# Installation

## XAMPP

XAMPP is an open-source software package created to provide an easy way for web developers to **install, configure and manage** necessary components to develop and run PHP-based web-based applications. The ultimate goal is to provide a full-fledged **local web server** to work on.

It contains four main components in a cross-platform (**X**) manner:

* **A**pache web server
* **M**ySQL database server
* **P**HP language
* **P**erl language

Besides, XAMPP also provides phpMyAdmin, MariaDB, FileZilla FTP server, Fake Sendmail, Mercury Mail server, JSP server Tomcat, Webalizer.

For more details, check following posts:

* How to install at [here](https://pureinfotech.com/install-xampp-windows-10/#install_xampp_windows10).
* How to configure at [here](https://pureinfotech.com/install-xampp-windows-10/#configure_xampp_windows10).

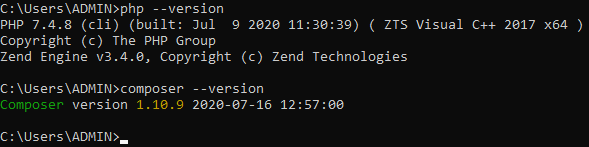
## Composer

Composer is the most popular **dependency manager for PHP**. We'll use it to install Laravel framework and many necessary dependencies for PHP-based webs/applications.

* How to install at [here](https://getcomposer.org/download/).
* Full list of CLIs at [here](https://getcomposer.org/doc/03-cli.md).
* Most common CLIs at [this section](#_Composer_CLIs).

**Note**: Composer automatically add itself and PHP to the PATH environment variable.





## Laravel Framework

For how to install it, check [here](https://laravel.com/docs/7.x#installing-laravel).

# Composer CLIs

## require

**Usage**: Adds new package(s) to the composer.json file from the current directory. Then install / update the package(s). If composer.json does not exist, create it.

**Syntax**: composer require <package(s)>

**Example**:

If you do not want to choose requirements interactively, you can pass them to the command: composer require "vendor/package:2.\*" vendor/package2:dev-master

**Tip**: If you do not specify any package, composer will prompt you to search for a package, and given results, provide a list of matches to require.

## update

**Usage**: Download and install the latest versions of the dependencies described in composer.json file. Then update the composer.lock file.

If no composer.json is found, show error.

We usually use this command to retrieve packages when cloning from Git repo, as we don't commit folder vendor to reduce size.

**Syntax**: composer update [specific-package(s)]

**Examples:**

* Update all packages: composer update
* Only update package1 and package2: composer update vendor/package1 vendor/package2
* Update a bunch of packages at once: composer update "vendor/\*"

**Tip**: This command is also aliased as upgrade as it does the same as upgrade does if you are thinking of apt-get or similar package managers.

## remove

**Usage**: Removes package(s) from the composer.json file from the current directory. Then remove the package(s) from the disk.

**Syntax**: composer remove <package(s)>

## show

**Usage**: Lists all of the available packages.

**Syntax**: composer show [specific-packages(s)]

**Example**:

* Use wildcard: composer show monolog/\*

# Artisan CLIs

<https://laravel.com/docs/7.x/artisan>

# Welcome Page

Laravel always include a default project with welcoming message to their framework once we install it. We'll create this project, name it welcome and put in the htdocs directory of XAMPP (e.g. C:\xampp\htdocs).

**1.** Run following commands to create the welcome project:

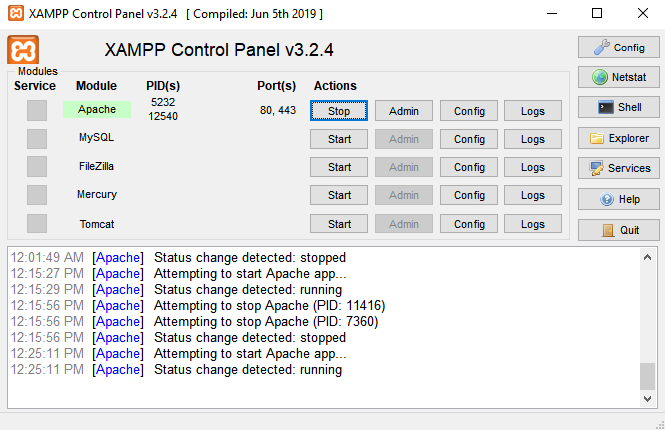
cd C:\xampp\htdocs

composer create-project --prefer-dist laravel/laravel welcome 5.7

**Note**:

* Here we use composer and specify version 5.7 for Laravel. If you want to install the latest version, just remove 5.7 from the command.
* The command not only creates the new project but also pull all source code, dependencies and configurations of Laravel from Github.

