Contents

HOW TO SETUP LARAGON

Why Laragon? Laragon is a localhost program for Windows machine. Laragon provides a portable, isolated, fast and powerful development for **PHP, Node.JS, Phython, Java, Go, and Ruby** for Windows computers. You can use **Apache or Nginx** as a web server. You can add a domain name and customize as you wish. **MySQL, PostgreSQL, MongoDB and Redis** options are available for database. Laragon can be used **as a development environment**.

1. Install Laragon.

Go to Laragon's official website (https://laragon.org/download) and download the latest version of the appropriate installer for your version of Windows. For the purpose of this tutorial, we shall using the full edition.

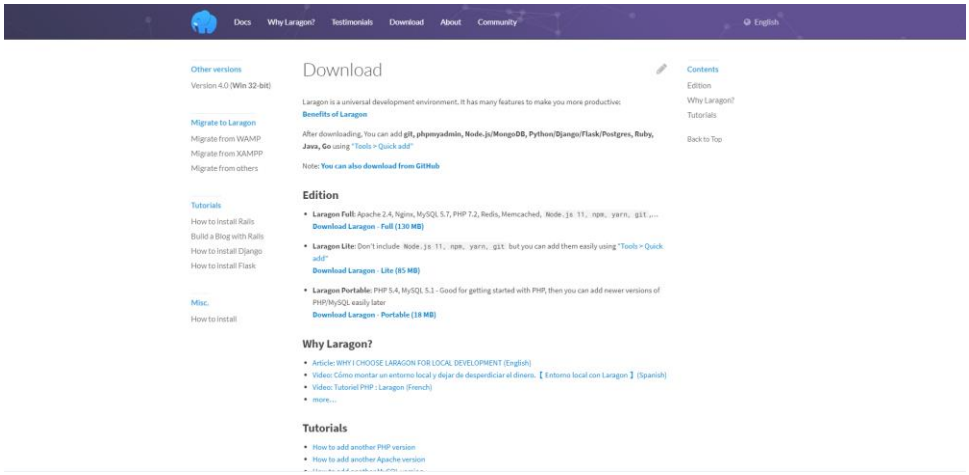


Figure 1: Laragon Website

After the download is complete, double-click on the Laragon Installer to initiate the installation process.

- On the ***"Select Setup Language"***, select your language of your choice. In this case, English then click on the OK button.
- On the ***"Select Destination Location"*** page, go with the default destination option which is C:\laragon and then click on the Next button. On the ***"Setup Options"*** page, it is my personal preference to uncheck these two checkboxes: '*Run Laragon when Windows starts*' and '*Add Notepad++ & Terminal to the Right-clik Menu*'. Now, click Next.

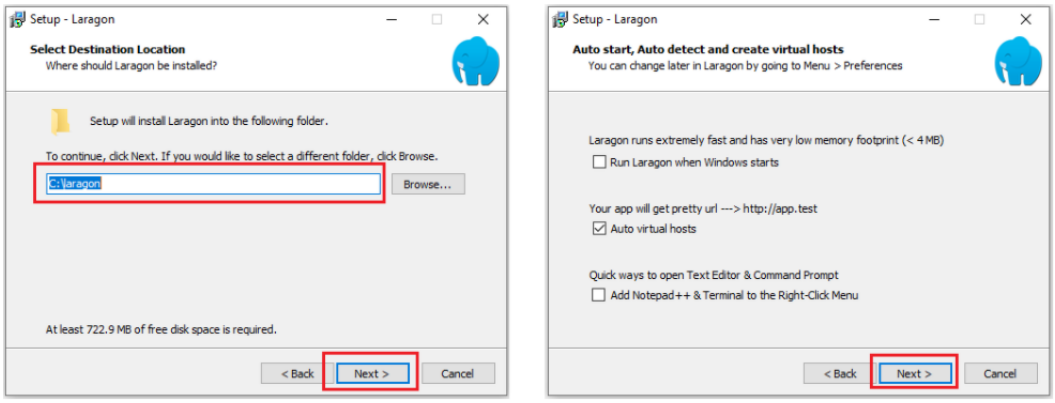


Figure 2: Setup Laragon

- On the "***Ready to Install***" page, click Install. When the installation is done, click on Finish so as to restart your machine.

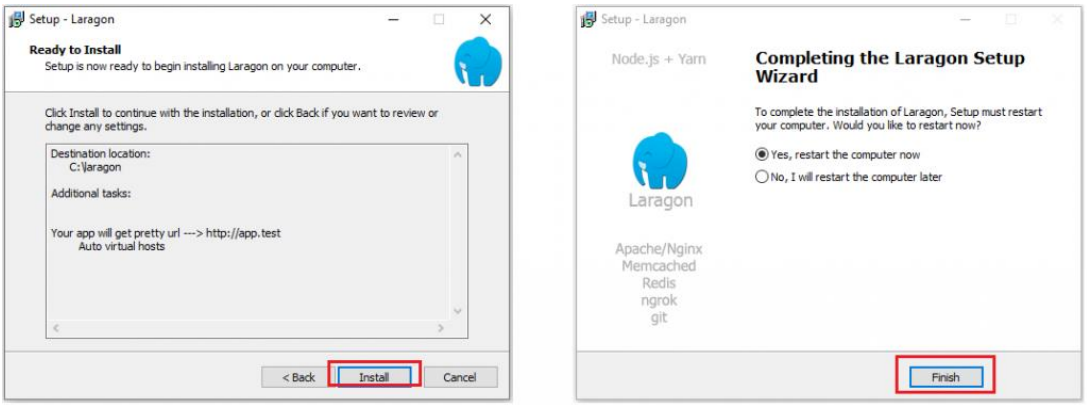


Figure 3: Setup Laragon

2. Start Laragon.

After your PC is done rebooting, launch the Laragon app from the desktop or from the Start menu. When Laragon opens, you should see an interface that looks like the image on the left below. Click on the **Start All** button to start Apache and MySQL.

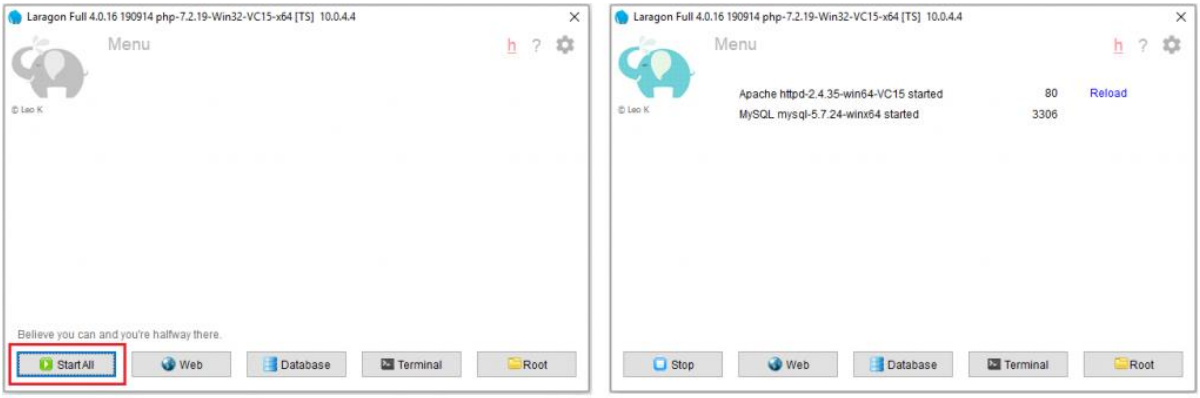


Figure 4: Laragon

In the event you get a firewall notice like the ones depicted in the image set below, you can choose to deny or allow access. But since I trust my private network, I normally click on checkbox for private networks and then the 'Allow access' button for both ***Apache HTTP Server*** and ***MySQL*** so they can communicate on my private network.

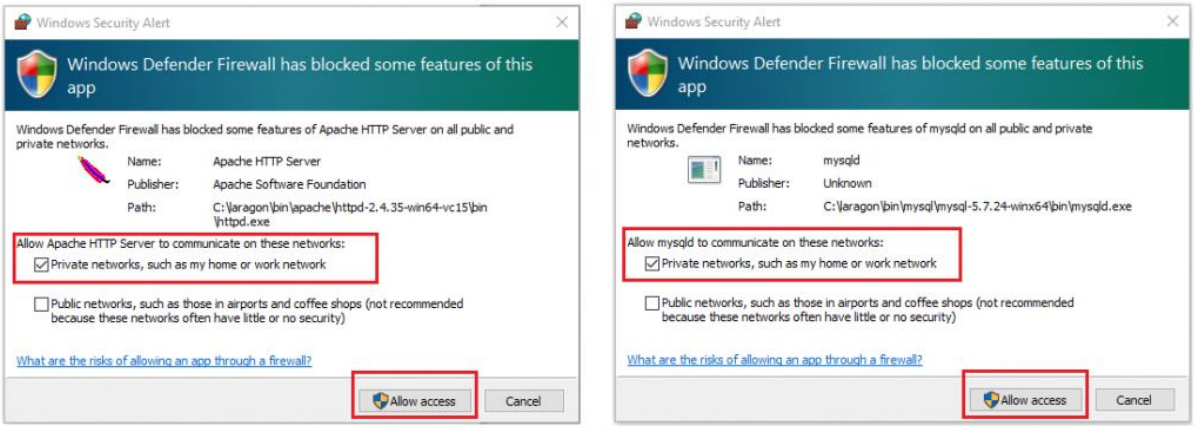


Figure 5: Windows Security Alert

HOW TO INSTALL VISUAL STUDIO CODE

If you already have VS Code installed in your machine, you can skip this section. However, if you want it as one of the apps installed within the Laragon directory then you have to first uninstall VS Code from your PC before continuing with the tutorial in this section.

Download the VS Code installer for your version of Windows from **(https://code.visualstudio.com/download)**. When the download is complete, double-click on the installer to commence the installation process.

1. On the ***"License Agreement"*** page, choose '*I accept the agreement*' and click Next. On the ***"Select Destination Location"*** page, click on the '*Browse*' button and navigate to the laragon\bin directory and click OK. If you have done it correctly you should see C:\laragon\bin\Microsoft VS Code in the text-field; click Next.

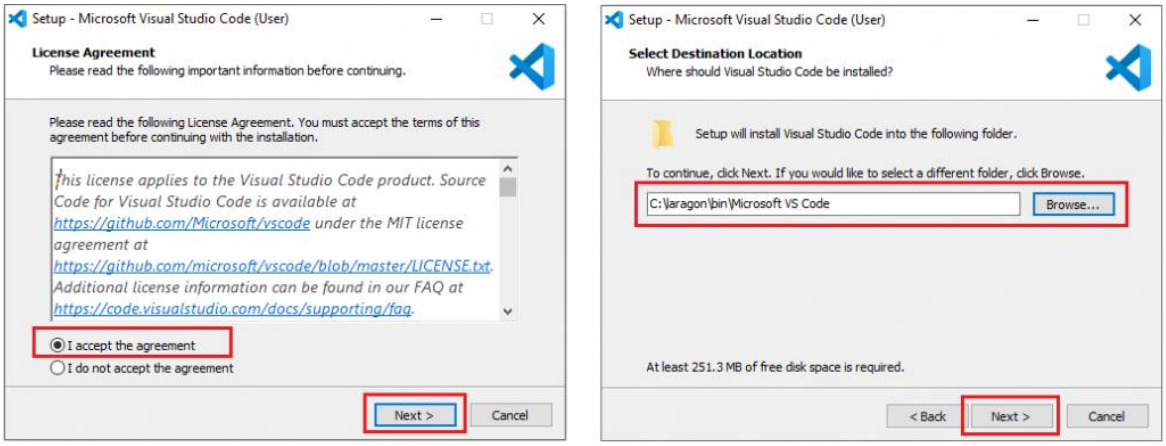


Figure 6: VS Code Setup

2. On the ***"Select Start Menu Folder"*** page, click Next. On the ***"Select Additional Tasks"*** page, ensure you check all the checkboxes, then click Next.

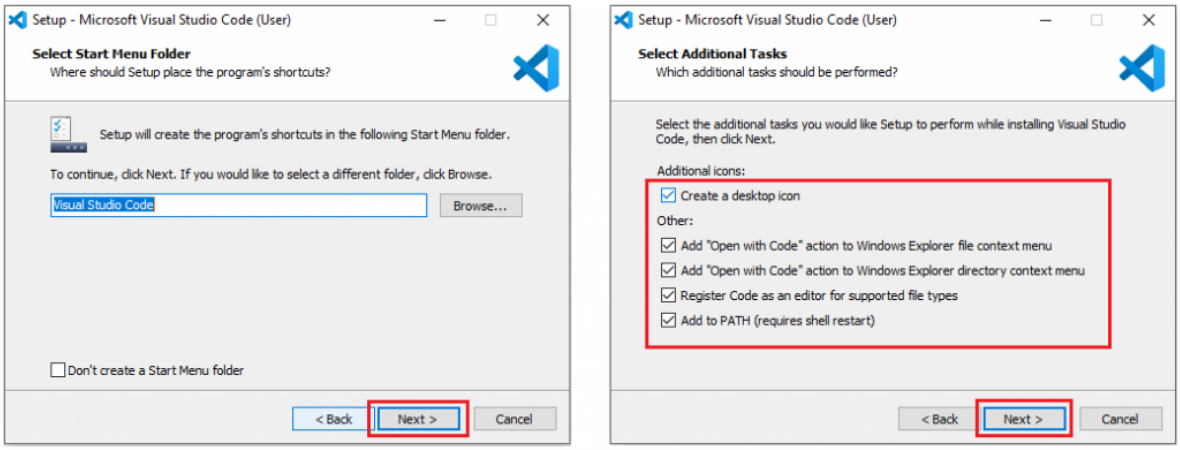


Figure 7: VS Code Setup

3. On the ***"Ready to Install"*** page, click Next. When the installation is complete, click Finish.

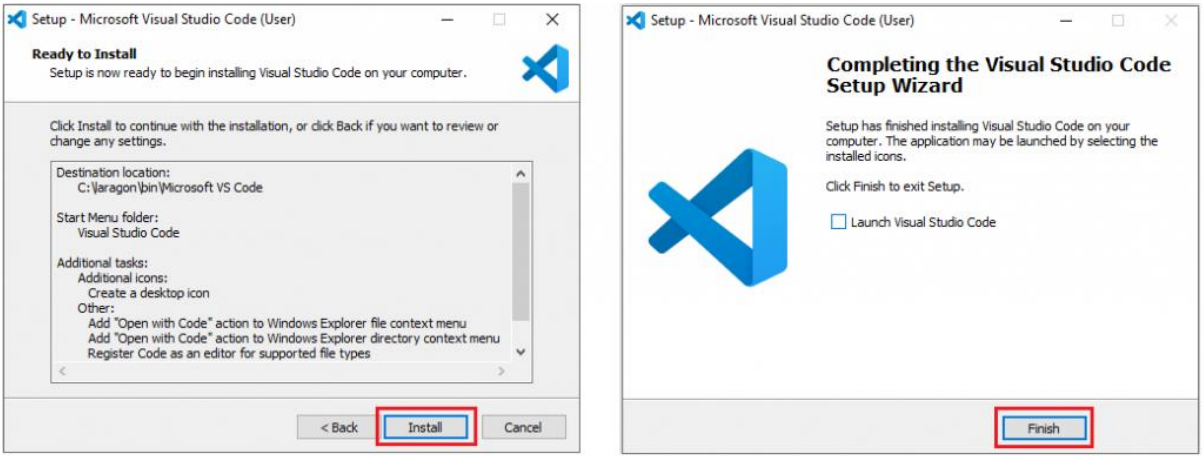


Figure 8: VS Code Setup

To start code, simply open visual studio code and click on the "File->Open Folder" and choose your Laravel project.

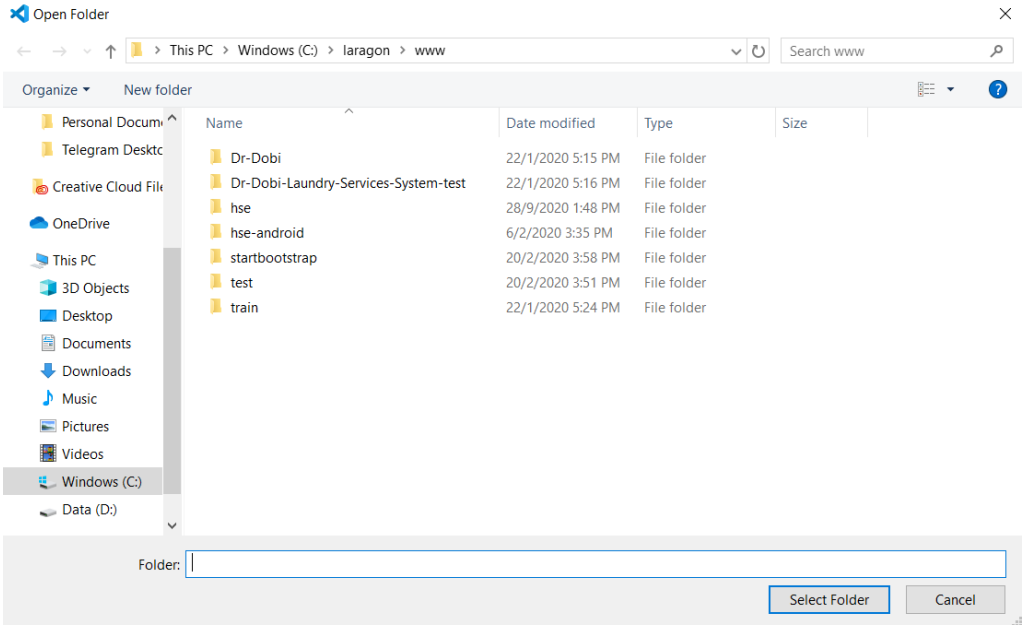


Figure 9: VS Code Open Folder

HOW TO START LARAVEL PROJECT

Since Laragon version 4.0.16 only provide PHP 7.2, you need to install latest version of PHP to start Laravel Jetstream project. Go to (https://windows.php.net/download) and download the ZIP file under VC15 x64 Thread Safe for PHP 7.4 (7.4.11) as shown below:

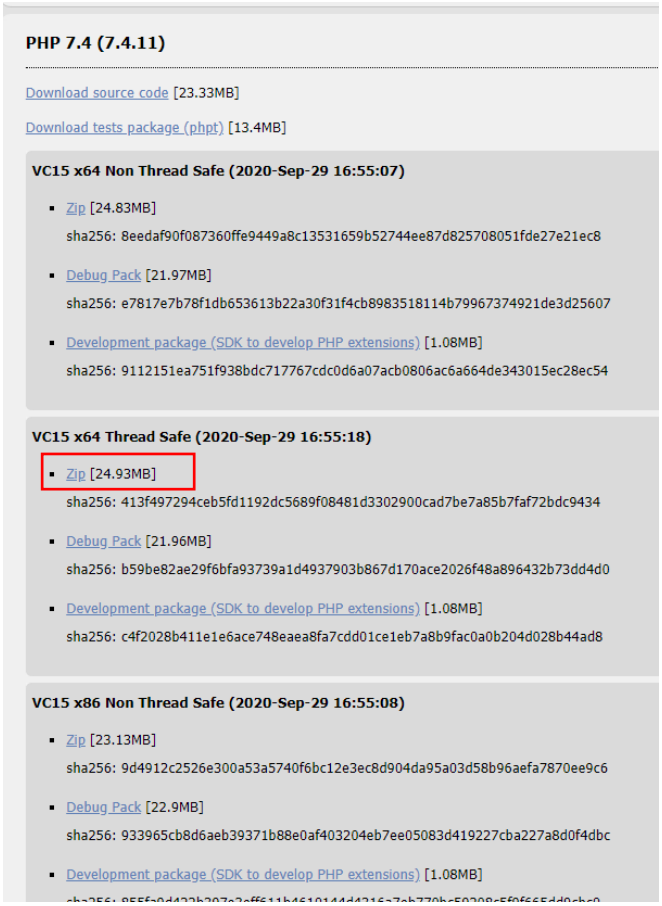


Figure 10:PHP website

After done downloading, extract the folder (use the **Extract to** option), copy and paste it to C->laragon->bin->php.

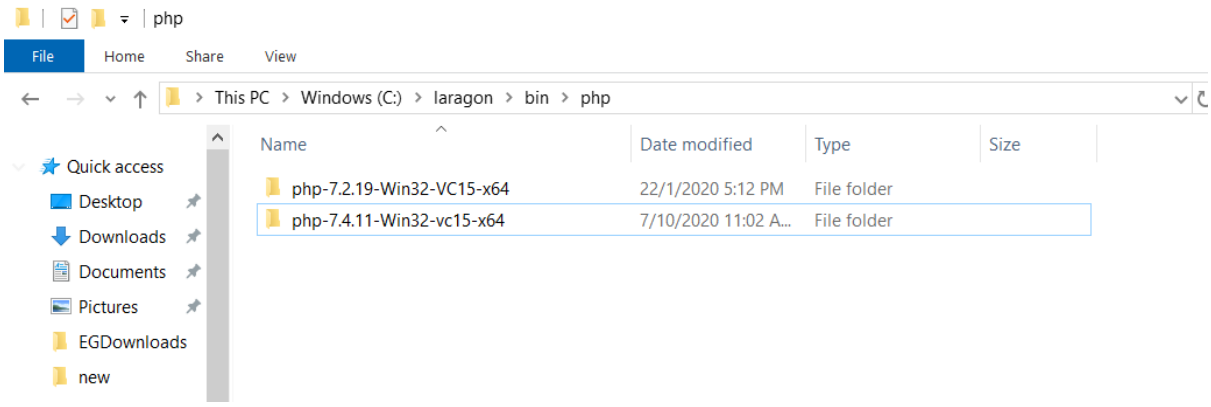


Figure 11: PHP Folder

Then, go to Laragon, right-click, go to PHP->version and choose PHP 7.4 as shown below:

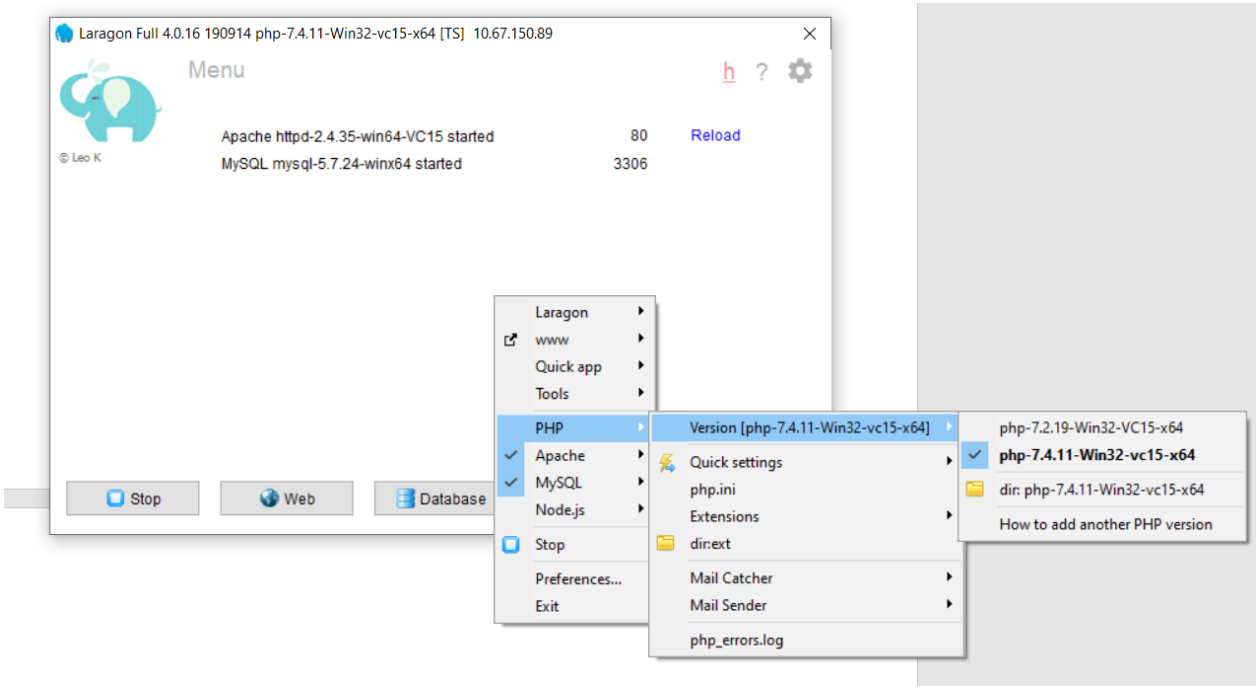


Figure 12:Change PHP version

Next restart your Laragon by click Stop and then Start All button. After that, Open Laragon “Terminal” and run these commands.

