 6 – Copy the following code to app/Http/ABCController.php file.
app/Http/ABCController.php

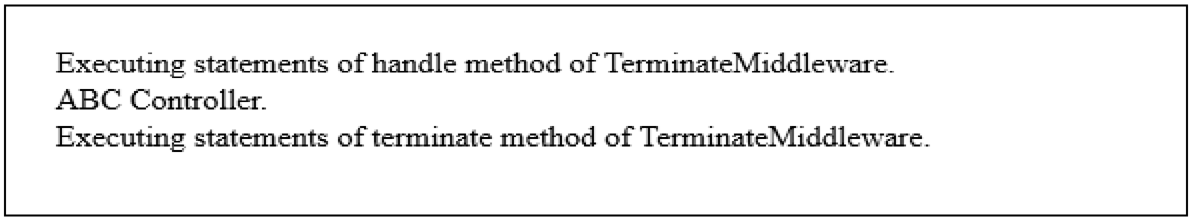
```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Http\Requests;
use App\Http\Controllers\Controller;
```

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

Step 7 – 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 8 – 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.



```
Executing statements of handle method of TerminateMiddleware.
ABC Controller.
Executing statements of terminate method of TerminateMiddleware.
```

Introduction to Controller

- Controllers can group related request handling logic into a single class.
- Controllers are stored in the app/Http/Controllers
- It acts as a directing traffic between Views and Models.

Defining Controllers:

- The controller extends the base controller class included with Laravel.
- 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>
```

Controller Creation

- Replace the <controller-name> with the name of your controller.
- 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.

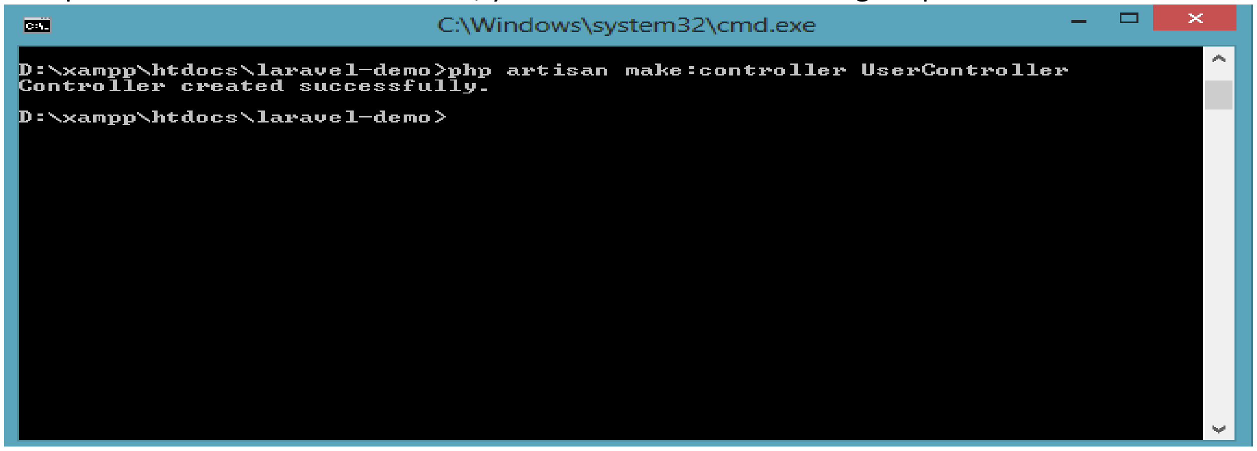
```
Route ::get('base URI' , 'controller@method');
```

Example :

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

```
Php artisan make:controller UserController
```

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



```
D:\xampp\htdocs\laravel-demo>php artisan make:controller UserController
Controller created successfully.

D:\xampp\htdocs\laravel-demo>
```

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;
class UserController extends Controller {
    public function show($id) {
    return view('user.profile', ['user' => User::findOrFail($id)]); }
}
```

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

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

Controllers & Namespaces:

It is very important to note that we did not need to specify the full controller namespace when defining the controller route. If you choose to nest your controllers deeper into the App\Http\Controllers directory, simply use the specific class name relative to the App\Http\Controllers root namespace. So, if your full controller class is App\Http\Controllers\Photos\AdminController, you should register routes to the controller like soe: For example, we want to have a sub-folder app/Http/Controllers/Admin and then inside of it we have our AdminController.php, that's fine. What we need to do inside of the file itself:

1. Correct namespace – specify the inner folder:
   Namespace App\Http\Controllers\Admin; get ('user/{id}', UserController@show');
2. Use Controller – from your inner-namespace Laravel won't "understand" extends Controller, so you need to add this:

```
Use App\Http\Controllers\Controller ;
```

3. Routes – specify full path

```
Route::get ('admin','Admin\AdminController@getHome');
```

Single Action Controllers

If you would like to define a controller that only handles a single action, you may place a single__invoke method on the controller

```
<?php
namespace App\Http\Controllers;
use App\User;
use App\Http\Controllers\Controller;
class ShowProfile extends Controller
{
    public function __invoke($id)
    {
        return view('user.profile', ['user' => User::findOrFail($id)]);
    }
}
```

When registering routes for single action controllers, you do not need to specify a method:

```
Route::get('user/{id}', 'ShowProfile');
```

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.

Example

Step 1 – Add the following lines 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.

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

Step 3 – Add the following code in 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 line.

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

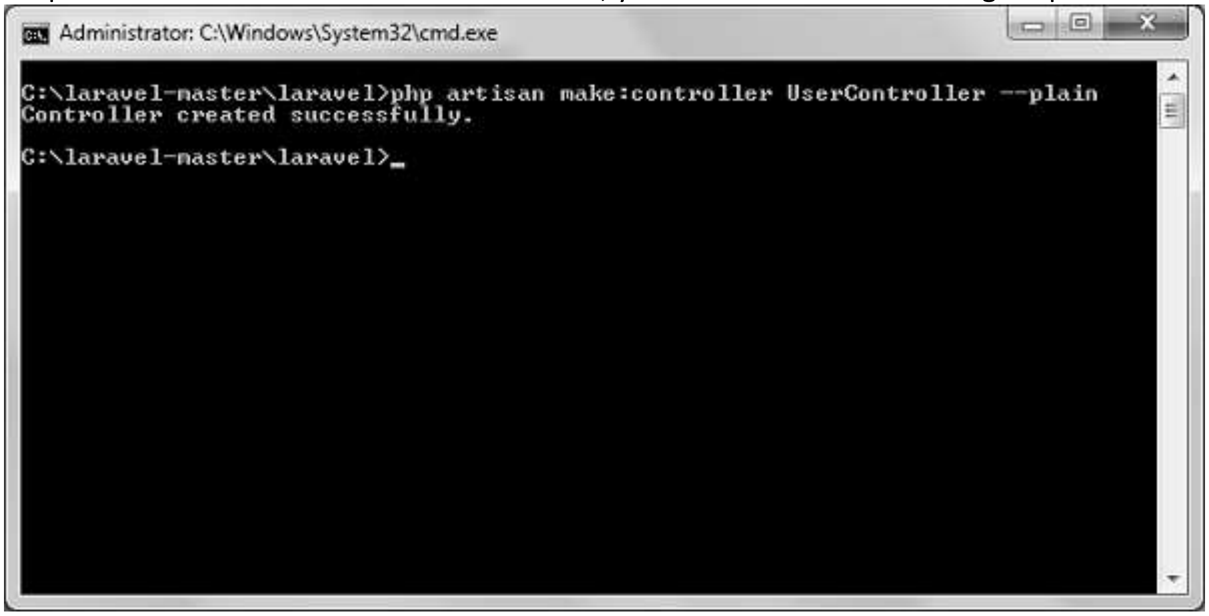
Step 5 – Add the following code in the handle method of the newly created SecondMiddleware at app/Http/Middleware.
SecondMiddleware.php

```
<?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
```

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



Administrator: C:\Windows\System32\cmd.exe

```
C:\laravel-master\laravel>php artisan make:controller UserController --plain
Controller created successfully.

C:\laravel-master\laravel>_
```

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

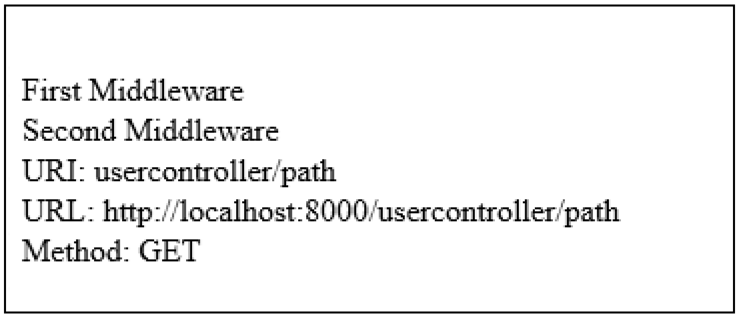
```
<?php
namespace App\Http\Controllers;
```

```
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.



```
First Middleware
Second Middleware
URI: usercontroller/path
URL: http://localhost:8000/usercontroller/path
Method: GET
```

Assigning Middleware within Controller's constructor

Middleware may be assigned to the controller's routes in your route files Assigning Middleware within Controller's constructorExample
Step 1 – Add the following lines 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.

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

Step 3 – Add the following code in 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 line.

```
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

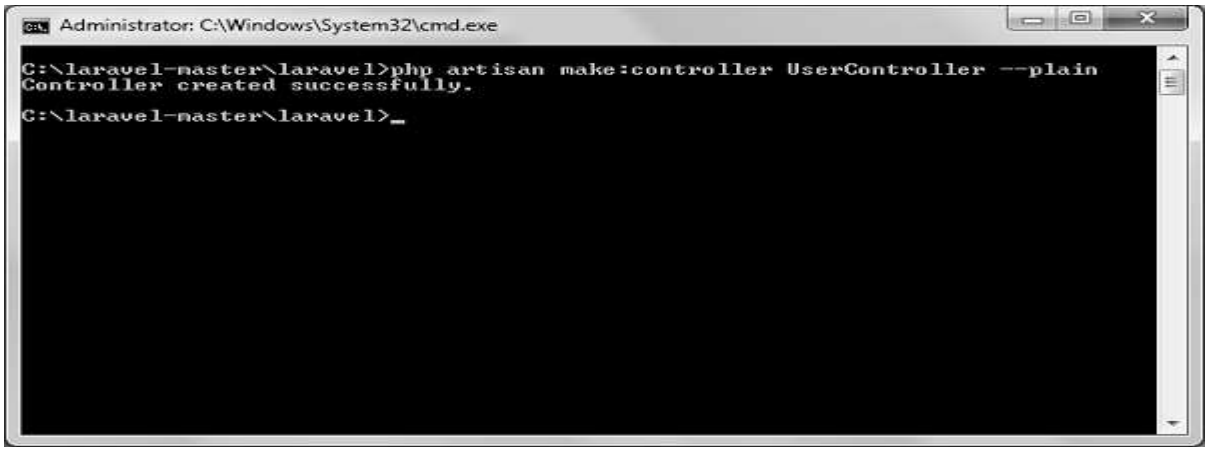
```
<?php
namespace App\Http\Middleware;
use Closure;
```

```
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
```

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


Administrator: C:\Windows\System32\cmd.exe

```
C:\laravel-master\laravel>php artisan make:controller UserController --plain
Controller created successfully.

C:\laravel-master\laravel>_
```

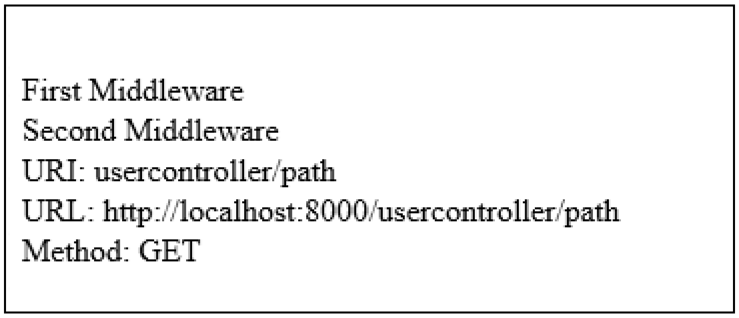
Step 8 – Copy the following code to app/Http/UserController.php file.
app/Http/UserController.php

```
<?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.



```
First Middleware
Second Middleware
URI: usercontroller/path
URL: http://localhost:8000/usercontroller/path
Method: GET
```

Resource Controllers

Laravel resource routing assigns the typical "CRUD" (Create, Read, Update, Delete) routes to a controller with a single line of code. 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.

Verb	Path	Action	Route Name
GET	/my	index	my.index
GET	/my/create	create	my.create
POST	/my	store	my.store
GET	/my/{my}	show	my.show
GET	/my/{my}/edit	edit	my.edit
PUT/PATCH	/my/{my}	update	my.update
DELETE	/my/{my}	destroy	my.destroy

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

URL	Description	Output Image
http://localhost:8000/my	Executes index method of MyController.php	index
http://localhost:8000/my/cre	Executes create method of	create

ate	MyController.php	
http://localhost:8000/my/1	Executes show method of MyController.php	show
http://localhost:8000/my/1/edit	Executes edit method of MyController.php	edit

Partial Resource Routes

When declaring a resource route, you may specify a subset of actions the controller should handle instead of the full set of default actions

```
Route::resource('photo', 'PhotoController', ['only' => [ 'index', 'show' ]]);
Route::resource('photo', 'PhotoController', ['except' => [ 'create', 'store',
'update', 'destroy' ]]);
```

Naming Resource Routes :

By default, all resource controller actions have a route name; however, you can override these names by passing a names array with your options:

```
Route::resource('photo', 'PhotoController', ['names' => [ 'create' =>
'photo.build' ]]);
```

Route Caching

If your application is exclusively using controller based routes, you should take advantage of Laravel's route cache. Using the route cache will drastically decrease the amount of time it takes to register all of your application's routes. In some cases, your route registration may even be up to 100x faster. To generate a route cache, just execute the route: cache Artisan command

```
Php artisan route:cache
```

After running this command, your cached routes file will be loaded on every request. Remember, if you add any new routes you will need to generate a fresh route cache. Because of this, you should only run the route:cache command during your project's deployment.

```
Php artisan route :clear
```

Retrieval

Basic Request Information:

The Illuminate\Http\Request instance provides a variety of methods for examining the HTTP request for your application and extends the Symfony\Component\HttpFoundation\Request class.

- Retrieving The Request URI
- The path method returns the request's URI. So, if the incoming request is targeted at http://domain.com/foo/bar, the path method will
return foo/bar:

```
$uri=$request ->path();
```

The is method allows you to verify that the incoming request URI matches a given pattern, You may use the * character as a wildcard when utilizing this method:

if ($request->is('admin/*')) { // }
To get the full URL, not just the path info, you may use the url or fullUrl methods on the request instance: $url = $request->url();
// With Query String...
$url = $request->fullUrl();
You may also get the full URL and append query parameters. For example, if the request is targeted at http://domain.com/foo, the following method will return http://domain.com/foo?bar=baz $url = $request->fullUrlWithQuery(['bar' => 'baz']);

Retrieval

Retrieving The Request Method:
The method method will return the HTTP verb for the request. You may also use the isMethodmethod to verify that the HTTP verb matches a given string:
$method = $request->method();
if ($request->isMethod('post')) { // }

PSR-7 Requests

The PSR-7 standard specifies interfaces for HTTP messages, including requests and responses. If you would like to obtain an instance of a PSR-7 request, you will first need to install a few libraries.Laravel uses the Symfony HTTP Message Bridge component to convert typical Laravel requests and responses into PSR-7 compatible implementations
composer require symfony/psr-http-message-bridge

```
composer require zendframework/zend-diactoros
```

Once you have installed these libraries, you may obtain a PSR-7 request by simply type-hinting the request type on your route or controller

```
use Psr\Http\Message\ServerRequestInterface;
Route::get('/', function (ServerRequestInterface $request) { // });
```

Retrieving Input

Retrieving An Input Value: Using a few simple methods, you may access all user input from your Illuminate\Http\Request instance.You do not need to worry about the HTTP verb used for the request, as input is accessed in the same way for all verbs:
You may pass a default value as the second argument to the input method. This value will be returned if the requested input value is not present on the request
$name = $request->input('name');
$name = $request->input('name', 'Sally');

When working on forms with array inputs, you may use "dot" notation to access the arrays
Retrieving All Input Data: You may also retrieve all of the input data as an array using the all method

$name = $request->input('products.0.name');
$names = $request->input('products.*.name');
$input = $request->all();
Retrieving A Portion Of The Input Data
If you need to retrieve a sub-set of the input data, you may use the only and except methods. Both of these methods will accept a single array or a dynamic list of arguments
$input = $request->only(['username', 'password']);
$input = $request->only('username', 'password');
$input = $request->except(['credit_card']);
$input = $request->except('credit_card');
Dynamic Properties You may also access user input using dynamic properties on the Illuminate\Http\Request instance. For example, if one of your application's forms contains a name field, you may access the value of the posted field like so When using dynamic properties, Laravel will first look for the parameter's value in the request payload and then in the route parameters.
$name = $request->name;
Old Input
Laravel allows you to keep input from one request during the next request.This feature is particularly useful for re-populating forms after detecting validation errors. However, if you are using Laravel's included validation services, it is unlikely you will need to manually use these methods, as some of Laravel's built-in validation facilities will call them automatically.

Flashing Input To The Session

The flash method on the Illuminate\Http\Request instance will flash the
current input to the session so that it is available during the user's next request to the application You may also use the flashOnly and flashExcept methods to flash a subset of the request data into the session:
$request->flash();
$request->flashOnly(['username', 'email']);
$request->flashExcept('password');

Flash Input Into Session Then Redirect

Since you often will want to flash input in association with a redirect to the previous page, you may easily chain input flashing onto a redirect using the with Input method
return redirect('form')->withInput();
return redirect('form')->withInput($request->except('password'));

Retrieving Old Data

To retrieve flashed input from the previous request, use the old method on the Request instance. The old method provides a convenient helper for pulling the flashed input data out of the session
• Laravel also provides a global old helper function. If you are displaying old input within a Blade template, it is more convenient to use the old helper.
• If no old input exists for the given string,null will be returned
$username = $request->old('username');
<input type="text" name="username" value="{{ old('username') }}">

Upload Files

Retrieving Uploaded Files:You may access uploaded files from a Illuminate\Http\Request instance using the file method or using dynamic properties.The file method returns an instance of theIlluminate\Http\UploadedFile class, which extends the PHP SplFileInfo class and provides a variety of methods for interacting with the file

```
$file = $request->file('photo');
$file = $request->photo;
```

You may determine if a file is present on the request using the hasFile method Validating Successful Uploads: In addition to checking if the file is present, you may verify that there were no problems uploading the file via the isValid method:

```
if ($request->hasFile('photo')) { // }
if ($request->file('photo')->isValid()) { // }
```

File Paths & Extensions: The Uploaded File class also contains methods for accessing the file's fullyqualified path and its extension. The extension method will attempt to guess the file's extension based on its contents. This extension may be different from the extension that was supplied by the client:

```
$path = $request->photo->path();
$extension = $request->photo->extension();
```

Storing Uploaded Files: To store an uploaded file, you will typically use one of your configured file systems. Then Uploaded File class has a store method which will move an uploaded file to one of your disks, which may be a location on your local filesystem or even a cloud storage location.

The store method accepts the path where the file should be stored relative to the filesystem's configured root directory.

This path should not contain a file name, since the name will automatically be generated using the MD5 hash of the file's contents.

The store method also accepts an optional second argument for the name of the disk that should be used to store the file. The method will return the path of the file relative to the disk's root:

If you do not want a file name to be automatically generated, you may use the storeAs method, which accepts the path, file name, and disk name as its arguments

```
$path = $request->photo->store('images');
$path = $request->photo->store('images', 's3');
$path = $request->photo->storeAs('images', 'filename.jpg');
$path = $request->photo->storeAs('images', 'filename.jpg', 's3');
```

Example

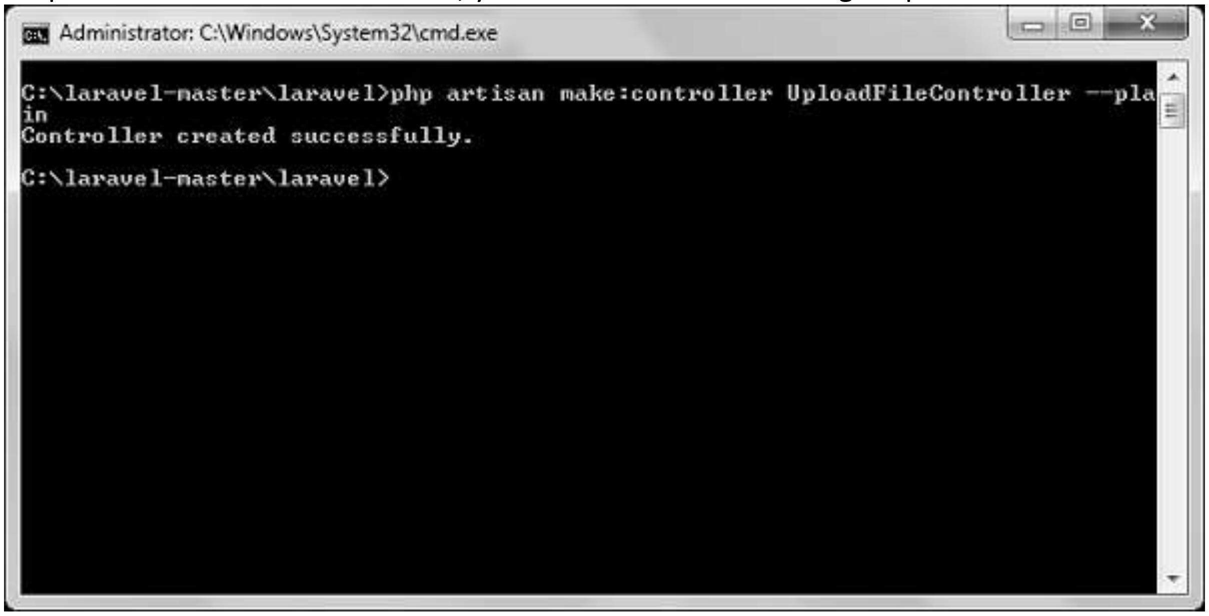
Step 1 – Create a view file called resources/views/uploadfile.php and copy the following code in that file. resources/views/uploadfile.php

```
<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
```

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



```
C:\laravel-master\laravel>php artisan make:controller UploadFileController --pla
in
Controller created successfully.

C:\laravel-master\laravel>
```

Step 4 – Copy the following code in app/Http/Controllers/UploadFileController.php file.

app/Http/Controllers/UploadFileController.php

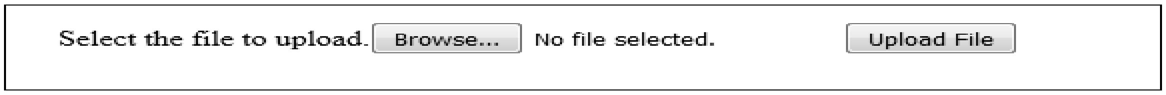
```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Http\Requests;
use App\Http\Controllers\Controller;
class UploadFileController extends Controller {
   public function index(){
      return view('uploadfile');
   }
   public function showUploadFile(Request $request){
      $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.


Select the file to upload. Browse... No file selected. Upload File

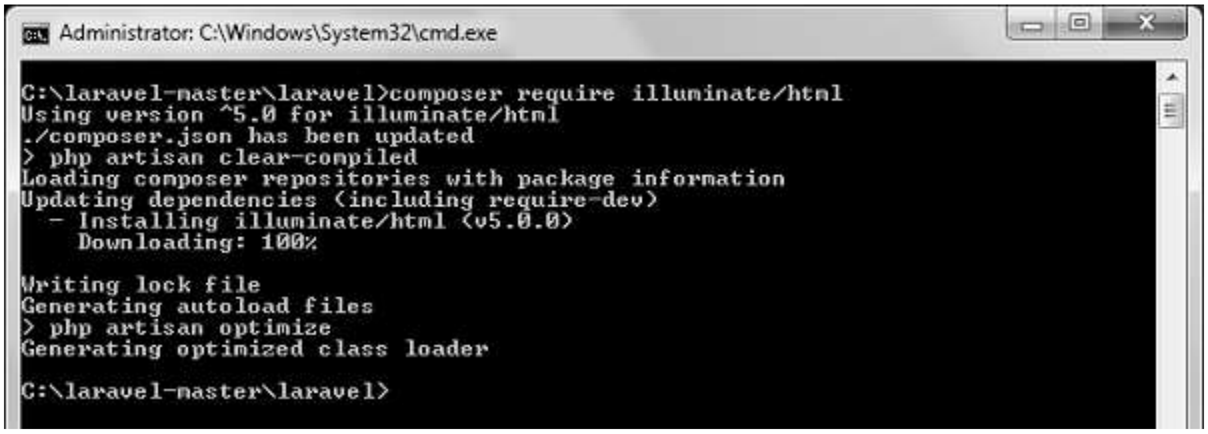
Forms & HTML

- 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.

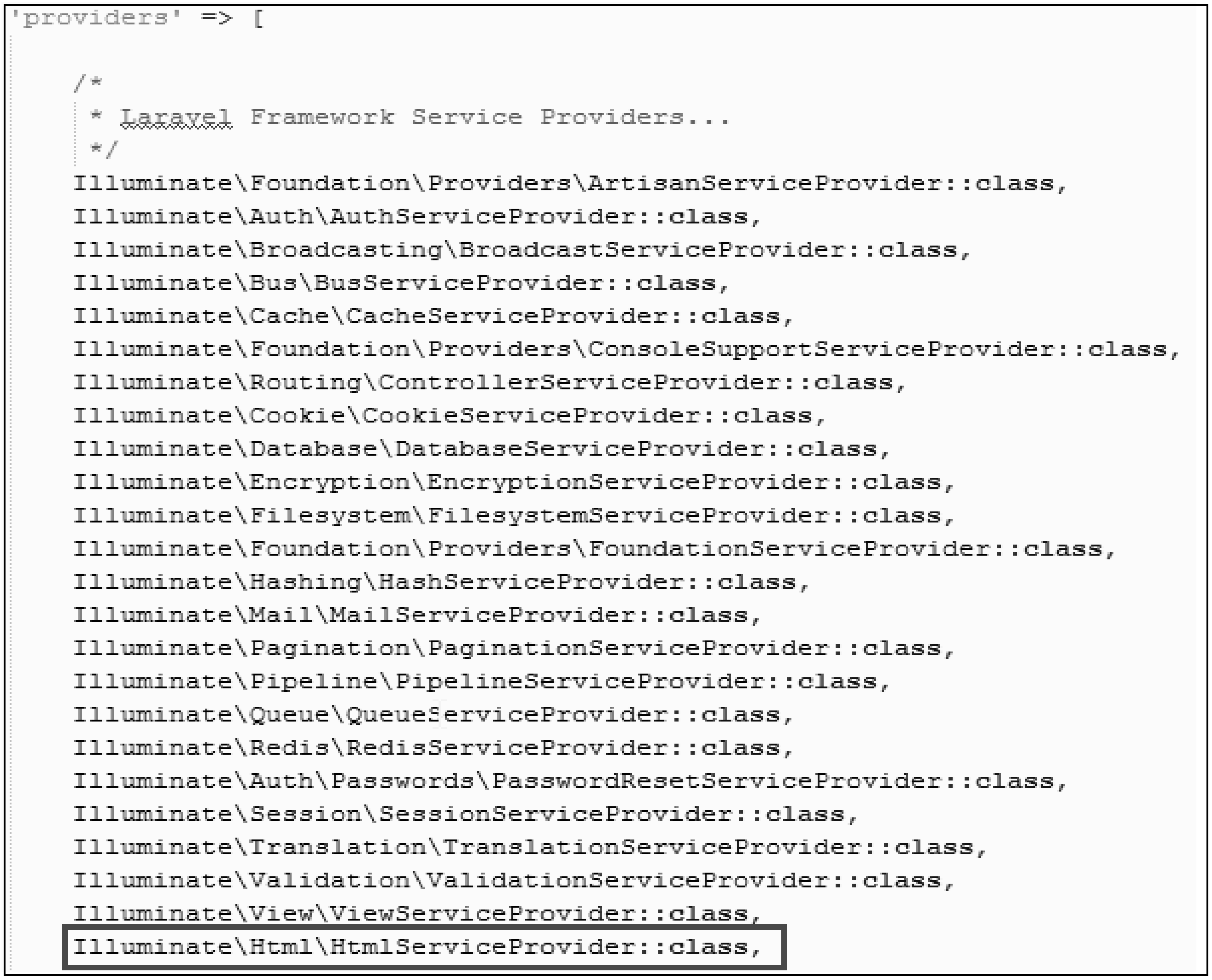
```
composer require illuminate/html
```

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



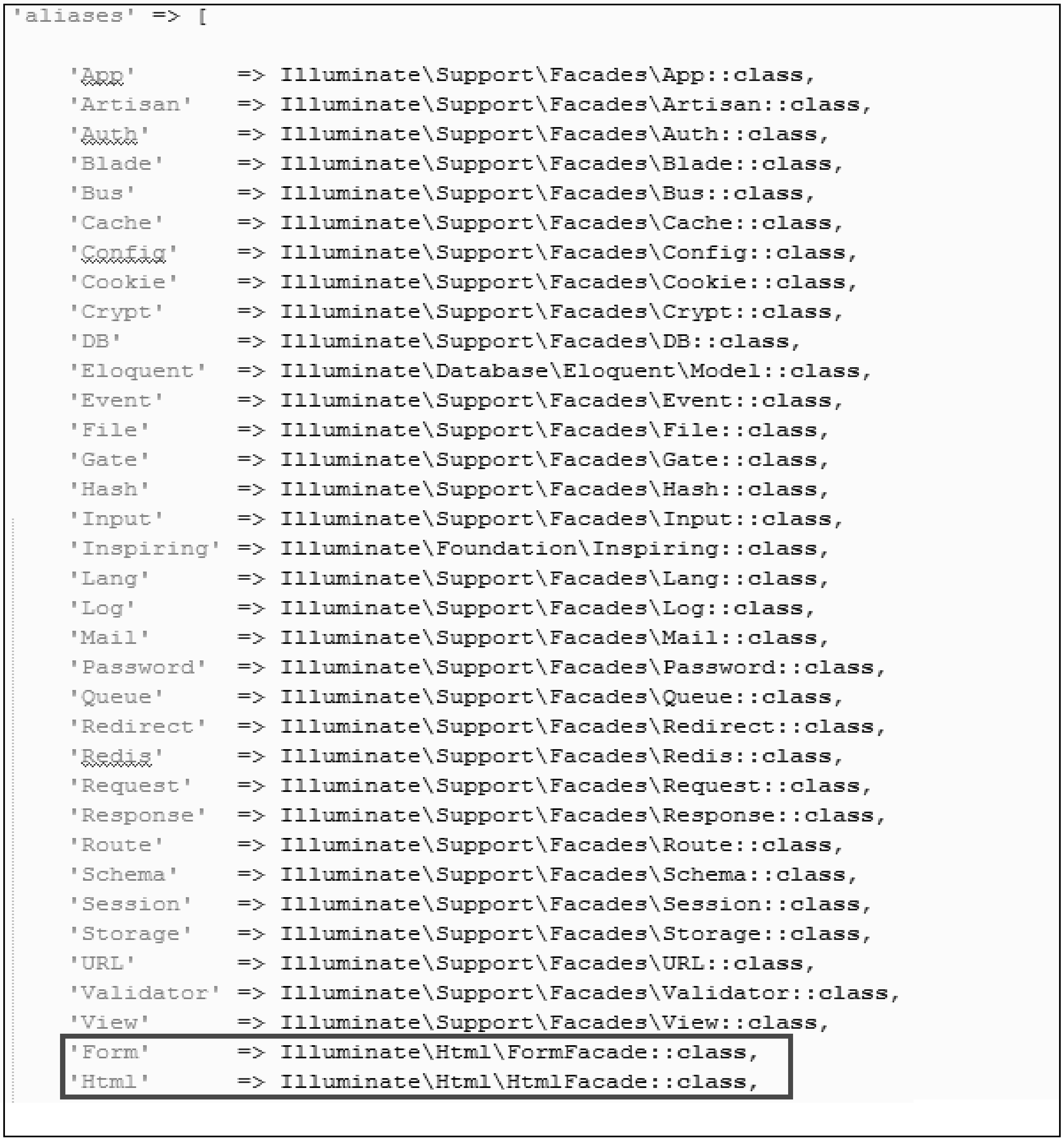
```
C:\laravel-master\laravel>composer require illuminate/html
Using version ^5.0 for illuminate/html
./composer.json has been updated
> php artisan clear-compiled
Loading composer repositories with package information
Updating dependencies (including require-dev)
  - Installing illuminate/html (v5.0.0)
    Downloading: 100%

Writing lock file
Generating autoload files
> php artisan optimize
Generating optimized class loader

C:\laravel-master\laravel>
```

Step 3 – Now, we need to add this package 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

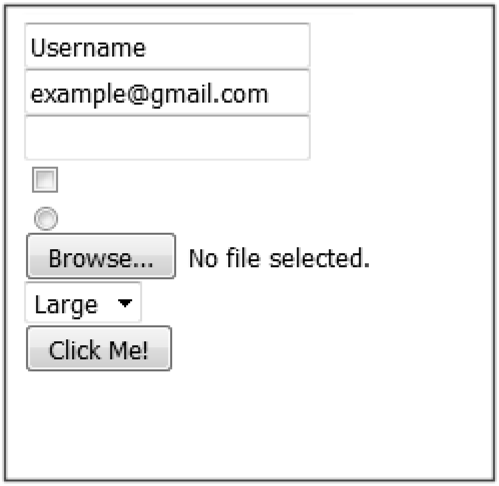
```
<html>   <body>
            <?php
         echo Form::open(array('url' => 'foo/bar'));
            echo Form::text('username','Username');
            echo '<br/>';
                echo Form::text('email', 'example@gmail.com');
            echo '<br/>';
           echo Form::password('password');
            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' => 'Small'));
        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.



Custom Macros :

Registering A Form Macro
It's easy to define your own custom Form class helpers called "macros". Here's how it works.
First, simply register the macro with a given name and a Closure:
Form::macro('myField', function() { return '<input type="awesome">'; });
Now you can call your macro using its name:

```
echo Form::myField();
```

Security

Security is important feature while designing web applications. It assures the users of the website that their data is secured.Laravel provides various mechanisms to secure website.

Some of the features are listed below Storing Passwords – Laravel provides a class called DzHashdz class which provides secure Bcrypt hashing. The password can be hashed in the following way.

```
$password = Hash::make('secret');
```

make() function will take a value as argument and will return the hashed value. The hashed value can be checked using the check()function in the following way.

```
Hash::check('secret', $hashedPassword)
```

The above function will return Boolean value. It will return true if password matched or false otherwise.

• Authenticating Users – The other main security features in Laravel is authenticating user and perform some action. Laravel has made this task easier and to do this we can use Auth::attempt method in the following
way.