**2.** Start Apache server from XAMPP control panel using the default ports:

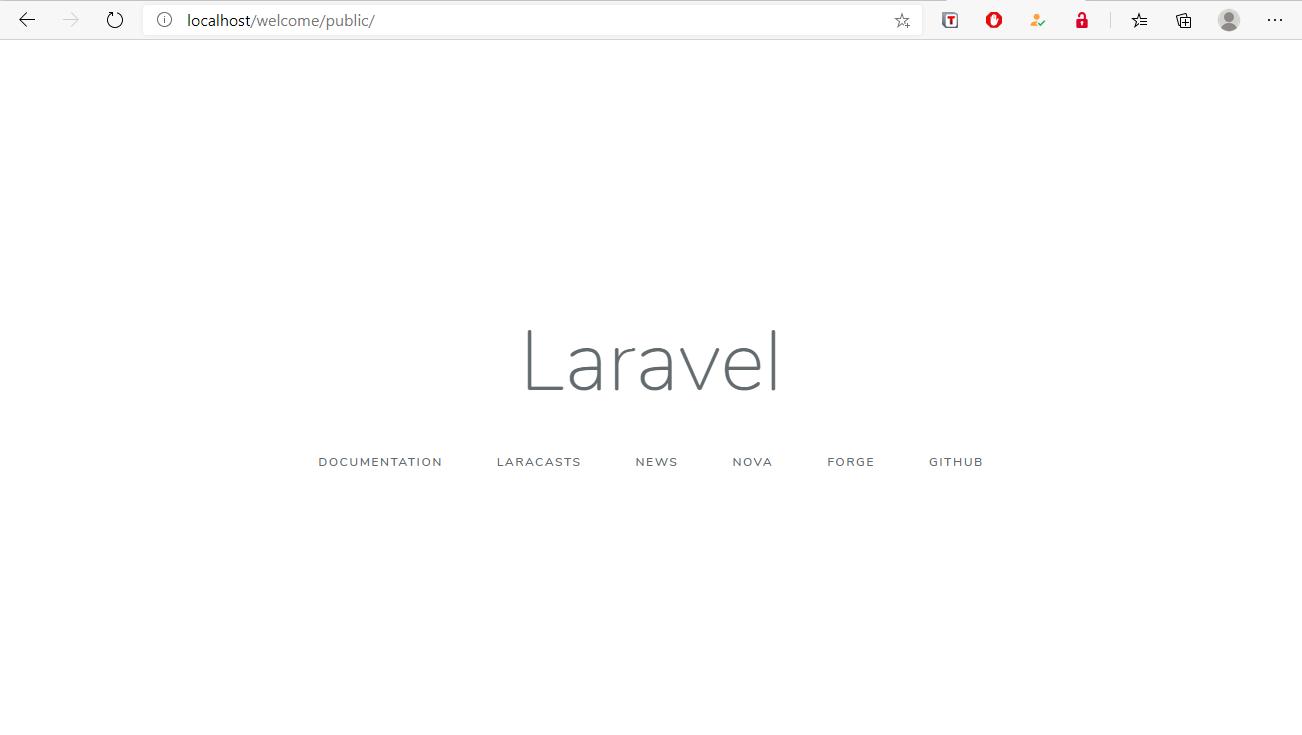


**3.** Now we can open the welcome's main page by running the following URL in the web browser:

localhost/welcome/public

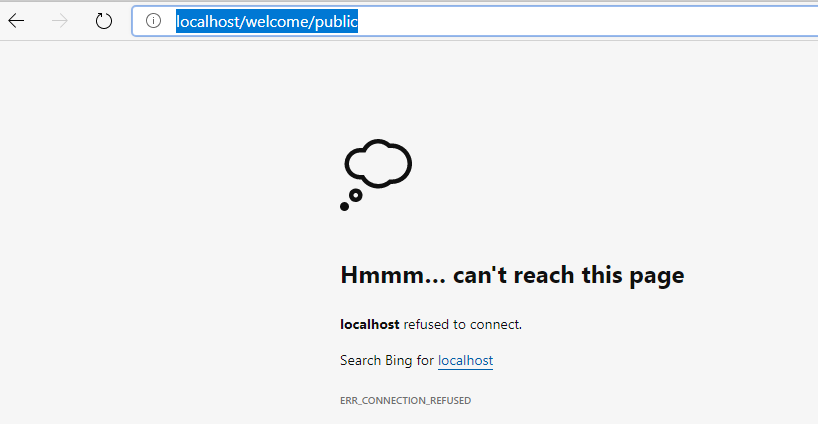
**Note**: Here localhost is our local web server (we can change it to 127.0.0.1) and welcome is the name of the project. We also need to add public at the end.

Here is the result:

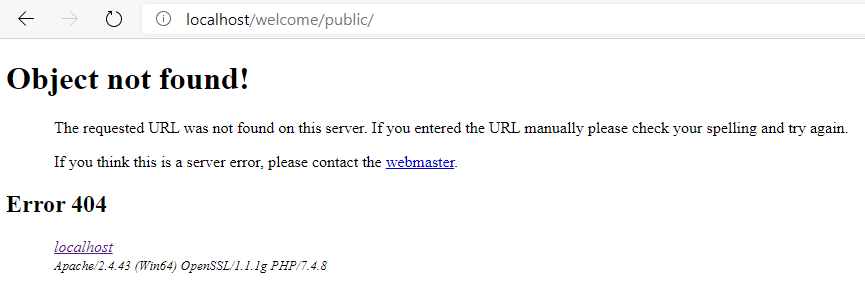


**Warning**:

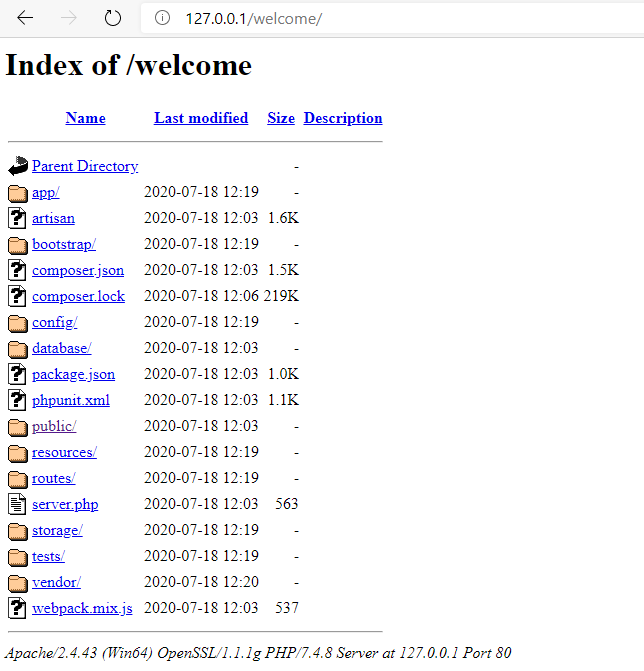
If you forget to start Apache web server, you'll get:



If you started Apache but forget to put welcome project to htdocs, you'll get:



If you forget to add /public, you'll get:



# Directory Structure in Laravel Project

For details, check <https://laravel.com/docs/5.7/structure>

## Directory Structure

|  |  |
| --- | --- |
| **Directory** | **Description** |
| app | Contains the **base code** for a Laravel application.  More detail at [this section](#_Detailed_'app'_Directory). |
| bootstrap | Contains all **bootstrapping scripts** used for the application. |
| config | Contains all project **configuration** files (.config). |
| database | Contains all **database** files.  More detail at [this section](#_Detailed_'database'_Directory). |
| public | Helps in starting the Laravel project and also holds other scripts (**JavaScript** and **CSS**), along with **images** required for the project. |
| resources | Contains all the **Sass** (Software as a service) files, **language** (localization) files, **templates** (if any). |
| routes | Contains all your definition files for routing such as web.php, console.php, api.php, channels.php, etc. |
| storage | Contains session files, cache, compiled templates, and miscellaneous files **generated by the framework**. |
| test | Contains all **test cases**. |
| vendor | Contains all composer **dependency files**. |

### Detailed 'app' Directory Structure

|  |  |
| --- | --- |
| **Directory** | **Description** |
| Console | Contains all project artisan commands. |
| Exceptions | Contains project's **exception handling files**, which handles all the exceptions thrown by the project. |
| Http | Contains different middleware (for filtering HTTP requests) and **controllers**. |
| Providers | Contains different **service providers**. |
| Broadcasting | Contains all of the **broadcast channel** classes.  It does not get created initially, instead run this artisan command:  php artisan make:channel |
| Jobs | Contains all lineup jobs in this directory.  It does not get created initially, instead run this artisan command:  php artisan make:job |
| Events | Contains **event files** that the application may pop up. Events are used for sending messages or signals to other parts of the Laravel project that any action had taken place within the project.  It does not get created initially, instead run these artisan commands:  php artisan make:generate  php artisan make:event |
| Listeners | Contains all project's handler classes used for **receiving and handling events**.  It does not get created initially, instead run these artisan commands:  php artisan make:generate  php artisan make:listener |
| Mail | Contains all emails send by through the project.  It does not get created initially, instead run this artisan command:  php artisan make:mail |
| Notifications | Contains all transactional notifications sent through the project.  It does not get created initially, instead run this artisan command:  php artisan make:notification |
| Policies | Contains different policies for the Laravel project.  It does not get created initially, instead run this artisan command:  php artisan make:policy |
| Rules | Contains different objects related to custom validation rules.  It does not get created initially, instead run this artisan command:  php artisan make:rule |

### Detailed 'database' Directory Structure

|  |  |
| --- | --- |
| **Directory** | **Description** |
| Seeds | Contains classes used for unit testing database. |
| Migrations | Helps in queries for migrating the database used in the web application. |
| Factories | Is used to generate large number of data records. |

## File

|  |  |
| --- | --- |
| **File** | **Description** |
| artisan | Command line interface used in Laravel. It includes a set of commands which assists in building a web application. |
| .env |  |
| .env.example | Backup version of .env file. |
| composer.json |  |

# Routing

Routing in Laravel allows you to **route all your application requests to its appropriate controller**. The main and primary routes in Laravel acknowledge and accept a URI (Uniform Resource Identifier) along with a closure, given that it should have to be a simple and expressive way of routing.

## Example

In routes/web.php:

use Illuminate\Support\Facades\Route;

Route::get('/', function() {

    return view('welcome');

});

In resources/views/welcome.blade.php:

<html>

<head>

   <title>Welcome</title>

</head>

<body>

   <h2>Welcome to Laravel</h2>

</body>

</html>

From the above example, you can see that the routing mechanism takes place in 2 steps:

* The URL you run needs to be matched exactly with the method (either get(), post(), put() or delete()) defined in the web.php file.

In this case, it is the root URL (/) of the project.

* The method calls the view() function with the file name located in resources/views/, and add the extension blade.php at the time of calling.

In this case, it's resources/views/welcome.blade.php.

Now when you run the root URL of the project in the browser, you'll see the welcome page.

**Warning:**

If you get error: *Call to undefined method Illuminate\Routing\Route::get()*

Then this is because your import is wrong: use Illuminate\Routing\Route;

You actually don't have to import any class as Laravel registers a global alias Route.

If you want to import the right class, that would be: use Illuminate\Support\Facades\Route;

## Route Parameters

In many cases, you had to capture the parameters send through the URL. Laravel provides two ways of capturing the passed parameter:

**Required Parameter**

At times you had to work with a segment(s) of the URL (Uniform Resource Locator) in your project. Route parameters are encapsulated within a "{}" with alphabets inside.