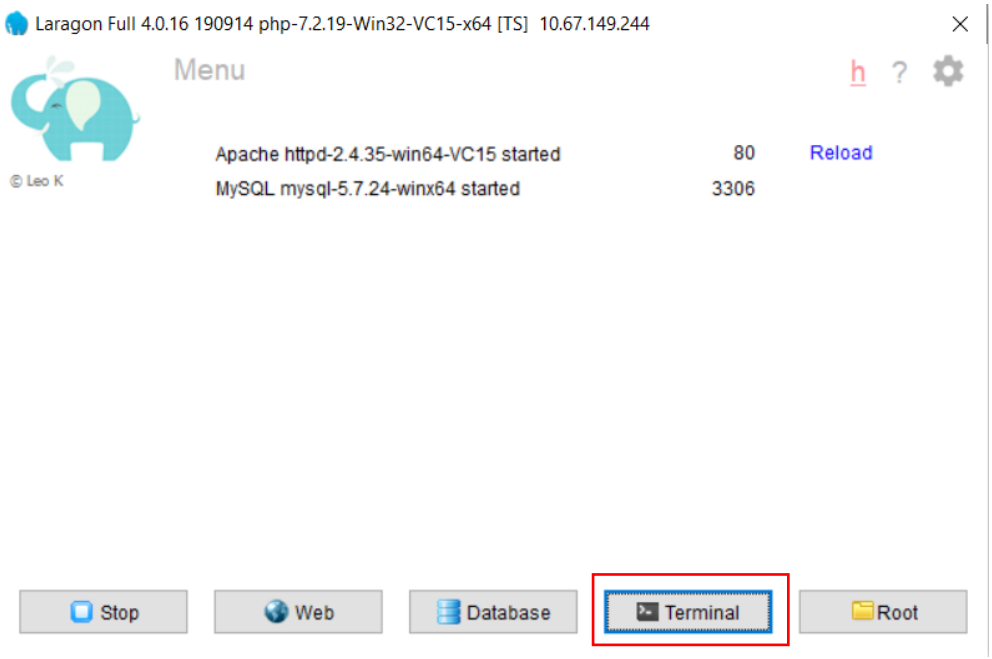


Figure 13:Laragon

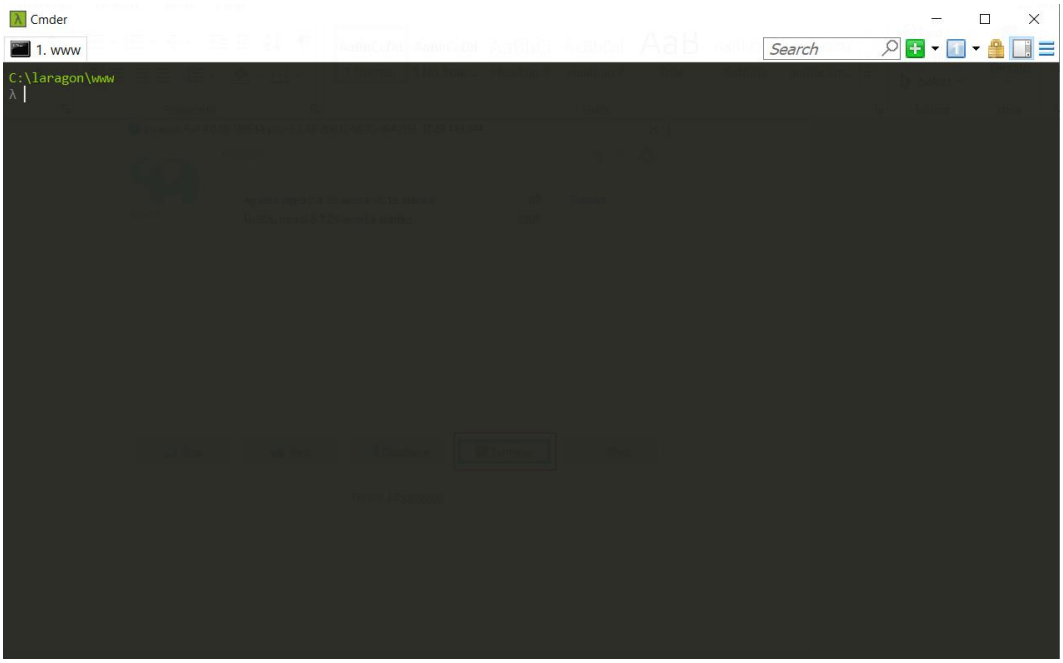


Figure 14:Terminal command prompt

Install Laravel

```
$ composer global require laravel/installer
```

Create new project as shown below

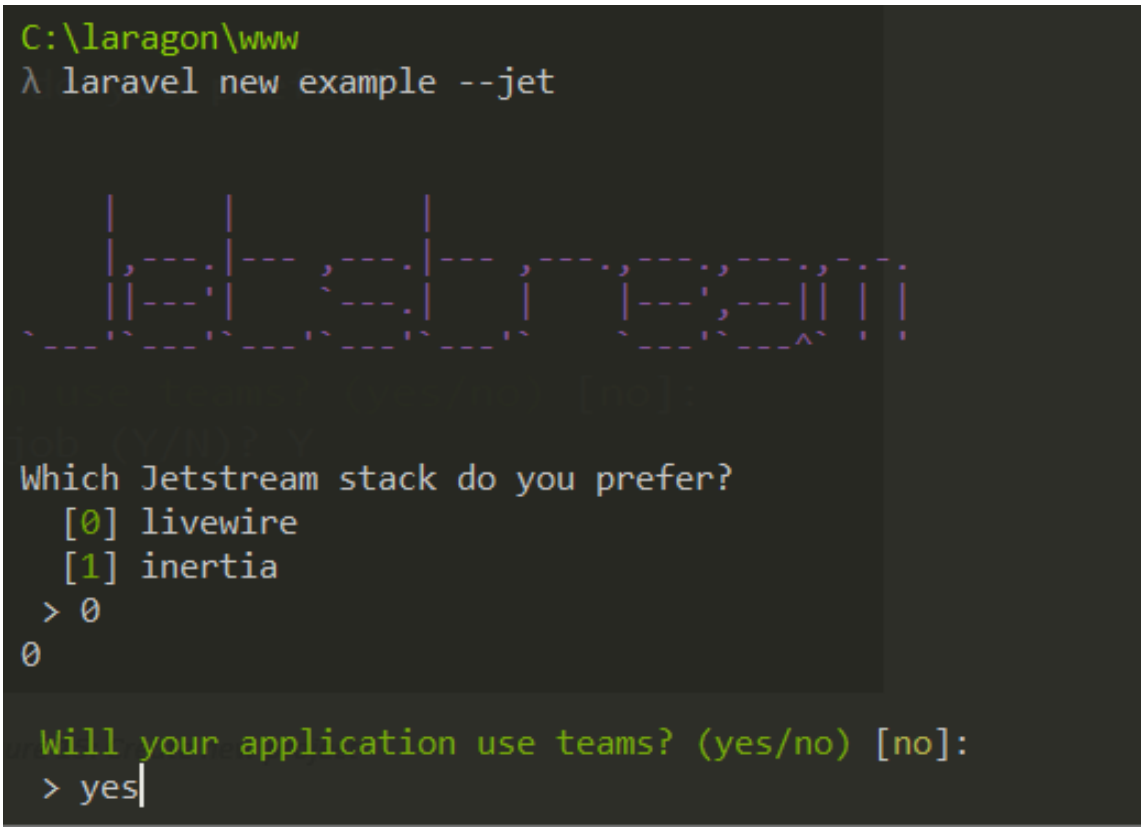


Figure 15: Create new project

```
$ Laravel new project-name --jet
```

*project-name is name of your project. For example in the figure above, example is the project name.

Choose livewire

```
$ 0
```

Depends to use team support

```
$ yes
```

We use HeidiSQL as Database Tool for this project. Click on **"Database"** button in Laragon, the session manager will prompt as shown below

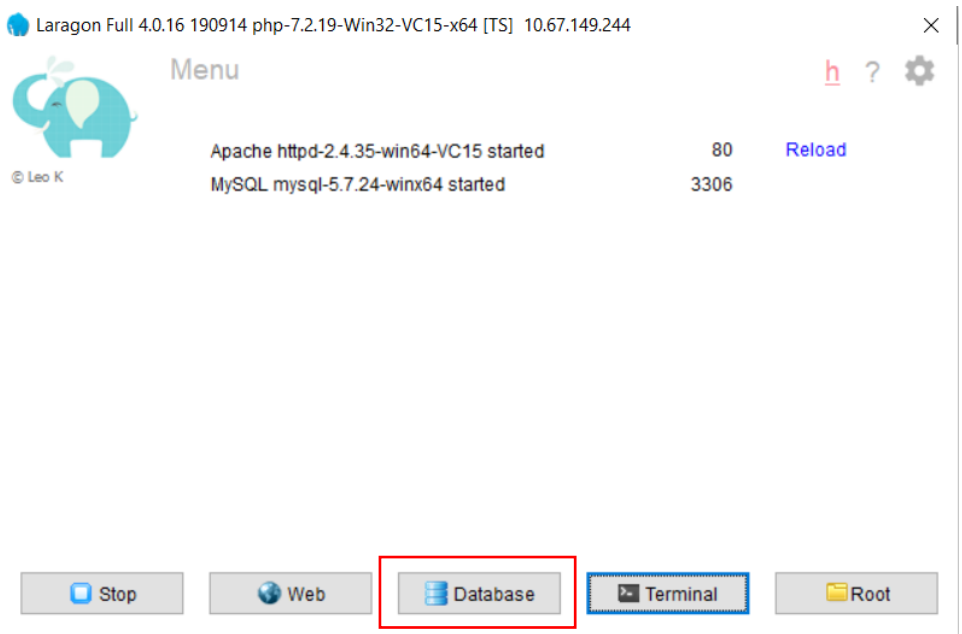


Figure 16:Laragon

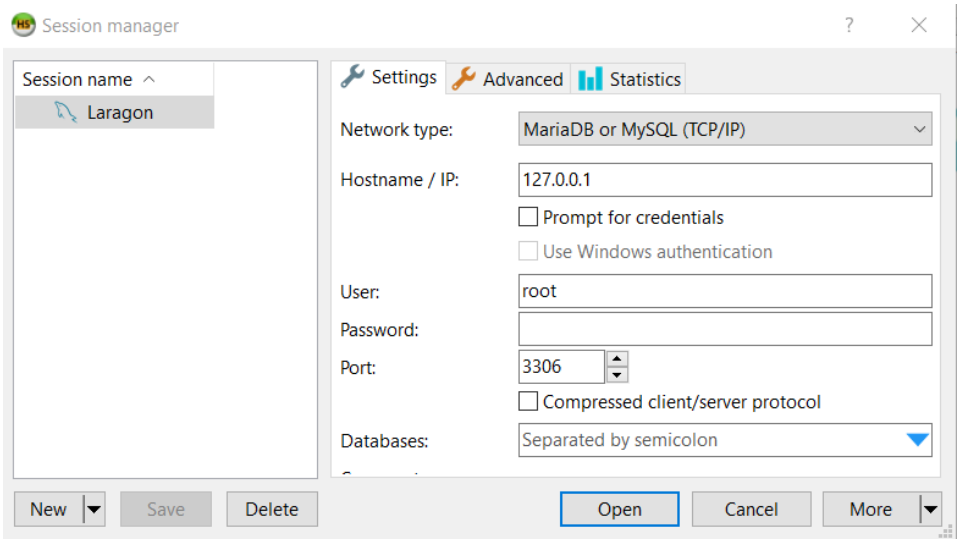


Figure 17: HeidiSQL

Click on the “New” on the left bottom and rename it to “Laragon” as shown above. Then, click “Open” to open HeidiSQL tool.

Right-click on the session name which is “Laragon”, then go to “Create new->Database” and name your database as shown below *(You can skip this if Laragon already automatically create database for you)*

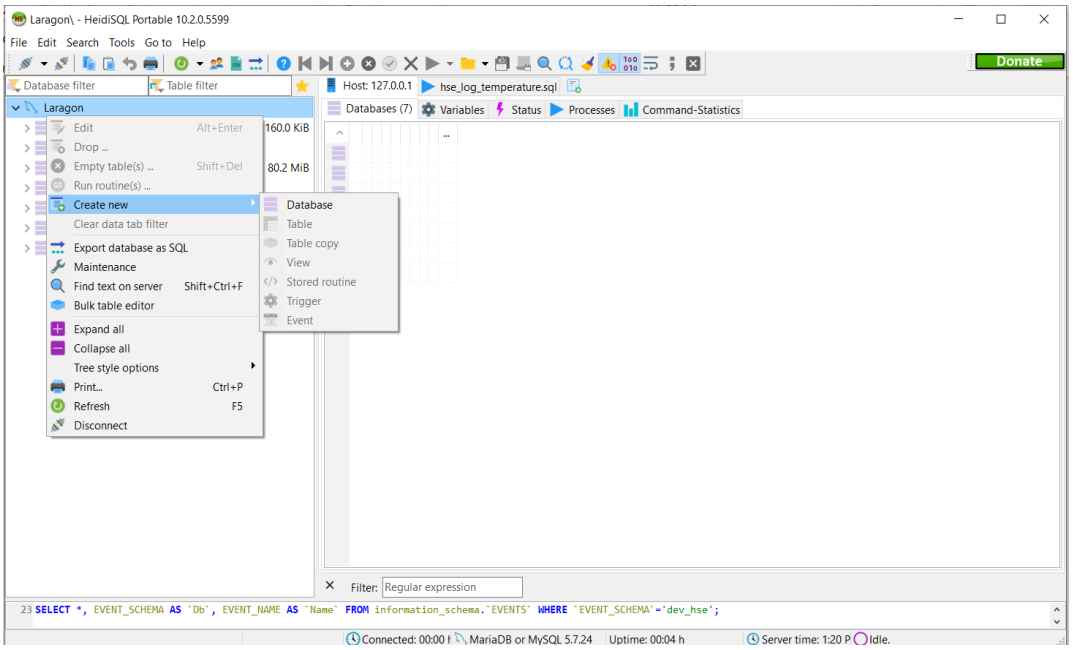


Figure 18: Create new database usin HeidiSQL

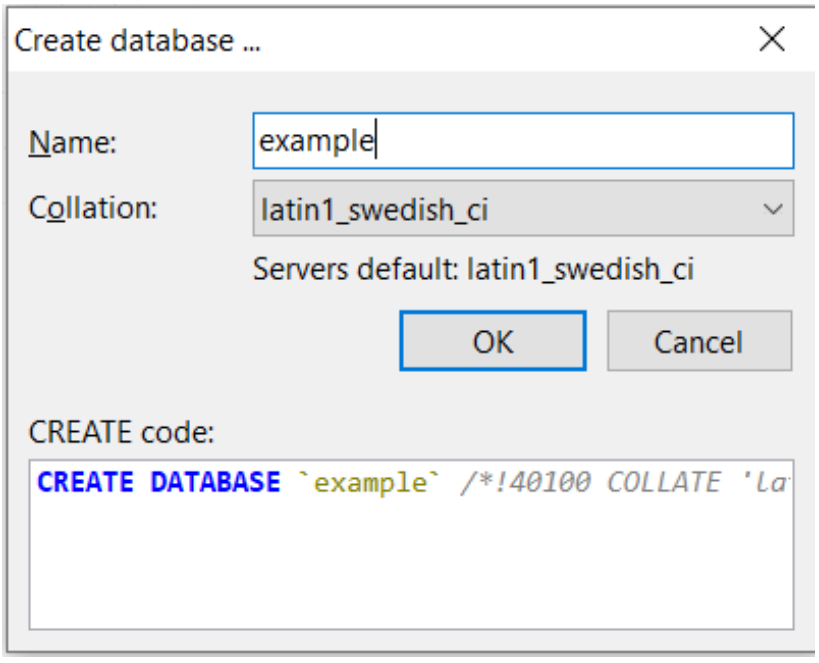


Figure 19: Name Database

Then, setup your database environment by open ".env" file in your Laravel project. Make sure "DB_DATABASE" is your database name in HeidiSQL

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=example
DB_USERNAME=root
DB_PASSWORD=
```

Figure 20: .env database setting

Then, Simply click "Terminal" button on Laragon and run the following command

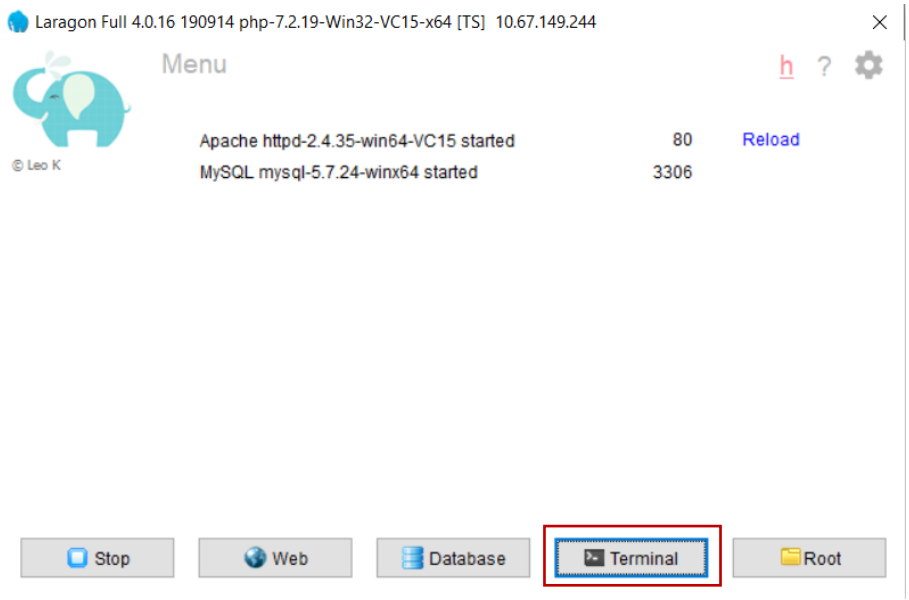


Figure 21: Laragon

Direct the terminal to your project. Example on the figure below.

```
$ cd {your-project-name}
```

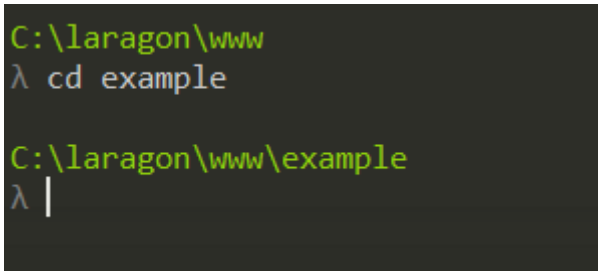


Figure 22: Direct terminal to project folder

Then, run the command below to migrate the tables that come with the project to test your connection is working as expected. Those migration files can be found in *"database/migrations/".* Command:

And then migrate;

```
$ php artisan migrate
```

Finalizing the installation by install and build your NPM dependencies and migrate your database:

```
$ npm install && npm run dev
```

```
$ php artisan migrate
```

Installing Composer package into your project;

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. Command:

```
$ composer install
```

Storage link;

In modern web development, file uploads are one of the most commonly used features and Laravel Storage provides an intuitive way to handle the file storage. Command:

```
$ php artisan storage:link
```

Generate new application key;

Application key is used for encryption and session. After you run the command below, you can check your key named "**APP_KEY**" in **".env"** file in your Laravel project. Command:

```
$ php artisan key:generate
```

To open the website, simply right-click laragon, navigate to “www” and to your project name

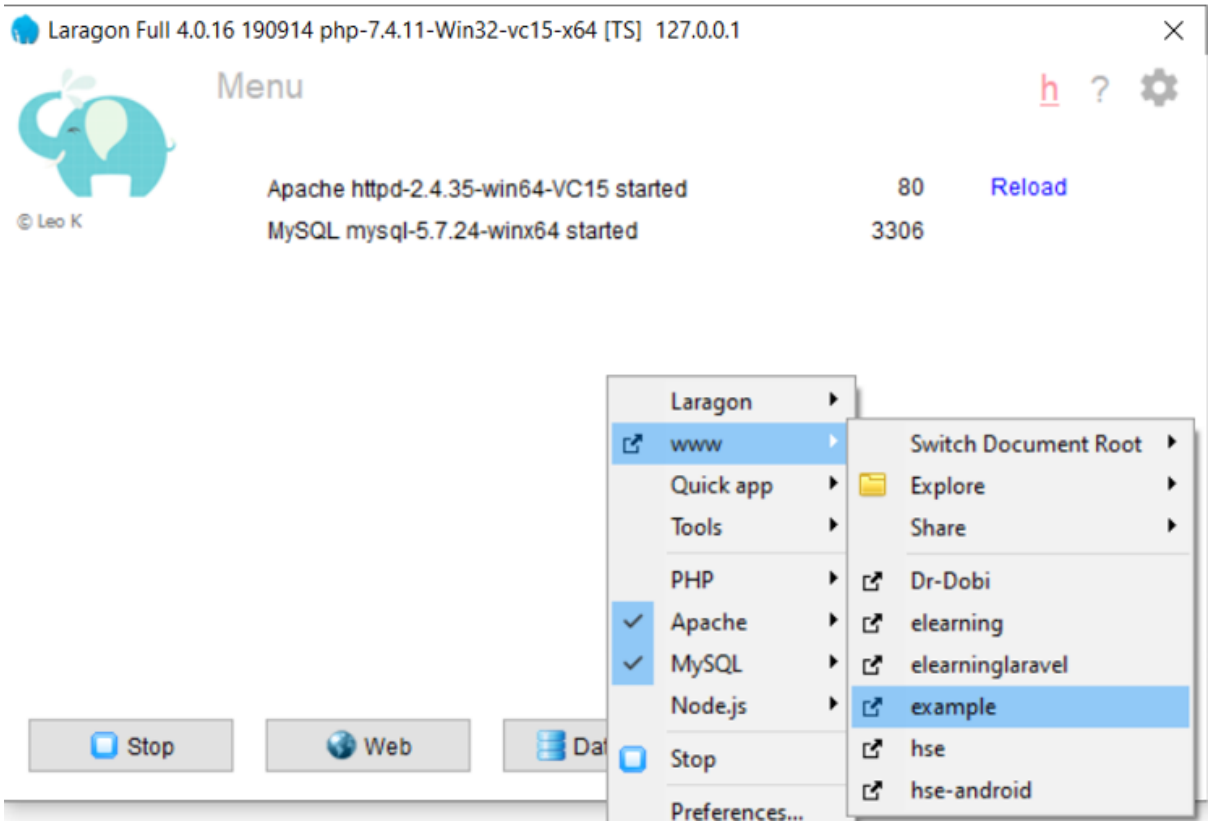


Figure 23: How to open the website through Laragon

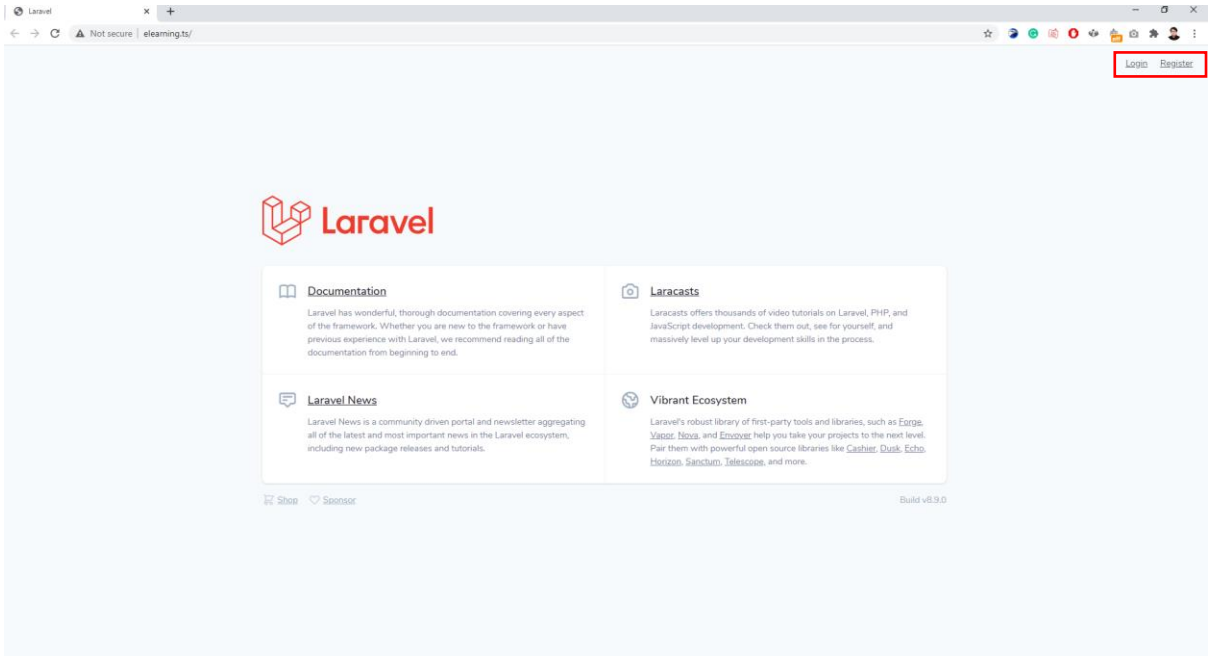


Figure 24: Laravel Website

You can see there are Login and Register button on the upper-right that Laravel has provided. Click Register button and register as a user. Look at the database, see if the data has been successfully enter in the “users” table.

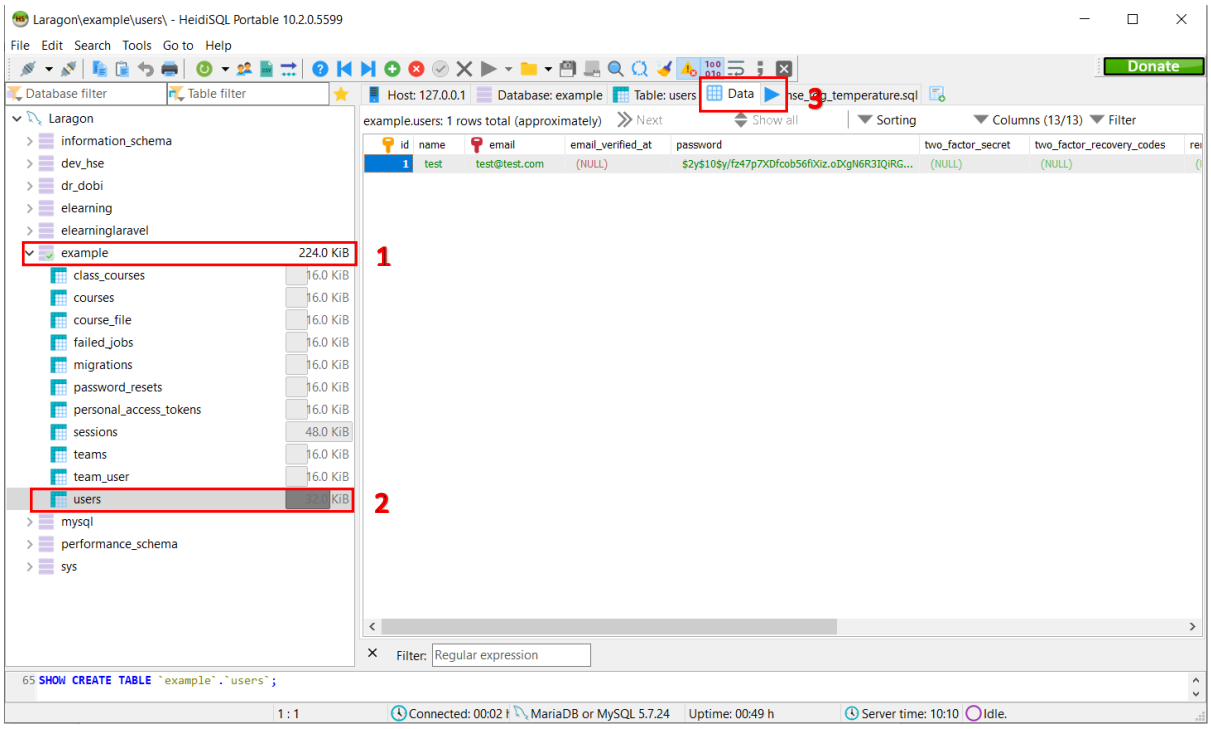


Figure 25: Check data has been successfully entered

CASE STUDY – E-LEARNING (MINI PROJECT)

This mini project require you to built a simple e-learning website using Laravel Jetstream framework. The project must have three different roles; Admin, Lecturer and Student. Every user has different and restricted view (Lecturer and Student). MVC architecture must be used in Laravel project. Various functionality provided by Laravel like migration, seeding and request must be used in this project. The functionality for each users have been described in the use case below:

Use Case

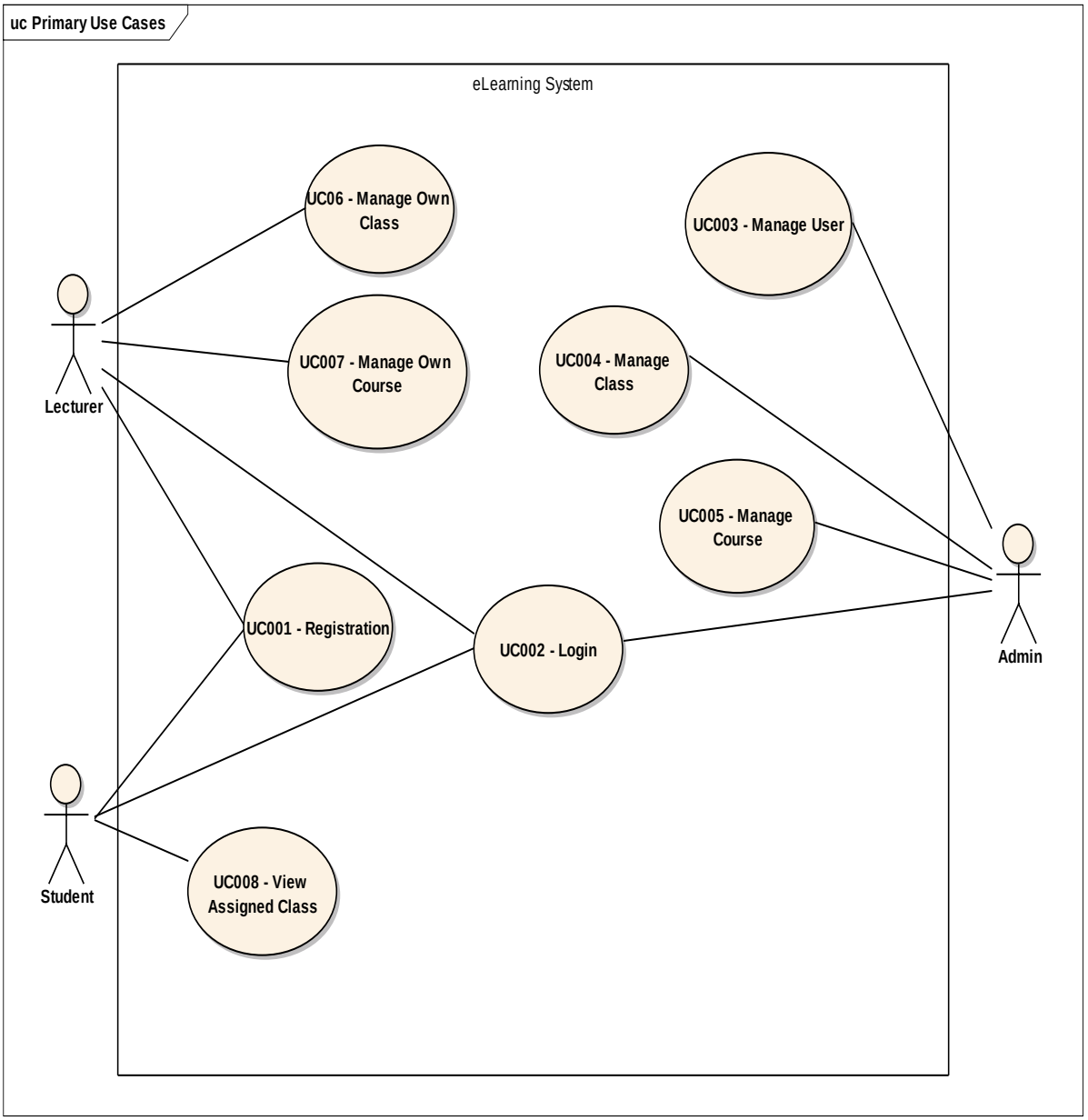


Figure 26: eLearning System Use Case

Use Case Specification

1) UC001 – Registration
   Student and lecturer can register themselves into the system

2) UC002 – Login
   Student, Lecturer and Admin can login into the system

3) UC003 – Manage User
   Admin can add, edit and delete student and lecturer information in the system

4) UC004 – Manage Class
   Admin can add, edit and delete class in the system. Admin also can assign student and lecturer into the class

5) UC005 – Manage Course
   Admin can add, edit and delete course in the system. Admin also can assign student and lecturer into the course

6) UC006 – Manage Own Course
   Lecturer can add, edit and delete learning material such as video, slide presentation etc into the assigned course

7) UC007 – Manage Own Class
   Lecturer can add, edit and delete course and student in its own class

8) UC008 – View Assigned Class
   Student can view their assigned class and courses

HOW TO SETUP – E-LEARNING (MINI PROJECT)

1) Copy the folder "**elearning**" to your laragon local web server directory. Basically the default path is : "**C:\laragon\www**".

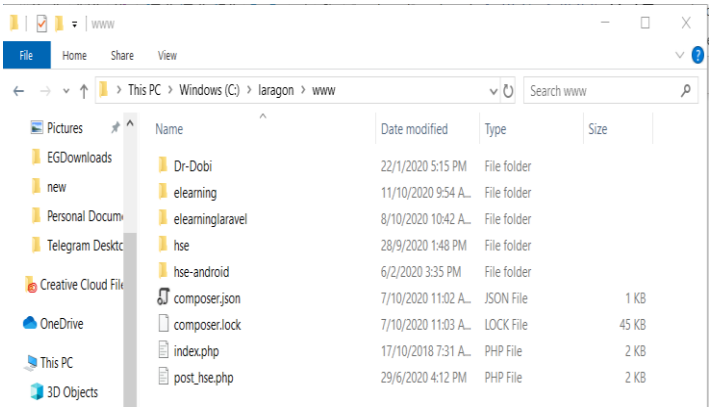


Figure 27: Laragon local web server directory

2) Open Laragon through the taskbar (if already running) and click "**Start All**". Make sure to open only one Laragon at a time. To check, view your taskbar and confirm that only one Laragon is running.

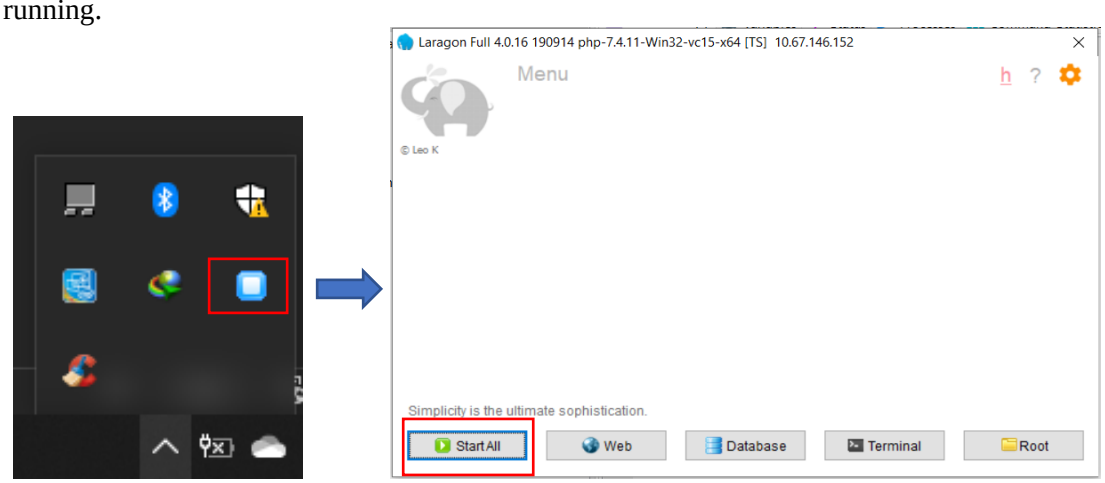


Figure 28: Laragon running at taskbar

3) Click on "**database**".

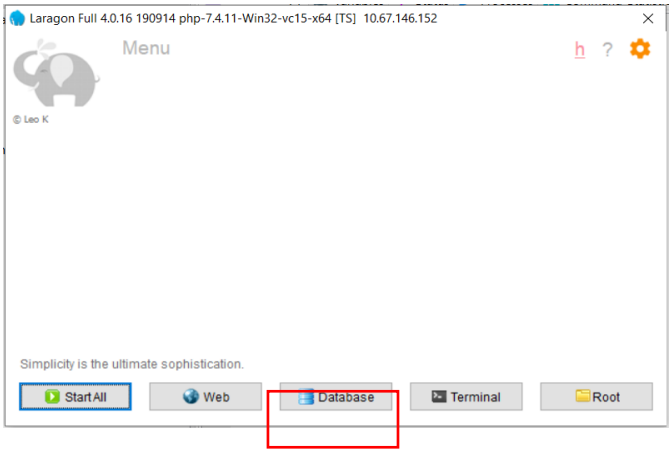


Figure 29: Laragon interface

4) Click “**Open**” to open HeidiSQL (DB Manager) tool.

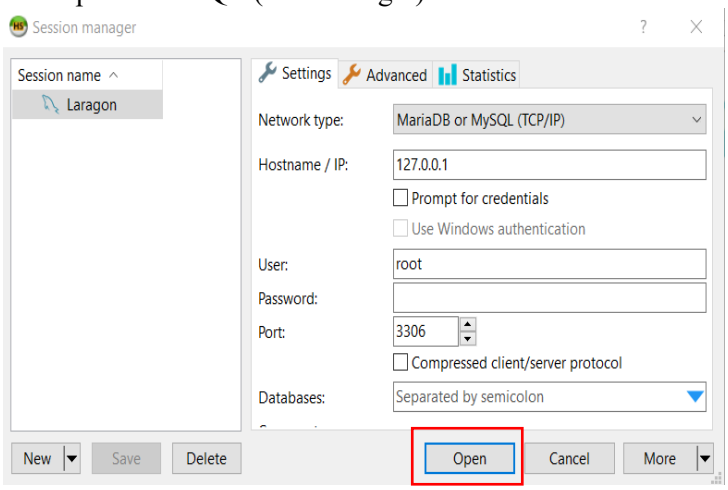


Figure 30: HeidiSQL Startup Interface

5) Right click on your server name and select “**Create New”** and choose “**database”**.

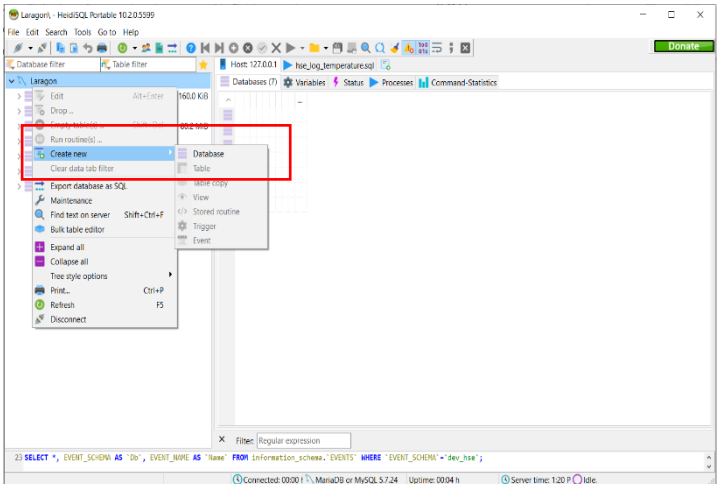


Figure 31: Create new database in HeidiSQL

6) Give your database name as “**elearning**”.

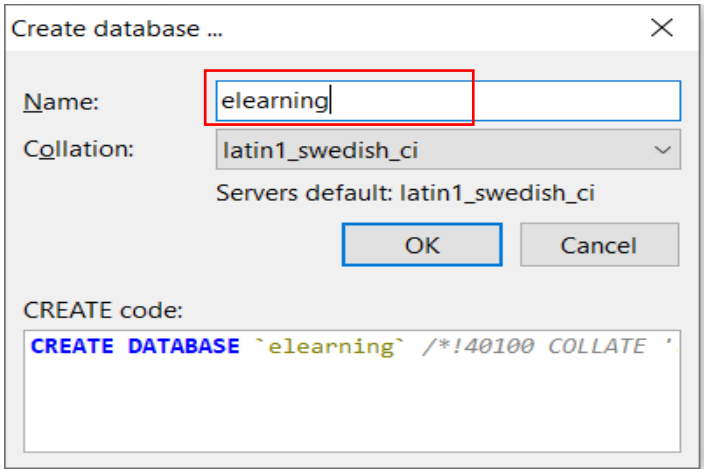


Figure 32: HeidiSQL new database name

7) To connect your project with database, we must configure database setting inside our project environment. Open your project code through Terminal provide by laragon.

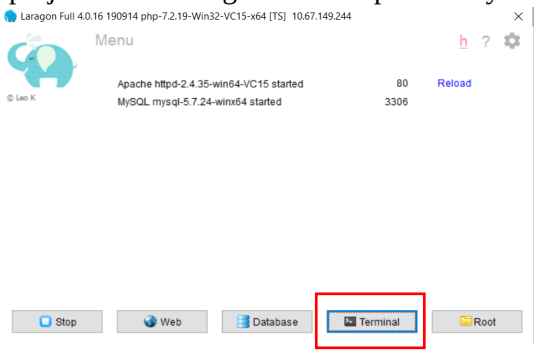


Figure 33: Opening terminal using Laragon

8) Navigate to your project folder by inserting command "**cd elearning**" into the terminal and insert command **"code ."** . This will open your code editor (VS Code).

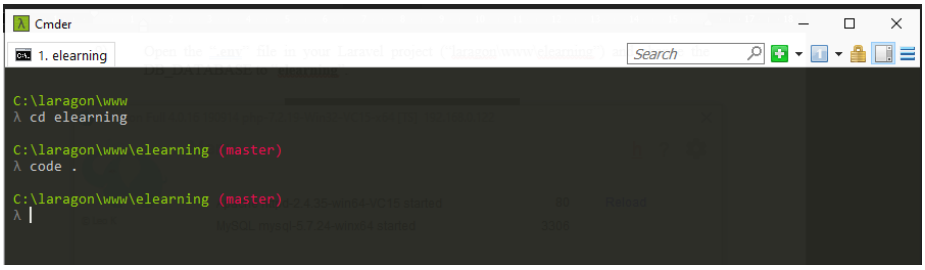


Figure 34: Navigate to project folder inside terminal

9) By using hotkey "**Ctrl+P**", search for file name "**.env**" file in your Laravel project ("laragon\www\elearning") and change the DB_DATABASE to "**elearning**". If cannot found the file open back your Terminal and run this command : **cp .env.example .env**. After that, search back the **.env** file in VS Code and change the DB_DATABASE to "**elearning**".

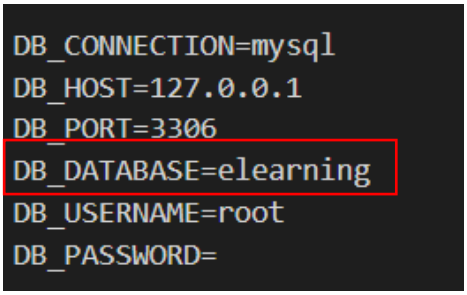


Figure 35: .env file inside root project directory

10) There are some command you need to run in the terminal before you can start running the system. Open back terminal and run the command below:

i) Install all laravel project dependencies : "**composer install**"
ii) Install and build NPM dependencies : "**npm install && npm run dev**"
iii) Database Migration : "**php artisan migrate**"
iv) Laravel storage link : "**php artisan storage:link**"
v) Laravel application key : "**php artisan key:generate**"

11) We already make **data seeder** to insert admin data into the database so you can use it to login into the system. Please run "**php artisan db:seed --class=UserSeeder**". On successful, run your web project through laragon and login using below credential.

Username : admin@mail.com
Password : 123456

12) Open Visual Studio Code and you are ready to code! Run Command: **code .**

13) Directory Structure :

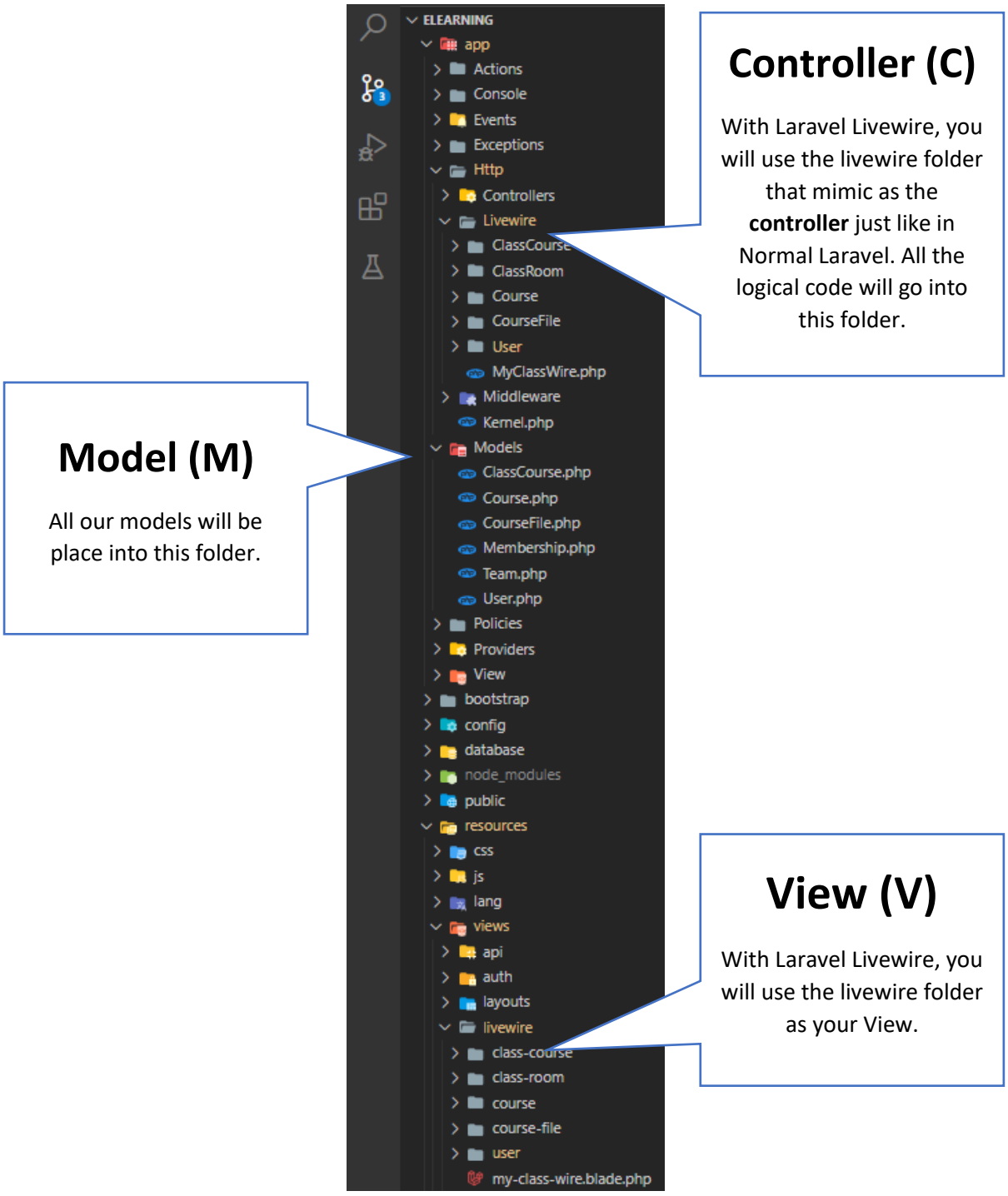


Figure 36: Project directory structure

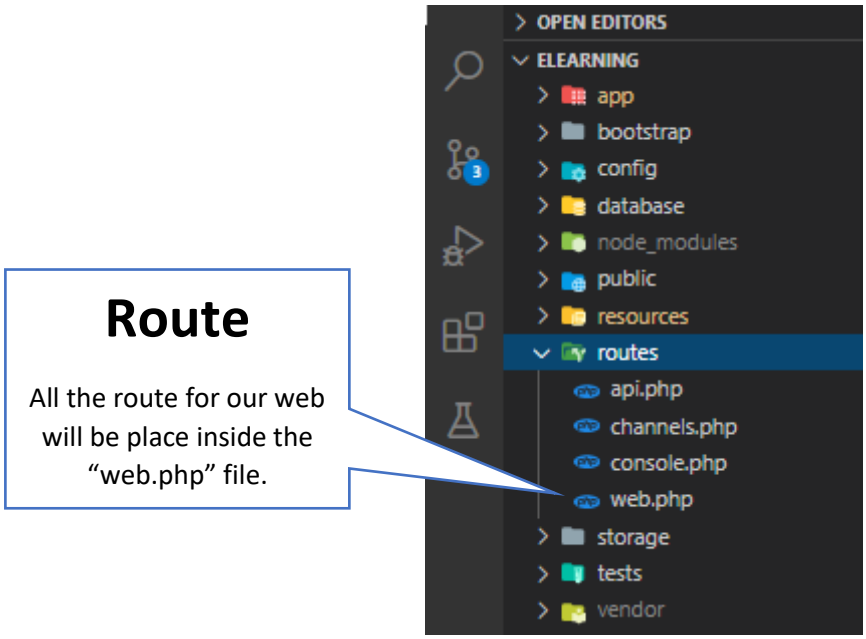


Figure 37: Route inside project directory

BRIEFING – E-LEARNING (MINI PROJECT)

As we use **Laravel Jetstream** that already comes with **Authentication Module**, you will directly dive into other module which is **User Module**, **Course Module** & **Class Module**.

We already made some modification to the Laravel Jetstream so it will match our project requirement and you can directly start with the other modules. The modification that has been made includes:

- Installing bootstrap 4
- Applied theme (sidebar & top bar)

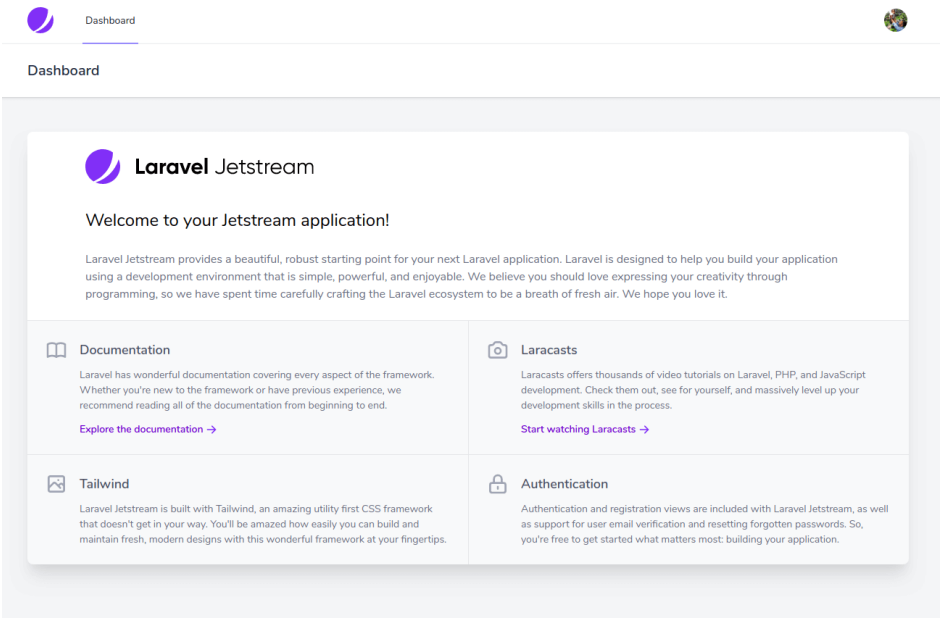


Figure 38: Default Laravel Jetstream dashboard page

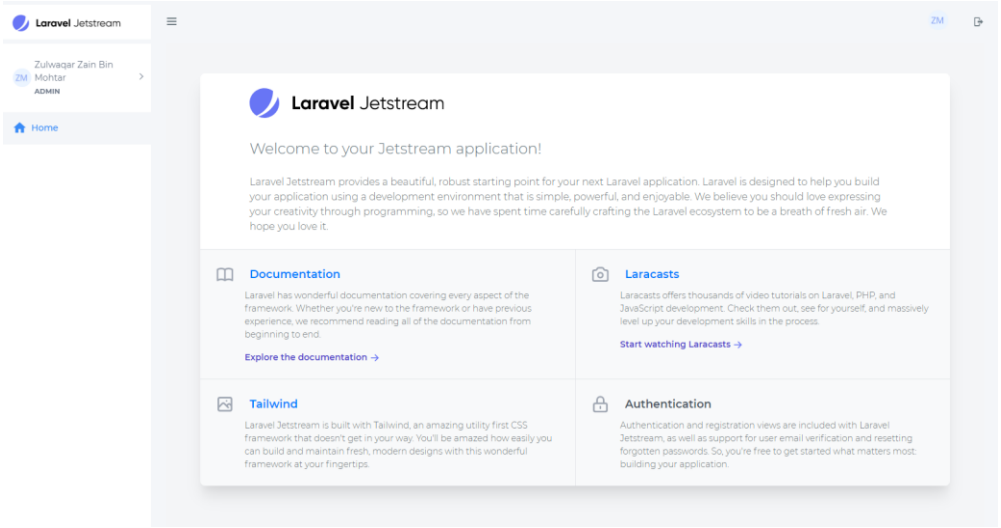


Figure 39: Dashboard page after modification

- Modification to user registration so it will include role when user register

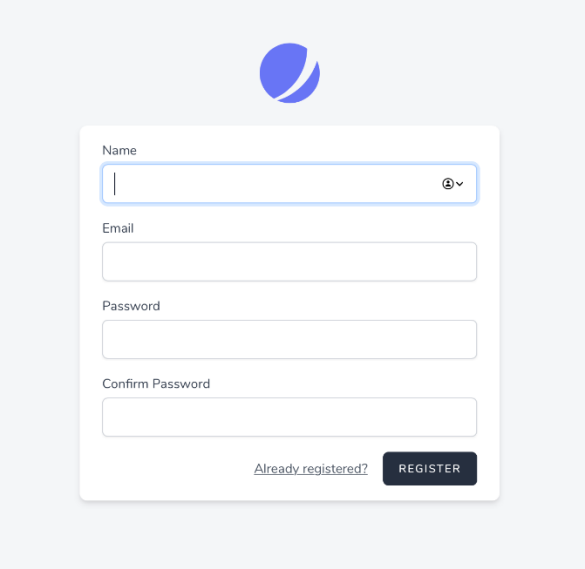


Figure 40: Default Laravel Jetstream login page

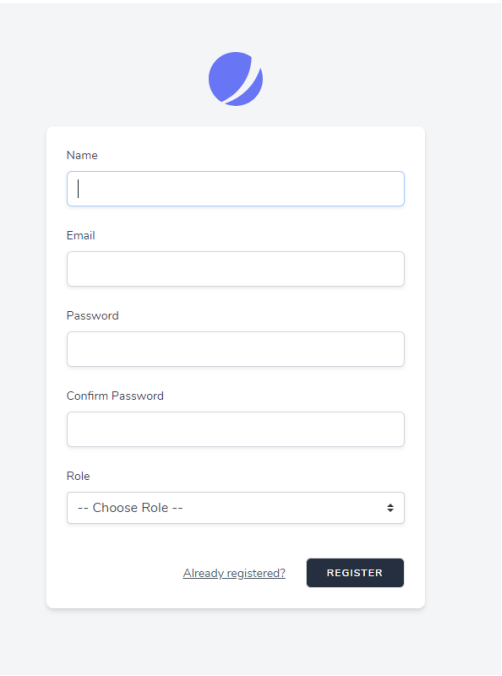


Figure 41: Login Page after modification

- Modification to team management where we change the name “team” to “class. Because the function of team is same as class, so we will reuse the team management provide by Laravel Jetstream as our Class module.

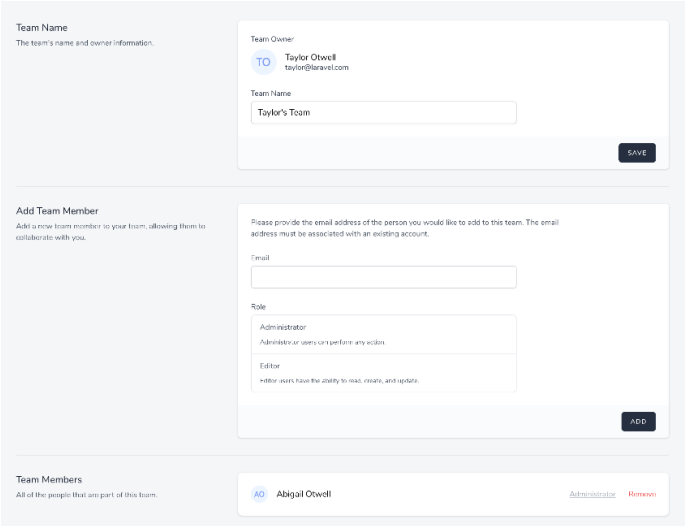


Figure 42: Default Laravel Jetstream Team Management page

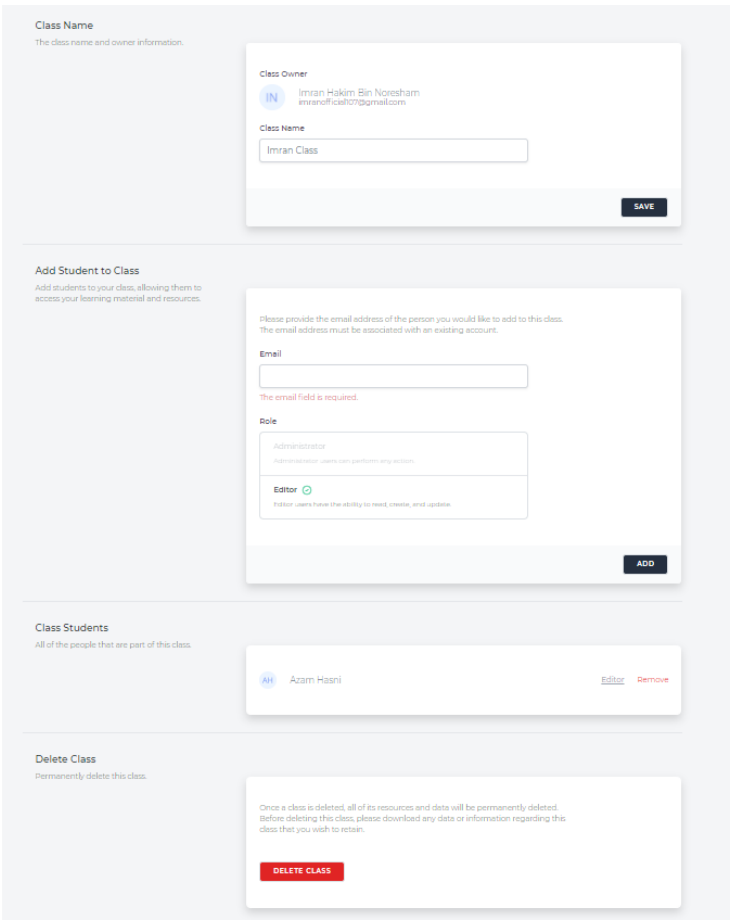


Figure 43: Modification to team page

Role: Administrator

User Module

Module Outcome – At the end of this module, you can develop the User Module for administration use. This module will include below function

i) Show user data
ii) Creating new user
iii) Edit the user
iv) Delete the user.

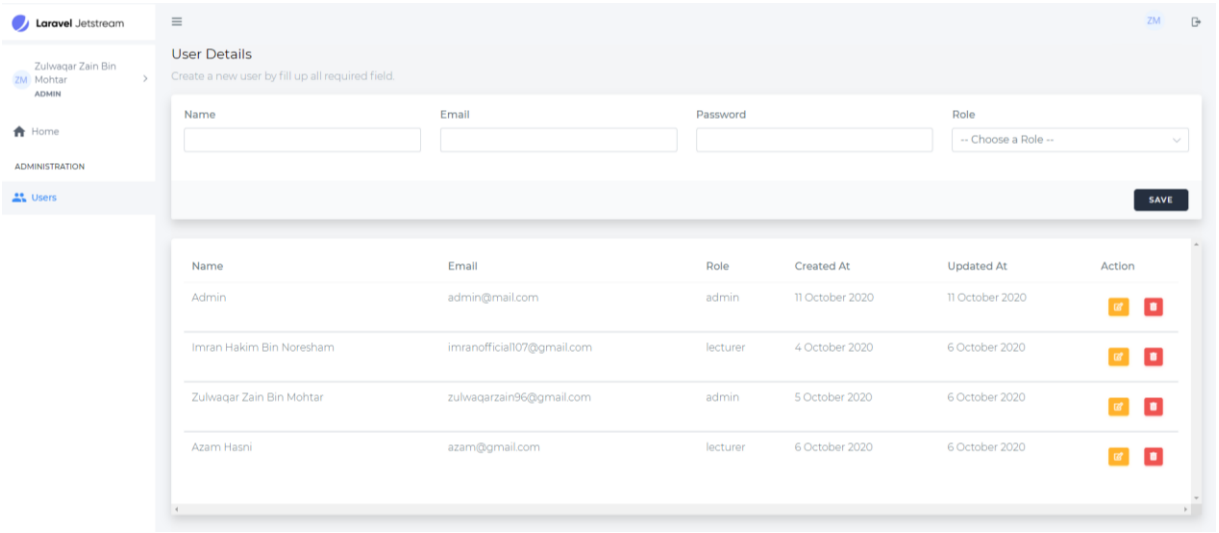


Figure 44: User management page for admin

1) To have like above page. We must have the **Model-View-Controller (MVC)** component.

2) **For Model**, because we want to build User page, so we need the User Model. But with Laravel Jetstream, they already come with user model. No need for us to create the user model. We will use the user model. You can find the model at "**app\Models**".

3) **For Controller**, Open the "**elearning-source**" folder and navigate to **Controller** folder. Copy the **"User"** folder and paste it to "**www\elearning\app\Http\Livewire**" at "**laragon\www\elearning**" project folder. This will be the logical code for user module.

4) **For View**, Open back "**elearning-source**" folder and navigate to **View** folder. Copy the **"user"** folder and paste it to "**www\elearning\resources\views\livewire**". This will be the view for user module.

   Explanation: We already provide the Controller and View file in the "**elearning-source**" folder to make things easier for you to develop. The original way of generating the component is through Laravel livewire artisan console command in terminal. The artisan command to generate the component is '**php artisan make:livewire User\\UserWire**'. This will create two empty file which is the Controller and View that located at location as below:

   a) **Controller file** : "**app\Http\Livewire\User\ UserWire.php**"
   b) **View file** : "**resources\views\livewire\user\user-wire.blade.php**".

Route

5) Route defines the URL pattern that you want to generate. Route will point to a controller. From the controller you can develop process logic or return to a view (V) or redirect to other url. Now open back your project inside the VS Code and lets take a look at the route located at web.php. For quick search, use the hotkey "**Ctrl+P**" and key in "**web.php**".

Uncomment below code. This is the route we use to navigate to user page. The **url "/user"** will point to "**UserWire.php**" class where the "**function render()**" inside "**UserWire.php**" class will be run to return the user to the view (V).