```
if (Auth::attempt(array('email' => $email, 'password' => $password)))
{ return Redirect::intended('home'); }
```

Security

The Auth::attempt method will take credentials as argument and will verify those credentials against the credentials stored in database and will return true if it is matched or false otherwise. CSRF Protection/Cross-site request forgery (XSS) – Cross-site scripting (XSS) attacks happen when attackers are able to place client-side JavaScript code in a page viewed by other users. To avoid this kind of attack, you should never trust any user-submitted data or escape any dangerous characters. You should favor the double-brace syntax ({{ $value }}) in your Blade templates, and only use the {!! $value !!} syntax, where you're certain the data is safe to display in its raw format.

Avoiding SQL injection – SQL injection vulnerability exists when an application inserts arbitrary and unfiltered user input in an SQL query. By default, Laravel will protect you against this type of attack since both the query builder and Eloquent use PHP Data Objects (PDO) class behind the scenes. PDO uses prepared statements, which allows you to safely pass any parameters without having to escape and sanitize them.

Cookies – Secure by default – Laravel makes it very easy to create, read, and expire cookies with its Cookie class. In Laravel all cookies are automatically signed and encrypted. This means that if they are tampered with, Laravel will automatically discard them. This also means that you will not be able to read them from the client side using JavaScript.

Forcing HTTPS when exchanging sensitive data – (TTPS prevents attackers on the same network to intercept private information such as session variables, and log in as the victim.

Blade Templates

Blade is the simple, yet powerful templating engine provided with Laravel. all Blade views are compiled into plain PHP code and cached until they are modified, meaning Blade adds essentially zero overhead to your application.

Blade view files use the .blade.php file extension and are typically stored in the esources/views directory.

Template Inheritance

Defining A Layout

Two of the primary benefits of using Blade are template inheritance and sections. To get started, let's take a look at a simple example. First, we will examine a "master" page layout. Since most web applications maintain the same general layout across various pages, it's convenient to define this layout as a single Blade view Store Layout in resources/views/layouts/app.blade.php

```
<html>
<head>
<title>
App Name - @yield('title')
</title>
</head>
<body>
@section('sidebar')
This is the master sidebar.
@show
<div class="container">
@yield('content')
</div>
</body>
</html>
```

• The @section directive, as the name implies, defines a section of content,
• the @yield directive is used to display the contents of a given section.

Extending A Layout

When defining a child view, use the Blade @extends directive to specify which layout the child view should "inherit". Views which extend a Blade layout may inject content into the layout's sections using @section directives.Remember, as seen in the example above, the contents of these sections will be displayed in the layout using @yield

Stored in resources/views/child.blade.php

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

• the sidebar section is utilizing the @parent directive to append (rather than overwriting) content to the layout's sidebar.

• The @parent directive will be replaced by the content of the layout when the view is rendered.
• Blade views may be returned from routes using the global view helper:

```
Route::get('blade', function () { return view('child'); });
```

Displaying Data :

You may display data passed to your Blade views by wrapping the variable in curly braces. For example, given the following route

```
Route::get('greeting', function () { return view('welcome', ['name' =>nidhi]); });
```

• You may display the contents of the name variable like
Hello, {{ $name }}
• Of course, you are not limited to displaying the contents of the variables passed to the view.
• You may also echo the results of any PHP function.
• In fact, you can put any PHP code you wish inside of a Blade echo statement

```
The current UNIX timestamp is {{ time() }}.
```

Echoing Data If It Exists

Sometimes you may wish to echo a variable, but you aren't sure if the variable has been set.We can express this in verbose PHP code like so
{{ isset($name) ? $name : 'Default' }}
However, instead of writing a ternary statement, Blade provides you with the following convenient short-cut, which will be compiled to the ternary statement above:

{{ $name or 'Default' }}

 In this example, if the $name variable exists, its value will be displayed. However, if it does not exist, the word Default will be displayed.

Displaying Unescaped Data

By default, Blade {{ }} statements are automatically sent through PHP's html entities function to prevent XSS attacks. If you do not want your data to be escaped, you may use the following syntax

```
Hello, {!! $name !!}.
```

Blade & JavaScript Frameworks

The @ symbol will be removed by Blade; however, {{ name }} expression will remain untouched by the Blade engine, allowing it to instead be rendered by your JavaScript framework.
<h1>Laravel</h1> Hello, @{{ name }}.

The @verbatim Directive

If you are displaying JavaScript variables in a large portion of your template, you may wrap the HTML in the @verbatim directive so that you do not have to prefix each Blade echo statement with an @ symbol

```
@verbatim
<div class="container"> Hello, {{ name }}. </div>
@endverbatim
```

Converting HTML 5 Template to Laravel Blade Templates

Let’s start with downloading the Admin template Unzip the contents of the download file.We will now create a new Laravel 5 project.

```
cd D:\xampp\htdocs
composer create-project laravel/laravel laradmin 5.2
composer create-project laravel/laravel Laravel-demo
```

Let’s now create the blade template files. We will adopt a modular approach to organization our template files. The following images shows the directories and files that we will create

E_shop_transactions – contains the template files for all items under the E-Shop Transactions panel in the admin panel

customers.blade.php : Lists all registered customers

orders.blade.php: Lists all customer orders

products.blade.php: Lists all ordered products

Frontend_data_entry - contains the template files for all items under the Frontend Data Entry panel in the admin panel

blog_posts.blade.php lists all the blog posts and modal form for creating and updating blog posts

pages.blade.php lists all the pages and modal form for creating and updating pages

Layouts – contains all master layout files.

layout.blade.php – contains code that is common to all pages i.e. headers, navigation, footer etc.

System – contains all templates files for items under the System Panel in the admin panel

database_backup.blade.php – displays the form for making database backups

export_to_csv.blade.php – provides options for exporting database records to the comma separated files (CSV)

import_from_csv.blade.php – provides options importing CSV file records into the database.

system_users.blade.php – provides options for managing users in the system.

The layout.blade.php template will have the following common format

```
<html>
<head>
<title>@yield('title')</title>
</head>
<body>
@section('sidebar')
This is the master sidebar.
@show
```

```
<div class="container">
@yield('content')
</div>
</body>
</html>
```

The rest of all the templates will have the following common format

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

Admin Panel Controllers :

Just like we did with the templates, we will also adopt a modular approach with our controllers for better file organization. We will create the following controllers

Admin Panel Routes

Route::group(['middleware' => ['web'], 'prefix' => 'admin'], function () {
…} groups the routes, applies the web middlewareand sets the admin prefix. The prefix will be automatically appended to the beginning of every route defined inside the group.

 Route::resourceî…Ò the resource route maps the HTTP verbs to the appropriate methods in the respective controllers.

Testing the waters

```
cd C:\xampp\htdocs\laradmin
```

Run the following command

```
php artisan serve
```

Load the following URL in your web browser

```
http://localhost:8000/admin/brands
```

Control Structures

In addition to template inheritance and displaying data, Blade also provides convenient short-cuts for common PHP control structures, such as conditional statements and loops.
These short-cuts provide a very clean, terse way of working with PHP control structures, while also remaining familiar to their PHP counterparts.

If Statements

You may construct if statements using the @if, @elseif, @else, and @endif directives. These directives function identically to their PHP counterparts:

Control Structures

```
@if (count($records) === 1)
I have one record!
@elseif (count($records) > 1)
I have multiple records!
@else
I don't have any records!
@endif
```

For convenience, Blade also provides an @unless directive

```
@unless (Auth::check())
You are not signed in.
@endunless
```

Loops

In addition to conditional statements, Blade provides simple directives for working with PHP's loop structures.Again, each of these directives functions identically to their PHP counterparts

```
@for ($i = 0; $i < 10; $i++)
The current value is {{ $i }}
@endfor
@foreach ($users as $user)
<p>This is user {{ $user->id }}</p>
@endforeach
@forelse ($users as $user)
<li>{{ $user->name }}</li>
@empty <p>No users</p>
@endforelse
@while (true)
<p>I'm looping forever.</p>
@endwhile
```

When using loops you may also end the loop or skip the current iteration

```
@foreach ($users as $user)
@if ($user->type == 1)
@continue
@endif
<li>{{ $user->name }}</li>
@if ($user->number == 5)
@break
@endif
@endforeach
```

You may also include the condition with the directive declaration in one line

```
@foreach ($users as $user)
@continue($user->type == 1)
<li>{{ $user->name }}</li>
@break($user->number == 5)
@endforeach
```

The Loop Variable

When looping, a $loop variable will be available inside of your loop. This variable provides access to some useful bits of information such as the current loop index and whether this is the first or last iteration through the loop

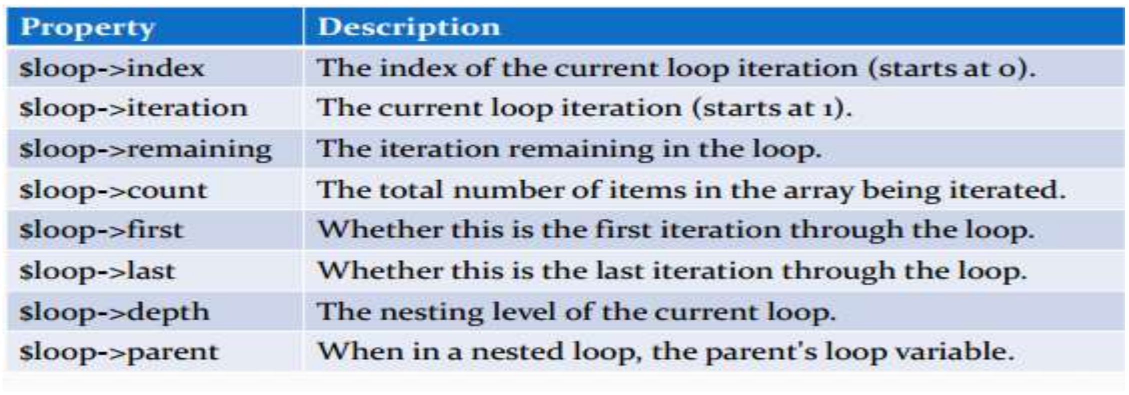
```
@foreach ($users as $user)
@if ($loop->first)
This is the first iteration.
@endif
@if ($loop->last)
This is the last iteration.
@endif
<p>This is user {{ $user->id }}</p>
@endforeach
```

The Loop Variable

If you are in a nested loop, you may access the parent loop's $loop variable via the parent property

```
@foreach ($users as $user)
@foreach ($user->posts as $post)
@if ($loop->parent->first)
This is first iteration of the parent loop.
@endif
@endforeach
@endforeach
```

The $loop variable also contains a variety of other useful properties



Property	Description
$loop->index	The index of the current loop iteration (starts at 0).
$loop->iteration	The current loop iteration (starts at 1).
$loop->remaining	The iteration remaining in the loop.
$loop->count	The total number of items in the array being iterated.
$loop->first	Whether this is the first iteration through the loop.
$loop->last	Whether this is the last iteration through the loop.
$loop->depth	The nesting level of the current loop.
$loop->parent	When in a nested loop, the parent's loop variable.

Comments

Blade also allows you to define comments in your views. unlike HTML comments, Blade comments are not included in the HTML returned by your application
{{-- This comment will not be present in the rendered HTML --}}

Database

Laravel makes interacting with databases extremely simple across a variety of database backends using either raw SQL, the fluent query builder, and the Eloquent ORM. Currently, Laravel supports four databases:

- MySQL
- Postgres
- SQLite
- SQL Server

Configuration

The database configuration for your application is located at config/database.php.
In this file you may define all of your database connections, as well as specify which connection should be used by default.
By default, Laravel's sample environment configuration is ready to use with Laravel Homestead, which is a convenient virtual machine for doing Laravel development on your local machine.
Of course, you are free to modify this configuration as
needed for your local database. Laravel supports SQL Server out of the box; however, you will need to add the connection configuration for the database to your config/database.php configuration file:

```
'mysql' => [
        'driver' => 'mysql',
        'host' => env('DB_HOST', 'localhost'),
        'port' => env('DB_PORT', '3306'),
        'database' => env('DB_DATABASE', 'abcd_laravel'),
        'username' => env('DB_USERNAME', 'root'),
        'password' => env('DB_PASSWORD', ''),
        'charset' => 'utf8',
        'collation' => 'utf8_unicode_ci',
        'prefix' => '',
        'strict' => false,
        'engine' => null,
    ],
```

SQLite Configuration

After creating a new SQLite database using a command such as touch atabase/database.sqlite, you can easily configure your environment variables to point to this newly created database by using the database's absolute path

```
 'sqlite' => [

        'driver' => 'sqlite',
        'database' => env('DB_DATABASE', database_path('database.sqlite')),

        'prefix' => '',

    ],
```

Read & Write Connections

Sometimes you may wish to use one database connection for SELECT statements, and another for INSERT, UPDATE, and DELETE statements.

Laravel makes this a breeze, and the proper connections will always be used whether you are using raw queries, the query builder, or the Eloquent ORM.

To see how read / write connections should be configured, let's look at this example:

```
'mysql' =>
[
'read' => [ 'host' => '192.168.1.1', ],
'write' => [ 'host' => '196.168.1.2' ],
'driver' => 'mysql',
'database' => 'database',
'username' => 'root',
'password' => '',
'charset' => 'utf8',
'collation' => 'utf8_unicode_ci',
'prefix' => '', ],
```

Note that two keys have been added to the configuration array: read and write. Both of these keys have array values containing a single key: host. The rest of the database options for the read and write connections will be merged from the
main mysql array

You only need to place items in the read and write arrays if you wish to override the values from the main array. So, in this case, 192.168.1.1 will be used as the host for the "read" connection, while 192.168.1.2 will be used for the "write" connection. The database credentials, prefix, character set, and all other options in the main mysql array will be shared across both connections.

Using Multiple Database Connections

When using multiple connections, you may access each connection via the connection method on the DB facade. The name passed to the connection method should correspond to one of the connections listed in your config/database.php configuration file You may also access the raw, underlying PDO instanceusing the getPdo method on a connection instance

Running Raw SQL Queries

Once you have configured your database connection, you may run queries using the DB facade. The DB facade provides methods for each type of query: select, update, insert, delete, and statement.

Running A Select Query

To run a basic query, you may use the select method on the DB facade:

```
<?php
namespace App\Http\Controllers;
use Illuminate\Support\Facades\DB;
use App\Http\Controllers\Controller;
class UserController extends Controller
{ /** * Show a list of all of the application's users. * * @return Response */
public function index()
{
$users = DB::select('select * from users where active = ?', [1]);
return view('user.index', ['users' => $users]);
}
}
```

The first argument passed to the select method is the raw SQL query, while the second argument is any parameter bindings that need to be bound to the query. Typically, these are the values of the where clause constraints. Parameter binding provides protection against SQL injection. The select method will always return an array of results.Each result within the array will be a PHP StdClass object, allowing you to access the values of the results

```
foreach ($users as $user) { echo $user->name; }
```

Running Query

Using Named Bindings

Instead of using ? to represent your parameter bindings, you may execute a query using named bindings:

```
$results = DB::select('select * from users where id = :id', ['id' => 1]);
```

Running An Insert Statement

To execute an insert statement, you may use the insert method on the DB facade. Like select, this method takes the raw SQL query as its first argument and bindings as its second argument:

DB::insert('insert into users (id, name) values (?, ?)', [1, Nidhi ']);

Running An Update Statement

The update method should be used to update existing records in the database. The number of rows affected by the statement will be returned:

$affected = DB::update('update users set votes = 100 where name = ?',
[Nidhi']);

Running A Delete Statement

The delete method should be used to delete records from the database. Like update, the number of rows affected will be returned:

```
$deleted = DB::delete('delete from users');
```

Running A General Statement

Some database statements do not return any value. For these types of operations, you may use the statement method on the DB facade:

DB::statement('drop table users');

Listening For Query Events

If you would like to receive each SQL query executed by your application, you may use the listen method.

This method is useful for logging queries or debugging. You may register your query listener in a service provider:

```
<?php
namespace App\Providers;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\ServiceProvider;
class AppServiceProvider extends ServiceProvider
{
/** * Bootstrap any application services. *
* @return void */
public function boot()
{
DB::listen(function ($query)
{
// $query->sql
// $query->bindings
// $query->time });
}
/**
* Register the service provider. *
* @return void */
public function register()
{
//
}
}
```

Database Transactions:

You may use the transaction method on the DB facade to run a set of operations within a database transaction. If an exception is thrown within the transaction Closure, the transaction will automatically be rolled back. If the Closure executes successfully, the transaction will automatically be committed. You don't need to worry about manually rolling back or committing while using the transaction method

```
DB::transaction(function ()
{ DB::table('users')->update(['votes' => 1]);
DB::table('posts')->delete(); });
```

Manually Using Transactions :

If you would like to begin a transaction manually and have complete control over rollbacks and commits, you may use the beginTransaction method on the DB facade

```
DB::beginTransaction();
```

You can rollback the transaction via the rollBack method:

```
DB::rollBack();
```

Lastly, you can commit a transaction via the commit method:

```
DB::commit();
```

Eloquent Queries :

The Eloquent ORM included with Laravel provides a beautiful, simple Active Record implementation for working with your database. Each database table has a corresponding "Model" which is used to interact with that table. Models allow you to query for data in your tables, as well as insert new records into the table. Before getting started, be sure to configure a database connection in config/database.php.

Defining Models : To get started, let's create an Eloquent model

Models typically live in the app directory, but you are free to place them anywhere that can be auto-loaded according to your composer.json file All Eloquent models extend Illuminate\Database\Eloquent\Model class.

The easiest way to create a model instance is using the make: model Artisan command:

```
php artisan make: model User
```

If you would like to generate a database migration when you generate the model, you may use the --migration or -m option:

php artisan make:model User --migration

php artisan make:model User –m

Eloquent Model Conventions

Now, let's look at an example Flight model, which we will use to retrieve and store information from our flights database table

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Flight extends Model
{ // }
```

Table Names : Note that we did not tell Eloquent which table to use for our Flight model. By convention, the "snake case", plural name of the class will be used as the table name unless another name is explicitly specified. So, in this case, Eloquent will assume the Flight model stores records in the flights table. You may specify a custom table by defining a table property on your model:

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Flight extends Model
{ /** * The table associated with the model. * * @var string */
protected $table = 'my_flights';
}
```

Primary Keys:

Eloquent will also assume that each table has a primary key column named id You may define a $primaryKey property to override this convention. Eloquent assumes that the primary key is an incrementing integer value, which means that by default the primary key will be cast to an int automatically. If you wish to use a non-incrementing or a non-numeric primary key you must set the public $incrementing property on your model to false.

Timestamps : Eloquent expects created_at and updated_at columns to exist on your tables. If you do not wish to have these columns automatically managed by Eloquent, set the $timestamps property on your model to false

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Flight extends Model
{ /** * Indicates if the model should be timestamped. * * @var bool */
public $timestamps = false;
}
```

If you need to customize the format of your timestamps, set the $dateFormat property on your model.

Timestamps : This property determines how date attributes are stored in the database, as well as their format when the model is serialized to an array or JSON

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Flight extends Model { /** * The storage format of the model's date columns. * *
@var string */
protected $dateFormat
```

If you need to customize the names of the columns used to store the timestamps, you may set the CREATED_AT and UPDATED_AT constants in your model:

```
<?php
class Flight extends Model
{
const CREATED_AT = 'creation_date';
const UPDATED_AT = 'last_update';
}
```

Database Connection

By default, all Eloquent models will use the default database connection configured for your application.

If you would like to specify a different connection for the model, use the $connectionproperty

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Flight extends Model
{ /**
•The connection name for the model. *
•* @var string */
•protected $connection = 'connection-name'; }
```

Retrieving Models

Once you have created a model and its associated database table, you are ready to start retrieving data from your database. Think of each Eloquent model as a powerful query builder allowing you to fluently query the database table associated with the model. For example

```
<?php
use App\Flight;
$flights = App\Flight::all();
foreach ($flights as $flight)
{ echo $flight->name; }
```

Adding Additional Constraints :

The Eloquent all method will return all of the results in the model's table. Since each Eloquent model serves as a query builder, you may also add constraints to queries, and then use `the getmethod to retrieve the results

```
  flights = App\Flight::where('active', 1)
->orderBy('name', 'desc')
->take(10)
->get();
```

Collections:

For Eloquent methods like all and get which retrieve multiple results, an instance of Illuminate\Database\Eloquent\Collection will be returned. The Collection class provides a variety of helpful methods for working with your eloquent results:

```
$flights = $flights->reject(function ($flight)
{ return $flight->cancelled; });
```

Of course, you may also simply loop over the collection like an array:

```
foreach ($flights as $flight) { echo $flight->name; }
```

Chunking Results

If you need to process thousands of Eloquent records, use the chunk command. The chunk method will retrieve a "chunk" of eloquent models, feeding them to a given Closure for processing. Using the chunk method will conserve memory when working with large result sets:

```
Flight::chunk(200, function ($flights)
{ foreach ($flights as $flight) { // } });
```

The first argument passed to the method is the number of records you wish to receive per "chunk". The Closure passed as the second argument will be called for each chunk that is retrieved from the database. A database query will be executed to retrieve each chunk of records passed to the Closure.