The following example captures the ID of the customer or employee from the generated URL:

// Route::get('/home', function() {

//     return view('home');

// });

Route::get('home/{id}', function($id) {

    echo 'Emp '.$id;

});

Output:

* If you run http://localhost/myapp/public/home, you'll see the page with error 404.
* If you run http://localhost/myapp/public/home/01, you'll see the page with content "Emp 01".

**Optional Parameter**

There are many parameters which do not remain present within the URL, but the developers had to use them. So such parameters get indicated by a "?" following the name of the parameter.

For example:

// Route::get('/home', function() {

//     return view('home');

// });

Route::get('home/{name?}', function($name = 'guest') {

    echo $name;

});

Output:

* If you run http://localhost/myapp/public/home, you'll see the page with content "guest".
* If you run http://localhost/myapp/public/home/guest, you'll see the page with content "guest".
* If you run http://localhost/myapp/public/home/guest1, you'll see the page with content "guest1".

## Named Routes

Named route is used to give specific name to a route. The chaining of routes can be specified using name() method onto the route definition.

The following code shows an example for creating named routes with controller:

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

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

The idea of defining name() in routes is for easier code maintenance in the future, it's not mandatory.

Say for example you have few places which uses the route login, one fine day you update the route to user-login. You will have to find and update all the route being used, changing from url('login') to url('user-login').

If you have a route name defined, you will be using route('login'), when you update your route url, there's no need to update all the other files that you're using that route.

# Redirection

## Redirecting to Named Routes

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

For example:

In resources/views/test.php:

<html>

   <body>

      <h1>Example of Redirecting to Named Routes</h1>

   </body>

</html>

In routes/web.php:

Route::get('/test', ['as' => 'testing', function() {

    return view('test');

}]);

Route::get('/redirect', function() {

    return redirect()->route('testing');

});

Here we set up the route for test.php file. We have renamed it to testing. We have also set up another route redirect which will redirect the request to the named route testing.

Output:

Run <http://localhost/redirect>, then you will be redirected to <http://localhost/test>, and the page with content "Example of Redirecting to Named Routes" is displayed.

## Redirecting to Controller Actions

To redirect to controller actions, simply pass the controller and name of the action to the action method. If you want to pass a parameter, you can pass it as the second argument of the action method.

For example:

Execute the following command to create a controller called RedirectController:

php artisan make:controller RedirectController

In app/Http/Controllers/RedirectController.php:

<?php

namespace App\Http\Controllers;

class RedirectController extends Controller {

    public function index() {

       echo "Redirecting to controller's action";

    }

}

In routes/web.php:

Route::get('/rc', 'RedirectController@index');

Route::get('/redirectcontroller', function() {

   return redirect()->action('RedirectController@index');

});

Output:

Run <http://localhost/redirectcontroller>, then you will be redirected to <http://localhost/rc>, and the page with content "Redirecting to controller's action" is displayed.

# Views

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

## Example

In resources/views/test.blade.php:

<html>

   <body>

      <h1>Hello, World</h1>

   </body>

</html>

In routes/web.php:

Route::get('/test', function() {

    return view('test');

});

Output:

Run <http://localhost/test>, and the page with content "Hello, World" will be displayed.

## Passing Data to Views

To pass data to the views, we pass an array to view helper function. After passing an array, we can use the key to get the value of that key in the HTML file.

**For example:**

In resources/views/test.blade.php:

<html>

   <body>

      <h1>Welcome {{ $name }}</h1>

   </body>

</html>

In routes/web.php:

// The value of the key name will be passed to test.blade.php file

// and $name will be replaced by that value.

Route::get('/test', function() {

    return view('test', ['name' => 'John']);

});

Output:

Run <http://localhost/test>, and the page with content "Welcome John" will be displayed.

**Remember:**

To display data passed to views, wrap the variable in curly braces ( {{ $var-name }} ).

- {{ $var-name }} is the same as <?php echo $var-name; ?>

- In addition to {{ ... }} we also have {!! ... !!}. Here is the difference between them:

$name = "<strong>hello world<strong>";

{{ $name }} 🡪 **hello world**

{!! $name !!} 🡪 <strong>hello world<strong>

## Sharing Data with All Views

There is a need to pass data to all the views, not just a/some view(s). To do that, we use a method called share() – it takes two arguments, key and value. Typically share() can be called from boot method of service provider. We can use any service provider, AppServiceProvider or our own service provider.

For example:

In routes/web.php:

Route::get('/test', function() {

    return view('test');

});

 Route::get('/test2', function() {

    return view('test2');

});

In resources/views/test.blade.php and in resources/views/test2.blade.php:

<html>

   <body>

      <h1>Welcome {{ $name }}</h1>

   </body>

</html>

In app/Providers/AppServiceProvider.php:

<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider {

   /\*\*

      \* Bootstrap any application services.

      \*

      \* @return void

   \*/

   public function boot() {

      view()->share('name', 'John');

   }

   /\*\*

      \* Register any application services.

      \*

      \* @return void

   \*/

   public function register() {

      //

   }

}

Output:

Run <http://localhost/test> or <http://localhost/test2>, you will see two different pages but having the same content: "Welcome John".

# Controllers

In the MVC framework, the letter 'C' stands for Controller. It acts as a directing traffic between Views and Models.

## Creating a Controller

Run the below artisan command from your project working directory:

php artisan make:controller <controller-name>

This will create a plain constructor, located at app/Http/Controllers.