```
Route::get('/user', UserWire::class);
```

Figure 45: Insert route to UserWire class

And you need to import the class so the route can access to the right class that you have assigned.

```
App\Http\Livewire\User\UserWire;
```

Figure 46: Import class of UserWire

The "**web.php**" should look like below after you uncomment the above code.

```
<?php

use Illuminate\Support\Facades\Route;

use App\Http\Livewire\User\UserWire;

// use App\Http\Livewire\Course\CourseWire;
// use App\Http\Livewire\CourseFile\CourseFileWire;

// use App\Http\Livewire\ClassRoom\ClassRoomWire;
// use App\Http\Livewire\ClassCourse\ClassCourseWire;

// use App\Http\Livewire\MyClassWire;
/*
|--------------------------------------------------------------------------
| Web Routes
|--------------------------------------------------------------------------
|
| Here is where you can register web routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| contains the "web" middleware group. Now create something great!
|
*/

Route::get('/', function () {
    return redirect('login');
});
////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////
```

```php
//Laravel Livewire
////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////
Route::middleware('auth')->group(function () {

    Route::get('/user', UserWire::class);

    // Route::get('/course',    CourseWire::class);
    // Route::get('/coursefile',    CourseFileWire::class);

    // Route::get('/class', ClassRoomWire::class);
    // Route::get('/classcourse',   ClassCourseWire::class);

    // Route::get('/myclass',   MyClassWire::class);
});
////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////

Route::middleware(['auth:sanctum', 'verified'])->get('/dashboard', function () {
    return view('dashboard');
})->name('dashboard');
```

Figure 47: Route "web.php" source code

6) Now open the system and try insert the **url "/user"** after your local domain.
   Example: **elearning.test/user**

This will bring you to the empty user page. This is because the route pointing to the "**UserWire.php**" class where inside the class the "**function render()**" is run and the function return a view "**user-wire.blade.php**". Inside that view you will find the frontend code to generate below page.

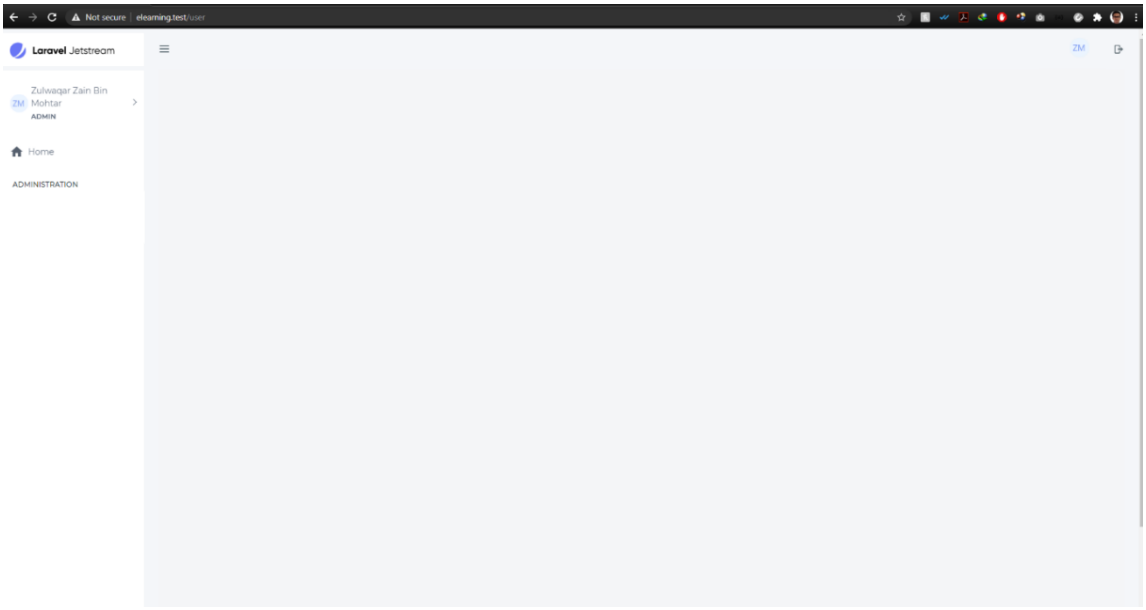


Figure 48: Empty user page

7) The sidebar and topbar of the system using layout file call "**app.blade.php**". This layout will be use by all other pages so we don't need to build the topbar and sidebar everytime we want to create a new page.
8) By using "**Ctrl+p**" search for "**app.blade.php**" and open it. We need to have User menu at the sidebar. To have that menu, uncomment the user menu under administration section. This will display the user menu at system sidebar. We use <a> tag as a link to the user page when user click on the user menu.

```
{{-------- START SECTION - ADMINISTRATION --------}}
@if (auth()->user()->role == 'admin')
<li class="nav-small-cap">ADMINISTRATION</li>
<li> <a class="waves-effect waves-dark" href="{{URL::to('user')}}" aria-
expanded="false"><i class="mdi mdi-account-multiple"></i><span class="hide-menu">Users </span></a>
</li>
{{-- <li> <a class="has-arrow waves-effect waves-dark" href="#" aria-
expanded="false"><i class="mdi mdi-bullseye"></i><span class="hide-menu">Courses </span></a>
        <ul aria-expanded="false" class="collapse">
                <li><a href="{{URL::to('course')}}">Create New Courses</a></li>
                <li><a href="{{URL::to('coursefile')}}">Add Resources to Course</a></li>
        </ul>
</li> --}}
{{-- <li> <a class="has-arrow waves-effect waves-dark" href="#" aria-
expanded="false"><i class="mdi mdi-view-dashboard"></i><span class="hide-menu">Classes </span></a>
        <ul aria-expanded="false" class="collapse">
                <li><a href="{{URL::to('class')}}">Create New Class</a></li>
                <li><a href="{{URL::to('classcourse')}}">Add Course to Class</a></li>
        </ul>
</li> --}}
@endif
{{-------- END SECTION - ADMINISTRATION --------}}
```

Figure 49: "app.blade.php" source code

9) Now open back the system and you can see user menu has been display at the sidebar but with an empty content.

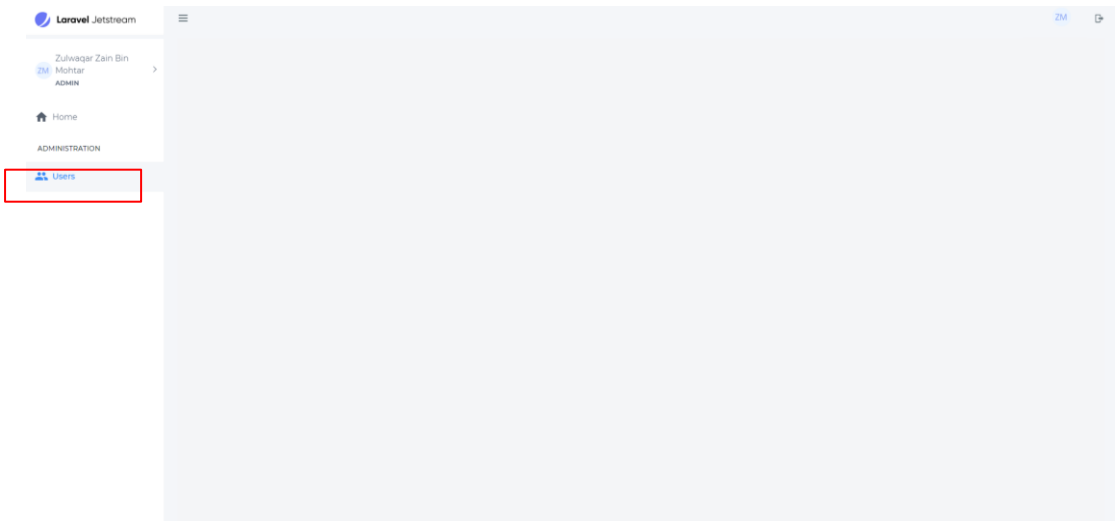


Figure 50: Empty user page with sidebar

Show User Data

10) Laravel Jetstream already come with user model. No need for us to create the user model. Search for "**UserWire.php**" and uncomment the code like below:

```php
<?php

namespace App\Http\Livewire\User;
use Livewire\Component;
use App\Models\User;

class UserWire extends Component
{

    // public $id_user;
    // public $action;

    // protected $listeners = [
    //     'refreshParent' => '$refresh',
    //     'delete'
    // ];


    // public function selectItem($id_user , $action)
    // {
    //     $this->id_user = $id_user;
    //     $this->action = $action;
    //     if($action == "update")
    //     {
    //         $this->emit('getModelId' , $this->id_user);
    //     }

    // }

    public function render()
    {
        $users = User::all();
        return view('livewire.user.user-wire')->with(compact('users'));

        // return view('livewire.user.user-wire');
    }
}
```

Import user model so we can use it to fetch user data.

The render method gets called on the initial page load & every subsequent component update. Here we fetch all user data and return the data to user view using "with(compact())"

We use eloquent ORM provide by Laravel to fetch user data from database. Eloquent ORM is a beautiful, simple ActiveRecord implementation for working with your database.

Figure 51: "UserWire.php" source code

11) Search for "**user-wire.blade.php**" and uncomment the section "**DATATABLE USER**" like below. This is a table to show the list of user.

