

Step by Step on How to Build a Simple Laravel 10 Website from Ground Zero

This article is dedicated to beginners finding it difficult to get started with Laravel. Keep calm and follow through.

What is Laravel?

Laravel is an easy-to-use web framework that will help you create extensible PHP-based websites and web applications at scale.

Pre-requisites

You need to be good at HTML, core PHP, and advanced PHP (still possible to skip).

STEP 1: Download and Install

The obvious first step is for you to download and set up all the necessary software that you would need for this course. The following are the list of software you need to download and install on your computer:

1. Node.js <https://nodejs.org/en>
2. Composer <https://getcomposer.org/>
3. wamp/xampp/lamp <https://www.wampserver.com/en/> or <https://www.apachefriends.org/>
4. Text editor (preferably VS Code) <https://code.visualstudio.com/>

Node.js

This would assist us in running some command lines.

Composer

According to their docs, 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.

WAMP/XAMPP/LAMP

You can use any of the above software, but not two or more at a time.

WampServer is a Web development platform on Windows that allows you to create dynamic Web applications with Apache2, PHP, MySQL

and MariaDB. WampServer automatically installs everything you need to intuitively develop Web applications.

XAMPP is a short form for Cross-Platform, Apache, MySQL, PHP, and Perl. XAMPP is a free and open-source cross-platform web server. XAMPP is simply a local host or server that is used to test clients or websites before publishing them to a remote web server. The XAMPP server software on a local computer provides an appropriate environment for testing MYSQL, PHP, Apache, and Perl projects.

LAMP stands for Linux, Apache, MySQL, and PHP. Together, they provide a proven set of software for delivering high-performance web applications.

VS Code

Visual Studio Code is a streamlined code editor with support for development operations like debugging, task running, and version control.

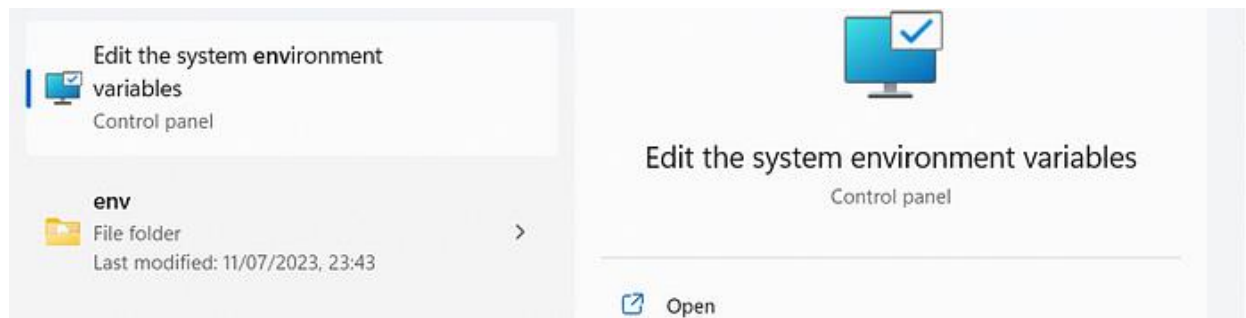
Final Note: Make sure you read about the software and their functions before installing them. You can also go on YouTube to check how to install each of the software.

Once you're done installing the software listed above, you can then proceed to step 2.

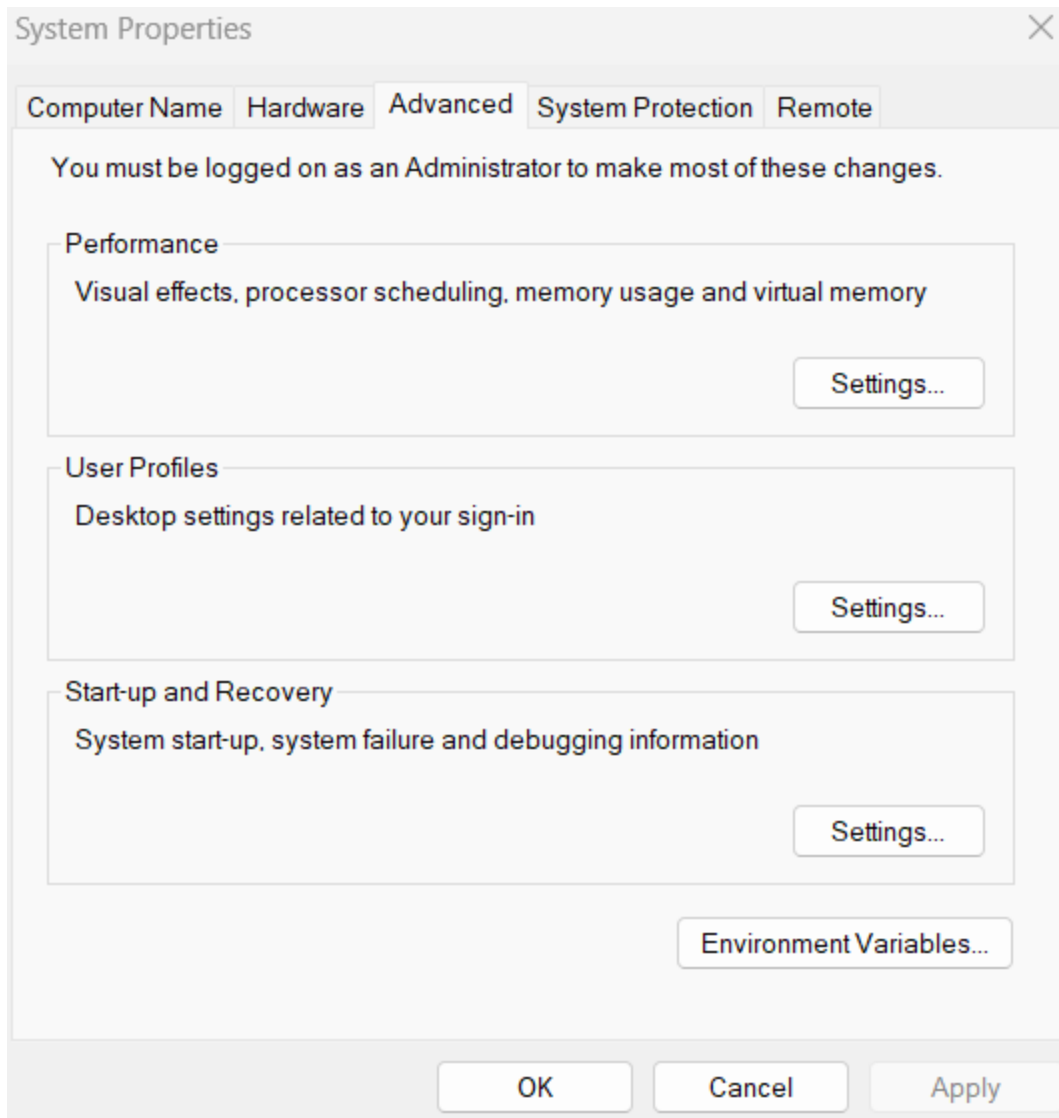
STEP 2: Environment Variables

On your Windows' computer search box, type **env**, click on **edit the system environment variables**, then select **path** in the system variables box, click on edit.