And it can be called from routes/web.php by the following syntax:

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

**For example:**

Create UserController controller:

php artisan make:controller UserController

The newly created app/Http/Controller/UserController.php will have following default content:

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller {

   //

}

## Restful Resource Controllers

Many times, web applications need to perform CRUD (Create, Read, Update, Delete) operations. Laravel makes this job easy by providing option --resource when creating controllers with artisan command:

php artisan make:controller MyController --resource

**For example:**

Create a resource controller and name it as ResourceController:

php artisan make:controller ResourceController --resource

Now open app/Http/Controllers/ResourceController.php, and you will see following available methods:

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class ResourceController extends Controller

{

    /\*\*

     \* Display a listing of the resource.

     \*

     \* @return \Illuminate\Http\Response

     \*/

    public function index()

    {

        //

    }

    /\*\*

     \* Show the form for creating a new resource.

     \*

     \* @return \Illuminate\Http\Response

     \*/

    public function create()

    {

        //

    }

    /\*\*

     \* Store a newly created resource in storage.

     \*

     \* @param  \Illuminate\Http\Request  $request

     \* @return \Illuminate\Http\Response

     \*/

    public function store(Request $request)

    {

        //

    }

    /\*\*

     \* Display the specified resource.

     \*

     \* @param  int  $id

     \* @return \Illuminate\Http\Response

     \*/

    public function show($id)

    {

        //

    }

    /\*\*

     \* Show the form for editing the specified resource.

     \*

     \* @param  int  $id

     \* @return \Illuminate\Http\Response

     \*/

    public function edit($id)

    {

        //

    }

    /\*\*

     \* Update the specified resource in storage.

     \*

     \* @param  \Illuminate\Http\Request  $request

     \* @param  int  $id

     \* @return \Illuminate\Http\Response

     \*/

    public function update(Request $request, $id)

    {

        //

    }

    /\*\*

     \* Remove the specified resource from storage.

     \*

     \* @param  int  $id

     \* @return \Illuminate\Http\Response

     \*/

    public function destroy($id)

    {

        //

    }

}

If you want to create route for each of these methods, you can use the resource() method to create all the routes (instead of creating each by each with the get() method). Here is the code:

Route::resource('/resources', 'ResourceController');

To check these routes, run this artisan command:

php artisan route:list

+--------+-----------+---------------------------+-------------------+----------------------------------------------------+------------------+

| Domain | Method | URI | Name | Action | Middleware |

+--------+-----------+---------------------------+-------------------+----------------------------------------------------+------------------+

...

| | GET|HEAD | resources | resources.index | App\Http\Controllers\ResourceController@index | web |

| | POST | resources | resources.store | App\Http\Controllers\ResourceController@store | web |

| | GET|HEAD | resources/create | resources.create | App\Http\Controllers\ResourceController@create | web |

| | GET|HEAD | resources/{resource} | resources.show | App\Http\Controllers\ResourceController@show | web |

| | PUT|PATCH | resources/{resource} | resources.update | App\Http\Controllers\ResourceController@update | web |

| | DELETE | resources/{resource} | resources.destroy | App\Http\Controllers\ResourceController@destroy | web |

| | GET|HEAD | resources/{resource}/edit | resources.edit | App\Http\Controllers\ResourceController@edit | web |

...

+--------+-----------+---------------------------+-------------------+----------------------------------------------------+------------------+

## Controller Middleware

Middleware can be used with controller, either assigned to controller’s route or within controller’s constructor.

**For example:**

In app\Http\Kernel.php, register your middleware:

protected $routeMiddleware = [

    ...

    'first' => \App\Http\Middleware\FirstMiddleware::class,

    'second' => \App\Http\Middleware\SecondMiddleware::class,

];

In routes\web.php:

Route::get('/usercontroller/path', [

    'middleware' => 'first', // Assigning middleware to route

    'uses' => 'UserController@showPath'

]);

In app\Http\Middleware\FirstMiddleware.php:

<?php

namespace App\Http\Middleware;

use Closure;

class FirstMiddleware

{

    /\*\*

     \* Handle an incoming request.

     \*

     \* @param  \Illuminate\Http\Request  $request

     \* @param  \Closure  $next

     \* @return mixed

     \*/

    public function handle($request, Closure $next)

    {

        echo '<br>First Middleware';

        return $next($request);

    }

}

In app\Http\Middleware\SecondMiddleware.php:

<?php

namespace App\Http\Middleware;

use Closure;

class SecondMiddleware

{

    /\*\*

     \* Handle an incoming request.

     \*

     \* @param  \Illuminate\Http\Request  $request

     \* @param  \Closure  $next

     \* @return mixed

     \*/

    public function handle($request, Closure $next)

    {

        echo '<br>Second Middleware';

        return $next($request);

    }

}

In app\Http\Controllers\UserController.php:

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller

{

    public function \_\_construct() {

// Assigning middleware within controller's constructor

        $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;

    }

}

Output:

First Middleware

Second Middleware

URI: usercontroller/path

URL: http://localhost/laravel\_training\_triho/public/usercontroller/path

Method: GET

# Request

## Getting the Request URI

The Request class provides some useful built-in functions to get URI data.