```
<div>

{{-- START SECTION - USER FORM  --}}
{{-- @livewire('user.user-form-wire') --}}
{{-- END SECTION - USER FORM  --}}

{{-- START SECTION - DATATABLE USER  --}}
<div class="row">
    <div class="col-md-12">
        <div class="card" style="box-shadow: 0 .5rem 1rem rgba(0,0,0,.15)!important;border-
radius: 5px;">
            <div class="card-body" style="overflow:scroll">
                <table class="table table-hover" >
                    <thead>
                        <tr>
                            <th>Name</th>
                            <th>Email</th>
                            <th>Role</th>
                            <th>Created At</th>
                            <th>Updated At</th>
                            <th style='width:40px'>Action</th>
                        </tr>
                    </thead>
                    @foreach ($users as $user)
                        <tbody>
                            <tr>
                                <td>{{$user->name}}</td>
                                <td>{{$user->email}}</td>
                                <td>{{$user->role}}</td>
                                <td>{{date('j F Y', strtotime($user->created_at))}}</td>
                                <td>{{date('j F Y', strtotime($user->updated_at))}}</td>
                                 <td>

                                    <table style="border:none">
                                        <tr>
                                            <td style="border:none">
                                                <button type="button" wire:click="selectItem({{$user-
>id}} , 'update' )" class="btn btn-sm waves-effect waves-light btn-warning"><i class="far fa-
edit"></i></button>
                                            </td>
                                            <td style="border:none">
                                                <button type="button" wire:click="selectItem({{$user-
>id}} , 'delete' )" class="btn btn-sm waves-effect waves-light btn-danger data-delete" data-
form="{{$user->id}}"><i class="fas fa-trash-alt"></i></button>
                                            </td>
```

User data send from controller is use to show all user exist in system.

```
                                    </tr>
                                </table>
                            </td>
                                    </tr>
                                </table>
                            </td>

                        </tr>
                    </tbody>
                @endforeach
            </table>
        </div>
    </div>
</div>
{{-- END SECTION - DATATABLE USER  --}}

@push('scripts')

{{-- START SECTION - SCRIPT FOR DELETE BUTTON  --}}
{{-- <script>
  document.addEventListener('livewire:load', function () {


    $(document).on("click", ".data-delete", function (e)
        {
            e.preventDefault();
            swal({
            title: "Are you sure?",
            text: "Once deleted, you will not be able to recover!",
            icon: "warning",
            buttons: true,
            dangerMode: true,
            })
            .then((willDelete) => {
            if (willDelete) {
                e.preventDefault();
                Livewire.emit('delete')
            }
            });
        });

  })
</script> --}}
{{-- END SECTION - SCRIPT FOR DELETE BUTTON  --}}

@endpush
</div>
```

Figure 52: "user-wire.blade" source code

12) Now open the system and you will see table of user with its data.

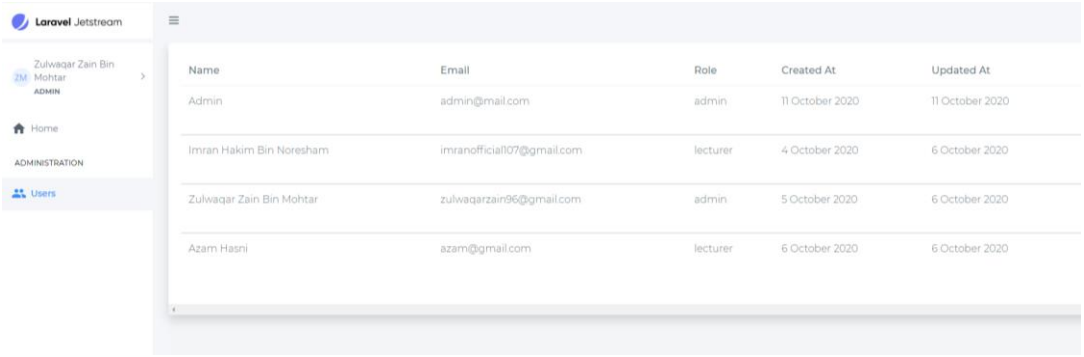


Figure 53: User page with table of user

Creating New User Function

13) Now, we already have view for table of user. To have a create function, search for "**UserFormWire.php**" and uncomment the code like below:

```php
<?php

namespace App\Http\Livewire\User;
use Livewire\Component;
use App\Models\User;
use Illuminate\Support\Facades\Hash;

class UserFormWire extends Component
{
    public $name;
    public $email;
    public $password;
    public $role;
    // public $model_id;

    // protected $listeners = [
    //     'getModelId'
    // ];

    // public function getModelId($model_id)
    // {
    //     $user = User::find($model_id);
    //     $this->model_id = $user->id;
    //     $this->name = $user->name;
    //     $this->email = $user->email;
    //     $this->role = $user->role;
    // }

    public function store()
    {
        // if($this->model_id)
        // {
        //     $this->validate([
        //         'name' => 'required|string|max:255',
        //         'email' => 'required|string|email|max:255',
        //         'role' => 'required',
        //     ]);

        //     $update = User::find($this->model_id);
        //     $update->name = $this->name;
        //     $update->email = $this->email;

        //     if(($this->password != null) || ($this->password != ''))
        //     {
```

Import user model so we can use it to fetch user data.

Import the hash package to hash the password when creating user.

Public properties in Livewire are automatically made available to the view. You need to declare all the properties needed in view(V) here.

```
        //          $this->validate([
        //              'password' => 'required|string|min:6',
        //          ]);
        //          $update->password = Hash::make($this->password);
        //      }
        //      $update->role = $this->role;
        //      $update->save();
        //      session()->flash('message', 'User suc
        // }
        // else
        // {
            $this->validate([
                'name' => 'required|string|max:255',
                'email' => 'required|string|email|max:255|unique:users',
                'password' => 'required|string|min:6',
                'role' => 'required',
            ]);

            $add = New User;
            $add->name = $this->name;
            $add->email = $this->email;
            $add->password = Hash::make($this->password);
            $add->role = $this->role;
            $add->save();

            session()->flash('message', 'New user successfully added');
        // }

        // $this->emit('refreshParent');

    }

    public function render()
    {
        return view('livewire.user.user-form-wire');
    }
}
```

Store function for creating new user. We must first validate the input from view(V) . If validation is fail, error message is return to view.

If validation is successful. We use the eloquent ORM to store user data into database.

Send message to view on successful store user into database.

Figure 54: "UserFormWire.php"

14) Now go to "**user-form-wire.blade.php**" and uncomment the section "**USER FORM**".

```
<div>
    {{-- START SECTION - USER FORM  --}}
    <div class="row">
        <div class="col-lg-4">
            <h3 style="color:black">User Details</h3>
            <p class="text-muted">Create a new user by fill up all required field.</p>
        </div>
```

```
        <div class="col-lg-12">

            <form wire:submit.prevent="store">
            <div class="card" style="box-shadow: 0 .5rem 1
radius: 5px;">
                <div class="card-body">
                    <div class="row">
                        <div class="col-md-3">
                            <div class="form-group">
                                <label class="control-label" style
                                <input wire:model="name" type="text" id="name" name="name" class="form
-control" >
                                @error('name') <span class="error">{{ $message }}</span> @enderror
                            </div>
                        </div>

                        <div class="col-md-3">
                            <div class="form-group">
                                <label class="control-labell" style="font-weight:500">Email</label>
                                <input wire:model="email" type="email" id="email" name="email" class="
form-control" >
                                @error('email') <span class="error">{{ $message }}</span> @enderror
                            </div>
                        </div>

                        <div class="col-md-3">
                            <div class="form-group">
                                <label class="control-label" style="font-weight:500">Password</label>
                                <input wire:model="password" type="password" id="password" name="passw
ord" class="form-control" placeholder="">
                                @error('password') <span class="error">{{ $message }}</span> @enderror
                            </div>
                        </div>

                        <div class="col-md-3">
                            <div class="form-group">
                                <label class="control-label" style="font-weight:500">Role</label>
                                <select wire:model="role" name="role" id="role" class="form-
control custom-select" data-placeholder="Choose a Role" tabindex="1">
                                    <option value="">-- Choose a Role --</option>
                                    <option value="admin">Admin</option>
                                    <option value="lecturer">Lecturer</option>
                                    <option value="student">Student</option>
                                </select>
                                @error('role') <span class="error">{{ $message }}</span> @enderror
                            </div>
                        </div>
                    </div>
```

Laravel livewire use wire:submit.prevent = ”function in controller” to submit form.

Laravel livewire use wire:model to synchronize value of form input and public properties in the controller.

If validation is fail at controller. It will return the error message to view and the message will show up here.

```
                </div>

                <div class="card-footer" style="background-color: #f9fafb !important;border-
top: none;">
                    <div class="row">
                        <div class="col-md-12 text-right">
                            @if (session()->has('message'))
                                {{ session('message') }}

                            @endif
                            <button type="submit" class="inline-flex items-center px-4 py-2 bg-gray-
800 border border-transparent rounded-md font-semibold text-xs text-white uppercase tracking-
widest hover:bg-gray-700 active:bg-gray-900 focus:outline-none focus:border-gray-900 focus:shadow-
outline-gray disabled:opacity-25 transition ease-in-out duration-150">
                                Save
                            </button>
                        </div>
                    </div>
                </div>

            </div>

        </form>

        </div>
    </div>
    {{-- END SECTION - USER FORM  --}}
</div>
```

Figure 55: "user-form-wire.blade.php" source code

15) To include "**user-form-wire.blade**" component inside "**user-wire.blade**". Go to "**user-wire.blade**" and uncomment the section "**USER FORM**".

```
{{-- START SECTION - USER FORM  --}}
@livewire('user.user-form-wire')
{{-- END SECTION - USER FORM  --}}
```

Figure 56: "user-wire.blade" User Form section

16) Now go back to system and you will see the create form for user , you can create user after fill in all the required field and click the save button. Go on and try. On successful , you will got a message notified that user successfully created but if you notice, the user table does not update and show the latest user that you just create in real-time, this mean you need to refresh the page to see the change.

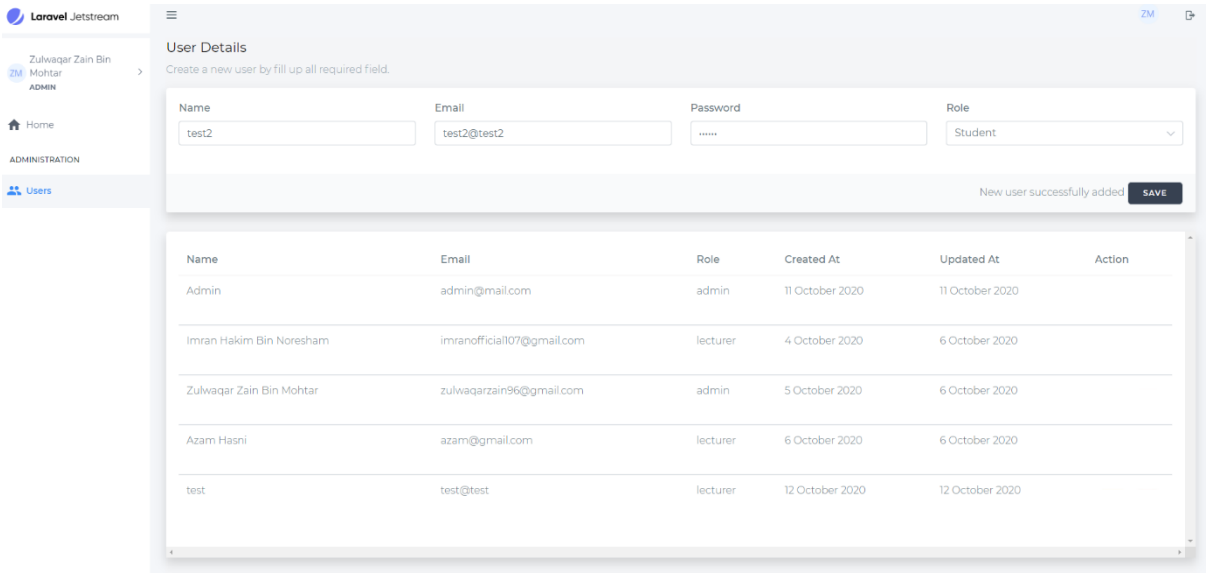


Figure 57: User page with table of user & create user component

17) To have a real-time update on the user table go back to "**UserWire.php**" and uncomment the code other than `// 'delete'`. This is the listener to listen on succesful store event of user.

```
protected $listeners = [
        'refreshParent' => '$refresh',
        // 'delete'
    ];
```

Figure 58: "UserWire.php" listener

18) After that go to "**UserFormWire.php**" and uncomment below code. This code is use to triggered the listener.

```
    $this->emit('refreshParent');
```

Figure 59: "UserFormWire.php" emitting to listener

19) We use listener on store function in "**UserFormWire.php**" to detect if user have been successfully created. First, we define the listener in "**UserWire.php**" just like in the code above. Then we call the listener by using **emit('refreshParent').** When the listener is triggered, the **magic keyword** provide by Laravel livewire which is "**$refresh**" will refresh the "**UserWire.php**" component which is the user table so we will have the latest data that we just created. Go on and try to insert back new user.

Edit the User Function

20) If you noticed we already uncomment some code inside "**user-wire.blade**". The code will fire when user click on the button. The **wire:click** event will run the **selectItem** function inside "**UserWire.php**" to set all the public properties to a specific user that we have choose.

```
<td style="border:none">
     <button type="button" wire:click="selectItem({{$user->id}} , 'update' )" class="btn btn-sm waves-
effect waves-light btn-warning"><i class="far fa-edit"></i></button>
</td>
```

21) Go to "**UserWire.php**" and uncomment like below code. The "**selectItem**" function will run on "**wire:click**" that we define at "**user-wire.blade.php**". We write the logic '**if($action == "update")**' if the parameter "**$action**" pass an "**update**" value, thus it will emit listener name "**getModelId**" with parameter of **user id**.

```
public function selectItem($id_user , $action)
    {
        $this->id_user = $id_user;
        $this->action = $action;
        if($action == "update")
        {
            $this->emit('getModelId' , $this->id_user);
        }
    }
```

Figure 60: "UserWire.php" selectItem function

22) After that, go to "**UserFormWire.php**" and uncomment below code. This code is to listen to an event call "**getModelID**" from the "**UserWire.php**". The listener will run the function **getModelId** and set all the data of user. This data will synchronise automatically with the view where we define **wire:model** at "**user-form-wire.blade.php**".

```
<?php
namespace App\Http\Livewire\User;
use Livewire\Component;
use App\Models\User;
// use Illuminate\Support\Facades\Hash;

class UserFormWire extends Component
{
    public $name;
    public $email;
    public $password;
    public $role;
    public $model_id;

    protected $listeners = [
        'getModelId'
    ];
```

```php
public function getModelId($model_id)
{
    $user = User::find($model_id);
    $this->model_id = $user->id;
    $this->name = $user->name;
    $this->email = $user->email;
    $this->role = $user->role;
}
```

Figure 61: "UserFormWire.php"

23) Now go on to system and click at the edit button and you will see the data that you want to edit will automatically fill in the user form component.

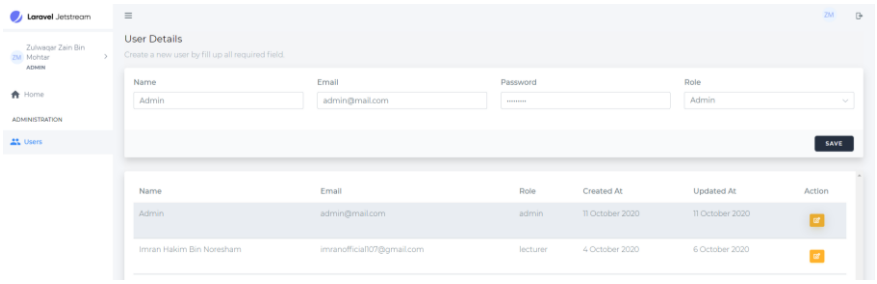


Figure 62: User page with applied real-time update

24) But If you click on the save button, the data will create a new user. To enable the edit function let's do some logic on the store function. Go back to "**UserFormWire.php**" and uncomment all the code located in the store function.

```php
public function store()
    {
        if($this->model_id)
        {
            $this->validate([
                'name' => 'required|string|max:255',
                'email' => 'required|string|email|max:255',
                'role' => 'required',
            ]);
            $update = User::find($this->model_id);
            $update->name = $this->name;
            $update->email = $this->email;

            if(($this->password != null) || ($this->password != ''))
            {
                $this->validate([
                    'password' => 'required|string|min:6',
                ]);
                $update->password = Hash::make($this->password);
            }
            $update->role = $this->role;
            $update->save();
```

If this function detect "model_id" has a value, then it will do an update to the exist user data.

```
            session()->flash('message', 'User successfully updated');
        }
        else
        {
            $this->validate([
                'name' => 'required|string|max:255',
                'email' => 'required|string|email|max:255|unique:users',
                'password' => 'required|string|min:6',
                'role' => 'required',
            ]);
            $add = New User;
            $add->name = $this->name;
            $add->email = $this->email;
            $add->password = Hash::make($this->password);
            $add->role = $this->role;
            $add->save();
            session()->flash('message', 'New user successfully added');
        }
        $this->emit('refreshParent');
    }
```

Figure 63: "UserFormWire.php" source code

25) Lets jump back to the system and edit some user. The edit function will be successful.

Delete the User Function

26) If you noticed we already uncomment some code inside "**user-wire.blade**". The code will fire when user click on the button. The **wire:click** event will run the **selectItem** function inside "**UserWire.php**" to set all the public properties to a specific user that we have choose.

```
<td style="border:none">
     <button type="button" wire:click="selectItem({{$user->id}} , 'delete' )" class="btn btn-sm waves-
effect waves-light btn-danger data-delete" data-form="{{$user->id}}"><i class="fas fa-trash-
alt"></i></button>
</td>
```

27) Beside using wire:click event, to include delete function, go to "**user-wire.blade.php**" and uncomment on section "**Script for Delete Button**" like below. This script use JS Package - SweetAlert to popup a message of confirmation on deletion of user.

```
{{-- START SECTION - SCRIPT FOR DELETE BUTTON  --}}
<script>
  document.addEventListener('livewire:load', function () {
    $(document).on("click", ".data-delete", function (e) {
            e.preventDefault();
            swal({
            title: "Are you sure?",
            text: "Once deleted, you will not be able to recover!",
            icon: "warning",
            buttons: true,
            dangerMode: true,
            })
            .then((willDelete) => {
            if (willDelete) {
                e.preventDefault();
                Livewire.emit('delete')
            }
            });
        });
  })
</script>
{{-- END SECTION - SCRIPT FOR DELETE BUTTON  --}}
```

This script will run on user click on tag that has a "data-delete" class. If you see at step 27, we define a "data-delete" class at the delete button.

This emit function of delete will sent a message to the listener located at "UserWire.php" and call the function delete.

Figure 64: "user-wire.blade.php" Script for Delete Button section

28) After that navigate to “UserWire.php” and uncomment below code inside the file.

```php
public $id_user;
public $action;
protected $listeners = [
    'refreshParent' => '$refresh',
    'delete'
];
public function delete()
{
    $user = User::find($this->id_user);
    $user->delete();
}
```

Because we already set the public properties on wire:click event at step 28. We can use the value to search for the user and delete it.

Figure 65: “UserWire.php”

29) Now lets navigate back to system and try the delete function by clicking the delete button as below.

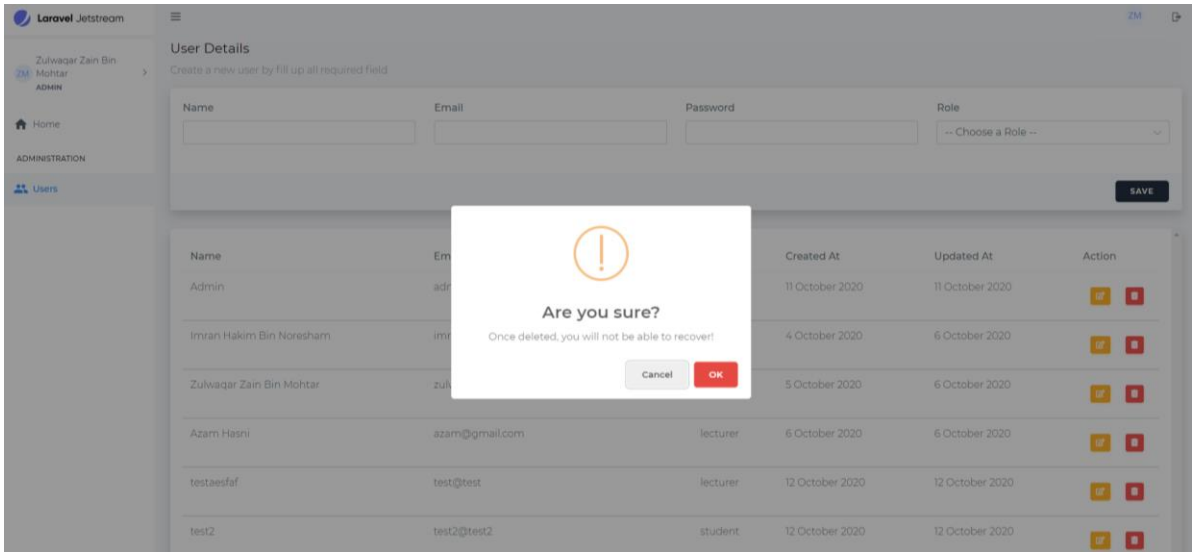


Figure 66: User Page with delete function

Course Module

Submodule – Create New Course

Submodule Outcome – At the end of this module, you can develop the Submodule Create New Course for administration use. This module will include below function:

i) Show the course data
ii) Creating the course
iii) Edit the course
iv) Delete the course.

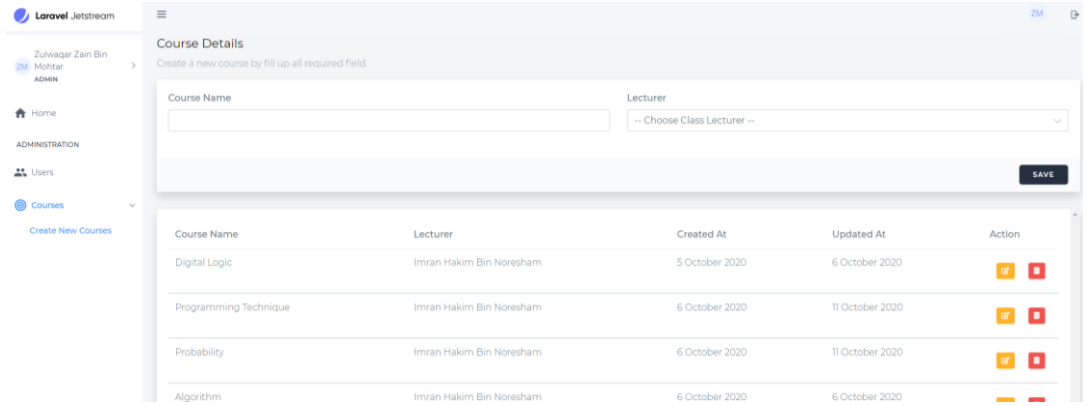


Figure 67: Course management page for admin

1) To have like above page. We must have the **Model-View-Controller (MVC)** component.

2) **For Model**, open the "**elearning-source**" folder and navigate to **Model** folder. Copy the **"Course.php"** file and paste it to "**www\elearning\app\Models\"** in your project folder. This will be the model for course.

   Explanation: We already provide the Model file in the "**elearning-source**" folder to make things easier for you to develop. The original way of generating Course model is through Laravel artisan command in terminal which is "**php artisan make:model Course**". This will create model file inside "**app\Models\Course.php**".

3) **For Controller**, open the "**elearning-source**" folder and navigate to **Controller** folder. Copy the **"Course"** folder and paste it to "**laragon\www\elearning\app\Http\Livewire**" (project folder)**.** This will be the logical code for this submodule.

4) **For View,** open back "**elearning-source**" folder and navigate to **View** folder. Copy the **"course"** folder and paste it to "**laragon\www\elearning\resources\views\livewire**" (project folder). This will be the view for this submodule.

   Explanation: We also provide the Controller and View file in the "**elearning-source**" folder to make things easier for you to develop. The original way of generating the component is through artisan console command in terminal. The artisan command to generate the component is '**php artisan make:livewire Course\\CourseWire**'. This will create the Controller file inside "**app\Http\Livewire\Course\ CourseWire.php**" and View file inside "**resources\views\livewire\course\course-wire.php**".

Route

5) Route defines the URL pattern that you want to generate. Route will point to a controller. From the controller you can develop process logic or return to a view (V) or redirect to other url. Now open back your project inside the VS Code and lets take a look at the route located at web.php. For quick search, use the hotkey "**Ctrl+P**" and key in "**web.php**".

Uncomment below code. This is route we use to navigate to course page. The **url "/course"** will point to "**CourseWire.php**" class where the "**function render()**" inside "**CourseWire.php**" class will be run to return the user to the view (V)

```
Route::get('/course', CourseWire::class);
```

Figure 68: Import class of CourseWire

And you need to import the class so the route can access to the right class that you have assigned.