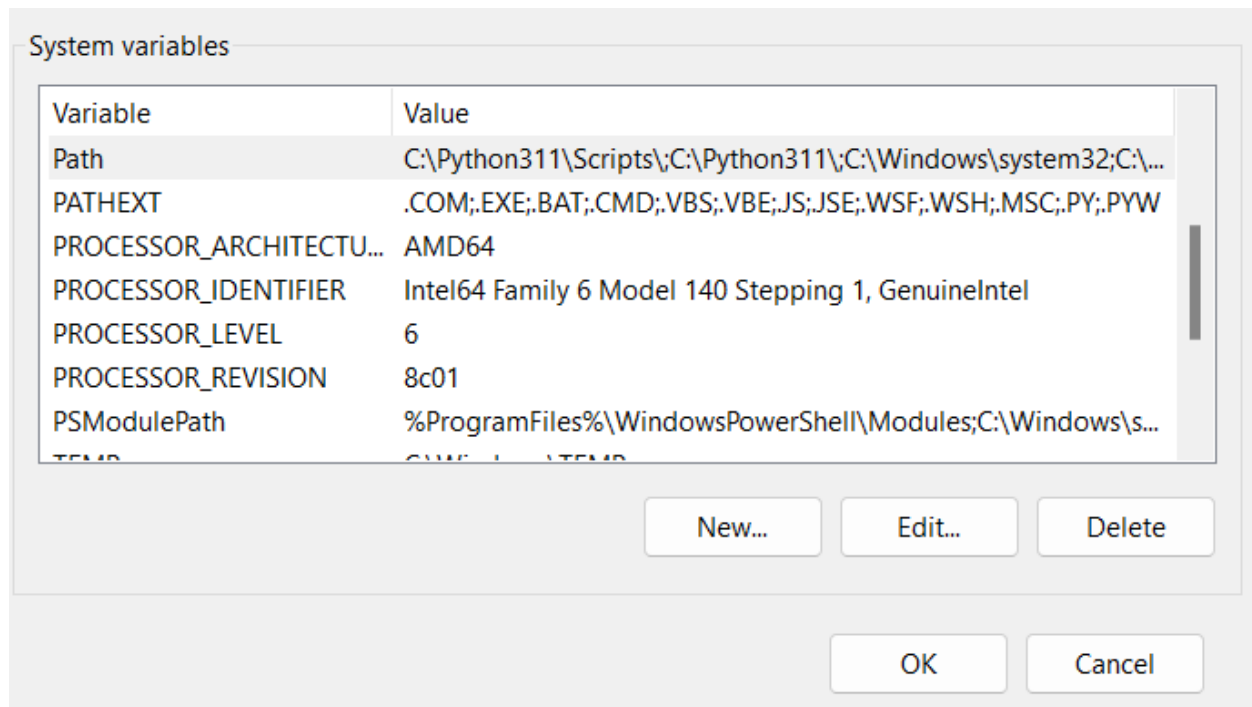
1.



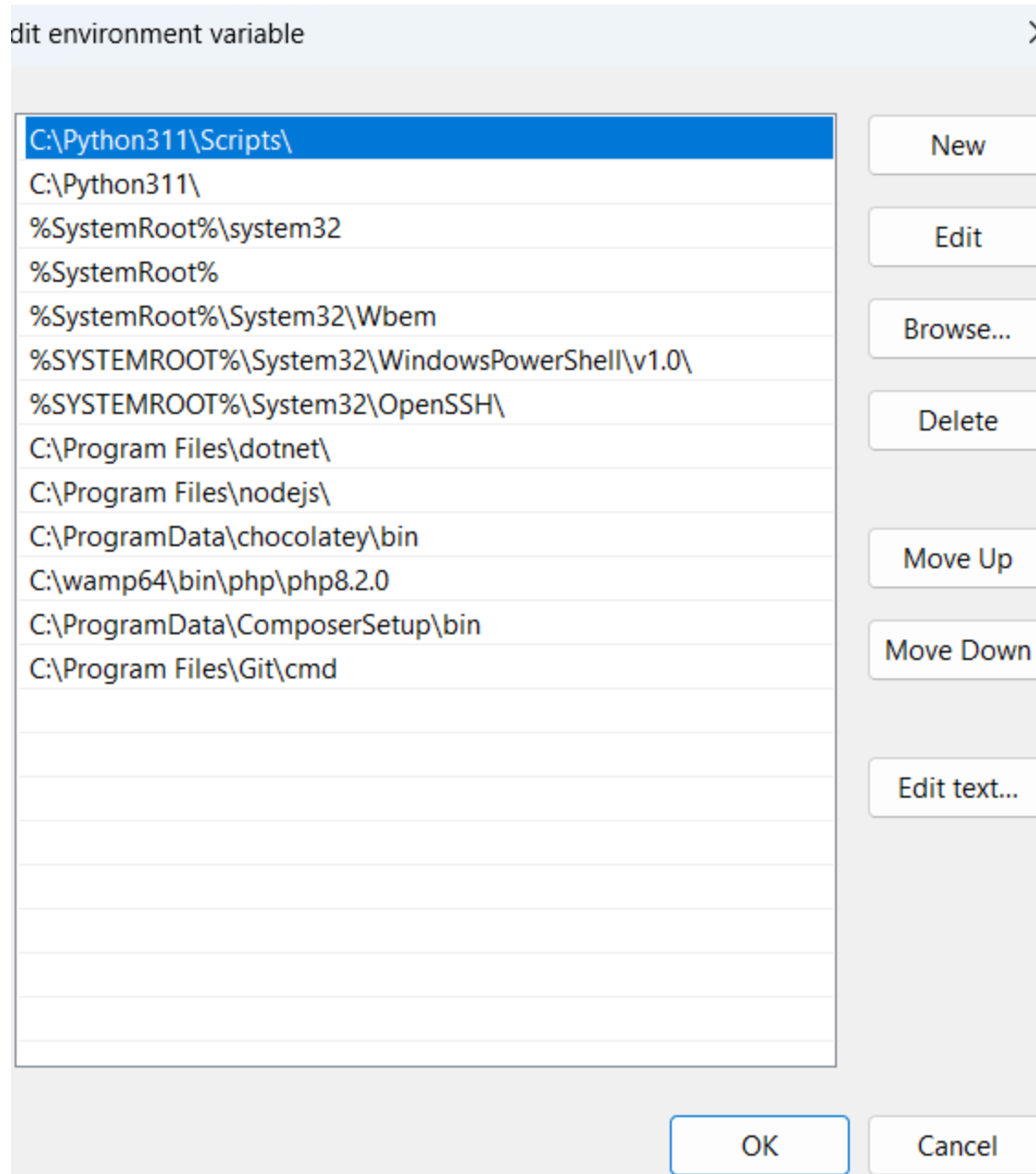
2.