Using Cursors

The cursor method allows you to iterate through your database records using a cursor, which will only execute a single query. When processing large amounts of data, the cursor method may be used to greatly reduce your memory usage:

```
foreach (Flight::where('foo', 'bar')->cursor() as $flight) { // }
```

Retrieving Single Models /Aggregates

Of course, in addition to retrieving all of the records for a given table, you may also retrieve single records using find and first. Instead of returning a collection of models, these methods return a single model instance:

```
$flight = App\Flight::find(1);
// Retrieve the first model matching the query constraints...
$flight = App\Flight::where('active', 1)->first();
```

You may also call the find method with an array of primary keys, which will return a collection of the matching records:

```
$flights = App\Flight::find([1, 2, 3]);
```

Retrieving Single Models / Aggregates

Not Found Exceptions : Sometimes you may wish to throw an exception if a model is not found.

- This is particularly useful in routes or controllers.

- The findOrFail and firstOrFail methods will retrieve the first result of the query; however, if no result is found, a Illuminate\Database\Eloquent\ModelNotFoundException will be thrown:

```
        $model = App\Flight::findOrFail(1);
$model = App\Flight::where('legs', '>', 100)->firstOrFail();
```

If the exception is not caught, a 404 HTTP response is automaticallysent back to the user. It is not necessary to write explicit checks to return 404 responses when using these methods:

```
Route::get('/api/flights/{id}', function ($id) { return
App\Flight::findOrFail($id); });
```

Retrieving Single Models / Aggregates

Retrieving Aggregates: You may also use the count, sum, max, and other aggregate methods provided by the query builder. These methods return the appropriate scalar value instead of a full model instance:

```
$count = App\Flight::where('active', 1)->count();
$max = App\Flight::where('active', 1)->max('price');
```

Inserting Models

To create a new record in the database, simply create a new model instance, set attributes on the model, then call the save method:

```
namespace App\Http\Controllers;
use App\Flight;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;
class FlightController extends Controller
{
public function store(Request $request)
{ // Validate the request... $flight = new Flight;
$flight->name = $request->name;
$flight->save();
} }
```

In this example, we simply assign the name parameter from the incoming HTTP request to the name attribute of the App\Flight model instance. When we call the save method, a record will be inserted into the database. The created at and updated at timestamps will automatically be set when the save method is called, so there is no need to set them manually.

Updates Models

The save method may also be used to update models that already exist in the database.
To update a model, you should retrieve it, set any attributes you wish to update, and then call the save method. Again, the updated at timestamp will automatically be updated, so there is no need to manually set its value:

```
$flight = App\Flight::find(1);
$flight->name = 'New Flight Name';
$flight->save();
```

Mass Updates

Updates can also be performed against any number of models that match a given query.
In this example, all flights that are active and have a destination of San Diego will be marked as delayed:The update method expects an array of column and value pairs representing the columns that should be updated.

```

App\Flight::where('active', 1)
->where('destination', 'San Diego')
->update(['delayed' => 1]);
```

Deleting Models

To delete a model, call the delete method on a model instance:

```
$flight = App\Flight::find(1);
$flight->delete();
```

Deleting An Existing Model By Key :

we are retrieving the model from the database before calling the delete method. if you know the primary key of the model, you may delete the model without retrieving it. To do so, call the destroy method:

```
App\Flight::destroy(1);
App\Flight::destroy([1, 2, 3]);
App\Flight::destroy(1, 2, 3);
```

Of course, you may also run a delete statement on a set of models. In this example, we will delete all flights that are marked as inactive. Like mass updates, mass deletes will not fire any model events for the models that are deleted

```
$deletedRows = App\Flight::where('active', 0)->delete();
```

Query Builder

Laravel's database query builder provides a convenient, fluent interface to creating and running database queries. It can be used to perform most database operations in your application and works on all supported database systems. The Laravel query builder uses PDO parameter binding to protect your application against SQL injection attacks. There is no need to clean strings being passed as bindings.

Retrieving Results

Retrieving All Rows From A Table You may use the table method on the DB facade to begin a query. The table method returns a fluent query builder instance for the given table, allowing you to chain more constraints onto the query and then finally get the results using the get method

```
<?php
namespace App\Http\Controllers;
use Illuminate\Support\Facades\DB;
use App\Http\Controllers\Controller;
class UserController extends Controller
{
public function index()
{
$users = DB::table('users')->get();
return view('user.index', ['users' => $users]);
}
}
```

The get method returns an Illuminate\Support\Collection containing the results where each result is an instance of the PHP StdClass object.
You may access each column's value by accessing the column as a property of the object:

```
foreach ($users as $user) { echo $user->name; }
```

Retrieving A Single Row / Column From A Table :

If you just need to retrieve a single row from the database table, you may use the first method. This method will return a single StdClass object:

```
$user = DB::table('users')->where('name', 'John')->first();
echo $user->name;
```

If you don't even need an entire row, you may extract a single value from a record using the value method.
This method will return the value of the column directly:

```
$email = DB::table('users')->where('name', 'John')->value('email');
```

Retrieving A List Of Column Values :

If you would like to retrieve an array containing the values of a single column, you may use the pluck method. In this example, we'll retrieve an array of role titles:

```
$titles = DB::table('roles')->pluck('title');
foreach ($titles as $title) { echo $title; }
```

Retrieving A List Of Column Values :

You may also specify a custom key column for the returned array:

```
$roles = DB::table('roles')->pluck('title', 'name');
foreach ($roles as $name => $title) { echo $title; }
```

Chunking Results:

If you need to work with thousands of database records, consider using the chunk method. This method retrieves a small chunk of the results at a time and feeds each chunk into a Closure for processing. This method is very useful for writing Artisan commands that process thousands of records.

Retrieving A List Of Column Values :

```
DB::table('users')->orderBy('id')->chunk(100, function ($users)
{ foreach ($users as $user) { // } });
```

You may stop further chunks from being processed by returning false from the Closure:

```
DB::table('users')->orderBy('id')->chunk(100, function ($users)
{ // Process the records... return false; });
```

Aggregates:

The query builder also provides a variety of aggregate methods such as count, max, min, avg, and sum. You may call any of these methods after constructing your query:
$users = DB::table('users')->count();
$price = DB::table('orders')->max('price');
Of course, you may combine these methods with other clauses

```
$price = DB::table('orders') ->where('finalized', 1) ->avg('price');
```

Selects :

Specifying A Select Clause : Of course, you may not always want to select all columns from a database table. Using the selectmethod, you can specify a
custom select clause for the query:

```
$users = DB::table('users')->select('name', 'email as user_email')->get();
```

The distinct method allows you to force the query to return distinct results:

```
$users = DB::table('users')->distinct()->get();
```

If you already have a query builder instance and you wish to add a column to its existing select clause, you may use the addSelect method:

```
$query = DB::table('users')->select('name');
$users = $query->addSelect('age')->get();
```

Raw Expressions

Sometimes you may need to use a raw expression in a query. To create a raw expression, you may use the DB::raw method:

```
$users = DB::table('users')
->select(DB::raw('count(*) as user_count, status'))
->where('status', '<>', 1)
->groupBy('status') ->get();
```

Joins

Inner Join Clause

The query builder may also be used to write join statements. To perform a basic "inner join", you may use the join method on a query builder instance. The first argument passed to the joinmethod is the name of the table you need to join to, while the remaining arguments specify the column constraints for the join. Of course, as you can see, you can join to multiple tables in a single query:

```
$users = DB::table('users') ->join('contacts', 'users.id', '=',
'contacts.user_id')
->join('orders', 'users.id', '=', 'orders.user_id')
->select('users.*', 'contacts.phone', 'orders.price')
->get();
```

Left Join Clause

If you would like to perform a "left join" instead of an "inner join", use the leftJoin method. The leftJoin method has the same signature as the join method:

```
$users = DB::table('users')
->leftJoin('posts', 'users.id', '=', 'posts.user_id')
->get();
```

Cross Join Clause

To perform a "cross join" use the crossJoin method with the name of the table you wish to cross join to. Cross joins generate a cartesian product between the first table and the joined table

```
$users = DB::table('sizes') ->crossJoin('colours') ->get();
```

Advanced Join Clauses

You may also specify more advanced join clauses. To get started, pass a Closure as the second argument into the join method.

The Closure will receive a JoinClause object which allows you to specify constraints on the join clause:
DB::table('users')
->join('contacts', function ($join) { $join->on('users.id', '=',
'contacts.user_id')->orOn(...); })
->get();

If you would like to use a "where" style clause on your joins, you may use the where and orWheremethods on a join. Instead of comparing two columns, these methods will compare the column against a value:

```
DB::table('users')
->join('contacts', function ($join)
{
$join->on('users.id', '=', 'contacts.user_id')
->where('contacts.user_id', '>', 5); })
->get();
```

Unions

The query builder also provides a quick way to "union" two queries together.
For example, you may create an initial query and use the union method to union it with a second query:

```
$first = DB::table('users') ->whereNull('first_name');
$users = DB::table('users') ->whereNull('last_name') ->union($first) ->get
```

Where Clauses

Simple Where Clauses

You may use the where method on a query builder instance to add where clauses to the query. The most basic call to where requires three arguments. The first argument is the name of the column. The second argument is an operator, which can be any of the database's supported operators. Finally, the third argument is the value to evaluate against the column. For example, here is a query that verifies the value of the
"votes" column is equal to 100:

```
  users = DB::table('users') ->where('votes', '<>', 100) ->get();
```

Of course, you may use a variety of other operators when writing a where clause:

```
$users = DB::table('users') ->where('votes', '>=', 100) ->get();
$users = DB::table('users') ->where('name', 'like', 'T%') ->get();
$users = DB::table('users') ->where('name', 'like', 'T%') ->get();
```

You may also pass an array of conditions to the where function:

```
$users = DB::table('users')->where([ ['status', '=', '1'], ['subscribed', '<>', '1'], ])->get();
```

Or Statements:

You may chain where constraints together as well as add or clauses to the query. The or Where method accepts the same arguments as the where method

```
  users = DB::table('users')
->where('votes', '>', 100)
->orWhere('name', 'John') ->get();
```

Additional Where Clauses

whereBetween: The whereBetween method verifies that a column's value is between two values:

```
$users = DB::table('users') ->whereBetween('votes', [1, 100])->get();
```

whereNotBetween:

The whereNotBetween method verifies that a column's value lies outside of two values:

```
$users = DB::table('users')
->whereNotBetween('votes', [1, 100])
->get();
```

whereIn / whereNotIn : The whereIn method verifies that a given column's value is contained within the given array:

```
$users = DB::table('users') ->whereIn('id', [1, 2, 3]) ->get();
```

The whereNotIn method verifies that the given column's value is not contained in the given array:

```
$users = DB::table('users') ->whereNotIn('id', [1, 2, 3]) ->get();
```

whereNull / whereNotNull

The whereNull method verifies that the value of the given column is NULL:

```
$users = DB::table('users') ->whereNull('updated_at') ->get();
```

The whereNotNull method verifies that the column's value is not NULL:

```
$users = DB::table('users') ->whereNotNull('updated_at') ->get();
```

whereDate / whereMonth / whereDay / whereYear

The whereDate method may be used compare a column's value against a date:

```
$users = DB::table('users') ->whereDate('created_at', '2016-12-31') ->get();
```

The whereMonth method may be used compare a column's value against a specific month of a year:

```
$users = DB::table('users') ->whereMonth('created_at', '12') ->get();
```

The whereDay method may be used compare a column's value against a specific day of a month:

```
$users = DB::table('users') ->whereDay('created_at', '31') ->get();
```

The whereYear method may be used compare a column's value against a specific year:

```
$users = DB::table('users') ->whereYear('created_at', '2016') ->get();
```

whereColumn : The whereColumn method may be used to verify that two columns are equal:

```
$users = DB::table('users') ->whereColumn('first_name', 'last_name') ->get();
```

You may also pass a comparison operator to the method:

```
$users = DB::table('users')
->whereColumn('updated_at', '>', 'created_at') ->get();
```

The whereColumn method can also be passed an array of multiple conditions. These conditions will be joined using the and operator:

```
$users = DB::table('users')
->whereColumn([ ['first_name', '=', 'last_name'], ['updated_at', '>',
'created_at'] ])->get();
```

Ordering, Grouping, Limit, & Offset orderBy:

The orderBy method allows you to sort the result of the query by a given column. The first argument to the orderBy method should be the column you wish to sort by,
while the second argument controls the direction of the sort and may be either asc or desc

```
$users = DB::table('users') ->orderBy('name', 'desc') ->get();
```

latest / oldest :

The latest and oldest methods allow you to easily order results by date. By default, result will be ordered by the created_at column. Or, you may pass the column name that you wish to sort by

```
$user = DB::table('users') ->latest() ->first();
```

inRandomOrder:

The inRandomOrder method may be used to sort the query results randomly. For example, you may use this method to fetch a random user:

```
$randomUser = DB::table('users') ->inRandomOrder() ->first();
```

groupBy / having / havingRaw

The groupBy and having methods may be used to group the query results. The having method's signature is similar to that of the where method:

```
$users = DB::table('users')
->groupBy('account_id')
->having('account_id', '>', 100) ->get();
```

The havingRaw method may be used to set a raw string as the value of the having clause. For example, we can find all of the departments with sales greater than $2,500:

```
$users = DB::table('orders')
->select('department', DB::raw('SUM(price) as total_sales'))
->groupBy('department') ->havingRaw('SUM(price) > 2500')
->get();
```

skip / take:

To limit the number of results returned from the query, or to skip a given number of results in the query, you may use the skip and take methods:

```
$users = DB::table('users')->skip(10)->take(5)->get();
```

Alternatively, you may use the limit and offset methods:

```
$users = DB::table('users') ->offset(10) ->limit(5) ->get();
```

Conditional Clauses:

Sometimes you may want clauses to apply to a query only when something else is true. For instance you may only want to apply a where statement if a given input value is present on the incoming request.You may accomplish this using the when method:

```
$role = $request->input('role');
$users = DB::table('users') ->when($role, function ($query) use ($role)
{ return $query->where('role_id', $role); }) ->get();
```

The when method only executes the given Closure when the first parameter is true. If the first parameter is false, the Closure will not be executed. You may pass another Closure as the third parameter to the when method. This Closure will execute if the first parameter evaluates as false. To illustrate how this feature may be used, we will use it to configure the default sorting of a query:

```
$sortBy = null;
$users = DB::table('users') ->when($sortBy, function ($query) use ($sortBy) {
return $query->orderBy($sortBy); },
function ($query) { return $query->orderBy('name'); }) ->get();
```

Query Builder Inserts

The query builder also provides an insert method for inserting records into the database table. The insert method accepts an array of column names and values:

```
DB::table('users')->insert( ['email' => 'john@example.com', 'votes' => 0] );
```

You may even insert several records into the table with a single call to insert by passing an array of arrays.

```
DB::table('users')->insert([ ['email' => 'taylor@example.com', 'votes' => 0],
['email' => 'dayle@example.com', 'votes' => 0] ]);
```

Query Builder Updates :

Of course, in addition to inserting records into the database, the query builder can also update existing records using the update method.The update method, like the insert method, accepts an array of column and value pairs containing the columns to be updated. You may constrain the update query using where clauses:

```
DB::table('users') ->where('id', 1) ->update(['votes' => 1]);
```

Query Builder Deletes

The query builder may also be used to delete records from the table via the delete method. You may constrain delete statements by adding where clauses
before calling the delete method:

```
DB::table('users')->delete();
DB::table('users')->where('votes', '>', 100)->delete();
```

If you wish to truncate the entire table, which will remove all rows and reset the auto-incrementing ID to zero, you may use the truncate method:

```
DB::table('users')->truncate();
```

Migrations :

Laravel encourages an agile, iterative style of development. We don't expect to get everything right the first time. Instead we write code, tests and interact with our end-users to refine our understanding as we go. For that to work, we need a supporting set of practices. We use version control tools like subversion, git or mercurial to store our application's source code files, allowing us to undo mistakes and to track what changes during the course of development.

Migration Basics

A Laravel migration is simply a PHP source file in your application's app/database/migrations folder. Each file contains a discrete set of changes to the underlying database.Changes to the database are made in PHP rather than a database-specific flavour of SQL. Your PHP migration code ends up being converted into the DDL specific to your current database; this makes switching database platforms very easy. Since migrations are kept in their own directory, it's pragmatic to include them into version control just like any other project code. Laravel migrations are run explicitly from command-line using the artisan tool.

Running Migrations Forward and Backward

You apply migrations to the database using the artisan tool. Laravel provides a set of artisan tasks to work with migrations which boil down to running certain
sets of migrations.

Note: You can run artisan list to see the list of tasks it supports, and most of the database migration-related tasks are prefixed with migrate:There are only a few tasks that you really need to know:

```
migrate:install
```

The very first migration related artisan task you will use will probably be artisan migrate:install. Internally, Laravel uses a special table to keep track of which migrations have already run. To create this table, just use the artisan command-line tool:

```
php artisan migrate:install
```

Migrate:You'll run the migrate task frequently to update your database to support the latest tables and columns you've added to your application. In its most basic form it just runs the up() method for all the migrations that have not yet been run. If there are no such migrations, it exits. It will run these migrations in order based on the date of the migration.

migrate:rollback

Occasionally you will make a mistake when writing a migration. If you have already run the migration then you cannot just edit the migration and run the migration again: Laravel assumes it has already run the migration and so will do nothing when you run artisan migrate. You must rollback the migration using artisan migrate:rollback, edit your migration and then run artisan migrate to run the corrected version.

migrate:reset

This task will roll back all migrations that have ever run

migrate:refresh

The artisan migrate:refresh task will drop the database, recreate it and load the current schema into it. This is a convenience shortcut to run reset and subsequently re-run all migrations.

migrate:make

The artisan migrate:make command tells Laravel to generate a skeleton migration file (which is actually a PHP file) in the app/database/migrations folder. You can then edit this file to flesh out your table/index definition. Later, when you run the artisan migrate command, Artisan will consult this file to generate actual SQL DDL code.