```
use App\Http\Livewire\Course\CourseWire;
```

Figure 69: Insert route to CourseWire class

The "**web.php**" should look like below after you uncomment the above code.

```
<?php

use Illuminate\Support\Facades\Route;

use App\Http\Livewire\User\UserWire;

use App\Http\Livewire\Course\CourseWire;
// use App\Http\Livewire\CourseFile\CourseFileWire;

// use App\Http\Livewire\ClassRoom\ClassRoomWire;
// use App\Http\Livewire\ClassCourse\ClassCourseWire;

// use App\Http\Livewire\MyClassWire;
/*
|--------------------------------------------------------------------------
| Web Routes
|--------------------------------------------------------------------------
|
| Here is where you can register web routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| contains the "web" middleware group. Now create something great!
|
*/

Route::get('/', function () {
    return redirect('login');
});
////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////
```

```php
//Laravel Livewire
////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////
Route::middleware('auth')->group(function () {

    Route::get('/user', UserWire::class);

    Route::get('/course',   CourseWire::class);

    // Route::get('/coursefile',    CourseFileWire::class);

    // Route::get('/class', ClassRoomWire::class);
    // Route::get('/classcourse',   ClassCourseWire::class);

    // Route::get('/myclass',   MyClassWire::class);
});
////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////

Route::middleware(['auth:sanctum', 'verified'])->get('/dashboard', function () {
    return view('dashboard');
})->name('dashboard');
```

Figure 70: Route "web.php" source code

6) Now open the system and try insert the **url "/course"** after your local domain. Example: **elearning.test/course** This will bring you to the empty course page.

Explanation: The route pointing to the "**CourseWire.php**" class where inside the class the "**function render()**" is run and the function return a view "**course-wire.blade.php**". Inside that view you will find the view code to generate below page.

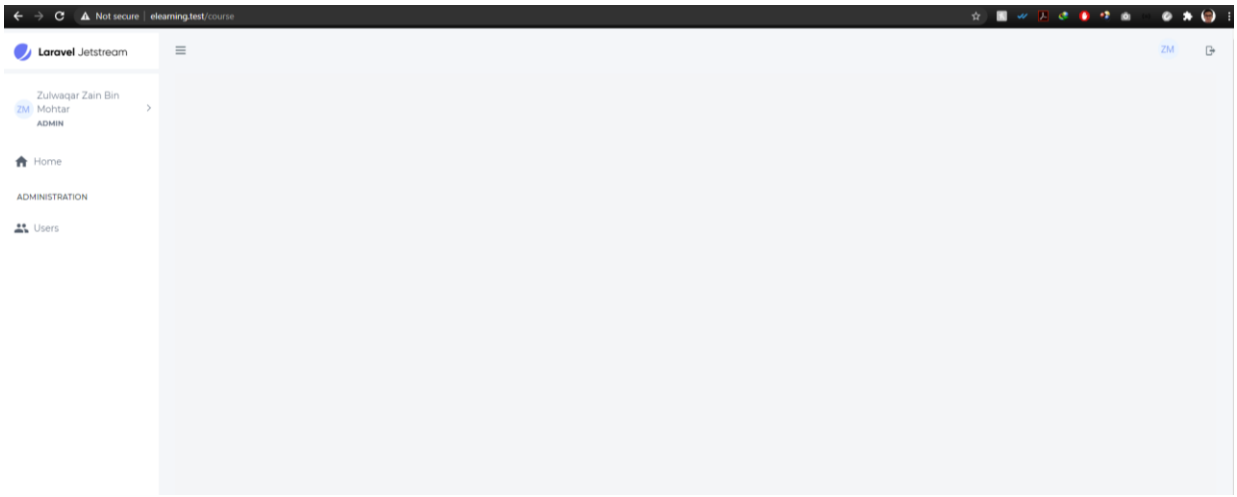


Figure 71: Empty course page

7) The sidebar and topbar of the system using layout file call "**app.blade.php**". This layout will be use by all other pages so we don't need to build the topbar and sidebar everytime we want to create a new page.
8) By using **Ctrl+p** search for "**app.blade.php**" and open it. We need to have Course menu at the sidebar. To have those menu, uncomment the course menu under administration section. This will display the course menu at system sidebar. We use <a> tag as a link to the Create New Course page when user click on the Create New Course menu.

```
{{-------- START SECTION - ADMINISTRATION --------}}
@if (auth()->user()->role == 'admin')
<li class="nav-small-cap">ADMINISTRATION</li>

<li> <a class="waves-effect waves-dark" href="{{URL::to('user')}}" aria-
expanded="false"><i class="mdi mdi-account-multiple"></i><span class="hide-menu">Users </span></a>
</li>

 <li> <a class="has-arrow waves-effect waves-dark" href="#" aria-expanded="false"><i class="mdi mdi-
bullseye"></i><span class="hide-menu">Courses </span></a>
        <ul aria-expanded="false" class="collapse">
                <li><a href="{{URL::to('course')}}">Create New Courses</a></li>
                <li><a href="{{URL::to('coursefile')}}">Add Resources to Course</a></li>
        </ul>
</li>
{{-- <li> <a class="has-arrow waves-effect waves-dark" href="#" aria-
expanded="false"><i class="mdi mdi-view-dashboard"></i><span class="hide-menu">Classes </span></a>
        <ul aria-expanded="false" class="collapse">
                <li><a href="{{URL::to('class')}}">Create New Class</a></li>
                <li><a href="{{URL::to('classcourse')}}">Add Course to Class</a></li>
        </ul>
</li> --}}
@endif
{{-------- END SECTION - ADMINISTRATION --------}}
```

Figure 72: "app.blade.php" section administration source code

9) Now open back the system and you can see course menu has been display at the sidebar but with an empty content.

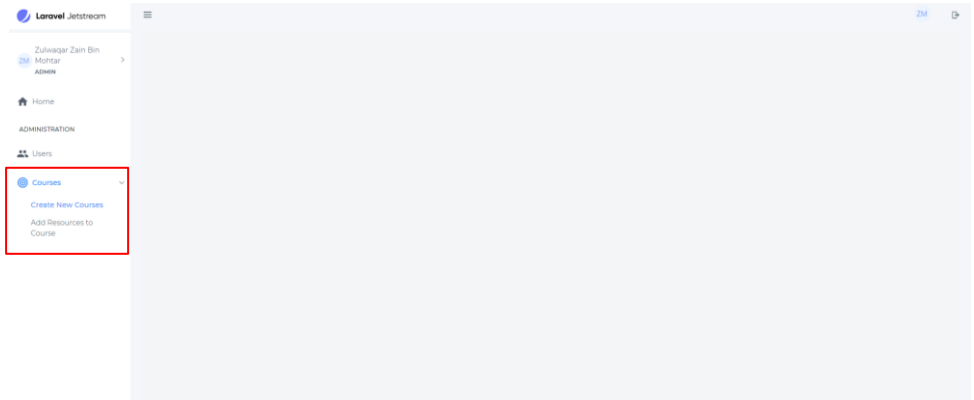


Figure 73: Empty course page with sidebar

Show the Course Data Function

10) Search for "**CourseWire.php**" and follow the code like below:

```php
<?php

namespace App\Http\Livewire\Course;

use Livewire\Component;
use App\Models\Course;

class CourseWire extends Component
{
    // public $course_id;
    // public $name;
    // public $action;

    // protected $listeners = [
    //     'refreshParent' => '$refresh',
    //     'delete'
    // ];

    // public function selectItem($modelid , $action)
    // {
    //     $this->course_id = $modelid;
    //     $this->action = $action;
    //     if($action == "update")
    //     {
    //         $this->emit('getModelId' , $this->course_id);
    //     }
    //     else if($action == "manageCourseResources")
    //     {
    //         $this->emit('getModelIdDeeper' , $this->course_id);
    //         $this->dispatchBrowserEvent('openModal_manageCourseResources');
    //     }
    // }

    public function render()
    {
        // if (auth()->user()->role == 'admin') {
            $courses = Course::all();
        // }
        // else if (auth()->user()->role == 'lecturer')
        // {
```

Import course model so we can use it to fetch course data.

The render method gets called on the initial page load & every subsequent component update.

We use eloquent ORM provide by Laravel to fetch course data from database. Eloquent ORM is a beautiful, simple ActiveRecord implementation for working with your database.

```
        //     $courses = Course::where('id_lecturer',auth()->user()->id)->get();
        // }
        return view('livewire.course.course-wire')->with(compact('courses'));

        // return view('livewire.course.course-wire');
    }
}
```

Course data that have been fetch are return to the view using "with(compact())"

Figure 74: "CourseWire.php" source code

11) Search for "**course-wire.blade.php**" and uncomment the section "**DATATABLE COURSE**" like below. This is a table to show the list of course.

```
<div>

{{-- START SECTION - COURSE FORM  --}}
{{-- @livewire('course.course-form-wire') --}}
{{-- END SECTION - COURSE FORM  --}}

{{-- START SECTION - DATATABLE COURSE  --}}
<div class="row">
    <div class="col-md-12">
        <div class="card" style="box-shadow: 0 .5rem 1rem rgba(0,0,0,.15)!important;border-
radius: 5px;">
            <div class="card-body" style="overflow:scroll">
                <table class="table table-hover" >
                    <thead>
                        <tr>
                            <th>Course Name</th>
                            <th>Lecturer</th>
                            <th>Created At</th>
                            <th>Updated At</th>
                            <th style='width:40px'>Action</th>
                        </tr>
                    </thead>
                    @foreach ($courses as $course)
                        <tbody>
                            <tr>
                                <td>{{$course->name}}</td>
                                <td>{{$course->lecturer ? $course->lecturer->name : 'undefined'}}</td>
                                <td>{{date('j F Y', strtotime($course->created_at))}}</td>
                                <td>{{date('j F Y', strtotime($course->updated_at))}}</td>
                                <td>

                                    <table style="border:none">
                                        <tr>
                                            <td style="border:none">
                                                <button type="button" wire:click="selectItem({{$course
->id}} , 'update' )" class="btn btn-sm waves-effect waves-light btn-warning"><i class="far fa-
edit"></i></button>
                                            </td>
```

Course data send from controller is use to show all course exist in system.

```
                                        <td style="border:none">
                                            <button type="button" wire:click="selectItem({{$course
->id}} , 'delete' )" class="btn btn-sm waves-effect waves-light btn-danger data-delete" data-
form="{{$course->id}}"><i class="fas fa-trash-alt"></i></button>
                                        </td>
                                    </tr>
                                </table>
                            </td>
                        </tr>
                    </tbody>
                @endforeach
            </table>
        </div>
    </div>
</div>
{{-- END SECTION - DATATABLE COURSE  --}}

@push('scripts')

{{-- START SECTION - SCRIPT FOR DELETE BUTTON  --}}
{{-- <script>

  document.addEventListener('livewire:load', function () {
    $(document).on("click", ".data-delete", function (e)
        {
            e.preventDefault();
            swal({
            title: "Are you sure?",
            text: "Once deleted, you will not be able to recover!",
            icon: "warning",
            buttons: true,
            dangerMode: true,
            })
            .then((willDelete) => {
            if (willDelete) {
                e.preventDefault();
                Livewire.emit('delete')
            }
            });
        });
  })
</script> --}}
{{-- END SECTION - SCRIPT FOR DELETE BUTTON  --}}
@endpush
</div>
```

Figure 75: "course-wire.blade" source code

12) Now open the system and you will see table of course with its data.

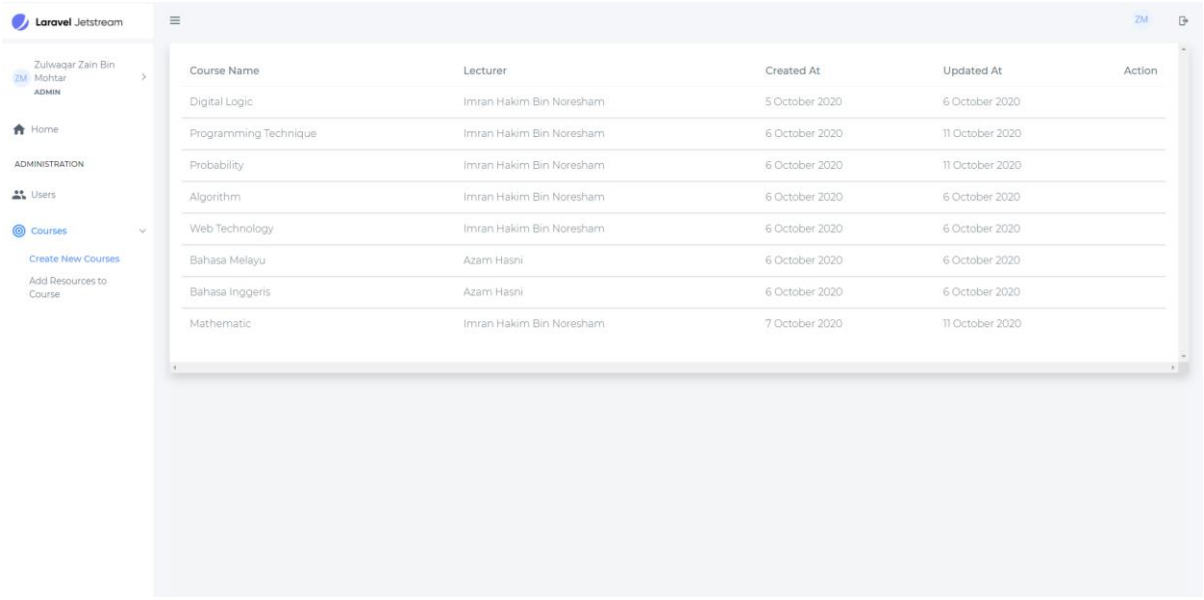


Figure 76: Create New Course page with table of course

Creating the course Function

13) Now, we already have view for table of course. To have a create function, search for "**CourseFormWire.php**" and uncomment the code like below:

```php
<?php

namespace App\Http\Livewire\Course;

use Livewire\Component;
use App\Models\User;
use App\Models\Course;

class CourseFormWire extends Component
{
    public $name;
    public $id_lecturer;
    // public $model_id;

    // protected $listeners = [
    //     'getModelId'
    // ];

    // public function getModelId($model_id)
    // {
    //     $course = Course::find($model_id);

    //     $this->model_id = $course->id;
    //     $this->name = $course->name;
    //     $this->id_lecturer = $course->id_lecturer;

    // }

    public function store()
    {
    //     if($this->model_id)
    //     {
    //         $this->validate([
    //             'name' => 'required|string|max:255',
    //         ]);

    //         $update = Course::find($this->model_id);
    //         $update->name = $this->name;
    //         $update->id_lecturer = $this->id_lecturer;
    //         $update->save();

    //         session()->flash('message', 'Course successfully updated');
    //     }
    //     else
```

Import user and course model so we can use it to fetch the data.

Public properties in Livewire are automatically made available to the view. You need to declare all the properties needed in view(V) here.

```
    //     {
            $this->validate([
                'name' => 'required|string|max:255',
                'id_lecturer' => 'required',
            ]);

            $add = New Course;
            $add->name = $this->name;
            $add->id_lecturer = $this->id_lecturer;
            $add->save();

            session()->flash('message', 'New course
                successfully added');
    //     }

        // $this->emit('refreshParent');

    }

    public function render()
    {
        $lecturers = User::where('role' , 'lecturer')->get();

        return view('livewire.course.course-form-wire')->with(compact('lecturers'));
        // return view('livewire.course.course-form-wire');

    }
}
```

Store function for creating new course. We must first validate the input from view(V) . If validation is fail, error message is return to view.

If validation is successful. We use the eloquent ORM to store course data into database.

Send message to view on successful store course into database.

We use eloquent ORM provide by Laravel to fetch user data with role == lecturer. By using the where method.

Figure 77: "CourseFormWire.php"

14) Now go to "course-form-wire.blade.php" and uncomment the section "COURSE FORM".

```
<div>
    {{-- START SECTION - COURSE FORM  --}}
    <div class="row">
        <div class="col-lg-4">
            <h3 style="color:black">Course Details</h3>
            <p class="text-muted">Create a new course by fill up a
        </div>
        <div class="col-lg-12">

            <form wire:submit.prevent="store">
            <div class="card" style="box-shadow: 0 .5rem 1rem rgba(0,0,0,.15)!important;border-
radius: 5px;">
                <div class="card-body" >
                    <div class="row">
                        <div class="col-md-6">
                            <div class="form-group">
```

Laravel livewire use wire:submit.prevent = "function in controller" to submit form.

```
                                <label class="control-label" style="font-
weight:500">Course Name</label>
                                <input wire:model="name" type="text" id="name" name="name" class="form
-control" >
                                @error('name') <span class="error">{
                            </div>
                        </div>
                        <div class="col-md-6">
                            <div class="form-group">
                                <label class="control-label" style="font-weight:500">Lecturer</label>

                                   @if (auth()->user()->role == 'admin')
                                   <select wire:model="id_lecturer" name="id_lecturer" id="id_lecturer
" class="form-control custom-select" data-placeholder="Choose Class Owner" tabindex="1">
                                        <option value="">-- Choose Class Lecturer --</option>
                                        @foreach ($lecturers as $lecturer)
                                            <option value="{{$lecturer->id}}">{{$lecturer-
>name}}</option>
                                        @endforeach
                                    </select>
                                   @else
                                   <select wire:model="id_lecturer" name="id_lecturer" id="id_lecturer
" class="form-control custom-select" data-placeholder="Choose Class Owner" tabindex="1">
                                    <option value="">-- Choose Class Lecturer --</option>
                                    <option value="{{auth()->user()->id}}">{{auth()->user()-
>name}}</option>
                                   </select>
                                   @endif

                                @error('id_lecturer') <span class="error">{{ $message }}</span> @ender
ror
                            </div>
                        </div>
                    </div>
                </div>

                <div class="card-footer" style="background-color: #f9fafb !important;border-
top: none;">
                    <div class="row">
                        <div class="col-md-12 text-right">
                            @if (session()->has('message'))
                                {{ session('message') }}

                            @endif
                            <button type="submit" class="inline-flex items-center px-4 py-2 bg-gray-
800 border border-transparent rounded-md font-semibold text-xs text-white uppercase tracking-
widest hover:bg-gray-700 active:bg-gray-900 focus:outline-none focus:border-gray-900 focus:shadow-
outline-gray disabled:opacity-25 transition ease-in-out duration-150">
```

Laravel livewire use wire:model to synchronize value of form input and public properties in the controller.

If validation is fail at controller. It will return the error message to view and the message will show up here.

```
                                Save
                            </button>
                        </div>
                    </div>
                </div>
            </div>
        </form>

        </div>
    </div>
    {{-- END SECTION - COURSE FORM  --}}
</div>
```

Figure 78: "course-form-wire.blade.php" source code

15) To include "**course-form-wire.blade**" component inside "**course-wire.blade**". Go to "**course-wire.blade**" and uncomment the section "**COURSE FORM**".

```
{{-- START SECTION - COURSE FORM  --}}
@livewire('course.course-form-wire')
{{-- END SECTION - COURSE FORM  --}}
```

Figure 79: "course-wire.blade" Course Form section

16) Now go back to system and you will see the create form for course , you can create new course after fill in all the required field and click the save button. Go on and try. On successful , you will got a message notified that course successfully created but if you notice, the course table does not update and show the latest course that you just create in real-time, this mean you need to refresh the page to see the change.

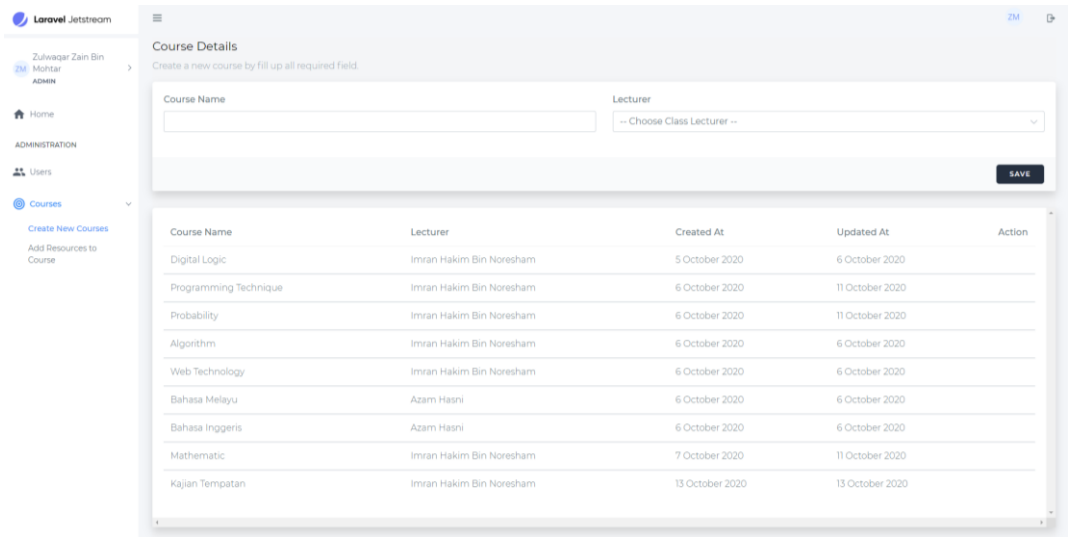


Figure 80: Create New Course page with table of course & create new course component

17) To have a real-time update on the course table go back to "**CourseWire.php**" and uncomment the code other than `// 'delete'` . This is the listener to listen on succesful store event of course.

```
protected $listeners = [
        'refreshParent' => '$refresh',
        // 'delete'
    ];
```

Figure 81: "CourseWire.php" listener

18) After that go to "**CourseFormWire.php**" and uncomment below code. This code is use to triggered the listener.

```
    $this->emit('refreshParent');
```

Figure 82: "CourseFormWire.php" emitting to listener

19) We use listener on store function in "**CourseFormWire.php**" to detect if course have been successfully created. First, we **define the listener** in "**CourseWire.php**" just like in the code above. Then we call the listener by using **emit('refreshParent')**. When the listener is triggered, the **magic keyword** provide by Laravel livewire which is **"$refresh"** will refresh the "**CourseWire.php**" component which is the course table so we will have the latest data that we just created. Go on and try to insert back new course.

Edit the Course Function

20) If you noticed we already uncomment some code inside "**course-wire.blade**". The code will fire when user click on the button. The **wire:click** event will run the **selectItem** function inside "**CourseWire.php**" to set all the public properties to a specific course that we have choose.

```
<td style="border:none">
    <button type="button" wire:click="selectItem({{$course->id}} , 'update' )" class="btn btn-
sm waves-effect waves-light btn-warning"><i class="far fa-edit"></i></button>
</td>
```

21) Go to "**CourseWire.php**" and uncomment below code. The "**selectItem**" function will run on "**wire:click**" that we define at "**course-wire.blade.php**". We write the logic '**if($action == "update")**' if the parameter "**$action**" pass an "**update**" value, thus it will emit listener name "**getModelId**" with parameter of **course id**.

```
public function selectItem($modelid , $action)
    {
        $this->course_id = $modelid;
        $this->action = $action;

        if($action == "update")
        {
            $this->emit('getModelId' , $this->course_id);
        }
        else if($action == "manageCourseResources")
        {
            $this->emit('getModelIdDeeper' , $this->course_id);
            $this->dispatchBrowserEvent('openModal_manageCourseResources');
        }
    }
```

Figure 83: "CourseWire.php" selectItem function

22) After that, go to "**CourseFormWire.php**" and uncomment below code. This code is to listen to an event call "**getModelID**" from the "**CourseWire.php**". The listener will run the function **getModelId** and set all the data of course. This data will synchronise automatically with the view where we define **wire:model** at "**course-wire-form.blade.php**"..