3.



4.



Go to this path on your computer `C:\wamp64\bin\php\php8.2.0`

I am currently using php8.2.0, but yours might be different, if it is, it doesn't affect what we are about to do. Once you are able to locate this

path on your system, copy the path and add it as new in the dialog box that popped up.

Check if your composer has also been added to the system variable, if it's not, add it too.

STEP 3: Check Versions

Launch your node.js, type **php -v** to get the version of php you're using. For composer, type **composer -v**. If the versions are not returned, it's probably that you did not install the software correctly or they are not available in the system environment. When you check the version of composer, you will see something like the image below:



STEP 4: Install Laravel

Launch your node.js command prompt, type the following commands:

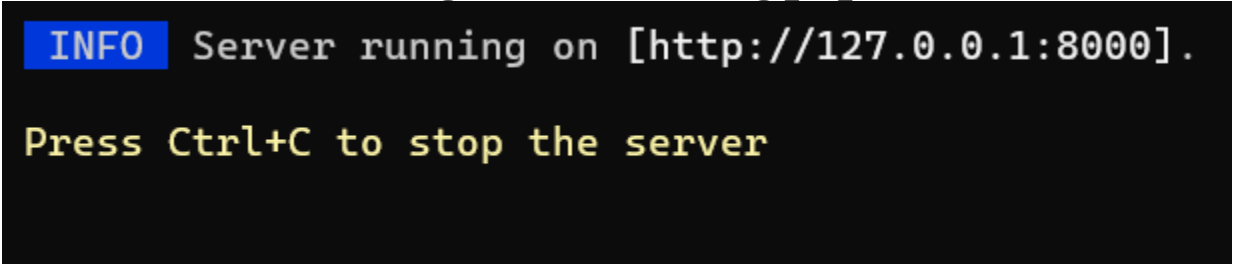
```
$ composer create-project laravel/laravel simple-project
```



```
$ cd simple-project
```

```
$ php artisan serve
```

You should see something like after running **php artisan serve**



```
INFO Server running on [http://127.0.0.1:8000].  
Press Ctrl+C to stop the server
```

Quick Explanation:

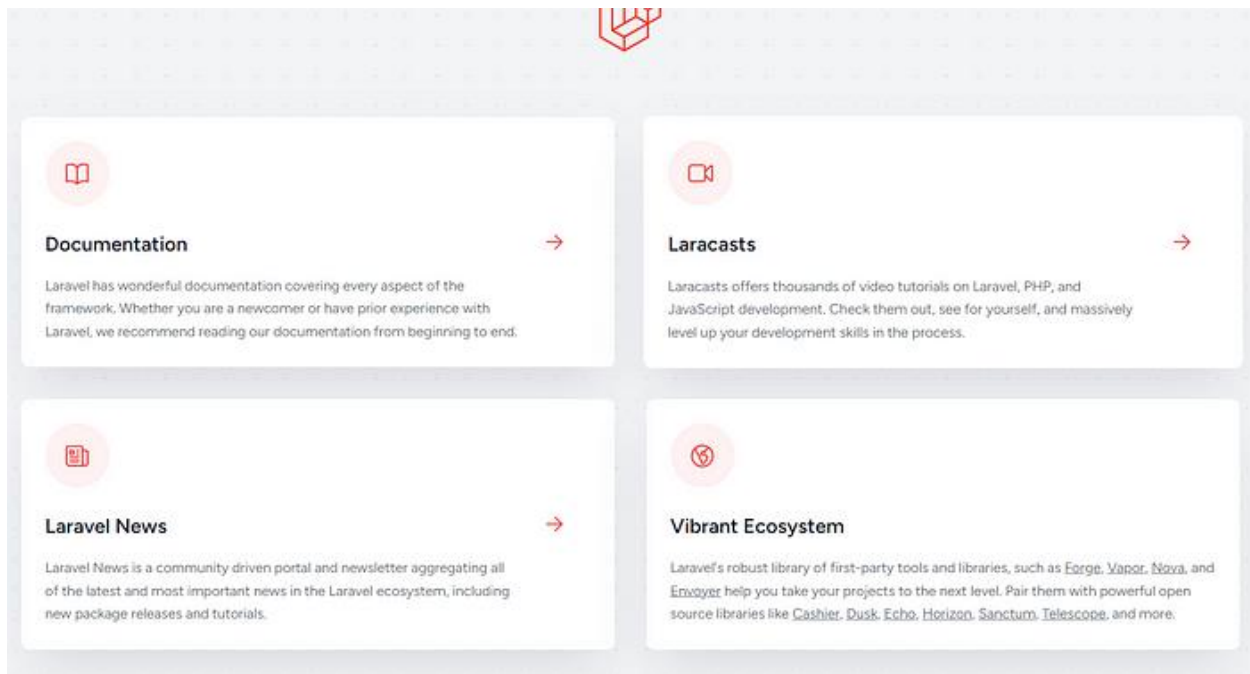
The first command was to create a new laravel project called **simple-project**

The second command was to move into a new directory (folder) named **simple-project**. **cd** means **change directory**.

The third command **php artisan serve** is used to start up the server. Copy the link and paste it on your browser's search box. The default port is 8000, but if you decide to use another port, you should rewrite your command like this:

```
$ php artisan serve --port=8001
```

Your default Laravel page should show like this



STEP 5: SETUP DATABASE

Launch your server (WAMP for windows), start all services. On your browser, type **localhost/phpmyadmin**. Create a new database, **simple_project**.

In your simple-project directory, open **.env** file, update the **DB_DATABASE** value with the name of the database you just created.