**For example:**

Create a UriController controller by running this artisan command:

php artisan make:controller UriController

In app/Http/Controllers/UriController.php:

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UriController extends Controller {

   public function index(Request $request) {

      // Get the requested URI

      $path = $request->path();

      echo 'Path Method: '.$path;

      echo '<br>';

      // Get the requested URI which matches the particular pattern specified in the argument of the method

      $pattern = $request->is('test/\*');

      echo 'is Method: '.$pattern;

      echo '<br>';

      // Get the full URL

      $url = $request->url();

      echo 'URL method: '.$url;

   }

}

In routes/web.php:

Route::get('/test/uri, 'UriController@index');

Output:

Run <http://localhost/laravel_training_triho/public/test/uri>, and the output will appear as following content:

Path Method: test/uri  
is Method: 1  
URL method: http://localhost/laravel\_training\_triho/public/test/uri

## Getting Input from Forms

The input values can be easily retrieved in Laravel no matter what method was used **GET** or **POST**. There are two ways we can retrieve the input values:

* Using the input('field-name') method of Request class
* Using the properties of Request class

**For example:**

In resources/views/register.blade.php:

<html>

Another way:

action = "{{ url('/user/register') }}"

   <head>

      <title>Form Example</title>

   </head>

   <body>

      <form action = "{{ action('RegisterController@postRegister') }}" method = "POST">

         <input type = "hidden" name = "\_token" value = "<?php echo csrf\_token() ?>">

         <table>

            <tr>

               <td>Name</td>

               <td><input type = "text" name = "name" /></td>

            </tr>

            <tr>

               <td>Username</td>

               <td><input type = "text" name = "username" /></td>

            </tr>

            <tr>

               <td>Password</td>

               <td><input type = "text" name = "password" /></td>

            </tr>

            <tr>

               <td colspan = "2" align = "center">

                  <input type = "submit" value = "Register" />

               </td>

            </tr>

         </table>

      </form>

   </body>

</html>

Create a RegisterController controller by running this artisan command:

php artisan make:controller RegisterController

In app/Http/Controllers/RegisterController.php:

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class RegisterController extends Controller {

    public function postRegister(Request $request) {

        // Retrieve the name input

        $name = $request->name;     // Same as: $name = $request->input('name');

        echo 'Name: '.$name;

        echo '<br>';

        // Retrieve the username input

        $username = $request->username;

        echo 'Username: '.$username;

        echo '<br>';

        // Retrieve the password input

        $password = $request->password;

        echo 'Password: '.$password;

    }

}

In routes/web.php:

Another way:

Route::post('/user/register', array(

    'uses' => 'RegisterController@postRegister'

));

Route::get('/user/register', function() {

    return view('register');

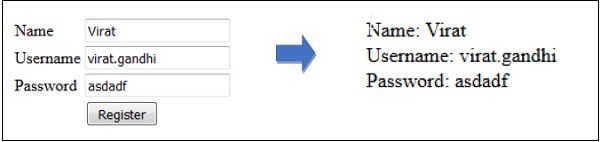
});

// Because we use POST method in register.blade.php, we have to use Route::post here too

Route::post('/user/register', 'RegisterController@postRegister');

Output:

Run <http://localhost/laravel_training_triho/user/register>, then enter registration details and click Register. You will see on the second page that we have retrieved and displayed the user registration details, as shown in below the following image:



# Response

A web application responds to a user’s request in many ways depending on many parameters. Laravel provides several different ways to return response. Response can be sent either from route or from controller.

## Basic Response

The basic response that can be sent is simple string as shown in the below sample code. This string will be automatically converted to appropriate HTTP response.

For example:

In routes/web.php:

Route::get('/basic\_response', function() {

    return 'Hello World';

});

Visit <http://localhost/basic_response>, and you'll see the page with content 'Hello World'.

## Attaching Headers

<https://www.tutorialspoint.com/laravel/laravel_response.htm>

## Attaching Cookies

<https://www.tutorialspoint.com/laravel/laravel_response.htm>

## JSON Response

<https://www.tutorialspoint.com/laravel/laravel_response.htm>

# Models

# Middleware

<https://www.w3schools.in/laravel-tutorial/middleware/>

# Namespaces

<https://www.tutorialspoint.com/laravel/laravel_namespaces.htm>

# Blade Templates

Blade is a *templating engine* provided with Laravel. When compared to other PHP templating engines, Blade is unique in the following ways:

* It does not restrict the developer from using plain PHP code in views.
* The blade views thus designed, are compiled and cached until they are modified. That means Blade adds essentially zero overhead to your application.

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

## Template Inheritance

### Example

**Example 1**: Inherit the master template but overwrite its content

First define a "master" template. Laravel already puts it in resources/views/layouts/app.blade.php:

<head>

    <meta charset="utf-8">

    <meta name="viewport" content="width=device-width, initial-scale=1">

    <!-- CSRF Token -->

    <meta name="csrf-token" content="{{ csrf\_token() }}">

    <title>{{ config('app.name', 'Laravel') }}</title>

    <!-- Fonts -->

    <link href="https://fonts.googleapis.com/css?family=Nunito" rel="stylesheet" type="text/css">

    <!-- Styles -->

    <link href="{{ asset('css/app.css') }}" rel="stylesheet">

</head>

<body>

    <div id="app">

        <nav class="navbar navbar-expand-md navbar-light navbar-laravel">

            <div class="container">

                <a class="navbar-brand" href="{{ url('/') }}">

                    Page title: @yield('title')

                </a>

            </div>

        </nav>

        <div class="container">

            <div>

                @section('table-of-content')

                    This is the master table of content.