Creating Laravel_db database

1. Open PHPMyAdmin or what ever MySQL database management tool that you are using.
2. Run the following command to create a database

```
CREATE DATABASE `laravel_db`;
```

HERE,

• CREATE DATABASE laravel_db; creates a database called laravel_db in MySQL

Setting database connection parameters for Laravel

1. Open /config/database.php
2. Locate the following lines of code

```
'mysql' => [
    'driver'    => 'mysql',
    'host'      => env('DB_HOST', 'localhost'),
    'database'  => env('DB_DATABASE', 'forge'),
    'username'  => env('DB_USERNAME', 'forge'),
    'password'  => env('DB_PASSWORD', ''),
    'charset'   => 'utf8',
    'collation' => 'utf8_unicode_ci',
    'prefix'    => '',
    'strict'    => false,
],
```

Update the following array keys to match the settings on your instance of MySQL

```
'database' => env('DB_DATABASE', ' laravel_db'),
'username'  => env('DB_USERNAME', 'root'),
'password'  => env('DB_PASSWORD', ''),
```

Setting database connection parameters for Artisan One of the challenges that most developers face when working with migrations in Laravel 5 from the artisan command line tool is the following message. Access denied for user 'homestead'@' localhost' (using password: YES)

You will get the above message even if you have set the correct parameters in /config/database.php This is because the artisan command line tool uses the database connection parameters specified in .env file.

The solution

1. Go to your project route using windows explorer or whatever tool you use to browser files on your system.
2. Open /.env file

You will get the following

```
APP_ENV=local
APP_DEBUG=true
APP_KEY=aqk5XHULL8TZ8t6pXE43o7MBSFchfgy2

DB_HOST=localhost
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret

CACHE_DRIVER=file
SESSION_DRIVER=file
QUEUE_DRIVER=sync

MAIL_DRIVER=smtp
MAIL_HOST=mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
Update the following variables

DB_HOST=localhost
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret
```

to

```
DB_HOST=localhost
DB_DATABASE= laravel_db
DB_USERNAME=root
```

```
DB_PASSWORD=
```

Note: the database, username and password must match the ones on your system. Save the changes

Artisan migration command

In this section, we will create;

1. The migration table in our database
2. A migration file that we will use to create a table for hard drinks.

When you create a migration file, Laravel stores it in /database/migrations directory. You can specify a different path if you would like to but we won't cover that in this tutorial. We will work with the default path. Open the command prompt or terminal depending on your operating system For this tutorial, we are using windows. Run the following command to browse to the command prompt.

Create migration table

Run the following artisan command to create a migration table in Larashop database.

```
php artisan migrate:install
```

- You will get the following message

Migration table created successfully.

- Run the following command to create a migration file

```
php artisan make:migration create_drinks_table
```

Migration structure

We will now examine the contents of the created migration file Open the file /database/migrations/2015*0827072434createdrinks*table.php You will get the following file. Note: all comments have been removed for brevity's sake in this example.

```
<?php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;
class CreateDrinksTable extends Migration
{
   public function up()
   {
      //
   }
      public function down()
   {
      //
   }
}
```

HERE,

- class CreateDrinksTable extends Migration defines the CreateDrinksTable class that extends Migration class
- public function up() defines the function that is executed when the migration is run
- public function down() defines the function that is executed when you run migration rollback

How to create a table using a migration

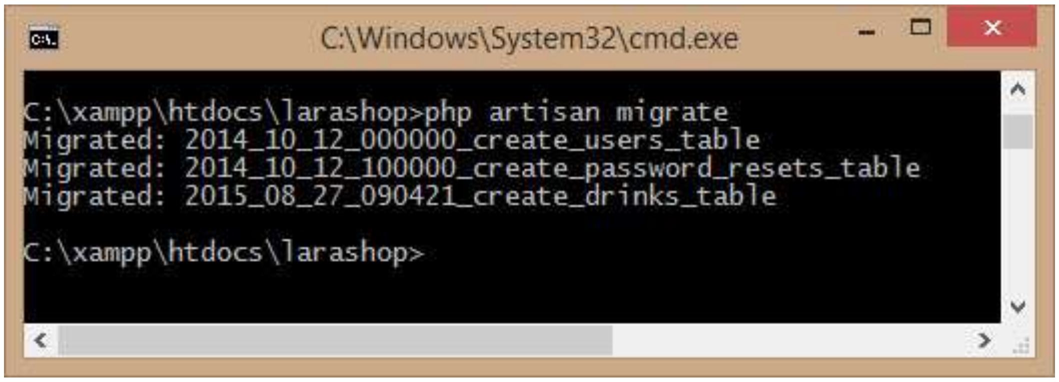
Now that we have successfully created a migration file, we will add the table definition fields in the migration Modify the contents of /database/migrations/2015*0827072434createdrinks*table.php

```
<?php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;
class CreateDrinksTable extends Migration {
   /**
    * Run the migrations.
    *
    * @return void
    */
   public function up() {
      Schema::create('drinks', function (Blueprint $table) {
         $table->increments('id');
         $table->string('name',75)->unique();
         $table->text('comments')->nullable();
         $table->integer('rating');
         $table->date('brew_date');
         $table->timestamps();      });
   }
   /**
    * Reverse the migrations.
    *
    * @return void
    */
   public function down() {
      Schema::drop('drinks');
    }
}
```

Go back to the command prompt Run the following command

```
php artisan migrate
```

You will get the following output



```
C:\xampp\htdocs\larashop>php artisan migrate
Migrated: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_100000_create_password_resets_table
Migrated: 2015_08_27_090421_create_drinks_table

C:\xampp\htdocs\larashop>
```

Eloquent ORM Models :

Eloquent models are models that extend the Illuminate\Database\Eloquent\Model class In the MVC architectural design, models are used to interact with data sources. Eloquent ORM is an Object Relational Mapper developed by Laravel. It implements the active record pattern and is used to interact with relational databases. Eloquent ORM can be used inside Laravel or outside Laravel since itʒs a complete package on its own. Models are Laravel 5 are located in the /app directory. Before getting started, be sure to configure a database connection in app/config/database.php

Naming conventions:

The law of the gods state that model names should be singular and table names should be plural. Laravel automatically pluralizes the model name to get the table name.
Eloquent Models provide the above as the default implementation. You can explicitly specify a table name if you want to. We will now create a model for the categories table. The name of the model will be category. This is in accordance with the law of the gods.

- Open the command prompt and browser to the project root.

Run the following artisan command

```
php artisan make:model Category
```

Open the newly created model in /app/Category.php
You will get the following

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Category extends Model
{
   //
}
```

Creating Model Factories

When creating seed data in the past you could utilize Eloquent or Laravel’s query builder and that way is still supported. However, with the introduction of model factories you can use them to build out “Dummy “ models which can be used for both seed data and in testing.

Open database/factories/ModelFactory.php and you will see a default one already defined:

```
$factory->define(App\User::class, function (Faker\Generator
$faker) {
return [
'name' => $faker->name,
'email' => $faker->email,
'password' => bcrypt(str_random(10)),
'remember_token' => str_random(10),
];
});
```

To explain this, we are defining the "App\User::class" model as the first parameter and then a callback that defines the data that goes in the columns. This callback also injects Faker which is a PHP library that generates fake data. Faker is powerful and can be used for a number of different field types. Here is an example of the ones shown:
$fake->name – "nidhi shah"
$faker->email –"nidhi@tops-int.com"

Eloquent ORM INSERT

Step -1 : open /app/Http/routes.php

```
Route::get('/insert', function() {
  App\Category::create(array('name' => 'Music'));
 return 'category added';
});
```

Step -2 : Load the following URL

```
http://localhost:8000/insert
```

Eloquent ORM READ

Step -1 : open /app/Http/routes.php

```
Route::get('/read', function() {
    $category = new App\Category();

    $data = $category->all(array('name','id'));

    foreach ($data as $list) {
        echo $list->id . ' ' . $list->name . '
';
    }
});
```

Step -2 : Load the following URL

```
http://localhost:8000/read
```

Eloquent ORM Update

Step -1 : open /app/Http/routes.php

```
Route::get('/update', function() {
   $category = App\Category::find(16);
   $category->name = 'HEAVY METAL';
   $category->save();
      $data = $category->all(array('name','id'));
   foreach ($data as $list) {
      echo $list->id . ' ' . $list->name . '
';
   }
});
```

Step -2 : Load the following URL

```
http://localhost:8000/update
```

Eloquent ORM Delete

Step -1 : open /app/Http/routes.php

```
Route::get('/delete', function() {
   $category = App\Category::find(5);
   $category->delete();
      $data = $category->all(array('name','id'));
   foreach ($data as $list) {
      echo $list->id . ' ' . $list->name . '
';
   }
});
```

Step -2 : Load the following URL

```
http://localhost:8000/delete
```

Eloquent Relationships :

Introduction:

Database tables are often related to one another. Eloquent makes managing and working with these relationships easy, and supports several different types of relationships

- One To One
- One To Many
- Many To Many

- Has Many Through
- Polymorphic Relations
- Many To Many Polymorphic Relations

Defining Relationships :

Eloquent relationships are defined as functions on your Eloquent model classes.

One To One :

A one-to-one relationship is a very basic relation. For example, a User model might be associated with one Phone. To define this relationship, we place a phone method on the User model. The phone method should call the hasOne method and return its result

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class User extends Model
{
public function phone()
{
return $this->hasOne('App\Phone');
}
}
```

The first argument passed to the hasOne method is the name of the related model. Once the relationship is defined, we may retrieve the related record using Eloquent's dynamic properties Dynamic properties allow you to access relationship functions as if they were properties defined on the model

```
$phone = User::find(1)->phone;
```

Eloquent determines the foreign key of the relationship based on the model name. In this case, thePhone model is automatically assumed to have a user_id foreign key. If you wish to override this convention, you may pass a second argument to the hasOne method

```
return $this->hasOne('App\Phone', 'foreign_key');
```

Eloquent assumes that the foreign key should have a value matching the id (or the custom $primaryKey) column of the parent. Eloquent will look for the value of the user's id column in the user_id column of the Phone record If you would like the relationship to use a value other than id, you may pass a third argument to the hasOne method specifying your custom key:

```
return $this->hasOne('App\Phone', 'foreign_key', 'local_key');
```

Defining The Inverse Of The Relationship

we can access the Phone model from our User. We can define the inverse of a hasOnerelationship using the belongsTo method

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Phone extends Model { /* Get the user that owns the phone. */
public function user() {
return $this->belongsTo('App\User');
}
}
```

Eloquent will try to match the user_id from the Phone model to an id on the User model. Eloquent determines the default foreign key name by examining the name of the relationship method and suffixing the method name with _id.
if the foreign key on thePhone model is not user_id, you may pass a custom key name as the second argument to the belongsTo method

```
public function user()
{ return $this->belongsTo('App\User', 'foreign_key'); }
```

If your parent model does not use id as its primary key, or you wish to join the child model to a different column, you may pass a third argument to the belongsTo method specifying your parent table's custom key

```
public function user()
{ return $this->belongsTo('App\User', 'foreign_key', 'other_key'); }
```

One To Many

A "one-to-many" relationship is used to define relationships where a single model owns any amount of other models. For example, a blog post may have an infinite number of comments. Like all other Eloquent relationships, one-to-many relationships are defined by placing a function on your Eloquent model

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Post extends Model {
public function comments() {
return $this->hasMany('App\Comment'); } }
```

Eloquent will automatically determine the proper foreign key column on the Comment model. By convention, Eloquent will take the "snake case" name of the owning model and suffix it with _id. Eloquent will assume the foreign key on the Comment model ispost_id.
Once the relationship has been defined, we can access the collection of comments by accessing the comments property.

```
$comments = App\Post::find(1)->comments;
foreach ($comments as $comment) { // }
```

All relationships also serve as query builders, you can add further constraints to which comments are retrieved by calling the comments method and continuing to chain conditions onto the query

```
$comments = App\Post::find(1)->comments()->where('title', 'foo')->first();
```

the hasOne method, you may also override the foreign and local keys by passing additional arguments to the hasMany method

```
return $this->hasMany('App\Comment', 'foreign_key');
return $this->hasMany('App\Comment', 'foreign_key', 'local_key');
```

One To Many (Inverse)

that we can access all of a post's comments, let's define a relationship to allow a comment to access its parent post.
To define the inverse of a hasMany relationship, define a relationship function on the child model which calls the belongsTo method

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Comment extends Model
{ /** * Get the post that owns the comment. */
public function post()
{ return $this->belongsTo('App\Post'); } }
```

Now that we can access all of a post's comments To define the inverse of a hasMany relationship, define a relationship function on the child model which calls the belongsTo method

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Comment extends Model
{
public function post()
{
return $this->belongsTo('App\Post');
}
}
```

Once the relationship has been defined, we can retrieve the Post model for a Comment by accessing the post "dynamic property”

```
$comment = App\Comment::find(1);
echo $comment->post->title;
```

Eloquent will try to match the post_id from the Comment model to an id on the Post model.
Eloquent determines the default foreign key name by examining the name of the relationship method and suffixing the method name with _id.
if the foreign key on theComment model is not post_id, you may pass a custom key name as the second argument to thebelongsTo method

```
public function post() { return $this->belongsTo('App\Post', 'foreign_key'); }
```

If your parent model does not use id as its primary key, or you wish to join the child model to a different column, you may pass a third argument to the belongsTo method specifying your parent table's custom key
public function post()
{ return $this->belongsTo('App\Post', 'foreign_key', 'other_key'); }

Many To Many

Many-to-many relations are slightly more complicated than hasOne and hasMany relationships. An example of such a relationship is a user with many roles, where the roles are also shared by other users. For example, many users may have the role of "Admin".
To define this relationship, three database tables are needed: users, roles, and role user.
The role_user table is derived from the alphabetical order of the related model names, and contains the user_id and role_id columns Many-to-many relationships are defined by writing a method that returns the result of the belongsToMany method
For example, let's define the roles method on our User model

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class User extends Model
{ /** * The roles that belong to the user. */
public function roles()
{
return $this->belongsToMany('App\Role');
}
}
```

Once the relationship is defined, you may access the user's roles using the roles dynamic property:

```
$user = App\User::find(1); foreach ($user->roles as $role) { // }
```

Of course, like all other relationship types, you may call the roles method to continue chaining query constraints onto the relationship

```
$roles = App\User::find(1)->roles()->orderBy('name')->get();
```

Eloquent will join the two related model names in alphabetical order. However, you are free to override this convention. You may do so by passing a second argument to the belongsToMany method

```
return $this->belongsToMany('App\Role', 'role_user');
```

In addition to customizing the name of the joining table, you may also customize the column names of the keys on the table by passing additional arguments to the belongsToMany method. The third argument is the foreign key name of the model on which you are defining the relationship, while the fourth argument is the foreign key name of the model that you are joining to

```
return $this->belongsToMany('App\Role', 'role_user', 'user_id', 'role_id');
```

Retrieving Intermediate Table Columns

let's assume our User object has many Role objects that it is related to. After accessing this relationship, we may access the intermediate table using the pivot attribute on the models

```
$user = App\User::find(1);
foreach ($user->roles as $role)
{ echo $role->pivot->created_at; }
```

Notice that each Role model we retrieve is automatically assigned a pivot attribute. This attribute contains a model representing the intermediate table, and may be used like any other Eloquent model. By default, only the model keys will be present on the pivot object. If your pivot table contains extra attributes, you must specify them when defining the relationship

```
return $this->belongsToMany('App\Role')->withPivot('column1', 'column2');
```

If you want your pivot table to have automatically maintained created_at and updated_attimestamps, use the withTimestamps method on the relationship definition

```
return $this->belongsToMany('App\Role')->withTimestamps();
```

Filtering Relationships Via Intermediate Table Columns

You can also filter the results returned by belongsToMany using the wherePivot and wherePivotInmethods when defining the relationship

```
return $this->belongsToMany('App\Role')->wherePivot('approved', 1);
return $this->belongsToMany('App\Role')->wherePivotIn('priority', [1, 2]);
```

Has Many Through

The "has-many-through" relationship provides a convenient short-cut for accessing distant relations via an intermediate relation. For example, a Country model might have many Post models through an intermediate User model. In this example, you could easily gather all blog posts for a given country. Let's look at the tables required to define this relationship.

```
Countries
id - integer
name - string
users
id - integer
country_id – integer
name - string
posts
id - integer
user_id - integer
title - string
```

Though posts does not contain a country_id column, the hasManyThrough relation provides access to a country's posts via $country->posts. To perform this query, Eloquent inspects the country_idon the intermediate users table. After finding the matching user IDs, they are used to query the posts table.

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Country extends Model
{ /** * Get all of the posts for the country. */
public function posts() { return $this->hasManyThrough('App\Post',
'App\User'); } }
```

The first argument passed to the hasManyThrough method is the name of the final model we wish to access, while the second argument is the name of the intermediate model.

Defining The Inverse Of The Relationship

To define the inverse of a many-to-many relationship, you simply place another call tobelongsToMany on your related model. To continue our user roles example, let's define the usersmethod on the Role model

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Role extends Model
{
public function users()
{
return $this->belongsToMany('App\User');
}
}
```

Typical Eloquent foreign key conventions will be used when performing the relationship's queries. If you would like to customize the keys of the relationship, you may pass them as the third and fourth arguments to the hasManyThrough method.
The third argument is the name of the foreign key on the intermediate model, the fourth argument is the name of the foreign key on the final model, and the fifth argument is the local key

```
class Country extends Model
{ public function posts() { return $this->hasManyThrough( 'App\Post',
'App\User', 'country_id', 'user_id', 'id' ); } }
```

Polymorphic Relations

Table Structure:

Polymorphic relations allow a model to belong to more than one other model on a single association. Using polymorphic relationships, you can use a single comments table for both of these scenarios.

Model Structure

We're ready to define the relationships on the model. The Post and Video models will both have a tags method that calls the morphToMany method on
the base Eloquent class

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
```

```
class Post extends Model
{ /** * Get all of the tags for the post. */
public function tags()
{ return $this->morphToMany('App\Tag', 'taggable'); } }
```

Next, on the Tag model, you should define a method for each of its related models. So, for this example, we will define a posts method and a videos method

```
<?php
namespace App; use Illuminate\Database\Eloquent\Model;
class Tag extends Model
{
/** * Get all of the posts that are assigned this tag. */
public function posts() {
return $this->morphedByMany('App\Post', 'taggable'); }
/** * Get all of the videos that are assigned this tag. */
public function videos()
{ return $this->morphedByMany('App\Video', 'taggable'); } }
```

Retrieving The Relationship

Once your database table and models are defined, you may access th relationships via your models. For example, to access all of the tags for a post, you can simply use the tags dynamic property

```
$post = App\Post::find(1); foreach ($post->tags as $tag) { // }
```

You may also retrieve the owner of a polymorphic relation from the polymorphic model by accessing the name of the method that performs the call to morphedByMany.
In our case, that is the posts or videos methods on the Tag model. So, you will access those methods as dynamic properties

```
$tag = App\Tag::find(1); foreach ($tag->videos as $video) { // }
```

Querying Relations

all types of Eloquent relationships also serve as query builders, allowing you to continue to chain constraints onto the relationship query before finally executing the SQL against your database.