Open the database.php file in the config directory, since we are using MySQL, update the **engine's** value. **engine=InnoDB**

STEP 6: INSTALL BREEZE FOR AUTHENTICATION

Laravel Breeze is a minimal, simple implementation of all of Laravel's authentication features, including login, registration, password reset, email verification, and password confirmation. You may check the documentation for more information.

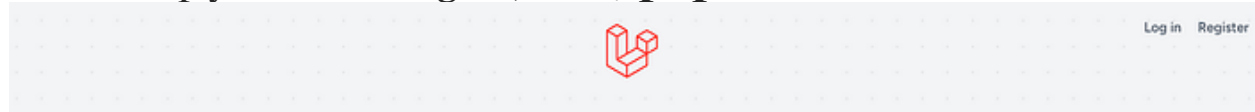
To stop your server from running, press **CTRL + C**

```
$ composer require laravel/breeze --dev
```

```
$ php artisan breeze:install blade
```

```
$ php artisan migrate
```

To start up your server again, run **\$ php artisan serve**



You should see **Login** and **Register** at the top right of your page.

Quick Explanation:

The first command helps to install the Laravel breeze package using composer. The second command publishes all of its codes (like authentication, routes, views, controllers, and other resources) to your application. We need to set up our data before we run the third command. The third command automatically publishes all schemas to

the database. Once this is done, you can check your database to see the new tables added to your database.

STEP 7: Update Users Schema

In the **app/database/migration** directory, select the file that contains “**create_users_table**” as its file name. The file should look like this:

```
<?php

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

return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('users', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('email')->unique();
            $table->timestamp('email_verified_at')->nullable();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down(): void
    {
        Schema::dropIfExists('users');
    }
}
```

```
}  
};
```

Let's update the schema with phone, role, and status fields.

```
Schema::create('users', function (Blueprint $table) {  
    $table->id();  
    $table->string('name');  
    $table->string('email')->unique();  
    $table->timestamp('email_verified_at')->nullable();  
    $table->string('password');  
    $table->string('phone')->nullable();  
    $table->enum('role', ['admin', 'user'])->default('user');  
    $table->enum('status', ['active',  
'inactive'])->default('active');  
    $table->rememberToken();  
    $table->timestamps();  
});
```

Now, that we are done updating our Users schema, we can now create data for the users table

STEP 8: CREATE DEMO DATA (SEEDERS)

Let's create demo data before we migrate all our tables to the database.

Run

```
$ php artisan make:seeder UsersTableSeeder
```

This command will create **UsersTableSeeder.php** file in **app/database/seeder** directory. We will create two records: an admin record and a user record.

```

<?php

namespace Database\Seeders;

use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\{Hash, DB};

class UsersTableSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run(): void
    {
        //Admin
        DB::table('users')->insert([
            [
                'name' => 'Lagbaja Tamedo',
                'email' => 'lagbaja@gmail.com',
                'password' => Hash::make(12345),
                'role' => 'admin',
                'status' => 'active'
            ],

            //Users
            [
                'name' => 'John Doe',
                'email' => 'doe@gmail.com',
                'password' => Hash::make(12345),
                'role' => 'user',
                'status' => 'active'
            ]
        ]);
    }
}

```

We are done creating demo data in the **UsersTableSeeder.php** file. Let's add more data to the Users table by creating fake random data, that's our next step.

STEP 9: CREATE DEMO DATA (FAKE)

Let's update our UserFactory.php file
in **app.database/factories** directory as thus:

```
<?php

namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;
use Illuminate\Support\Str;

/**
 * @extends \Illuminate\Database\Eloquent\Factories\Factory<\App\Models\User>
 */
class UserFactory extends Factory
{
    /**
     * Define the model's default state.
     *
     * @return array<string, mixed>
     */
    public function definition(): array
    {
        return [
            'name' => fake()->name(),
            'email' => fake()->unique()->safeEmail(),
            'email_verified_at' => now(),
            'role' => fake()->randomElement(['admin', 'user']),
            'status' => fake()->randomElement(['active', 'inactive']),
            'phone' => fake()->phoneNumber(),
            'password' =>
                '$2y$10$92IXUNpkjO0rQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi', // password
            'remember_token' => Str::random(10),
        ];
    }

    /**
     * Indicate that the model's email address should be unverified.
     */
    public function unverified(): static
    {
        return $this->state(fn (array $attributes) => [
            'email_verified_at' => null,
        ]);
    }
}
```

```
}  
}
```

In the **DatabaseSeeder.php** file in the database directory, we need to specify the number of fake data we want. The code below should throw more light to that.

```
<?php  
  
namespace Database\Seeders;  
  
// use Illuminate\Database\Console\Seeds\WithoutModelEvents;  
use Illuminate\Database\Seeder;  
  
class DatabaseSeeder extends Seeder  
{  
    /**  
     * Seed the application's database.  
     */  
    public function run(): void  
    {  
        $this->call(UsersTableSeeder::class);  
        \App\Models\User::factory(2)->create();  
    }  
}
```

In the above code, we decided to add 2 records. If we want more data, change the number in the factory() method.

Run this command `$ php artisan migrate:fresh --seed`

This command will drop all tables (if there is any) and migrate them afresh along with the demo data we created. Check your phpmyadmin interface, you should see the following:

Database Tables

Table	Action	Rows	Type	Collation	Size	Overhead
<input type="checkbox"/> failed_jobs		0	InnoDB	utf8mb4_unicode_ci	16.0 KiB	-
<input type="checkbox"/> migrations		4	InnoDB	utf8mb4_unicode_ci	16.0 KiB	-
<input type="checkbox"/> password_reset_tokens		0	InnoDB	utf8mb4_unicode_ci	16.0 KiB	-
<input type="checkbox"/> personal_access_tokens		0	InnoDB	utf8mb4_unicode_ci	16.0 KiB	-
<input type="checkbox"/> users		4	InnoDB	utf8mb4_unicode_ci	16.0 KiB	-
5 tables	Sum	8	MyISAM	utf8mb4_general_ci	80.0 KiB	0 B