                @show

            </div>

            <br/>

            <main class="py-4">

                @yield('content')

            </main>

        </div>

    </div>

</body>

</html>

**Note**: Here we changed the default app.blade.php a bit, making it much simpler, to fit our example.

Now in resources\views\test.blade.php:

@extends('layouts.app')

@section('title', 'Input the page title here')

@section('table-of-content')

    <p>This is the child table of content.</p>

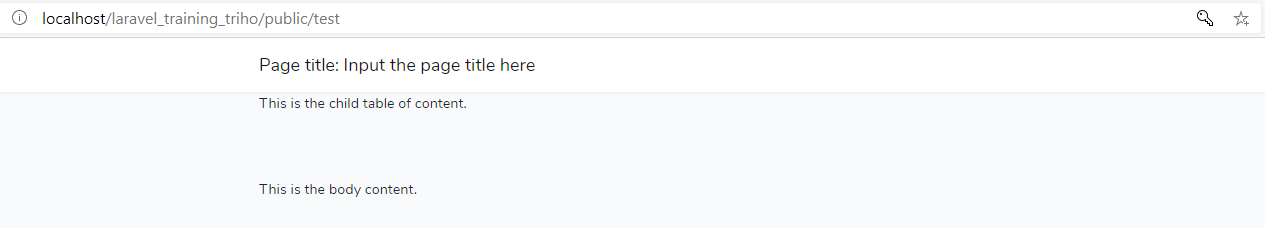
@endsection

@section('content')

    <p>This is the body content.</p>

@endsection

Output:



You can see that the child page inherits the template defined by the master template. **It does not display only plain text**, but have navigation bar, container, background and font although we do not put any CSS scripts to test.blade.php.

**Example 2**: Inherit the master template and append its content

In resources/views/layouts/app.blade.php, use the same source code as example 1.

In resources\views\test.blade.php, use the same source code as example 1. But add a @parent directive:

@extends('layouts.app')

@section('title', 'Input the page title here')

@section('table-of-content')

    @parent

    <p>This is the child table of content.</p>

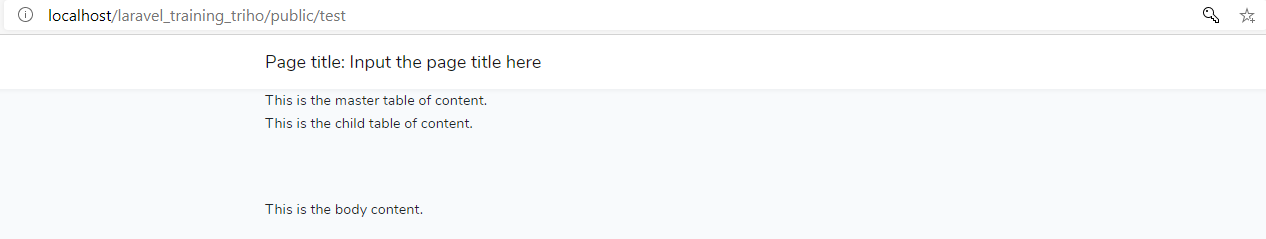
@endsection

@section('content')

    <p>This is the body content.</p>

@endsection

Output:



By using the @parent directive, we append (rather than overwriting) content to the section table of content.

### Passing Data to Templates

Passing data to Blade templates is same as [passing data to views](#_Passing_Data_to).

**For example:**

In resources/views/layouts/app.blade.php, use the same source code as example 1.

In resources\views\test.blade.php, use the same source code as example 1. But add a welcome message with variable $name:

@extends('layouts.app')

@section('title', 'Input the page title here')

@section('table-of-content')

    @parent

    <p>This is the child table of content.</p>

@endsection

@section('content')

    <p>This is the body content.</p>

    <p>Welcome {{ $name }}.</p>

@endsection

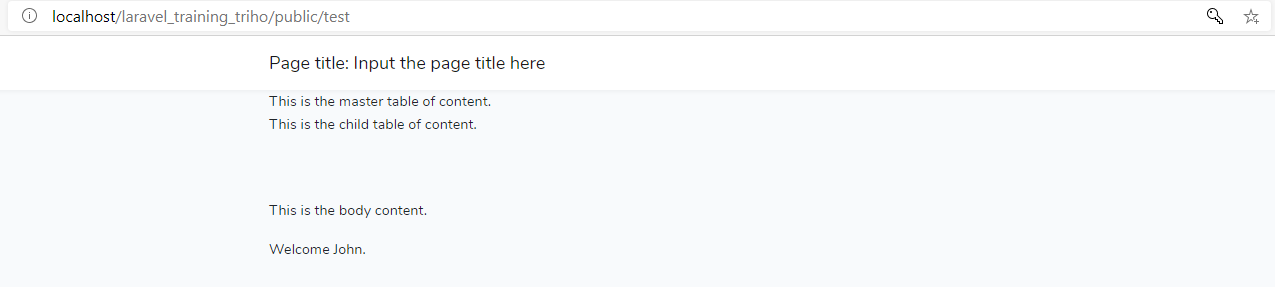
In routes/web.php, add a second parameter to method view():

Route::get('/test', function() {

    return view('test', ['name' => 'John']);

});

Output:



## Control Structures

<https://laravel.com/docs/5.7/blade#control-structures>

## Forms

<https://laravel.com/docs/5.7/blade#forms>

## Including Sub-Views

<https://laravel.com/docs/5.7/blade#including-sub-views>

# Working with DB

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

* MySQL
* Postgres
* SQLite
* SQL Server

## Connecting to Database

In this example, we use MySQL. We create a DB schema and name it "student\_db". In this schema, we create a table named "students" with following columns:

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Column Datatype** | **Extra** |
| id | int(11) | Primary key | Auto increment |
| name | varchar(25) |  |

In .env file:

DB\_CONNECTION=mysql

DB\_HOST=127.0.0.1

DB\_PORT=3306

DB\_DATABASE=student\_db

DB\_USERNAME=root

DB\_PASSWORD=

## Inserting / Retrieving / Updating / Deleting Records

In resources\views\student\_index.blade.php:

<html>

<head>

    <title>View Student Records</title>

</head>

<body>

    <table border="1">

        <tr>

            <td>ID</td>

            <td>Name</td>

            <td>Edit</td>

            <td>Delete</td>

        </tr>

        @foreach ($users as $user)

        <tr>

            <td>{{ $user->id }}</td>

            <td>{{ $user->name }}</td>

            <td><a href='{{ url('students/update') . '/' . $user->id }}'>Edit</a></td>

            <td><a href='{{ url('students/delete') . '/' . $user->id }}'>Delete</a></td>

        </tr>

        @endforeach

    </table>

</body>

</html>

In resources\views\student\_insert.blade.php:

<html>

<head>

    <title>Student Management | Add</title>

</head>

<body>

    <form action="{{ url('students/insert') }}" method="POST">

        <input type="hidden" name="\_token" value="{{ csrf\_token() }}">

        <table>

            <tr>

                <td>Name</td>

                <td><input type='text' name='stud\_name' /></td>

            </tr>

            <tr>

                <td colspan='2'>

                    <input type='submit' value="Add student" />

                </td>

            </tr>

        </table>

    </form>

</body>

</html>

In resources\views\student\_update.blade.php:

<html>

<head>

    <title>Student Management | Edit</title>

</head>

<body>

    <form action="{{ url('students/update') . '/' . $users[0]->id }}" method="POST">

        <input type="hidden" name="\_token" value="{{ csrf\_token() }}">

        <table>

            <tr>

                <td>Name</td>

                <td>

                    <input type='text' name='stud\_name' value='{{ $users[0]->name }}' />

                </td>

            </tr>

            <tr>

                <td colspan='2'>

                    <input type='submit' value="Update student" />

                </td>

            </tr>

        </table>

    </form>

</body>

</html>

In app\Http\Controllers\StudentController.php:

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use Illuminate\Support\Facades\DB;

use App\Http\Controllers\Controller;

class StudentController extends Controller {

    public function index() {

        $users = DB::select('select \* from students');

        return view('student\_index', ['users' => $users]);

    }

    public function insertForm() {

        return view('student\_insert');

    }

    public function insert(Request $request) {

        $stud\_name = $request->input('stud\_name');

        DB::insert('INSERT INTO students (name) values(?)', [$stud\_name]);

        echo "Record inserted successfully.<br/>";

        $back\_url = $request->getBaseUrl() . '/students';

        echo "<a href = $back\_url >Click Here</a> to go back.";

    }

    public function updateForm($id) {

        $users = DB::select('SELECT \* FROM students WHERE id = ?', [$id]);

        return view('student\_update', ['users' => $users]);

    }

    public function update(Request $request, $id) {

        $name = $request->input('stud\_name');

        DB::update('UPDATE students SET name = ? WHERE id = ?', [$name, $id]);

        echo "Record updated successfully.<br/>";

        $back\_url = $request->getBaseUrl() . '/students';

        echo "<a href = $back\_url >Click Here</a> to go back.";

    }

    public function delete(Request $request, $id) {

        DB::delete('DELETE FROM students WHERE id = ?', [$id]);

        echo "Record deleted successfully.<br/>";

        $back\_url = $request->getBaseUrl() . '/students';

        echo "<a href = $back\_url >Click Here</a> to go back.";

    }

}

In routes\web.php:

Route::get('/students', 'StudentController@index');

Route::post('/students', 'StudentController@index');

Route::get('/students/insert', 'StudentController@insertForm');

Route::post('/students/insert', 'StudentController@insert');

Route::get('/students/update/{id}', 'StudentController@updateForm');

Route::post('/students/update/{id}', 'StudentController@update');

Route::get('/students/delete/{id}', 'StudentController@delete');

**Output:**

Test insert feature:

Visit <http://localhost/laravel_training_triho/public/students/insert>, then add two students:



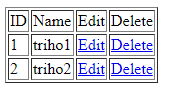
And



After click on Add Student, a page with following content is display:

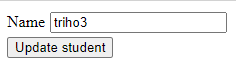


Click on Click Here to go back to <http://localhost/laravel_training_triho/public/students>, you will two newly added students:



Test update feature:

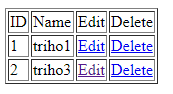
From the table, click Edit to change "triho2" to "triho3":



After click on Update Student, a page with following content is display:



Click on Click Here to go back to <http://localhost/laravel_training_triho/public/students>, you will see "triho2" was changed to "triho3":



Test delete feature:

From the table, click Delete for "triho3":

Once done, a page with following content is display:



Click on Click Here to go back to <http://localhost/laravel_training_triho/public/students>, you will see "triho3" was deleted



# Forms

So far, we're using pure HTML tags to create forms and buttons. Laravel also provides various built-in classes and methods to handle all the major elements of HTML easily and securely.

<https://www.tutorialspoint.com/laravel/laravel_forms.htm>

# Localization

<https://www.tutorialspoint.com/laravel/laravel_localization.htm>