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class User extends Model
{ /** * Get all of the posts for the user. */
public function posts() { return $this->hasMany('App\Post'); } }
```

You may query the posts relationship and add additional constraints to the relationship like so.

```
$user = App\User::find(1); $user->posts()->where('active', 1)->get();
```

Relationship Methods Vs. Dynamic Properties

If you do not need to add additional constraints to an Eloquent relationship query, you may simply access the relationship as if it were a property.

```
$user = App\User::find(1); foreach ($user->posts as $post) { // }
```

Dynamic properties are "lazy loading", meaning they will only load their relationship data when you actually access them. Because of this, developers often use eager loading to preload relationships they know will be accessed after loading the model. Eager loading provides a significant reduction in SQL queries that must be executed to load a model's relations.

Querying Relationship Existence

When accessing the records for a model, you may wish to limit your results based on the existence of a relationship. For example, imagine you want to retrieve all blog posts that have at least one comment.To do so, you may pass the name of the relationship to the has method

```
$posts = App\Post::has('comments')->get();
```

You may also specify an operator and count to further customize the query

```
$posts = Post::has('comments', '>=', 3)->get();
```

Nested has statements may also be constructed using "dot" notation.

```
// Retrieve all posts that have at least one comment with votes...
$posts = Post::has('comments.votes')->get();
```

Querying Relationship Existence

If you need even more power, you may use the whereHas and orWhereHas methods to put "where" conditions on your has queries. These methods allow you to add customized constraints to a relationship constraint, such as checking the content of a comment

```
// Retrieve all posts with at least one comment containing words like foo%
$posts = Post::whereHas('comments', function ($query)
{ $query->where('content', 'like', 'foo%'); })->get();
```

Querying Relationship Absence

When accessing the records for a model, you may wish to limit your results based on the absence of a relationship.

To do so, you may pass the name of the relationship to the doesntHave method

```
$posts = App\Post::doesntHave('comments')->get();
```

If you need even more power, you may use the whereDoesntHave method to put "where" conditions on your doesntHave queries. This method allows you to add customized constraints to a relationship constraint, such as checking the content of a comment

```
$posts = Post::whereDoesntHave('comments', function ($query) { $query-
>where('content', 'like', 'foo%'); })->get();
```

Counting Related Models

If you want to count the number of results from a relationship without actually loading them you may use the withCount method, which will place a {relation}_count column on your resulting models.

```
$posts = App\Post::withCount('comments')->get();
foreach ($posts as $post) { echo $post->comments_count; }
```

You may add retrieve the "counts" for multiple relations as well as add constraints to the queries

```
$posts = Post::withCount(['votes', 'comments' => function ($query)
{ $query->where('content', 'like', 'foo%'); }])->get();
echo $posts[0]->votes_count;
echo $posts[0]->comments_count;
```

Eager Loading :

When accessing Eloquent relationships as properties, the relationship data is "lazy loaded". This means the relationship data is not actually loaded until you first access the property. Eloquent can "eager load" relationships at the time you query the parent model. Eager loading alleviates the N + 1 query problem. To illustrate the N + 1 query problem, consider a Book model that is related to Author

```
<?php
namespace App;
use Illuminate\Database\Eloquent\Model;
class Book extends Model
{
public function author()
{
return $this->belongsTo('App\Author');
}
}
```

Now, let's retrieve all books and their authors

```
$books = App\Book::all();
foreach ($books as $book)
{ echo $book->author->nam
```

This loop will execute 1 query to retrieve all of the books on the table, then another query for each book to retrieve the author.

```
$books = App\Book::with('author')->get();
foreach ($books as $book) { echo $book->author->name; }
```

Eager Loading Multiple Relationships

Sometimes you may need to eager load several different relationships in a single operation. To do so, just pass additional arguments to the with method

```
$books = App\Book::with('author', 'publisher')->get();=
```

To eager load nested relationships, you may use "dot" syntax. For example, let's eager load all of the book's authors and all of the author's personal contacts in one Eloquent statement

```
$books = App\Book::with('author.contacts')->get();, 'publisher')->get();
```

Constraining Eager Loads

Sometimes you may wish to eager load a relationship, but also specify additional query constraints for the eager loading query.

```
$users = App\User::with(['posts' => function ($query)
{ $query->where('title', 'like', '%first%'); }])->get();
```

In this example, Eloquent will only eager load posts where the post's title column contains the word first. Of course, you may call other query builder methods to further customize the eager loading operation

```
$users = App\User::with(['posts' => function ($query)
{ $query->orderBy('created_at', 'desc'); }])->get();
```

Inserting & Updating Related Models

The Save Method: Eloquent provides convenient methods for adding new models to relationships. For example,perhaps you need to insert a new Comment for a Post model. Instead of manually setting the post_id attribute on the Comment, you may insert the Comment directly from the relationship's save method

```
$comment = new App\Comment(['message' => 'A new comment.']);
$post = App\Post::find(1);
$post->comments()->save($comment);
```

Notice that we did not access the comments relationship as a dynamic property. Instead, we called the comments method to obtain an instance of the relationship. The save method will automatically add the appropriate post_id value to the new Comment model. If you need to save multiple related models, you may use the saveMany method

```
$post = App\Post::find(1);
$post->comments()->saveMany
([ new App\Comment(['message' => 'A new comment.']), new
App\Comment(['message' => 'Another comment.']), ]);
```

The Create Method

In addition to the save and saveMany methods, you may also use the create method, which accepts an array of attributes, creates a model, and inserts it into the database. the difference between save and create is that save accepts a full Eloquent model instance while createaccepts a plain PHP array

```
$post = App\Post::find(1);
$comment = $post->comments()->create([ 'message' => 'A new
comment.', ]);
```

Belongs To Relationships

When updating a belongsTo relationship, you may use the associate method. This method will set the foreign key on the child model

```
$account = App\Account::find(10);
$user->account()->associate($account);
$user->save();
```

When removing a belongsTo relationship, you may use the dissociate method. This method will set the relationship's foreign key to null

```
$user->account()->dissociate();
$user->save();
```

Sessions

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.

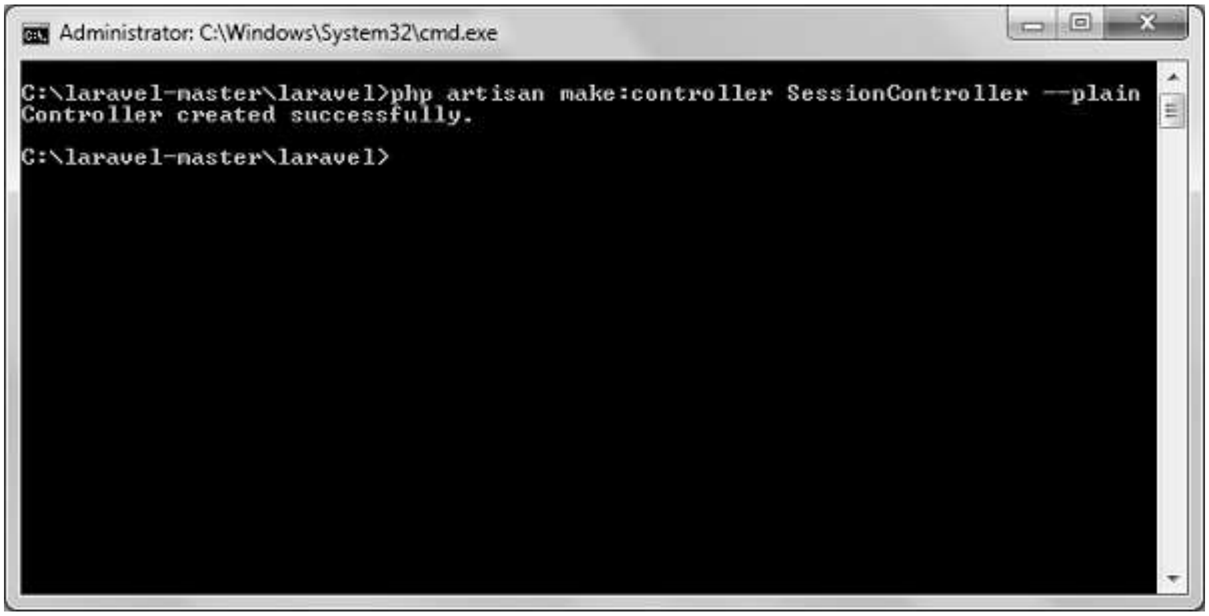
```
$request->session()->forget('key');
```

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.

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

```
php artisan make:controller SessionController
```

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



```
C:\laravel-master\laravel>php artisan make:controller SessionController --plain
Controller created successfully.

C:\laravel-master\laravel>
```

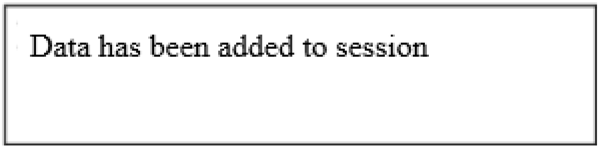
Step 3 – Copy the following code in a file at **app/Http/Controllers/SessionController.php.**
app/Http/Controllers/SessionController.php

```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','Nidhi Shah');
    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.

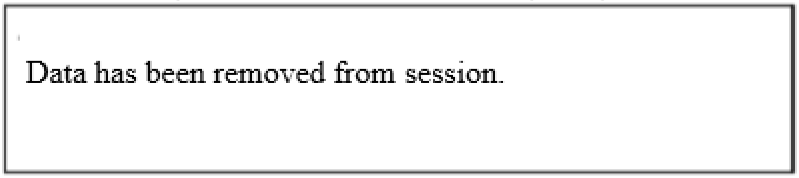


```
Data has been added to session
```

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.

```
Nidhi Shah
```

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.



```
Data has been removed from session.
```

Events :

Laravel's events provides a simple observer implementation, allowing you to subscribe and listen for various events that occur in your application. Event classes are typically stored in the app/Eventsdirectory, while their listeners are stored in app/Listeners.
Events serve as a great way to decouple various aspects of your application, since a single event can have multiple listeners that do not depend on each other.

Registering Events & Listeners

The EventServiceProvider included with your Laravel application provides a convenient place to register all of your application's event listeners. The listen property contains an array of all events (keys) and their listeners (values). Of course, you may add as many events to this array as your application requires.

Generating Events & Listeners

simply add listeners and events to your EventServiceProvider and use the event:generate command. This command will generate any events or listeners that are listed in your EventServiceProvider. Of course, events and listeners that already exist will be left untouched

```
php artisan event:generate
```

Manually Registering Events

Events should be registered via the EventServiceProvider $listen array; however, you may also register Closure based events manually in the boot method of your EventServiceProvider

```
/** * Register any other events for your application. * * @return void */
public function boot()
{
parent::boot(); Event::listen('event.name', function ($foo, $bar)
{ // });
}
```

Wildcard Event Listeners

You may even register listeners using the * as a wildcard parameter, allowing you to catch multiple events on the same listener. Wildcard listeners receive the entire event data array as a single argument

```
Event::listen('event.*', function (array $data) { // });
```

Defining Events:

An event class is simply a data container which holds the information related to the event. For example, let's assume our generated OrderShipped event receives an Eloquent ORM object

```
<?php
namespace App\Events;
use App\Order;
use Illuminate\Queue\SerializesModels;
class OrderShipped
{
use SerializesModels;
public $order;
/**
* Create a new event instance.
*
* @param Order $order
* @return void
*/
public function __construct(Order $order)
{
$this->order = $order;
}
}
```

As you can see, this event class contains no logic. It is simply a container for the Order instance that was purchased. The SerializesModels trait used by the event will gracefully serialize any Eloquent models if the event object is serialized using PHP's serialize function

Defining Listeners

Event listeners receive the event instance in their handle method.The event:generate command will automatically import the proper event class and type-hint the event on the handle method.Within the handle method, you may perform any actions necessary to respond to the event

```
<?php
namespace App\Listeners;
use App\Events\OrderShipped;
class SendShipmentNotification
{
/**
* Create the event listener.
* @return void
*/
public function __construct()
{ // }
/**
* Handle the event.
* @param OrderShipped $event
* @return void */
public function handle(OrderShipped $event)
{ // Access the order using $event->order... }
}
```

Firing Events

To fire an event, you may pass an instance of the event to the event helper.The helper will dispatch the event to all of its registered listeners.Since the event helper is globally available, you may call it from anywhere in your application:

```
<?php
namespace App\Http\Controllers;
use App\Order;
use App\Events\OrderShipped;
use App\Http\Controllers\Controller;
class OrderController extends Controller
{
/**
* Ship the given order.
* @param int $orderId
* @return Response
*/
public function ship($orderId)
{
$order = Order::findOrFail($orderId);
// Order shipment logic...
event(new OrderShipped($order));
}
}
```

Event Subscribers :

Writing Event Subscribers:

Event subscribers are classes that may subscribe to multiple events from within the class itself, allowing you to define several event handlers within a single class.
Subscribers should define a subscribe method, which will be passed an event dispatcher instance. You may call the listenmethod on the given dispatcher to register event listeners:

```
<?php
namespace App\Listeners;
class UserEventSubscriber { /**
* Handle user login events.
*/
public function onUserLogin($event)
{} /**
* Handle user logout events.
*/ public function onUserLogout($event) {} /**
•Register the listeners for the subscriber. *
•* @param Illuminate\Events\Dispatcher $events */
•public function subscribe($events) {
$events->listen( 'Illuminate\Auth\Events\Login',
'App\Listeners\UserEventSubscriber@onUserLogin' ); $events->listen(
'Illuminate\Auth\Events\Logout',
'App\Listeners\UserEventSubscriber@onUserLogout' );
} }
```

Registering Event Subscribers

After writing the subscriber, you are ready to register it with the event dispatcher.
You may register subscribers using the $subscribe property on the EventServiceProvider. For example, let's add the UserEventSubscriber to the list

```
<?php
namespace App\Providers;
use Illuminate\Foundation\Support\Providers\EventServiceProvider as
ServiceProvider;
class EventServiceProvider extends ServiceProvider
{
/** * The event listener mappings for the application. *
•@var array */
protected $listen = [ // ];
/** *
The subscriber classes to register. * * @var array */
protected $subscribe =
[ 'App\Listeners\UserEventSubscriber', ];
}
```

Eloquent: Collections

All multi-result sets returned by Eloquent are instances of the Illuminate\Database\Eloquent\Collection object, including results retrieved via the get method or accessed via a relationship.The Eloquent collection object extends the Laravel base collection, so it naturally inherits dozens of methods used to fluently work with the underlying array of Eloquent models. Of course, all collections also serve as iterators, allowing you to loop over them as if they were simple PHP arrays:

```
$users = App\User::where('active', 1)->get();
foreach ($users as $user) { echo $user->name; }
```

collections are much more powerful than arrays and expose a variety of map /reduce pep rations that may be chained using an intuitive interface. For example, let's remove all inactive models and gather the first name for each remaining user:

```
$users = App\User::where('active', 1)->get();
$names = $users->reject(function ($user)
{
return $user->active === false; }) ->map(function ($user)
{
return $user->name;
}
);
```

Available Methods

The Base Collection All Eloquent collections extend the base Laravel collection object; therefore, they inherit all of the powerful methods provided by the base collection class:

all() : The all method returns the underlying array represented by the collection

```
collect([1, 2, 3])->all();
OUTPUT : // [1, 2, 3]
```

avg() : The avg method returns the average of all items in the collection:

```
collect([1, 2, 3, 4, 5])->avg();
OUTPUT :// 3
```

Method Listing :

If the collection contains nested arrays or objects, you should pass a key to use for determining which values to calculate the average

```
$collection = collect([
['name' => 'JavaScript: The Good Parts', 'pages' => 176],
['name' => 'JavaScript: The Definitive Guide', 'pages' => 1096],
]);
$collection->avg('pages');
OUTPUT: // 636
```

chunk(): The chunk method breaks the collection into multiple, smaller collections of a given size:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7]);
$chunks = $collection->chunk(4);
$chunks->toArray();
OUTPUT :// [[1, 2, 3, 4], [5, 6, 7]]
```

This method is especially useful in views when working with a grid system such as Bootstrap. Imagine you have a collection of Eloquent models you want to display in a grid:

```
@foreach ($products->chunk(3) as $chunk)
<div class="row">
@foreach ($chunk as $product)
<div class="col-xs-4">{{ $product->name }}</div>
@endforeach
</div>
@endforeach
```

collapse(): The collapse method collapses a collection of arrays into a single, flat collection Method Listing

```
$collection = collect([[1, 2, 3], [4, 5, 6], [7, 8, 9]]);
$collapsed = $collection->collapse();
$collapsed->all();
OUTPUT :// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

combine() : The combine method combines the keys of the collection with the values of another array or collection

```
$collection = collect(['name', 'age']);
$combined = $collection->combine(['George', 29]);
$combined->all();
OUTPUT :// ['name' => 'George', 'age' => 29]
```

contains() : The contains method determines whether the collection contains a given item:

```
$collection = collect(['name' => 'Desk', 'price' => 100]);
$collection->contains('Desk');
OUTPUT : // true
$collection->contains('New York');
OUTPUT : // false
```

You may also pass a key / value pair to the contains method, which will determine if the given pair exists in the collection:

```
$collection = collect([
['product' => 'Desk', 'price' => 200], ['product' => 'Chair', 'price' => 100], ]
);
$collection->contains('product', 'Bookcase');
OUT PUT :// false
```

Finally, you may also pass a callback to the contains method to perform your own truth test:

```
$collection = collect([1, 2, 3, 4, 5]);
$collection->contains(function ($value, $key)
{ return $value > 5; });
OUTPUT : // false
```

count() : The count method returns the total number of items in the collection:

```
$collection = collect([1, 2, 3, 4]);
$collection->count();
OUTPUT : // 4
```

diff() : The diff method compares the collection against another collection or a plain PHP array based on its values.This method will return the values in the original collection that are not present in the given collection:

```
$collection = collect([1, 2, 3, 4, 5]);
$diff = $collection->diff([2, 4, 6, 8]);
$diff->all();
OUTPUT : // [1, 3, 5]
```

diffKeys() : The diffKeys method compares the collection against another collection or a plain PHP arraybased on its keys.This method will return the key / value pairs in the original collection that are not present in the given collection

```
$collection = collect([ 'one' => 10, 'two' => 20, 'three' => 30, 'four' => 40, 'five'
=> 50, ]);
$diff = $collection->diffKeys([ 'two' => 2, 'four' => 4, 'six' => 6, 'eight' => 8, ]);
$diff->all();
OUTPUT :// ['one' => 10, 'three' => 30, 'five' => 50]
```

each() : The each method iterates over the items in the collection and passes each item to a callback

```
$collection = $collection->each(function ($item, $key) { // });
```