☐ Check all With selected: ▼

user table

		id	name	email	email_verified_at	password	phone	role	status	remember_token	created_at	updated_at
<input type="checkbox"/>		1	Lagbaja Tamedo	lagbaja@gmail.com	NULL	\$2y\$10\$ja0VwWmcoS0pegkk2BhwK9Sle5SoEmGmQLAv	NULL	admin	active	NULL	NULL	NULL
<input type="checkbox"/>		2	John Doe	doe@gmail.com	NULL	\$2y\$10\$lg5minSU/KoKospVUTSmeyecowakf3JKoG3tznLj o3...	NULL	user	active	NULL	NULL	NULL
<input type="checkbox"/>		3	Jovan Pacocha DDS	hojt.hertzog@example.com	2023-07-29 21:40:01	\$2y\$10\$92XUNpkjQ2rOQ5byM:Ye4oKoEa3Rz9RC/ogia2...	+1 520 593 7878	admin	active	30969tABMq	2023-07-29 21:40:01	2023-07-29 21:40:01
<input type="checkbox"/>		4	Candelario Hudson	periozk@example.net	2023-07-29 21:40:01	\$2y\$10\$92XUNpkjQ2rOQ5byM:Ye4oKoEa3Rz9RC/ogia2...	754 480 9073	user	inactive	5Aw29mdDd	2023-07-29 21:40:01	2023-07-29 21:40:01

STEP 10: PLAY AROUND THE APP

You can try to register a new user or login with an existing user in the database. To login, using one of the demo data we created:

Email: lagbaja@gmail.com

Password: 12345

You should be able to login with the above information

STEP 11: REDIRECTING UNAUTHENTICATED USER

Logout of the app. Try visiting the dashboard without login in. Copy this link <http://localhost:8000/profile> and paste it in the URL search bar of your browser, you would be redirected to the login page, because you're not authenticated (logged in). If you want to change where an unauthenticated user should be redirected to, go to this file **app/Http/Middleware/Authenticate.php**

```
protected function redirectTo(Request $request): ?string
{
    return $request->expectsJson() ? null : route('login');
}
```

If you want the unauthenticated user to be redirected to the register page, change the “login” to “register”

STEP 12: USER BASED DIRECTORY

You can change the user's directory after logging in based on the role of the user (user or admin). To do this, go to this file **app/http/controllers/auth/AuthenticatedSessionController.php** and update it as it's been done below.

```
public function store(LoginRequest $request): RedirectResponse
{
    $request->authenticate();

    $request->session()->regenerate();
}
```

```

switch ($request->user()->role) {
    case 'admin':
        $url = "/profile";
        break;
    case 'user':
        $url = "/dashboard";
        break;
    default:
        $url = "";
        break;
}

return redirect()->intended($url);
}

```

STEP 13: CHANGE DIRECTORY AFTER LOGOUT

When you log out, you are automatically redirected to the login page.

But if you decide to redirect it to another page, go to this file, `app/http/controllers/auth/AuthenticatedSessionController.php`. In the `destroy()` method, edit the `redirect('/')` method.

```

public function destroy(Request $request): RedirectResponse
{
    Auth::guard('web')->logout();

    $request->session()->invalidate();

    $request->session()->regenerateToken();

    return redirect('/');
}

```

To redirect to the register page, you should do this

```
public function destroy(Request $request): RedirectResponse
{
    Auth::guard('web')->logout();

    $request->session()->invalidate();

    $request->session()->regenerateToken();

    return redirect('/register');
}
```

STEP 14: ROLES AND PERMISSIONS

We need to create roles and permissions for our users, if we want to limit access that all users can have on an application. In our users table, we have two roles: user and admin, though we can have more, if we want. Instead of going the long way doing all on our own, we can simply make use of a package called **spatie**. This package allows us to manage user roles and permissions in the database. To get started, we need to run some commands. Before we run the commands, let's go to this file `app/Models/User.php` and add **HasRoles** to the User class like this

```
use HasApiTokens, HasFactory, Notifiable, HasRoles;
```

\$ composer require spatie/laravel-permission

```
$ php artisan vendor:publish --  
provider="Spatie\Permission\PermissionServiceProvider"
```

You should clear your cache, if you have been caching your configuration locally. Simply run:

```
$ php artisan optimize:clear
```

Add the following to the App/Http/Kernel.php file

```
'role' =>  
\Spatie\Permission\Middlewares\RoleMiddleware::class,  
  
'permission' =>  
\Spatie\Permission\Middlewares\PermissionMiddleware::cl  
ass,  
  
'role_or_permission' =>  
\Spatie\Permission\Middlewares\RoleOrPermissionMiddle  
ware::class,
```

So, the file looks like this

```
protected $middlewareAliases = [  
    'auth' => \App\Http\Middleware\Authenticate::class,  
    'auth.basic' =>  
\Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,  
    'auth.session' =>  
\Illuminate\Session\Middleware\AuthenticateSession::class,  
    'cache.headers' =>
```

```

\Illuminate\Http\Middleware\SetCacheHeaders::class,
    'can' => \Illuminate\Auth\Middleware\Authorize::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'password.confirm' =>
\Illuminate\Auth\Middleware\RequirePassword::class,
    'precognitive' =>
\Illuminate\Foundation\Http\Middleware\HandlePrecognitiveRequests::class,
    'signed' => \App\Http\Middleware\ValidateSignature::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
    'verified' =>
\Illuminate\Auth\Middleware\EnsureEmailIsVerified::class,
    'role' => \Spatie\Permission\Middlewares\RoleMiddleware::class,
    'permission' =>
\Spatie\Permission\Middlewares\PermissionMiddleware::class,
    'role_or_permission' =>
\Spatie\Permission\Middlewares\RoleOrPermissionMiddleware::class,

];

```

Lastly, run `$ php artisan migrate`

Your database should be updated with new tables.

Table	Action	Rows	Type	Collation	Size	Overhead
<input type="checkbox"/> failed_jobs	★ Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_unicode_ci	16.0 KiB	-
<input type="checkbox"/> migrations	★ Browse Structure Search Insert Empty Drop	5	InnoDB	utf8mb4_unicode_ci	16.0 KiB	-
<input type="checkbox"/> model_has_permissions	★ Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_unicode_ci	32.0 KiB	-
<input type="checkbox"/> model_has_roles	★ Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_unicode_ci	32.0 KiB	-
<input type="checkbox"/> password_reset_tokens	★ Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_unicode_ci	16.0 KiB	-
<input type="checkbox"/> permissions	★ Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_unicode_ci	16.0 KiB	-
<input type="checkbox"/> personal_access_tokens	★ Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_unicode_ci	16.0 KiB	-
<input type="checkbox"/> roles	★ Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_unicode_ci	16.0 KiB	-
<input type="checkbox"/> role_has_permissions	★ Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_unicode_ci	32.0 KiB	-
<input type="checkbox"/> users	★ Browse Structure Search Insert Empty Drop	4	InnoDB	utf8mb4_unicode_ci	16.0 KiB	-
10 tables	Sum	9	MyISAM	utf8mb4_general_ci	208.0 KiB	0 B

This is great! We have come a long way. Relax for a few minutes before you continue.

Our next goal is to apply the **roles** and **permissions** on the **user files** and **profile files**.