```
<?php
namespace App\Http\Livewire\Course;
use Livewire\Component;
use App\Models\User;
use App\Models\Course;
class CourseFormWire extends Component
{
    public $name;
    public $id_lecturer;
    public $model_id;
```

```php
protected $listeners = [
    'getModelId'
];
public function getModelId($model_id)
{
    $course = Course::find($model_id);
    $this->model_id = $course->id;
    $this->name = $course->name;
    $this->id_lecturer = $course->id_lecturer;


}
```

Figure 84: "CourseFormWire.php"

23) Now go on to system and click at the edit button and you will see the data that you want to edit will automatically fill in the course form component.

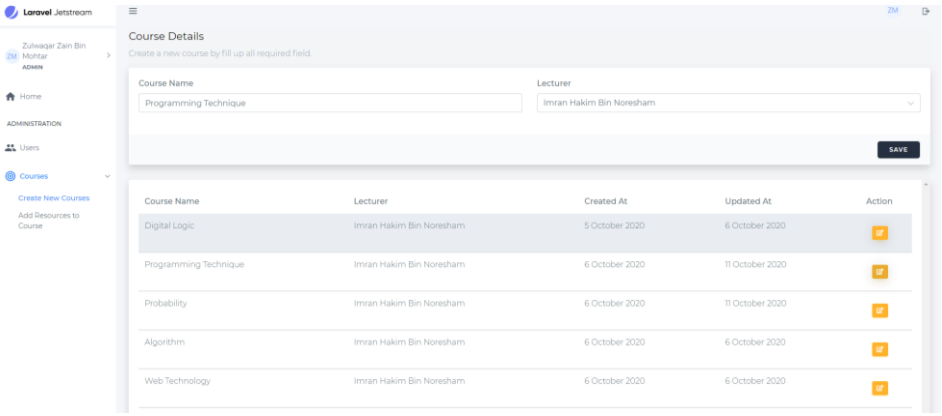


Figure 85: Create New Course page with applied real-time update

24) But If you click on the save button, the data will create a new course. To enable the edit function lets do some logic on the store function. Go back to "**CourseFormWire.php**" and uncomment all the code located in the store function.

```php
public function store()
    {
        if($this->model_id)
{
            $this->validate([
                'name' => 'required|string|max:255',
            ]);

            $update = Course::find($this->model_id);
            $update->name = $this->name;
            $update->id_lecturer = $this->id_lecturer;
            $update->save();

            session()->flash('message', 'Course successfully updated');
```

If this function detect "model_id" has a value, then it will do an update to the existed course data.

```
        }
        else
        {
            $this->validate([
                'name' => 'required|string|max:255',
                'id_lecturer' => 'required',
            ]);

            $add = New Course;
            $add->name = $this->name;
            $add->id_lecturer = $this->id_lecturer;
            $add->save();

            session()->flash('message', 'New course successfully added');
        }

        $this->emit('refreshParent');
    }
```

Figure 86: "CourseFormWire.php" source code

25) Lets jump back to the system and edit some course. The edit function will be successful.

Delete the course Function

26) If you noticed we already uncomment some code inside "**course-wire.blade**". The code will fire when user click on the button. The **wire:click** event will run the **selectItem** function inside "**CourseWire.php**" to set all the public properties to a specific user that we have choose.

```
<td style="border:none">
    <button type="button" wire:click="selectItem({{$course->id}} , 'delete' )" class="btn btn-
sm waves-effect waves-light btn-danger data-delete" data-form="{{$course->id}}"><i class="fas fa-
trash-alt"></i></button>
</td>
```

27) Beside using wire:click event, to include delete function, go to "**course-wire.blade.php**" and uncomment on section "**Script for Delete Button**" like below. This is the script to popup a message of confirmation on deletion of course.

```
{{-- START SECTION - SCRIPT FOR DELETE BUTTON  --}}
<script>

  document.addEventListener('livewire:load', function () {


    $(document).on("click", ".data-delete", function (e)
        {
            e.preventDefault();
            swal({
            title: "Are you sure?",
            text: "Once deleted, you will not be able to recover!",
            icon: "warning",
            buttons: true,
            dangerMode: true,
            })
            .then((willDelete) => {
            if (willDelete) {
                e.preventDefault();
                Livewire.emit('delete')
            }
            });
        });
  })
</script>
{{-- END SECTION - SCRIPT FOR DELETE BUTTON  --}}
```

This script will run on user click on tag that has a "data-delete" class. If you see at step 26, we define a "data-delete" class at the delete button.

This emit function of delete will sent a message to the listener located at "CourseWire.php" and call the function delete.

Figure 87: "course-wire.blade.php" Script for Delete Button section

28) After that navigate to "**CourseWire.php**" and uncomment below code inside the file.

```php
public $course_id;
public $name;
public $action;

protected $listeners = [
    'refreshParent' => '$refresh',
    'delete'
];
public function delete()
{
    $course = Course::find($this->course_id);
    $course->delete();

}
```

Because we already set the public properties on wire:click event at step 27. We can use the value to search for the course and delete it.

Figure 88: "CourseWire.php"

29) Now lets navigate back to system and try the delete function by clicking the delete button as below.

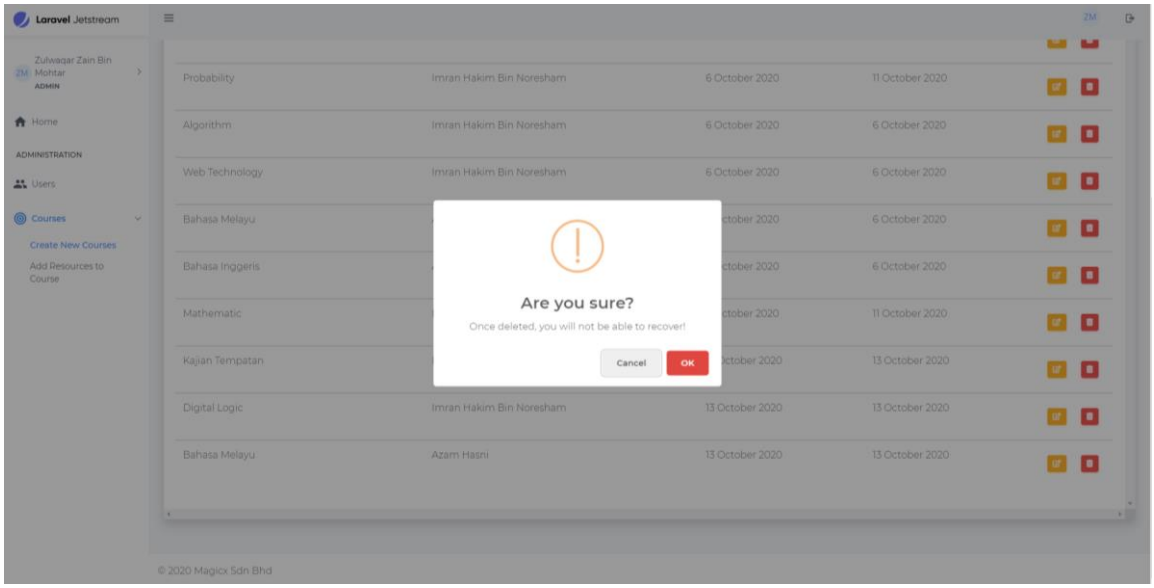


Figure 89: Course Page with delete function

Submodule – Add Resources to Course

Submodule Outcome – At the end of this module, you can develop the Submodule Add Resources to Course for administration use. This module will include below function:

i) Show the resources data
ii) Add the resources
iii) Delete the resources

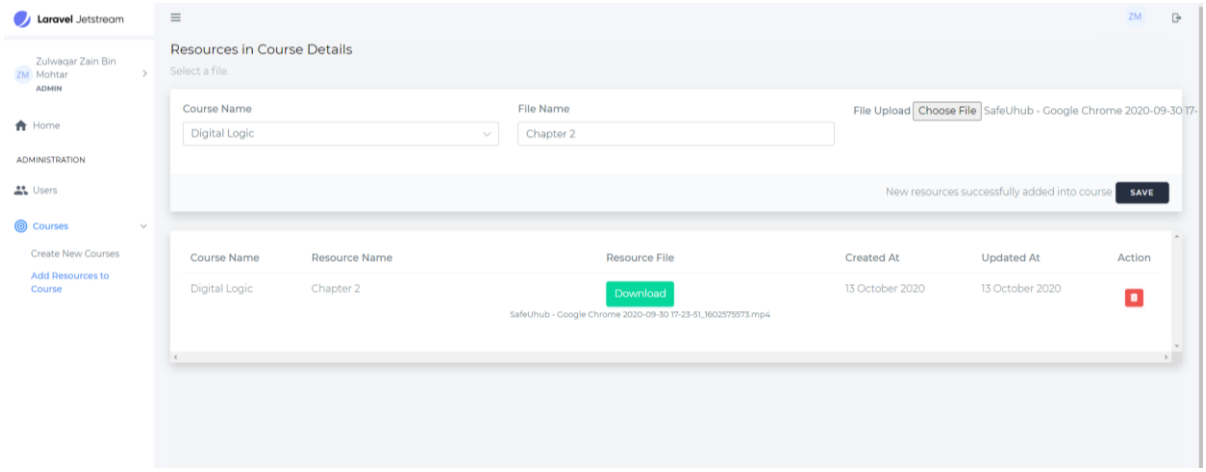


Figure 90: Course File management page for admin

1) To have like above page. We must have the **Model-View-Controller (MVC)** component.

2) **For Model**, open the "**elearning-source**" folder and navigate to **Model** folder. Copy the **"CourseFile.php"** file and paste it to "**www\elearning\app\Models\"** in your project folder. This will be the model for course file.
Explanation: We already provide the Model file in the "**elearning-source**" folder to make things easier for you to develop. The original way of generating Course File model is through Laravel artisan command in terminal which is "**php artisan make:model CourseFile**". This will create model file inside "**app\Models\CourseFile.php**".

3) **For Controller**, open the "**elearning-source**" folder and navigate to **Controller** folder. Copy the **"CourseFile"** folder and paste it to "**laragon\www\elearning\app\Http\Livewire**" (project folder)**.** This will be the logical code for this submodule.

4) **For View,** open back "**elearning-source**" folder and navigate to **View** folder. Copy the **"course-file"** folder and paste it to "**laragon\www\elearning\resources\views\livewire**" (project folder). This will be the view for this submodule.

   Explanation: We also provide the Controller and View file in the "**elearning-source**" folder to make things easier for you to develop. The original way of generating the component is through artisan console command in terminal. The artisan command to generate the component is '**php artisan make:livewire CourseFile\\CourseFileWire**'. This will create the Controller file inside "**app\Http\Livewire\CourseFile\ CourseFileWire.php**" and View file inside "**resources\views\livewire\course-file\course-file-wire.php**".

Route

5) Route defines the URL pattern that you want to generate. Route will point to a controller. From the controller you can develop process logic or return to a view (V) or redirect to other url. Now open back your project inside the VS Code and lets take a look at the route located at web.php. For quick search, use the hotkey "**Ctrl+P**" and key in "**web.php**".

Uncomment below code. This is route we use to navigate to Add Resource to Course page. The **url "/coursefile"** will point to **"CourseFileWire.php"** class where the "**function render()**" inside "**CourseFileWire.php**" class will be run to return the user to the view (V).

```
Route::get('/coursefile',    CourseFileWire::class);
```

Figure 91: Insert route to CourseFileWire class

And you need to import the class so the route can access to the right class that you have assigned.

```
use App\Http\Livewire\CourseFile\CourseFileWire;
```

Figure 92: Import class of CourseFileWire

The "**web.php**" should look like below after you uncomment the above code.

```
<?php

use Illuminate\Support\Facades\Route;

use App\Http\Livewire\User\UserWire;

use App\Http\Livewire\Course\CourseWire;
use App\Http\Livewire\CourseFile\CourseFileWire;

// use App\Http\Livewire\ClassRoom\ClassRoomWire;
// use App\Http\Livewire\ClassCourse\ClassCourseWire;

// use App\Http\Livewire\MyClassWire;
/*
|--------------------------------------------------------------------------
| Web Routes
|--------------------------------------------------------------------------
|
| Here is where you can register web routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| contains the "web" middleware group. Now create something great!
|
*/

Route::get('/', function () {
    return redirect('login');
});
////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////
//Laravel Livewire
```

```
//////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////
Route::middleware('auth')->group(function () {

    Route::get('/user', UserWire::class);

    Route::get('/course',    CourseWire::class);

    Route::get('/coursefile',    CourseFileWire::class);

    // Route::get('/class', ClassRoomWire::class);
    // Route::get('/classcourse',    ClassCourseWire::class);

    // Route::get('/myclass',    MyClassWire::class);
});
//////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////

Route::middleware(['auth:sanctum', 'verified'])->get('/dashboard', function () {
    return view('dashboard');
})->name('dashboard');
```

Figure 93: Route "web.php" source code

6) Now open the system and try insert the **url "/coursefile"** after your local domain. Example: **elearning.test/coursefile.** This will bring you to the empty add resources to course page.

Explanation: The route pointing to the "**CourseFileWire.php**" class where inside the class the "**function render()**" is run and the function return a view "**course-file-wire.blade.php**". Inside that view you will find the view code to generate below page.

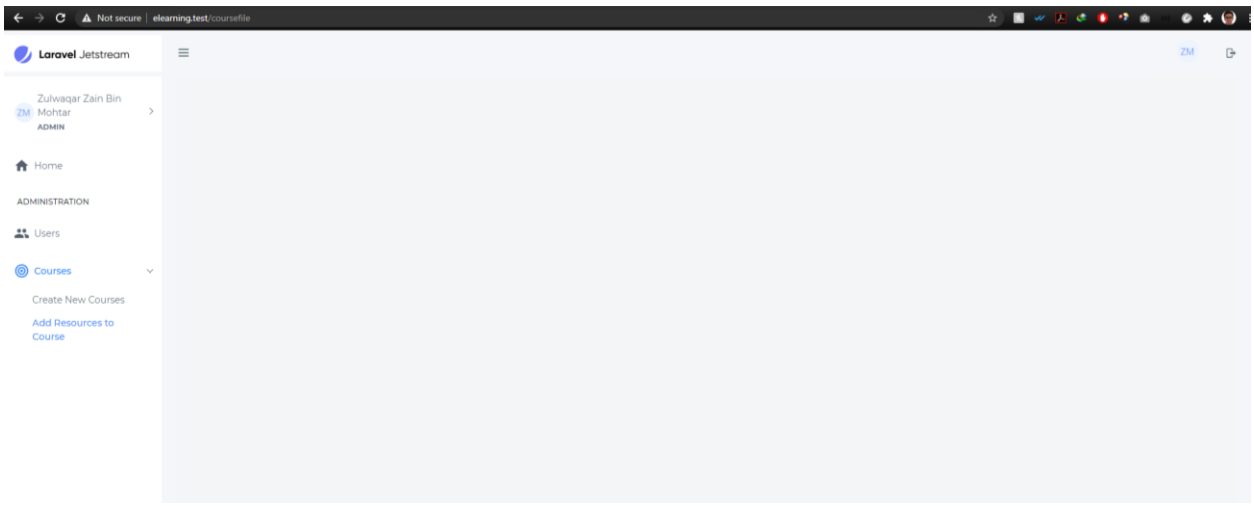


Figure 94: Empty add resources to course page

Show the resources data Function

7) Search for "**CourseFileWire.php**" and follow the code like below:

```php
<?php

namespace App\Http\Livewire\CourseFile;

use Livewire\Component;
use App\Models\CourseFile;

class CourseFileWire extends Component
{
    // public $id_coursefile;
    // public $action;

    // protected $listeners = [
    //     'refreshParent' => '$refresh',
    //     'delete'
    // ];

    // public function selectItem($modelid , $action)
    // {
    //     $this->id_coursefile = $modelid;
    //     $this->action = $action;
    // }

    public function render()
    {
        // if (auth()->user()->role == "admin") {

            $coursefiles = CourseFile::all();
        // } else if(auth()->user()->role == "lecturer"){
            // $coursefiles = CourseFile::whereHas('course' , function ($query) {
                // $query->where('id_lecturer' , auth()->user()->id);
            // })->get();
        // }

        return view('livewire.course-file.course-file-wire')->with(compact('coursefiles'));

        // return view('livewire.course-file.course-file-wire');
    }
}
```

Import course file model so we can use it to fetch course file

The render method gets called on the initial page load & every subsequent component update.

We use eloquent ORM provide by Laravel to fetch course file data from database. Eloquent ORM is a beautiful, simple ActiveRecord implementation for working with your database.

Course file data that have been fetch are return to the view using "with(compact())"

Figure 95: "CourseFileWire.php" source code

8) Search for "**course-file-wire.blade.php**" and uncomment the section "**DATATABLE COURSE FILE**" like below. This is a table to show the list of course file.

```
{{-- START SECTION - DATATABLE COURSE FILE  --}}
    <div class="row">
        <div class="col-md-12">
            <div class="card" style="box-shadow: 0 .5rem 1rem rgba(0,0,0,.15)!important;border-
radius: 5px;">
                <div class="card-body" style="overflow:scroll">
                    <table class="table table-hover" >
                        <thead>
                            <tr>
                                <th>Course Name</th>
                                <th>Resource Name</th>
                                <th class="text-center">Resource File</th>
                                <th>Created At</th>
                                <th>Updated At</th>
                                <th style='width:40px'>Action</th>
                            </tr>
                        </thead>

                            @foreach ($coursefiles as $coursefile)
                            <tbody>
                                <tr>
                                    <td>{{$coursefile->course ? $coursefile->course->name : ''}}</td>
                                    <td>{{$coursefile->name}}</td>
                                    <td class="text-center">
                                        <a href="{{URL::to(''.$coursefile-
>file_path.'')}}" target="_blank" class="btn btn-success">Download</a>
                                        <br>
                                        <small>{{$coursefile->file_name}}</small>
                                    </td>
                                    <td>{{date('j F Y', strtotime($coursefile->created_at))}}</td>
                                    <td>{{date('j F Y', strtotime($coursefile->updated_at))}}</td>
                                    <td>
                                        <table style="border:none">
                                            <tr>
                                                <td style="border:none">
                                                    <button type="button" wire:click="selectItem({{$co
ursefile->id}} , 'delete' )" class="btn btn-sm waves-effect waves-light btn-danger data-delete" data-
form="{{$coursefile->id}}"><i class="fas fa-trash-alt"></i></button>
                                                </td>
                                            </tr>
                                        </table>
                                    </td>
                                </tr>
                            </tbody>
                            @endforeach
```

Course file data send from controller is use to show all course file exist in system.

```
                </table>
            </div>
        </div>
    </div>
</div>
  {{-- END SECTION - DATATABLE COURSE FILE  --}}
```

Figure 96: "course-file-wire.blade" source code

9) Now open the system and you will see table of course file with its data.

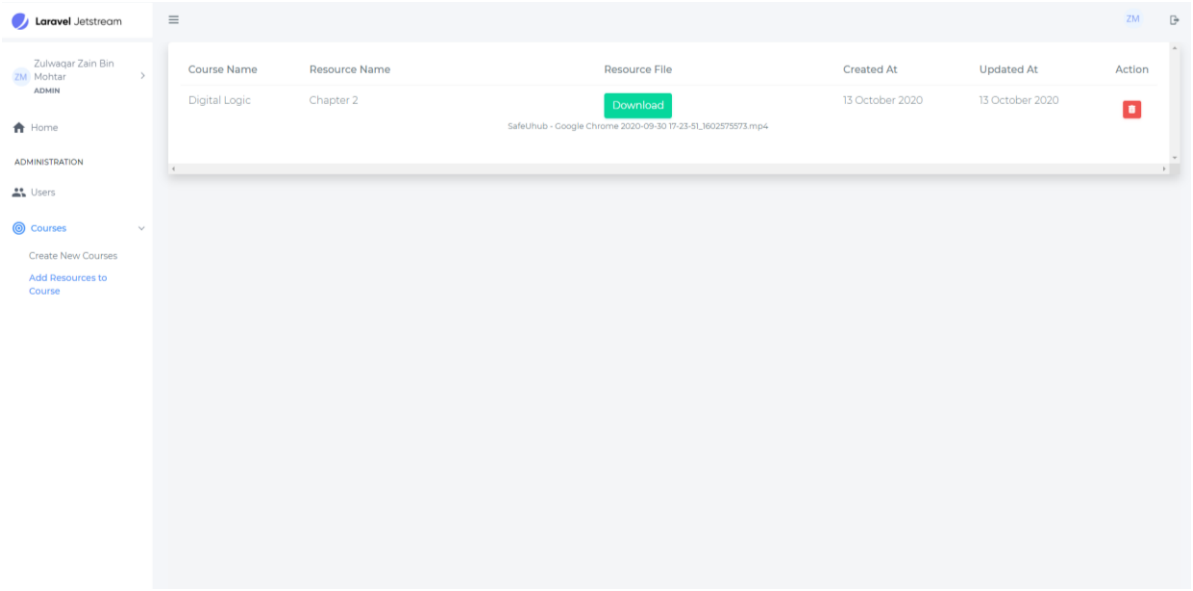


Figure 97: Add Resource to Course page with table of course file

Add the Resources Function

10) Now, we already have view for table of course file. To have a create function, search for "**CourseFileFormWire.php**" and follow the code like below:

```php
<?php

namespace App\Http\Livewire\CourseFile;

use Livewire\Component;
use App\Models\Course;
use App\Models\CourseFile;
use Livewire\WithFileUploads;

class CourseFileFormWire extends Component
{
    use WithFileUploads;

    public $id_coursefile;
    public $name;
    public $file_path;
    public $file_name;
    public $id_course;
    public $file_type;



    public function store()
    {
            $this->validate([
                'file_name' => 'required',
            ]);

            $add = New CourseFile;
            $add->name = $this->file_name;
            $add->id_course = $this->id_course;

            // Handle File Upload
            if ($this->file_path) {
                // Get filename with the extension
                $filenameWithExt = $this->file_path->getClientOriginalName();
                // Get just filename
                $filename = pathinfo($filenameWithExt, PATHINFO_FILENAME);
             // Get just ext
                if($this->file_type == "3dfile") {
                    $extension = "glb";
                } else {
                    $extension = $this->file_path->getClientOriginalExtension();
                }
```

Import course and course file model so we can use it to fetch the data.

Add livewire package "WithFileUploads" if you have file to upload to the server.

To make file upload available when using livewire, add the "use WithFileUploads".

Public properties in Livewire are automatically made available to the view. You need to declare all the properties needed in view(V) here.

Handle the file upload from view. The function will check if the file is exist then below code will be executed where it will save the file into the server.

```
                // Filename to store
                $fileNameToStore = $filename . '_' . time() . '.' . $extension;
                // Upload Image
                $this->file_path-
>storeAs('public' . DIRECTORY_SEPARATOR . 'coursefile', $fileNameToStore);
            }

else {
                $fileNameToStore = 'nofile_' . $this->id_course . '_' . time() . '.png';

                $img_path = public_path() . '' . DIRECTORY_SEPARATOR . 'storage' . DIRECTORY_SEPARATOR
 . 'coursefile' . DIRECTORY_SEPARATOR . 'nofile_' . $this->id_course->id . '_' . time() . '.png';

                copy(public_path() . '' . DIRECTORY_SEPARATOR . 'img' . DIRECTORY_SEPARATOR . 'noimage
.png', $img_path);
            }

            //path
            $path = '' . DIRECTORY_SEPARATOR . 'storage' . DIRECTORY_SEPARATOR . 'coursefile' . DIRECT
ORY_SEPARATOR . '' . $fileNameToStore;
            $add->file_type = $this->file_type;
            $add->file_name = $fileNameToStore;
            $add->file_path = $path;
            $add->save();

            session()->flash('message', 'New resources successfully added into course');


        // $this->emit('refreshParent');

    }


    public function render()
    {
        // if (auth()->user()->role == "admin") {
            $courses = Course::all();
        // } else if(auth()->user()->role == "lecturer") {
            // $courses = Course::where('id_lecturer',auth()->user()->id)->get();
        // }
        return view('livewire.course-file.course-file-form-wire')->with(compact('courses'));
        // return view('livewire.course-file.course-file-form-wire');
    }
}
```

Send message to view on successful store course file into database.

Figure 98: "CourseFileFormWire.php"

11) Now go to "**course-file-form-wire.blade.php**" and uncomment the section "**COURSE FILE FORM**".