If you would like to stop iterating through the items, you may return false from your callback:

```
$collection = $collection->each(function ($item, $key)
{ if (/* some condition */) { return false; } });
```

every() : The every method creates a new collection consisting of every n-th element:

```
$collection = collect(['a', 'b', 'c', 'd', 'e', 'f']);
$collection->every(4);
OUTPUT :// ['a', 'e']
```

You may optionally pass an offset as the second   argument:

```
$collection->every(4, 1);
// ['b', 'f']
```

except() : The except method returns all items in the collection except for those with the specified keys

```
$collection = collect(['product_id' => 1, 'price' => 100, 'discount' => false]);
$filtered = $collection->except(['price', 'discount']);
$filtered->all();
OUTPUT : // ['product_id' => 1]
```

filter() : The filter method filters the collection using the given callback, keeping only those items that pass a given truth test

```
$collection = collect([1, 2, 3, 4]);
$filtered = $collection->filter(function ($value, $key)
{ return $value > 2; });
$filtered->all();
OUTPUT : // [3, 4]
```

If no callback is supplied, all entries of the collection that are equivalent to false will be removed:

first() : The first method returns the first element in the collection that passes a given truth test

```
collect([1, 2, 3, 4])->first(function ($value, $key) { return $value > 2; });
OUTPUT : // 3
```

You may also call the first method with no arguments to get the first element in the collection. If the collection is empty, null is returned

```
collect([1, 2, 3, 4])->first();
OUTPUT : // 1
```

flatMap() : The flatMap method iterates through the collection and passes each value to the given callback The callback is free to modify the item and return it, thus forming a new collection of modified items Then, the array is flattened by a level:

```
$collection = collect([ ['name' => 'Sally'], ['school' => 'Arkansas'], ['age' => 28] ]);
$flattened = $collection->flatMap(function ($values)
{ return array_map('strtoupper', $values); });
$flattened->all();
OUTPUT : // ['name' => 'SALLY', 'school' => 'ARKANSAS', 'age' => '28'];
```

flatten() : The flatten method flattens a multidimensional collection into a single dimension:

```
$collection = collect(['name' => 'taylor', 'languages' => ['php', 'javascript']]);
$flattened = $collection->flatten();
$flattened->all();
OUTPTU : // ['taylor', 'php', 'javascript'];
```

You may optionally pass the function a "depth" argument:

```
$collection = collect([
'Apple' => [ ['name' => 'iPhone 6S', 'brand' => 'Apple'], ],
'Samsung' => [ ['name' => 'Galaxy S7', 'brand' => 'Samsung'] ], ]);
$products = $collection->flatten(1);
$products->values()->all();
/*
[
['name' => 'iPhone 6S', 'brand' => 'Apple'],
['name' => 'Galaxy S7', 'brand' => 'Samsung'],
] */
```

In this example, calling flatten without providing the depth would have also flattened the nested arrays, resulting in ['iPhone 6S', 'Apple', 'Galaxy S7', 'Samsung'].

Providing a depth allows you to restrict the levels of nested arrays that will be flattened.

flip() : The flip method swaps the collection's keys with their corresponding values

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);
$flipped = $collection->flip();
$flipped->all();
OUTPUT : // ['taylor' => 'name', 'laravel' => 'framework']
```

forget() : The forget method removes an item from the collection by its key

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);
$collection->forget('name');
$collection->all();
OUTPUT : // ['framework' => 'laravel']
```

forPage() : The forPage method returns a new collection containing the items that would be present on a given page number. The method accepts the page number as its first argument and the number of items to show per page as its second argument

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9]);
$chunk = $collection->forPage(2, 3);
$chunk->all(); // [4, 5, 6]
```

get() : The get method returns the item at a given key. If the key does not exist, null is returned:

```

$collection = collect(['name' => mansi', 'framework' => 'laravel']); $value =
$collection->get('name');
OUTPUT : // nidhi
```

You may optionally pass a default value as the second argument

```
$collection = collect(['name' => mansi', 'framework' => 'laravel']);
$value = $collection->get('foo', 'default-value');
// default-value
```

You may even pass a callback as the default value. The result of the callback will be returned if the specified key does not exist

```
$collection->get('email', function () { return 'default-value'; });
// default-value
```

groupBy() : The groupBy method groups the collection's items by a given key

```
$collection = collect([
['account_id' => 'account-x10', 'product' => 'Chair'], ['account_id' =>
'account-x10', 'product' => 'Bookcase'], ['account_id' => 'account-x11',
'product' => 'Desk'], ]);
$grouped = $collection->groupBy('account_id');
$grouped->toArray();
/* [ 'account-x10' => [ ['account_id' => 'account-x10', 'product' => 'Chair'],
['account_id' => 'account-x10', 'product' => 'Bookcase'], ], 'account-x11' => [
['account_id' => 'account-x11', 'product' => 'Desk'], ], ] */
```

In addition to passing a string key, you may also pass a callback. The callback should return the value you wish to key the group by

has() : The has method determines if a given key exists in the collection:

```
$collection = collect(['account_id' => 1, 'product' => 'Desk']);
$collection->has('product');
// true
```

implode() : The implode method joins the items in a collection. Its arguments depend on the type of items in the collection. If the collection contains arrays or objects, you should pass the key of the attributes you wish to join, and the "glue" string you wish to place between the values

```
$collection = collect([ ['account_id' => 1, 'product' => 'Desk'],
['account_id' => 2, 'product' => 'Chair'],
]);
$collection->implode('product', ', ');
OUTPUT : // Desk, Chair
```

If the collection contains simple strings or numeric values, simply pass the "glue" as the only argument to the method:

```
collect([1, 2, 3, 4, 5])->implode('-');
OUTPUT : // '1-2-3-4-5'
```

intersect() : The intersect method removes any values from the original collection that are not present in the given array or collection.

 The resulting collection will preserve the original collection's keys:

```
$collection = collect(['Desk', 'Sofa', 'Chair']);
$intersect = $collection->intersect(['Desk', 'Chair', 'Bookcase']);
$intersect->all();
// [0 => 'Desk', 2 => 'Chair']
```

isEmpty() : The isEmpty method returns true if the collection is empty; otherwise, false is returned:

```
collect([])->isEmpty(); // true
```

keyBy() : The keyBy method keys the collection by the given key. If multiple items have the same key, only the last one will appear in the new collection

```
$collection = collect([
['product_id' => 'prod-100', 'name' => 'desk'], ['product_id' => 'prod-200', 'name' => 'chair'],
]);
$keyed = $collection->keyBy('product_id');
$keyed->all();
OUTOUT:
/* [ 'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'], 'prod-200' => ['product_id' =>
'prod-200', 'name' => 'Chair'], ] */
```

You may also pass a callback to the method. The callback should return the value to key the collection by:

```
$keyed = $collection->keyBy(function ($item) { return
strtoupper($item['product_id']); });
$keyed->all();
OUTPUT :
/* [ 'PROD-100' => ['product_id' => 'prod-100', 'name' => 'Desk'], 'PROD-
200' => ['product_id' => 'prod-200', 'name' => 'Chair'], ] */
```

keys(): The keys method returns all of the collection's keys:

```
  collection = collect([ 'prod-100' => ['product_id' => 'prod-100', 'name' =>
'Desk'], 'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'], ]);
$keys = $collection->keys();
$keys->all();
OUTPUT : // ['prod-100', 'prod-200']
```

last() : The last method returns the last element in the collection that passes a given truth test:

```
collect([1, 2, 3, 4])->last(function ($value, $key)
{ return $value < 3; });
OUTPUT : // 2
```

You may also call the last method with no arguments to get the last element in the collection. If the collection is empty, null is returned

```
collect([1, 2, 3, 4])->last();
OUTPUT : // 4
```

map(): The map method iterates through the collection and passes each value to the given callback.The callback is free to modify the item and return it, thus forming a new collection of modified items:

```
$collection = collect([1, 2, 3, 4, 5]);
$multiplied = $collection->map(function ($item, $key) { return $item * 2; });
$multiplied->all();
OUTPUT : // [2, 4, 6, 8, 10]
```

mapWithKeys() : The mapWithKeys method iterates through the collection and passes each value to the given callback.The callback should return an associative array containing a single key / value pair

```
$collection = collect([ [ 'name' => 'John', 'department' => 'Sales', 'email' =>
'john@example.com' ], [ 'name' => 'Jane', 'department' => 'Marketing', 'email'
=> 'jane@example.com' ] ]);
$keyed = $collection->mapWithKeys(function ($item) { return [$item['email']
=> $item['name']]; });
$keyed->all();
OUTPUT : /* [ 'john@example.com' => 'John', 'jane@example.com' => 'Jane',
] */
```

max(): The max method returns the maximum value of a given key

```
$max = collect([['foo' => 10], ['foo' => 20]])->max('foo');
// 20
$max = collect([1, 2, 3, 4, 5])->max();
// 5
```

merge() : The merge method merges the given array into the original collection.
If a string key in the given array matches a string key in the original collection, the given array's value will overwrite the value in the original collection:

```
$collection = collect(['product_id' => 1, 'price' => 100]);
$merged = $collection->merge(['price' => 200, 'discount' => false]);
$merged->all();
OUTPUT : // ['product_id' => 1, 'price' => 200, 'discount' => false]
```

If the given array's keys are numeric, the values will be appended to the end of the collection:

```
$collection = collect(['Desk', 'Chair']);
$merged = $collection->merge(['Bookcase', 'Door']);
$merged->all();
OUTPUT : // ['Desk', 'Chair', 'Bookcase', 'Door']
```

min() : The min method returns the minimum value of a given key:

```
$min = collect([['foo' => 10], ['foo' => 20]])->min('foo');
// 10
$min = collect([1, 2, 3, 4, 5])->min();
// 1
```

only() : The only method returns the items in the collection with the specified keys:

```
$collection = collect(['product_id' => 1, 'name' => 'Desk', 'price' => 100,
'discount' => false]);
$filtered = $collection->only(['product_id', 'name']);
$filtered->all();
OUTPUT :// ['product_id' => 1, 'name' => 'Desk']
```

pipe(): The pipe method passes the collection to the given callback and returns the result:

```
$collection = collect([1, 2, 3]);
$piped = $collection->pipe(function ($collection) { return $collection-
>sum(); });
OUT PUT :// 6
```

pluck() : The pluck method retrieves all of the values for a given key:

```
$collection = collect([ ['product_id' => 'prod-100', 'name' => 'Desk'],
['product_id' => 'prod-200', 'name' => 'Chair'], ]);
$plucked = $collection->pluck('name');
$plucked->all();
OUTPUT : // ['Desk', 'Chair']
```

You may also specify how you wish the resulting collection to be keyed

```
$plucked = $collection->pluck('name', 'product_id');
$plucked->all();
OUTPUT : // ['prod-100' => 'Desk', 'prod-200' => 'Chair']
```

pop(): The pop method removes and returns the last item from the collection

```
$collection = collect([1, 2, 3, 4, 5]);
$collection->pop();
// 5
```

```
$collection->all();
// [1, 2, 3, 4]
```

prepend() : The prepend method adds an item to the beginning of the collection:

```
$collection = collect([1, 2, 3, 4, 5]);
$collection->prepend(0);
$collection->all();
// [0, 1, 2, 3, 4, 5]
```

You may also pass a second argument to set the key of the prepended item:

```
$collection = collect(['one' => 1, 'two', => 2]);
$collection->prepend(0, 'zero');
$collection->all();
OUTPUT : // ['zero' => 0, 'one' => 1, 'two', => 2]
```

pull() : The pull method removes and returns an item from the collection by its key:

```
$collection = collect(['product_id' => 'prod-100', 'name' => 'Desk']);
$collection->pull('name');
OUTPUT : // 'Desk'
$collection->all();
OUTPUT : // ['product_id' => 'prod-100']
```

push() : The push method appends an item to the end of the collection:

```
$collection = collect([1, 2, 3, 4]);
$collection->push(5);
$collection->all();
OUTPUT : // [1, 2, 3, 4, 5]
```

put() : The put method sets the given key and value in the collection

```
$collection = collect(['product_id' => 1, 'name' => 'Desk']);
$collection->put('price', 100);
$collection->all();
OUTPUT : // ['product_id' => 1, 'name' => 'Desk', 'price' => 100]
```

random() : The random method returns a random item from the collection

```
$collection = collect([1, 2, 3, 4, 5]);
$collection->random();
OUTPUT : // 4 - (retrieved randomly)
```

You may optionally pass an integer to random to specify how many items you would like to randomly retrieve

```
$random = $collection->random(3);
$random->all();
OUTPUT : // [2, 4, 5] - (retrieved randomly)
```

reduce() : The reduce method reduces the collection to a single value, passing the result of each iteration into the subsequent iteration

```
$collection = collect([1, 2, 3]);
$total = $collection->reduce(function ($carry, $item) { return $carry + $item;
});
OUTPUT : // 6
```

The value for $carry on the first iteration is null; however, you may specify its initial value by passing a second argument to reduce:

```
$collection->reduce(function ($carry, $item) { return $carry + $item; }, 4);
OUTPUT : // 10
```

reject() : The reject method filters the collection using the given callback. The callback should return trueif the item should be removed from the resulting
collection:

```
$collection = collect([1, 2, 3, 4]);
$filtered = $collection->reject(function ($value, $key) { return $value > 2; });
$filtered->all();
OUTPUT : // [1, 2]
```

For the inverse of the reject method, see the filter method.

reverse() : The reverse method reverses the order of the collection's items

```
$collection = collect([1, 2, 3, 4, 5]);
$reversed = $collection->reverse();
$reversed->all();
OUTPUT : // [5, 4, 3, 2, 1]
```

search() : The search method searches the collection for the given value and returns its key if found. If the item is not found, false is returned.

```
$collection = collect([2, 4, 6, 8]);
$collection->search(4);
OUTPUT : // 1
```

The search is done using a "loose" comparison, meaning a string with an integer value will be considered equal to an integer of the same value.
To use strict comparison, pass true as the second argument to the method:

```
$collection->search('4', true);
OUTPU : // false
```

Alternatively, you may pass in your own callback to search for the first item that passes your truth test

```
$collection->search(function ($item, $key) { return $item > 5; });
OUTPUT : // 2
```

shift() : The shift method removes and returns the first item from the collection:

```
$collection = collect([1, 2, 3, 4, 5]);
$collection->shift();
OUTPUT : // 1
$collection->all();
OUTPUT : // [2, 3, 4, 5]
```

shuffle() : The shuffle method randomly shuffles the items in the collection

```
$collection = collect([1, 2, 3, 4, 5]);
$shuffled = $collection->shuffle();
```

```
$shuffled->all();
OUTPUT : // [3, 2, 5, 1, 4] // (generated randomly)
```

slice() : The slice method returns a slice of the collection starting at the given index

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
$slice = $collection->slice(4);
$slice->all();
OUTPUT : // [5, 6, 7, 8, 9, 10]
```

If you would like to limit the size of the returned slice, pass the desired size as the second argument to the method

```
$slice = $collection->slice(4, 2);
$slice->all();
OUTPUT : // [5, 6]
```

The returned slice will preserve keys by default. If you do not wish to preserve the original keys, you can use the values method to reindex them

sort(): The sort method sorts the collection.

The sorted collection keeps the original array keys, so in this example we'll use the values method to reset the keys to consecutively numbered indexes

```
$collection = collect([5, 3, 1, 2, 4]);
$sorted = $collection->sort();
$sorted->values()->all();
OUTPUT : // [1, 2, 3, 4, 5]
```

sortBy() : The sortBy method sorts the collection by the given key. The sorted collection keeps the original array keys, so in this example we'll use the values method to reset the keys to consecutively numbered indexes:

```
$collection = collect([ ['name' => 'Desk', 'price' => 200], ['name' => 'Chair',
'price' => 100], ['name' => 'Bookcase', 'price' => 150], ]);
$sorted = $collection->sortBy('price');
$sorted->values()->all();
OUTPUT :
/* [ ['name' => 'Chair', 'price' => 100], ['name' => 'Bookcase', 'price' =>
150], ['name' => 'Desk', 'price' => 200], ] */
```

You can also pass your own callback to determine how to sort the collection values:

```
$collection = collect([ ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
['name' => 'Chair', 'colors' => ['Black']], ['name' => 'Bookcase', 'colors' =>
['Red', 'Beige', 'Brown']], ]);
$sorted = $collection->sortBy(function ($product, $key) { return
count($product['colors']); });
$sorted->values()->all();
OUTPUT :
/* [ ['name' => 'Chair', 'colors' => ['Black']], ['name' => 'Desk', 'colors' =>
['Black', 'Mahogany']], ['name' => 'Bookcase', 'colors' => ['Red', 'Beige',
'Brown']], ] */
```

sortByDesc() : This method has the same signature as the sortBy method, but will sort the collection in the opposite order.

splice() : The splice method removes and returns a slice of items starting at the specified index

```
$collection = collect([1, 2, 3, 4, 5]);
$chunk = $collection->splice(2);
$chunk->all();
// [3, 4, 5]
$collection->all();
// [1, 2]
```

You may pass a second argument to limit the size of the resulting chunk:

```
$collection = collect([1, 2, 3, 4, 5]);
$chunk = $collection->splice(2, 1);
$chunk->all();
// [3]
$collection->all();
// [1, 2, 4, 5]
```

In addition, you can pass a third argument containing the new items to replace the items removed from the collection:

```
$collection = collect([1, 2, 3, 4, 5]);
$chunk = $collection->splice(2, 1, [10, 11]);
$chunk->all();
// [3]
$collection->all();
// [1, 2, 10, 11, 4, 5]
```

split() : The split method breaks a collection into the given number of groups

```
$collection = collect([1, 2, 3, 4, 5]);
$groups = $collection->split(3);
$groups->toArray();
// [[1, 2], [3, 4], [5]]
```

sum() : The sum method returns the sum of all items in the collection:

```
collect([1, 2, 3, 4, 5])->sum();
// 15
```

take() : The take method returns a new collection with the specified number of items

```
$collection = collect([0, 1, 2, 3, 4, 5]);
$chunk = $collection->take(3);
$chunk->all();
// [0, 1, 2]
```