For better understanding of what we will be doing next, let's understand how **route, view and controller** work together.

STEP 15: ROUTE/VIEW/CONTROLLER

View: The location of the view files can always be found in **app/resources/views** directory. This is where all the files you want to display on your browser reside. You cannot view any of these files without configuring the corresponding routes.

Routes: The location of the routes configuration can be found in **app/routes/web.php** file. This is where a page's path is defined and the name of the route. A corresponding controller class can also be attached to the route, so also middlewares.

Below is an example of a route without a controller

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

get: This is the method for the route. There are others like post, patch, delete etc.

‘/’: That’s the path to go to in order to view the welcome page. Other paths could be **‘/login’** or **‘/register’** or depending on how you want to name your path

view(‘welcome’): This points to the welcome.blade.php file in the view folder. Once the path is visited, the view would load the content in the welcome.blade.php file to the browser.

If all these work well, what’s then the use of controllers? We use controllers to handle most of our request logics and for code organization. Instead of performing most of our response or request logics in the web.php file, it’s better we do them in our controllers.

```
Route::get('/users', [UserController::class, 'show'])->name('users.show');
```

The above code shows the relationships among the route, views and controllers.

The method used is **get**, the path is **users**, the controller is **UserController**, and the method where the view was called is in the **show** method. The route is also given a name, which is **user.show**.

Below is how the view is being called in the show() method of the UserController class


```
public function show(): View
{
    return view('users.show');
}
```

This is just the basics. For more information, kindly check Laravel's official documentation.

STEP 16: ADD DATA TO THE PERMISSION'S TABLE

Let's create a seeder file for our permission.

```
$ php artisan make:seeder PermissionTableSeeder
```

The database directory gets updated with a new file, PermissionTableSeeder. Then, we update the file as thus:

```
public function run(): void
{
    //
    $permissions = [
        'role-list',
        'role-create',
        'role-edit',
        'role-delete',
        'user-list',
        'user-create',
        'user-edit',
        'user-delete'
    ];

    foreach ($permissions as $permission) {
        Permission::create(['name' => $permission]);
    }
}
```

```
}  
}
```

This is to create a default list of permissions. As we did before, by updating the DatabaseSeeder.php file for the users, we also need to do the same for Permissions.

```
// use Illuminate\Database\Console\Seeds\WithoutModelEvents;  
use Illuminate\Database\Seeder;  
  
class DatabaseSeeder extends Seeder  
{  
    /**  
     * Seed the application's database.  
     */  
    public function run(): void  
    {  
        $this->call([UsersTableSeeder::class, PermissionTableSeeder::class]);  
        \App\Models\User::factory(2)->create();  
    }  
}
```

Take note, that the argument in the call() method has been changed to an array, so as to accommodate more seeders.

Let's migrate the permissions data to the database.

```
$ php artisan db:seed --class=PermissionTableSeeder
```

The above command includes —

class=PermissionTableSeeder. This is to enable the migration of the PermissionTableSeeder only without affecting others.

STEP 17: USER RESOURCE

Let's create all the needed routes and controller for the **user**. To get this done easily, we should run the command below:

```
$ php artisan make:controller UserController --resource
```

This command would automatically create a new UserController.php file in the controller directory with some predefined methods.

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Display a listing of the resource.
     */
    public function index()
    {
        //
        return "Index";
    }

    /**
     * Show the form for creating a new resource.
     */
    public function create()
    {
        //
        return "create";
    }

    /**
     * Store a newly created resource in storage.
     */
}
```

```

    */
    public function store(Request $request)
    {
        //
        return "store";
    }

    /**
     * Display the specified resource.
     */
    public function show(string $id)
    {
        //
        return "show";
    }

    /**
     * Show the form for editing the specified resource.
     */
    public function edit(string $id)
    {
        //
        return "edit";
    }

    /**
     * Update the specified resource in storage.
     */
    public function update(Request $request, string $id)
    {
        //
        return "update";
    }

    /**
     * Remove the specified resource from storage.
     */
    public function destroy(string $id)
    {
        //
        return "delete";
    }
}

```

The **return** keyword was added in order to explain this better.

To check all the route lists available.

Run:

```
$ php artisan route:list
```

STEP 18: ROLE RESOURCE

We can also do the same thing for role.

```
$ php artisan make:controller RoleController --resource
```

STEP 19: ROUTES

Add the user and roles routes to the web.php file.

```
Route::middleware(['auth'])->group(function () {  
    Route::resource('user', UserController::class);  
    Route::resource('roles', RoleController::class);  
});
```

The role resource has the following actions handled by the resource controllers as described below.

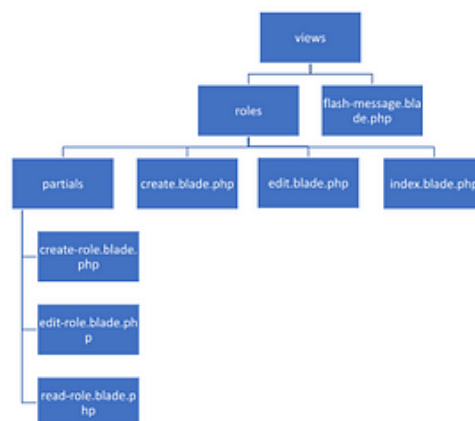
Verb	URI	Action	Route Name
GET	/roles/	index	roles.index
GET	/roles/create	create	roles.create
POST	/roles	store	roles.store
GET	/roles/{role}	show	roles.show
GET	/roles/{role}/edit	edit	roles.edit
PUT/PATCH	/roles/{role}	update	roles.update
DELETE	/roles/{role}	destroy	roles.destroy

STEP 19: CREATE ROLES

Since we have created the needed **permissions**, we can now create the **roles** that would be attached to the permissions chosen for a particular role. This time around, we are going to create the roles via the interface instead of pushing them directly from the files.

Role File Organization

Below is the file organisation for roles.



The styling in these files are going to be a combination of Laravel's default styling and Bootstrap styling. We won't be digging deep into the **Blade Templates**, but focus on what we want to achieve with it. In order to keep this simple, we are going to use Bootstrap CDN,

```
<link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@4.6.2/dist/css/bootstrap.min.css">
```

Add this link to the views/layouts/app.blade.php file.

views/layouts/navigation.blade.php

First off, let's create navigation links for the roles.

```
<x-dropdown-link :href="route('profile.edit')">
    {{ __('Profile') }}
</x-dropdown-link>
<x-dropdown-link :href="route('roles.create')">
    {{ __('Create Roles') }}
</x-dropdown-link>
<x-dropdown-link :href="route('roles.index')">
    {{ __('View Roles') }}
</x-dropdown-link>
<x-dropdown-link :href="route('profile.edit')">
    {{ __('Profile') }}
</x-dropdown-link>
<x-dropdown-link :href="route('roles.create')">
    {{ __('Create Roles') }}
</x-dropdown-link>
<x-dropdown-link :href="route('roles.index')">
    {{ __('View Roles') }}
</x-dropdown-link>
```

Add the **Create Roles** and **View Roles** to the navigation.blade.php file as done above. The **route('roles.create')** translates to **/roles/create** path and **route('roles.index')** translates to **/roles/** path.

flash-message.blade.php

This is where we keep **session messages** after an action has been performed.

```
@if ($message = Session::get('success'))
<div class="alert alert-success alert-dismissible fade show" role="alert">
    <strong>{{ $message }}</strong>
    <button type="button" class="btn-close" data-bs-dismiss="alert" aria-
label="Close"></button>
</div>
@endif

@if ($message = Session::get('error'))
<div class="alert alert-danger alert-dismissible fade show" role="alert">
    <strong>{{ $message }}</strong>
    <button type="button" class="btn-close" data-bs-dismiss="alert" aria-
label="Close"></button>
</div>
@endif
```

STEP 20: CREATE ROLES

Next, in the views directory, create the folders and files as seen in the flow-chart above.

Create Role

Name

Permissions

- ☐ role-list
- ☐ role-create
- ☐ role-edit
- ☐ role-delete
- ☐ user-list
- ☐ user-create
- ☐ user-edit
- ☐ user-delete

SAVE

The image above is the **create role** page we want to create now.

Add the codes below to the create.blade.php

```
<x-app-layout>
  <x-slot name="header">
    <h2 class="font-semibold text-xl text-gray-800 leading-tight">
      {{ __('Roles') }}
    </h2>
  </x-slot>

  <div class="py-12">
    <div class="max-w-7xl mx-auto sm:px-6 lg:px-8 space-y-6">
      <div class="p-4 sm:p-8 bg-white shadow sm:rounded-lg">
        <div class="max-w-xl">
          @include('flash-message')
          @include('roles.partials.create-role')
        </div>
      </div>
    </div>
  </div>
</x-app-layout>
```

@include('flash-message') means including the flash-message.blade.php file into the create.blade.php file.

@include('roles.partials.create-role') means including create-role.blade.php from the roles/partials directory into the create.blade.php file.

We are yet to create the create-role.blade.php file, so let's get this done.

```
<section>
  <header>
    <h2 class="text-lg font-medium text-gray-900">
      {{ __('Create Role') }}
    </h2>
  </header>

  <form method="post" action="{{ route('roles.store') }}" class="mt-6
space-y-6">
    @csrf
    @method('post')

    <div class="row mb-3">
      <label for="" class="col-sm-3 col-form-label">Name</label>
      <div class="col-sm-9">
        <input type="text" class="form-control" name="name"
autocomplete="off">
        @error('name')
          <span class="text-danger">{{ $message }}</span>
        @enderror
      </div>
    </div>

    <h5 class="pb-3">Permissions</h5>
    @error('permission')
      <span class="text-danger">{{ $message }}</span>
    @enderror

    @foreach($permissions as $item)
      <div class="form-check mb-2">
        <input type="checkbox" value="{{ $item->id }}" class="form-check-
input" name="permission[]">
```

```
        <label class="form-check-label" for="checkChecked">
            {{$item->name}}
        </label>
    </div>
@endforeach

    <div class="mt-3">
        <button type="submit" class="px-4 py-2 bg-red-600 text-
white">SAVE</button>
    </div>

</form>
</section>
```

This is a form for creating roles and attaching permissions to them.

@csrf CSRF means Cross-Site Request Forgeries. According to the Laravel documentation, Cross-site request forgeries are a type of malicious exploit whereby unauthorized commands are performed on behalf of an authenticated user. Laravel automatically generates a CSRF “token” for each active user session managed by the application. This token is used to verify that the authenticated user is the one actually making the requests to the application.

@error('name') This means that, if there’s a validation error in the name field, an error message should be displayed.

@foreach(\$permissions as \$item) This is a foreach loop that loops through a list of permissions.

In the RoleController.php, we need to update the create and the store methods as shown below.

```
public function create(): View
{
    $permissions = Permission::all();

    return view('roles.create', compact('permissions'));
}
```

Permission::all() — This is to get all the permissions stored in the permission's table

return view('roles.create', compact('permissions')) — display the content in the roles/create.blade.php file. **compact('permissions')** is a way of passing the variable **\$permissions** to the roles/create.blade.php file.

```
public function store(Request $request): RedirectResponse
{
    $this->validate($request, [
        'name' => 'required|unique:roles,name',
        'permission' => 'required',
    ]);

    $role = Role::create(['name' => $request->name]);
    $role->syncPermissions($request->permission);

    return back()->with('success', $request->name.' Role created successfully');
}
```

The `store()` method is where the form is being processed: the validation; and the storing of data happen here.

`$request` — This has the request header and body stored in it. We can retrieve the data input from the form by calling the properties or methods attached to the request. To get the value of a name field, we can either retrieve the value as **`$request->name`** or **`$request->input('name')`**

`$this->validate()` is for validation. If the validation conditions are not meant, it would return an error and not proceed to store the data.

`Role::create()` method would create a new name in the role's table.

`$role->syncPermissions($request->permission)` — this enables roles and permissions synchronization. Once you're able to create a role, do well to check the **`role_has_permissions`** table in the database, you should see the role's id and the permissions' ids stored in there like the image below.

<div> <div>←T→</div> <div>▼</div> </div>				permission_id	role_id
<input type="checkbox"/>	 Edit	 Copy	 Delete	1	1
<input type="checkbox"/>	 Edit	 Copy	 Delete	2	1
<input type="checkbox"/>	 Edit	 Copy	 Delete	3	1
<input type="checkbox"/>	 Edit	 Copy	 Delete	4	1
<input type="checkbox"/>	 Edit	 Copy	 Delete	5	1
<input type="checkbox"/>	 Edit	 Copy	 Delete	6	1
<input type="checkbox"/>	 Edit	 Copy	 Delete	7	1
<input type="checkbox"/>	 Edit	 Copy	 Delete	8	1

STEP 21: VIEW ROLES

At the end of this step, we should have a list of role names, edit buttons and delete buttons.

View Role			
#	Name	Action	
1	super	<button>Edit</button>	<button>Delete</button>
2	user	<button>Edit</button>	<button>Delete</button>
3	admin	<button>Edit</button>	<button>Delete</button>

Add the codes below to the index.blade.php

```
<x-app-layout>
  <x-slot name="header">
    <h2 class="font-semibold text-xl text-gray-800 leading-tight">
```

```

        {{ __('Roles') }}
    </h2>
</x-slot>

<div class="py-12">
    <div class="max-w-7xl mx-auto sm:px-6 lg:px-8 space-y-6">
        <div class="p-4 sm:p-8 bg-white shadow sm:rounded-lg">
            <div class="max-w-xl">
                @include('flash-message')
                @include('roles.partials.read-role')
            </div>
        </div>
    </div>
</div>
</x-app-layout>

```

This is similar to what we did with creating roles. Next, is to update the `roles/partials/read-role.blade.php` file.