You may also pass a negative integer to take the specified amount of items from the end of the collection:

```
$collection = collect([0, 1, 2, 3, 4, 5]);
$chunk = $collection->take(-2);
```

```
$chunk->all();
OUTPUT : // [4, 5]
```

toArray() : The toArray method converts the collection into a plain PHP array. If the collection's values are Eloquent models, the models will also be converted to arrays:

```
$collection = collect(['name' => 'Desk', 'price' => 200]);
$collection->toArray();
OPTPUT : /* [ ['name' => 'Desk', 'price' => 200], ] */
```

toJson() : The toJson method converts the collection into JSON:

```
$collection = collect(['name' => 'Desk', 'price' => 200]);
$collection->toJson();
OUTPUT : // '{"name":"Desk", "price":200}'
```

transform() : The transform method iterates over the collection and calls the given callback with each item in the collection. The items in the collection will be replaced by the values returned by the callback:

```
$collection = collect([1, 2, 3, 4, 5]);
$collection->transform(function ($item, $key) { return $item * 2; });
$collection->all();
OUTPUT : // [2, 4, 6, 8, 10]
```

union() : The union method adds the given array to the collection.If the given array contains keys that are already in the original collection, the original collection's values will be preferred:

```
$collection = collect([1 => ['a'], 2 => ['b']]);
$union = $collection->union([3 => ['c'], 1 => ['b']]);
$union->all();
OUTPUT : // [1 => ['a'], 2 => ['b'], [3 => ['c']]
```

unique() : The unique method returns all of the unique items in the collection. The returned collection keeps the original array keys, so in this example we'll use the values method to reset the keys to consecutively numbered indexes

```
$collection = collect([1, 1, 2, 2, 3, 4, 2]);
$unique = $collection->unique();
$unique->values()->all();
OUTPUT : // [1, 2, 3, 4]
```

values() : The values method returns a new collection with the keys reset to consecutive integers:

```
$collection = collect([ 10 => ['product' => 'Desk', 'price' => 200], 11 =>
['product' => 'Desk', 'price' => 200] ]);
$values = $collection->values();
$values->all();
OUTPUT :
/* [ 0 => ['product' => 'Desk', 'price' => 200], 1 => ['product' => 'Desk', 'price'
=> 200], ] */
```

where() : The where method filters the collection by a given key / value pair

```
$collection = collect([ ['product' => 'Desk', 'price' => 200], ['product' =>
'Chair', 'price' => 100], ['product' => 'Bookcase', 'price' => 150], ['product' =>
'Door', 'price' => 100], ]);
$filtered = $collection->where('price', 100);
$filtered->all();
OUTPUT :
/* [ ['product' => 'Chair', 'price' => 100], ['product' => 'Door', 'price' => 100],
] */
```

The where method uses loose comparisons when checking item values.

whereStrict() : This method has the same signature as the where method; however, all values are compared using "strict" comparisons.

whereIn() : The whereIn method filters the collection by a given key / value contained within the given array

```
$collection = collect([ ['product' => 'Desk', 'price' => 200], ['product' => 'Chair',
'price' => 100], ['product' => 'Bookcase', 'price' => 150], ['product' => 'Door',
'price' => 100], ]);
$filtered = $collection->whereIn('price', [150, 200]);
$filtered->all();
OUTPUT :
/* [ ['product' => 'Bookcase', 'price' => 150], ['product' => 'Desk', 'price' => 200],
] */
```

The whereIn method uses "loose" comparisons when checking item values. Use the whereInStrictmethod to filter using strict comparisons.

zip() : The zip method merges together the values of the given array with the values of the original collection at the corresponding index:

```
$collection = collect(['Chair', 'Desk']);
$zipped = $collection->zip([100, 200]);
$zipped->all();
OUTPUT : // [['Chair', 100], ['Desk', 20
```

Pagination

Basic Usage

Paginating Query Builder Results: There are several ways to paginate items.

The simplest is by using the paginate method on the query builder or an Eloquent query. The paginate method automatically takes care of setting the proper limit and offset based on the current page being viewed by the user.By default, the current page is detected by the value of the page query string argument on the HTTP request. Of course, this value is automatically detected by Laravel, and is also automatically inserted into links generated by the paginator.

"Simple Pagination": If you only need to display simple "Next" and "Previous" links in your pagination view, you may use the simplePaginate method to perform a more efficient query. This is very useful for large datasets when you do not need to display a link for each page number when rendering your view

```
$users = DB::table('users')->simplePaginate(15);
```

Paginating Eloquent Results : You may also paginate Eloquent queries. In this example, we will paginate the User model with 15items per page.As you can see, the syntax is nearly identical to paginating query builder results

```
$users = App\User::paginate(15);
```

Of course, you may call paginate after setting other constraints on the query, such as whereclauses:

```
$users = User::where('votes', '>', 100)->paginate(15);
```

You may also use the simplePaginate method when paginating Eloquent models:

```
$users = User::where('votes', '>', 100)->simplePaginate(15);
```

Manually Creating A Paginator

Sometimes you may wish to create a pagination   instance manually, passing it an array of items. You may do so by creating either an Illuminate\Pagination\Paginator or Illuminate\Pagination\LengthAwarePaginator instance,depending on your needs
The Paginator class does not need to know the total number of items in the result set; however, because of this, the class does not have methods for retrieving the index of the last page. The LengthAwarePaginator accepts almost the same arguments as the Paginator; however, it does require a count of the total number of items in the result set.
In other words, the Paginator corresponds to the simplePaginate method on the query builder and Eloquent, while the LengthAwarePaginator corresponds to the paginate method.

Displaying Pagination Results

When calling the paginate method, you will receive an instance of Illuminate\Pagination\LengthAwarePaginator.
When calling the simplePaginate method, you will receive an instance of Illuminate\Pagination\Paginator.
These objects provide several methods that describe the result set. In addition to these helpers methods, the paginator   instances are iterators and may be looped as an array.So, once you have retrieved the results, you may   display the results and render the page links   using Blade:

```
<div class="container">
@foreach ($users as $user)
{{ $user->name }}
@endforeach
</div>
{{ $users->links() }}
```

The links method will render the links to the rest of the pages in the result set. Each of these links will already contain the proper page query string variable.

Customizing The Paginator URI

The setPath method allows you to customize the URI used by the paginator when generating links. For example, if you want the paginator to generate links like http://example.com/custom/url?page=N, you should pass custom/url to the setPath method
Route::get('users', function () { $users = App\User::paginate(15);
$users->setPath('custom/url'); // });

Appending To Pagination Links

You may append to the query string of pagination links using the appends method. For example, to append sort=votes to each pagination link, you should make the following call to appends:

```
{{ $users->appends(['sort' => 'votes'])->links() }}
```

If you wish to append a "hash fragment" to the paginator's URLs, you may use the fragment method. For example, to append #foo to the end of each pagination link, make the following call to the fragment method:

```
{{ $users->fragment('foo')->links() }}
```

Customizing The Pagination View

By default, the views rendered to display the pagination links are compatible with the Bootstrap CSS framework. However, if you are not using Bootstrap, you are free to define your own views to render these links. When calling the links method on a paginator instance, pass the view name as the first argument to the method
{{ $paginator->links('view.name') }}

However, the easiest way to customize the pagination views is by exporting them to your resources/views/vendor directory using the vendor:publish command:

```
php artisan vendor:publish --tag=laravel-pagination
```

This command will place the views in the resources/views/vendor/pagination directory The default.blade.php file within this directory corresponds to the default pagination view. Simply edit this file to modify the pagination HTML.

Paginator Instance Methods

Each paginator instance provides additional pagination information via the following methods:

```
$results->count()
 $results->currentPage()
 $results->firstItem()
 $results->hasMorePages()
 $results->lastItem()
 $results->lastPage() (Not available when using simplePaginate)
 $results->nextPageUrl()
 $results->perPage()
 $results->previousPageUrl()
 $results->total() (Not available when using simplePaginate)
 $results->url($page)
```

Mail

Laravel provides a clean, simple API over the popular SwiftMailer library with drivers for SMTP, Mailgun, SparkPost, Amazon SES, PHP's mail function and sendmail, allowing you to quickly get started sending mail through a local or cloud based service of your choice.

Driver Prerequisites

The API based drivers such as Mailgun and SparkPost are often simpler and faster than SMTP servers. If possible, you should use one of these drivers. All of the API drivers require the Guzzle HTTP library, which may be installed via the Composer package manager

```
composer require guzzlehttp/guzzle
```

Mailgun Driver: To use the Mailgun driver, first install Guzzle, then set the driver option in your config/mail.phpconfiguration file to mailgun. Next, verify that your config/services.php configuration file contains the following options:

```
'mailgun' => [
'domain' => 'your-mailgun-domain',
'secret' => 'your-mailgun-key', ],
```

SparkPost Driver:

To use the SparkPost driver, first install Guzzle, then set the driver option in your config/mail.phpconfiguration file to sparkpost.Next, verify that your config/services.php configuration file contains the following options:

```
'sparkpost' => [ 'secret' => 'your-sparkpost-key', ],
```

SES Driver :

To use the Amazon SES driver you must first install the Amazon AWS SDK for PHP. You may install this library by adding the following line to your composer.json file's require section and running the composer update command:

```
"aws/aws-sdk-php": "~3.0"
```

Next, set the driver option in your config/mail.php configuration file to ses and verify that your config/services.php configuration file contains the following options:

```
'ses' =>
[
'key' => 'your-ses-key',
'secret' => 'your-ses-secret',
'region' => 'ses-region', // e.g. us-east-1
],
```

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 Tops Technology');
- $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
```

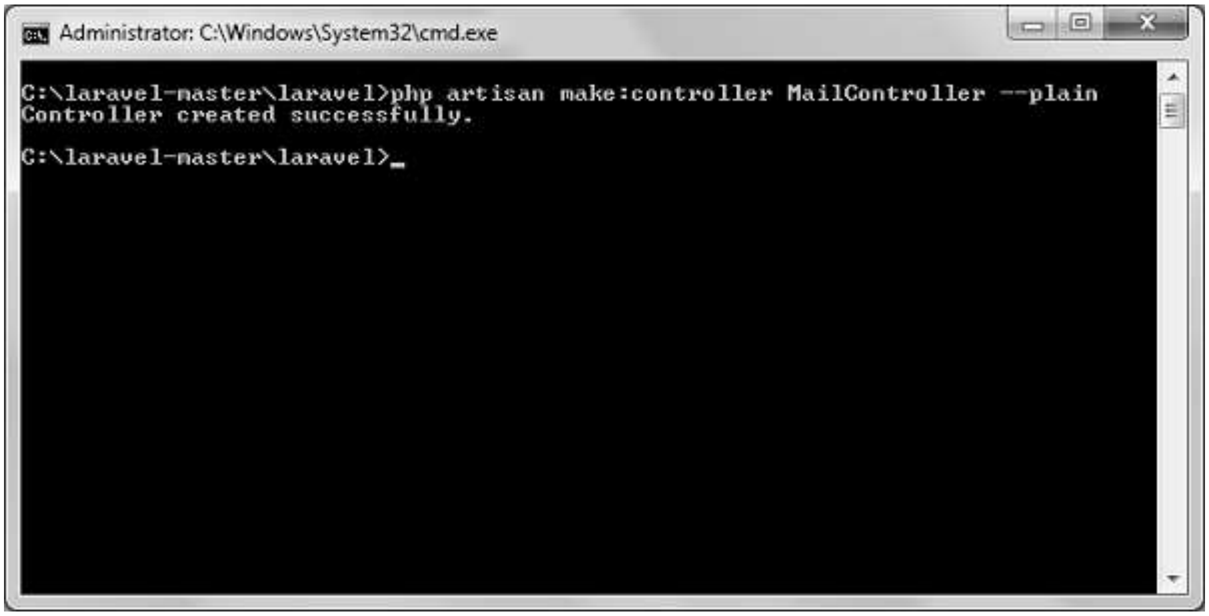
Step 2 − After changing the **.env** file execute the below two commands to clear the cache and restart the Laravel server.

```
php artisan config:cache
```

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

```
php artisan make:controller MailController
```

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



```
C:\laravel-master\laravel>php artisan make:controller MailController --plain
Controller created successfully.

C:\laravel-master\laravel>_
```

Step5 –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\image.png');
      $message->attach('C:\laravel-master\laravel\public\uploads\test.txt');
      $message->from('xyz@gmail.com','Virat Gandhi');
    });
    echo "Email Sent with attachment. Check your inbox.";
  }
}
```

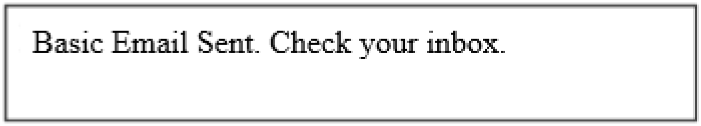
Step 6 – Copy the following code in **resources/views/mail.blade.php** file.
resources/views/mail.blade.php

```
<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

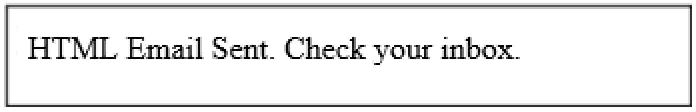
```
Route::get('sendbasicemail','MailController@basic_email');
Route::get('sendhtmlemail','MailController@html_email');
Route::get('sendattachmentemail','MailController@attachment_email');
```

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.



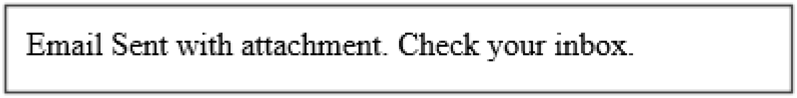
```
Basic Email Sent. Check your inbox.
```

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.



```
HTML Email Sent. Check your inbox.
```

Step 12 – Visit the following URL to test the HTML email with attachment.
http://localhost:8000/sendattachmentemail
Step 13 – The output screen will look something like this. Check your inbox to see the html email output with attachment.



```
Email Sent with attachment. Check your inbox.
```

Database Notifications :

Prerequisites

The database notification channel stores the notification information in a database table. This table will contain information such as the notification type as well as custom JSON data that describes the notification. You can query the table to display the notifications in your application's user interface. You may use the notifications:table command to generate a migration with the proper table schema:

```
php artisan notifications:table
php artisan migrate
```

Formatting Database Notifications:

If a notification supports being stored in a database table, you should define a toDatabase or toArray method on the notification class. This method will receive a $notifiable entity and should return a plain PHP array. The returned array will be encoded as JSON and stored in the datacolumn of your notifications table.

Let's take a look at an example toArray method:

```
public function toArray($notifiable)
{ return [ 'invoice_id' => $this->invoice->id, 'amount' => $this->invoice-
>amount, ]; }
```

toDatabase Vs. toArray

The toArray method is also used by the broadcast channel to determine which data to broadcast to your JavaScript client. If you would like to have two different array representations for the database and broadcast channels, you should define a toDatabase method

Accessing The Notifications

Once notifications are stored in the database, you need a convenient way to access them from your notifiable entities.The Illuminate\Notifications\Notifiable trait, which is included on Laravel's default App\User model, includes a notifications Eloquent relationship that returns the notifications for the entity. To fetch notifications, you may access this method like any other Eloquent relationship. By default, notifications will be sorted by the created_at timestamp

```
$user = App\User::find(1);
foreach ($user->notifications as $notification) { echo $notification->type; }
```

If you want to retrieve only the "unread" notifications, you may use the unreadNotificationsrelationship.

Again, these notifications will be sorted by the created_at timestamp Accessing The Notifications

```
$user = App\User::find(1);
foreach ($user->unreadNotifications as $notification) { echo $notification-
>type; }
```

Marking Notifications As Read

Typically, you will want to mark a notification as "read" when a user views it. The Illuminate\Notifications\Notifiable trait provides a markAsRead method, which updates the read_at column on the notification's database record:

```
$user = App\User::find(1);
foreach ($user->unreadNotifications as $notification)
{ $notification->markAsRead(); }
```

However, instead of looping through each notification, you may use the markAsRead method directly on a collection of notifications:

```
$user->unreadNotifications->markAsRead();
```

You may also use a mass-update query to mark all of the notifications as read without retrieving them from the database:

```
$user = App\User::find(1); $user->unreadNotifications()->update(['read_at' =>
Carbon::now()]);
```

Of course, you may delete the notifications to remove them from the table entirely:

```
$user->notifications()->delete();
```

Broadcast Notifications

Before broadcasting notifications, you should configure and be familiar with Laravel's event broadcasting services. Event broadcasting provides a way to react to server side fired Laravel events from your JavaScript client.

Formatting Broadcast Notifications

The broadcast channel broadcasts notifications using Laravel's event broadcasting services, allowing your JavaScript client to catch notifications in realtime. If a notification supports broadcasting, you should define a toBroadcast or toArray method on the notification class. This method will receive a $notifiable entity and should return a plain PHP array.The returned array will be encoded as JSON and broadcast to your JavaScript client. Let's take a look at an example toArray method

Listening For Notifications

Notifications will broadcast on a private channel formatted using a {notifiable}.{id} convention. So, if you are sending a notification to a App\User instance with an ID of 1, the notification will be broadcast on the App.User.1 private channel.When using Laravel Echo, you may easily listen for notifications on a channel using the notification helper method

```
Echo.private('App.User.' + userId)
.notification((notification) => { console
.log(notification.type); });
```

SMS Notifications

Sending SMS notifications in Laravel is powered by Nexmo. Before you can send notifications via Nexmo, you need to install the nexmo/client Composer package and add a few configuration options to your config/services.php configuration file. you may copy the example configuration below to get started:

```
'nexmo' =>
[ 'key' => env('NEXMO_KEY'),
'secret' => env('NEXMO_SECRET'),
'sms_from' => '15556666666', ],
```

The sms_from option is the phone number that your SMS messages will be sent from. You should generate a phone number for your application in the Nexmo control panel.

Formatting SMS Notifications

If a notification supports being sent as a SMS, you should define a toNexmo method on the notification class. This method will receive a $notifiable entity and should return a Illuminate\Notifications\Messages\NexmoMessage instance

```
public function toNexmo($notifiable) {
return (new NexmoMessage) ->content('Your SMS message content'); }
```

Customizing The "From" Number

If you would like to send some notifications from a phone number that is different from the phone number specified in your config/services.php file, you may use the from method on a NexmoMessage instance:

```
public function toNexmo($notifiable)
{
return (new NexmoMessage) ->
content('Your SMS message content') ->
from('15554443333'); }
```

Routing SMS Notifications

When sending notifications via the nexmo channel, the notification system will automatically look for a phone_number attribute on the notifiable entity. If you would like to customize the phone number the notification is delivered to, define a routeNotificationForNexmo method on the entity

```
<?php
namespace App;
use Illuminate\Notifications\Notifiable;
use Illuminate\Foundation\Auth\User as Authenticatable;
class User extends Authenticatable
{ use Notifiable;
/** * Route notifications for the Nexmo channel. *
@return string */
public function routeNotificationForNexmo()
{ return $this->phone; }
}
```