```

<section>
    <header>
        <h2 class="text-lg font-medium text-gray-900">
            {{ __('View Role') }}
        </h2>
    </header>

    <div class="table-responsive">
        <table class="table table-hover mb-0">
            <thead>
                <tr>
                    <th class="pt-0">#</th>
                    <th class="pt-0">Name</th>
                    <th class="pt-0" colspan="2">Action</th>
                </tr>
            </thead>
            <tbody>
                @foreach($roles as $item)
                    <tr>
                        <td>{{++$i}}</td>
                        <td>{{ $item->name }}</td>
                        <td>
                            <a class="btn btn-primary"
href="{{ route('roles.edit', $item->id) }}">Edit</a>
                        </td>
                        <td>
                            <form action="{{ route('roles.destroy', $item->id) }}"

```

```

method="post">
                                @csrf
                                @method('DELETE')
                                <button type="submit" class="px-4 py-2 bg-red-600
text-white">Delete</button>
                                </form>
                                </td>
                                </tr>
                                @endforeach
                                </tbody>
                                </table>
                                </div>

</section>

```

We have explained similar codes before, we can just skip the explanation for them, except for the **action** attribute in the form tag.

route('roles.destroy', \$item->id) — roles.destroy corresponds to the roles/destroy route, which points to the destroy() method in the RoleController class. \$item->id is the id of the role that would be passed to the destroy() method for processing.

There are two actions that would be performed here: redirects to the roles/{role}/edit page if you clicked on the edit button; while the edit button, if clicked, gets processed in the destroy() method.

Let's see how they were processed in the RoleController method

```

public function index(Request $request): View
{
    $roles = Role::all();
    $i = 0;

```



```
        return view('roles.index', compact('roles', 'i'));
    }
```

Similar to what we have done before, this is to retrieve all the roles that have been stored in the roles' table in the database.

```
public function destroy(Role $role): RedirectResponse
{
    $role->delete();

    return back()->with('success', 'Role deleted successfully');
}
```

This is simply to delete a role.

Click on the buttons to test out what you have done.

STEP 22: EDIT/UPDATE ROLES

This is the interface for the **edit role** page. It's quite similar to the **create role** page.

View Role

Name

Permissions

☒ role-list
☒ role-create
☐ role-edit
☐ role-delete
☒ user-list
☒ user-create
☐ user-edit
☐ user-delete

Save

Update the edit.blade.php file

```
<x-app-layout>
  <x-slot name="header">
    <h2 class="font-semibold text-xl text-gray-800 leading-tight">
      {{ __('Roles') }}
    </h2>
  </x-slot>

  <div class="py-12">
    <div class="max-w-7xl mx-auto sm:px-6 lg:px-8 space-y-6">
      <div class="p-4 sm:p-8 bg-white shadow sm:rounded-lg">
        <div class="max-w-xl">
          @include('flash-message')
          @include('roles.partials.edit-role')
        </div>
      </div>
    </div>
  </div>
</x-app-layout>
```

Update the roles/partials/edit-role.blade.php file

```
<section>
  <header>
    <h2 class="text-lg font-medium text-gray-900">
      {{ __('View Role') }}
    </h2>
```

```

</header>

<form class="forms-sample" method="post" action="{{ route('roles.update',
$role->id) }}">
    @csrf
    @method('PUT')
    <div class="row mb-3">
        <label for="exampleInputUsername2" class="col-sm-3 col-form-
label">Name</label>
        <div class="col-sm-9">
            <input type="text" value="{{ $role->name }}" class="form-
control" name="name" autocomplete="off">
            @error('name')
            <span class="text-danger">{{ $message }}</span>
            @enderror
        </div>
    </div>
    <h5 class="pb-3">Permissions</h5>
    @error('permission')
    <span class="text-danger">{{ $message }}</span>
    @enderror

    @foreach($permissions as $item)
    <div class="form-check mb-2">
        <input type="checkbox" @if(in_array($item->id, $rolePermission))
checked
@endif
        value="{{ $item->id }}" class="form-check-input"
name="permission[]">
        <label class="form-check-label" for="checkChecked">
            {{ $item->name }}
        </label>
    </div>
    @endforeach
    <div class="mt-3">
        <button type="submit" class="px-4 py-2 bg-red-600 text-
white">Save</button>
    </div>
</form>

</section>

```

Let's update the edit() method and the update() method

```

public function edit($id): View
{
    $role = Role::find($id);
    $permissions = Permission::get();
    $rolePermission =
DB::table("role_has_permissions")->where("role_has_permissions.role_id",$id)

->pluck('role_has_permissions.permission_id','role_has_permissions.permission
_id')
    ->all();

    return
view('roles.edit',compact('role','permissions','rolePermission'));
}

```

Role::find(\$id) selects a role that corresponds to the provided id

Permission::get() selects all the permissions available.

DB::table("role_has_permissions") calls the
role_has_permissions table.

where("role_has_permissions.role_id",\$id) selects the record
that has the **role_id** of the **role_has_permissions** table equal to
the provided id.

**pluck('role_has_permissions.permission_id','role_has_per
missions.permission_id')** selects specific values, in this regard, it
is the **permission_id** of the **role_has_permissions** table.

pluck() vs select()

The `pluck()` method returns an array of `permission_id`, while the `select()` method returns an array of objects.

```
public function update(Request $request, $id): RedirectResponse
{
    $this->validate($request, [
        'name' => 'required',
        'permission' => 'required',
    ]);

    $role = Role::find($id);
    $role->name = $request->name;
    $role->save();

    $role->syncPermissions($request->permission);

    return back()->with('success', 'Role updated successfully');
}
```

This is quite similar to what we did in the `store()` method. It validates, finds the corresponding role for the provided id, save it in the roles table and synchronises the roles and the permissions tables.