```
<div>
    {{-- START SECTION - COURSE FILE FORM  --}}
    <div class="row">
        <div class="col-lg-4">
            <h3 style="color:black">Resources in Course Details</h3>

            <p class="text-muted">Select a file.</p>
        </div>
        <div class="col-lg-12">

            <form wire:submit.prevent="store" enctype="multipart/form-data">

            <div class="card" style="box-shadow: 0 .5rem 1rem rgba(0,0,0,.15)!important;border-
radius: 5px;">
                <div class="card-body" >
                    <div class="row">
                        <div class="col-md-4">
                            <div class="form-group">
                                <label class="control-label" style="fo
weight:500">Course Name</label>
                                <select wire:model="id_course" name="id_course" id="id_course" class="
form-control custom-select" data-placeholder="Choose Class Course" tabindex="1">
                                    <option value="">-- Choose Course --</option>
                                    @foreach ($courses as $course)
                                        <option value="{{$course->id}}">{{$course->name}}</option>
                                    @endforeach
                                </select>
                                @error('id_course') <span class="error">{{ $message }}</span> @enderro
r
                            </div>
                        </div>
                        <div class="col-md-4">
                            <div class="form-group">
                                <label class="control-label" style="font-weight:500">File Name</label>
                                <input wire:model="file_name" type="text" id="file_name" name="file_na
me" class="form-control" >
                                @error('file_name') <span class="error">{{ $message }}</span> @enderro
r
                            </div>
                        </div>
                        <div class="col-md-4">
                            <div class="demo-radio-button">
                                <input name="group1" type="radio" id="radio_1" wire:model="file_type"
value="normalfile" />
                                <label for="radio_1">Normal File</label>
```

Laravel livewire use wire:submit.prevent = "function in controller" to submit form.

Laravel livewire use wire:model to synchronize value of form input and public properties in the controller.

If validation is fail at controller. It will return the error message to view and the message will show up here.

```
                                <input name="group1" type="radio" id="radio_2" wire:model="file_type"
value="3dfile"/>
                                <label for="radio_2">3D File</label>

                                <input name="group1" type="radio" id="radio_3" wire:model="file_type"
value="360file"/>
                                <label for="radio_3">360° Video</label>
                            </div>
                        </div>
                    </div>
                    <div class="row">
                        <div class="col-md-4">

                        </div>
                        <div class="col-md-4">

                        </div>
                        <div class="col-md-4" id="fileupload">
                            <div class="form-group">
                                <label class="control-label" style="font-
weight:500">File Upload</label>
                                <div
                                    x-data="{ isUploading: false, progress: 0 }"
                                    x-on:livewire-upload-start="isUploading = true"
                                    x-on:livewire-upload-finish="isUploading = false"
                                    x-on:livewire-upload-error="isUploading = false"
                                    x-on:livewire-upload-progress="progress = $event.detail.progress">
                                <div wire:loading wire:target="file_path"><i class="mdi mdi-
loading mdi-spin mdi-24px"></i></div>
                                    <input type="file" wire:model="file_path" id="file_path" name="fil
e_path" class="dropify" />
                                    @error('file_path') <span class="error">{{ $message }}</span> @end
error
                                    <div x-show="isUploading">
                                        <progress max="100" x-bind:value="progress"></progress>
                                    </div>
                                </div>
                            </div>
                        </div>
                    </div>
                </div>
                <div class="card-footer" style="background-color: #f9fafb !important;border-
top: none;">
                    <div class="row">
                        <div class="col-md-12 text-right">
                            @if (session()->has('message'))
                                {{ session('message') }}
                            @endif
```

```
                            <button type="submit" class="inline-flex items-center px-4 py-2 bg-gray-
800 border border-transparent rounded-md font-semibold text-xs text-white uppercase tracking-
widest hover:bg-gray-700 active:bg-gray-900 focus:outline-none focus:border-gray-900 focus:shadow-
outline-gray disabled:opacity-25 transition ease-in-out duration-150">
                                Save
                            </button>
                        </div>
                    </div>
                </div>
            </div>

        </form>

        </div>
    </div>
    {{-- END SECTION - COURSE FILE FORM  --}}
</div>
```

Figure 99: "course-file-form-wire.blade.php" source code

12) To include "**course-file-form-wire.blade**" component inside "**course-file-wire.blade**". Go to "**course-file-wire.blade**" and uncomment the section "**COURSE FILE FORM**".

```
    {{-- START SECTION - COURSE FILE FORM  --}}
    @livewire('course-file.course-file-form-wire')
    {{-- END SECTION - COURSE FILE FORM  --}}
```

Figure 100: "course-file-wire.blade" Course File Form section

13) Now go back to system and you will see the create form for add resource to course, you can add resource to course after fill in all the required field and click the save button. Go on and try. On successful , you will got a message notified that resource successfully added to the course but if you notice, the resource table does not update and show the latest resource that you just create in real-time, this mean you need to refresh the page to see the change.

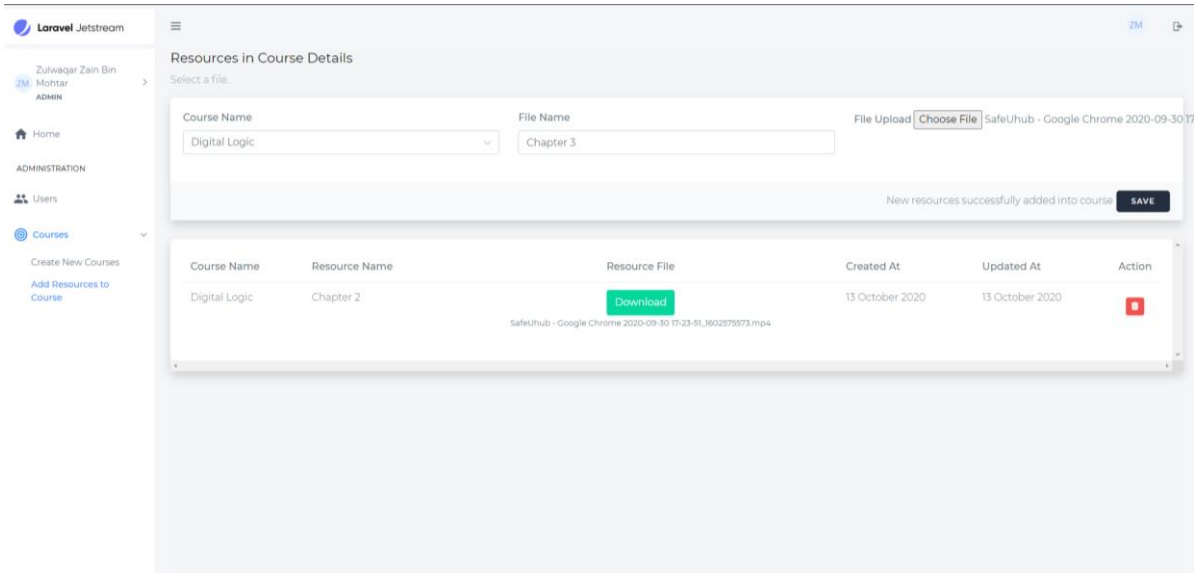


Figure 101: Add Resource to Course page with table of resource & add resource to course component

14) To have a real-time update on the resource table go back to "**CourseFileWire.php**" and uncomment the code other than `// 'delete'` . This is the listener to listen on succesful store event of course.

```
protected $listeners = [
        'refreshParent' => '$refresh',
        // 'delete'
    ];
```

Figure 102: "CourseFileWire.php" listener

15) After that go to "**CourseFileFormWire.php**" and uncomment below code. This code is use to triggered the listener.

```
    $this->emit('refreshParent');
```

Figure 103: "CourseFileFormWire.php" emitting to listener

16) We use listener on store function in "**CourseFileFormWire.php**" to detect if resource have been successfully added into course. First, we define the listener in "**CourseFileWire.php**" just like in the code above. Then we call the listener by using **emit('refreshParent').** When the listener is triggered, the **magic keyword** provide by Laravel livewire which is "**$refresh**" will refresh the "**CourseFileWire.php**" component which is the course table so we will have the latest data that we just created. Go on and try to add back new resource to course.

Delete the Resources Function

17) If you noticed we already uncomment some code inside "**course-file-wire.blade**". The code will fire when user click on the button. The **wire:click** event will run the **selectItem** function inside "**CourseFileWire.php**" to set all the public properties to a specific user that we have choose.

```
<td style="border:none">
          <button type="button" wire:click="selectItem({{$coursefile-
>id}} , 'delete' )" class="btn btn-sm waves-effect waves-light btn-danger data-delete" data-
form="{{$coursefile->id}}"><i class="fas fa-trash-alt"></i></button>
</td>
```

18) Beside using wire:click event, to include delete function, go to "**course-file-wire.blade.php**" and uncomment on section "**Script for Delete Button**" like below. This is the script to popup a message of confirmation on deletion of course.

```
{{-- START SECTION - SCRIPT FOR DELETE BUTTON  --}}
<script>

  document.addEventListener('livewire:load', function () {


    $(document).on("click", ".data-delete", function (e)
        {
            e.preventDefault();
            swal({
            title: "Are you sure?",
            text: "Once deleted, you will not be able to recover!",
            icon: "warning",
            buttons: true,
            dangerMode: true,
            })
            .then((willDelete) => {
            if (willDelete) {
                e.preventDefault();
                Livewire.emit('delete')
            }
            });
        });

  })
</script>
{{-- END SECTION - SCRIPT FOR DELETE BUTTON  --}}
```

This script will run on user click on tag that has a "data-delete" class. If you see at step 17, we define a "data-delete" class at the delete button.

This emit function of delete will sent a message to the listener located at "CourseFileWire.php" and call the function delete.

Figure 104: "user-file-wire.blade.php" Script for Delete Button section

19) After that navigate to "**CourseFileWire.php**" and uncomment below code inside the file.

```php
public $id_coursefile;
public $action;

protected $listeners = [
        'refreshParent' => '$refresh',
        'delete'
    ];
 public function selectItem($modelid , $action)
    {
        $this->id_coursefile = $modelid;
        $this->action = $action;
    }

    public function delete()
    {
        $coursefile = CourseFile::find($this->id_coursefile);
        $coursefile->delete();
    }
```

When user click at the delete button. This function will run and set the public properties.

Once user confirm the deletion. An emit event is send from the view and the listener 'delete' will trigger and run the function delete.

Figure 105: "CourseFileWire.php"

20) Now lets navigate back to system and try the delete function by clicking the delete button as below.

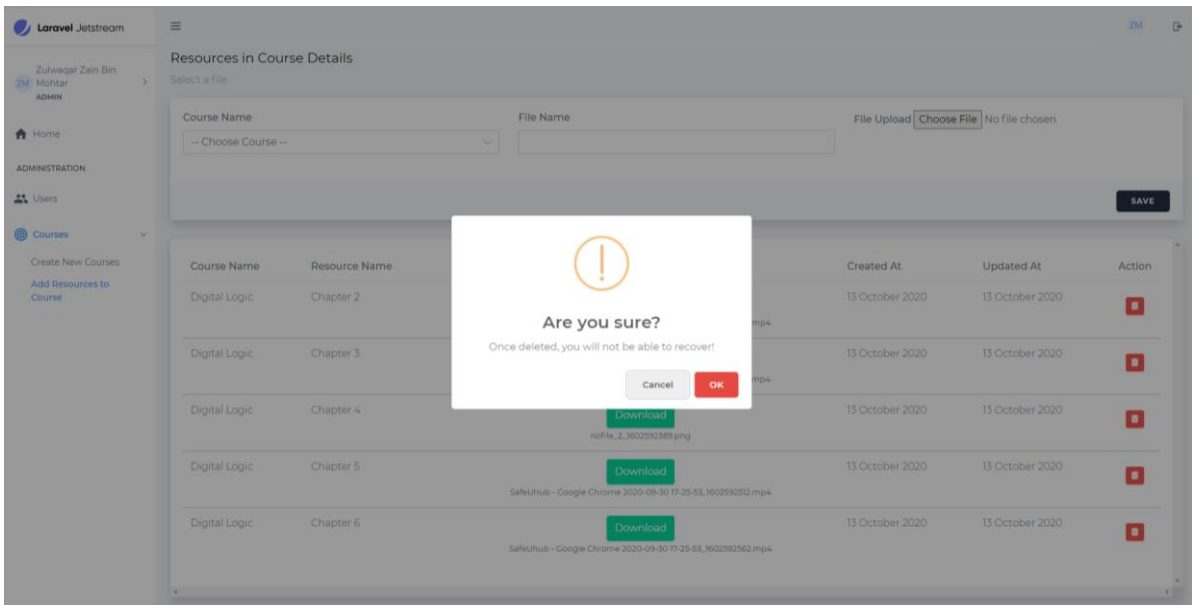


Figure 106: Add Resource to Course Page with delete function

Class Module

Submodule – Create New Class Room

Submodule Outcome – At the end of this module, you can develop the Submodule Create New Class for administration use. This module will include below function:

i) Show Class room data
ii) Create new Class room
iii) Edit the class room
iv) Delete the class room

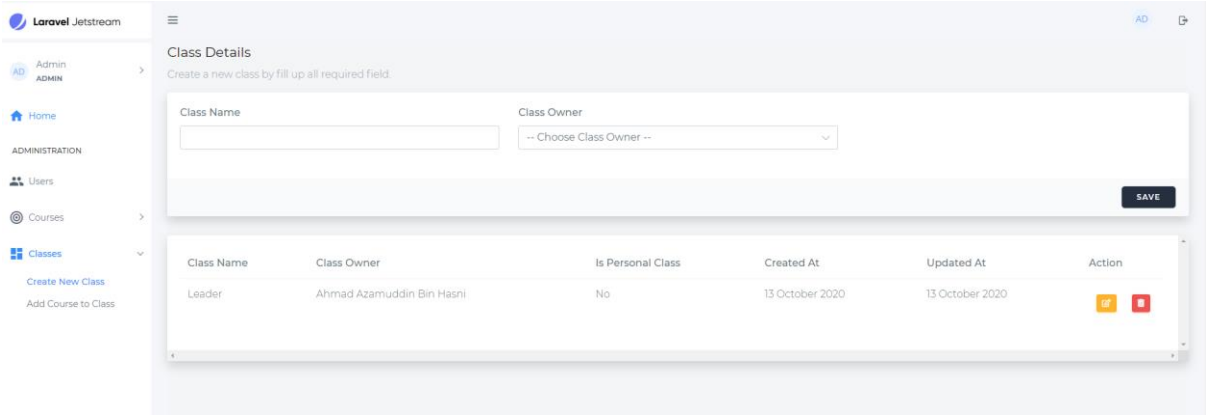


Figure 107: Class management page

1) To have like above page. We must have the **Model-View-Controller (MVC)** component.

2) **For Model**, because we want to build Class page, so we need the Class Model. But with Laravel Jetstream, they already come with a Team model where the function is just like Class that we want to develop. No need for us to create the Class model. We will use the team model as the class model. You can find the model at "**app\Models**".

3) **For Controller**, Open the "**elearning-source**" folder and navigate to **Controller** folder. Copy the **"ClassRoom"** folder and paste it to "**www\elearning\app\Http\Livewire**" at "**laragon\www\elearning**" project folder. This will be the logical code for this submodule.

4) **For View,** open back "**elearning-source**" folder and navigate to **View** folder. Copy the **"class-room"** folder and paste it to "**laragon\www\elearning\resources\views\livewire**" (project folder). This will be the view for this submodule.

Explanation: We already provide the Controller and View file in the "**elearning-source**" folder to make things easier for you to develop. The original way of generating the component is through artisan console command in terminal. The artisan command to generate the component is '**php artisan make:livewire ClassRoom\\ClassRoomWire**'. This will create the Controller file inside "**app\Http\Livewire\Class\ ClassRoomWire.php**" and View file inside "**resources\views\livewire\class\class-wire.php**".

Route

5) Route defines the URL pattern that you want to generate. Route will point to a controller. From the controller you can develop process logic or return to a view (V) or redirect to other url. Now open back your project inside the VS Code and lets take a look at the route located at web.php. For quick search, use the hotkey "**Ctrl+P**" and key in "**web.php**".

Uncomment below code. This is route we use to navigate to create new class page. The **url "/class"** will point to "**ClassRoomWire.php**" class where the "**function render()**" inside "**ClassRoomWire.php**" class will be run to return the user to the view (V).

```
Route::get('/class',    ClassRoomWire::class);
```

Figure 109: Insert route to ClassRoomWire class

And you need to import the class so the route can access to the right class that you have assigned.

```
use App\Http\Livewire\ClassRoom\ClassRoomWire;
```

Figure 110: Import class of ClassRoomWire

The "**web.php**" file should look like below after you uncomment the code above

```
<?php

use Illuminate\Support\Facades\Route;

use App\Http\Livewire\User\UserWire;

use App\Http\Livewire\Course\CourseWire;
use App\Http\Livewire\CourseFile\CourseFileWire;

use App\Http\Livewire\ClassRoom\ClassRoomWire;
// use App\Http\Livewire\ClassCourse\ClassCourseWire;

// use App\Http\Livewire\MyClassWire;


/*
|--------------------------------------------------------------------------
| Web Routes
|--------------------------------------------------------------------------
|
| Here is where you can register web routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| contains the "web" middleware group. Now create something great!
|
*/

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

```
    return redirect('login');
});
////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////
//Laravel Livewire
////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////
Route::middleware('auth')->group(function () {

    Route::get('/user', UserWire::class);

    Route::get('/course',   CourseWire::class);
    Route::get('/coursefile',   CourseFileWire::class);

    Route::get('/class',    ClassRoomWire::class);
   // Route::get('/classcourse',   ClassCourseWire::class);

    // Route::get('/myclass',   MyClassWire::class);


});
////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////

Route::middleware(['auth:sanctum', 'verified'])->get('/dashboard', function () {
    return view('dashboard');
})->name('dashboard');
```

Figure 111: Route "web.php" source code

6) Now open the system and try insert **url "/class"** after your local domain.
   Example: **elearning.test/class.** This will bring you to the empty class room page

Explanation: The route pointing to the "**ClassRoomWire.php**" class where inside the class the "**function render()**" is run and the function return a view "**class-room-wire.blade.php**". Inside that view you will find the view code to generate below page.

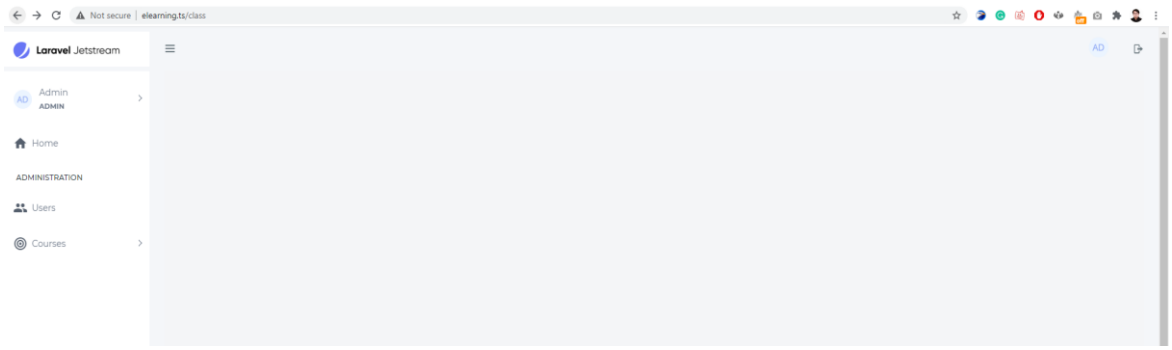


Figure 112: Class Room Page

7) The sidebar and topbar of the system using layout file call "**app.blade.php**". This layout will be use by all other pages so we don't need to build the topbar and sidebar everytime we want to create a new page.

8) By using **Ctrl+p** search for "**app.blade.php**" and open it. We need to have Class menu at the sidebar. To have that menu, uncomment the class menu under administration section. This will display the class menu at system sidebar. We use <a> tag as a link to the class page when user click on the class menu.

```
{{-------- START SECTION - ADMINISTRATION --------}}
@if (auth()->user()->role == 'admin')
    <li class="nav-small-cap">ADMINISTRATION</li>

    <li> <a class="waves-effect waves-dark" href="{{URL::to('user')}}" aria-
expanded="false"><i class="mdi mdi-account-multiple"></i><span class="hide-menu">Users </span></a>
    </li>
    <li> <a class="has-arrow waves-effect waves-dark" href="#" aria-
expanded="false"><i class="mdi mdi-bullseye"></i><span class="hide-menu">Courses </span></a>
        <ul aria-expanded="false" class="collapse">
            <li><a href="{{URL::to('course')}}">Create New Courses</a></li>
            <li><a href="{{URL::to('coursefile')}}">Add Resources to Course</a></li>
        </ul>
    </li>

    <li> <a class="has-arrow waves-effect waves-dark" href="#" aria-
expanded="false"><i class="mdi mdi-view-dashboard"></i><span class="hide-menu">Classes </span></a>
        <ul aria-expanded="false" class="collapse">
            <li><a href="{{URL::to('class')}}">Create New Class</a></li>
            <li><a href="{{URL::to('classcourse')}}">Add Course to Class</a></li>
        </ul>
    </li>
@endif
{{-------- END SECTION - ADMINISTRATION --------}}
```

Figure 113: "app.blade.php" source code

9) Now open back the system and you can see class menu has been display at the sidebar but with an empty content

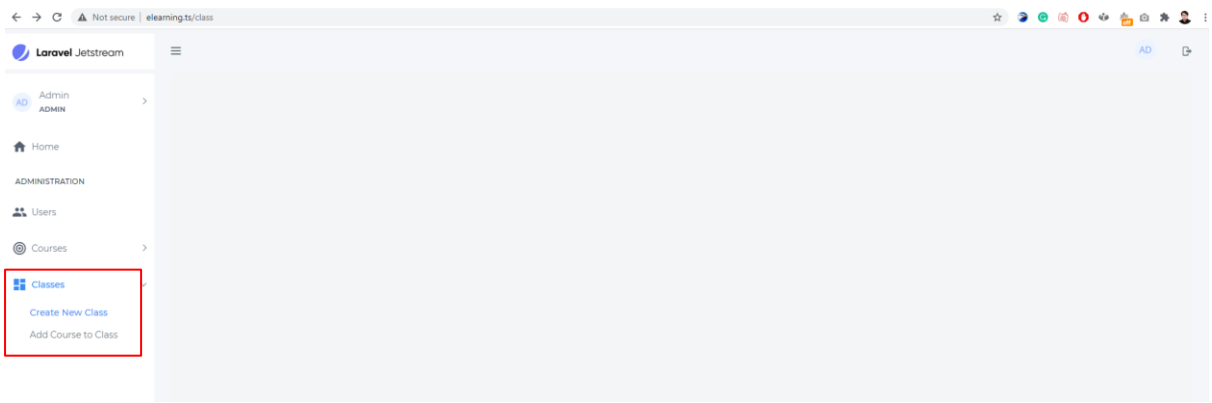


Figure 114: Empty Create New Class page with sidebar

Show Class Room Data Function

10) Search for "**ClassRoomWire.php**" and uncomment the code like below:

```php
<?php


namespace App\Http\Livewire\ClassRoom;
use Livewire\Component;
use App\Models\Team;

class ClassRoomWire extends Component
{
    // public $class_id;
    // public $owner_id;
    // public $name;
    // public $personal_team;
    // public $action;

    // protected $listeners = [
    //     'refreshParent' => '$refresh',
    //     'delete',
    // ];

    // public function selectItem($modelid , $action)
    // {
    //     $class = Team::find($modelid);

    //     $this->class_id = $modelid;
    //     $this->name = $class->name;
    //     $this->action = $action;

    //     if($action == "update")
    //     {
    //         $this->emit('getModelId' , $this->class_id);
    //     }
    //     else if($action == "manageClassCourse")
    //     {
    //         $this->emit('getModelIdDeeper' , $this->class_id);
    //         $this->dispatchBrowserEvent('openModal_manageCourseModal');
    //     }
```

Import Team model so we can use it to fetch class data.

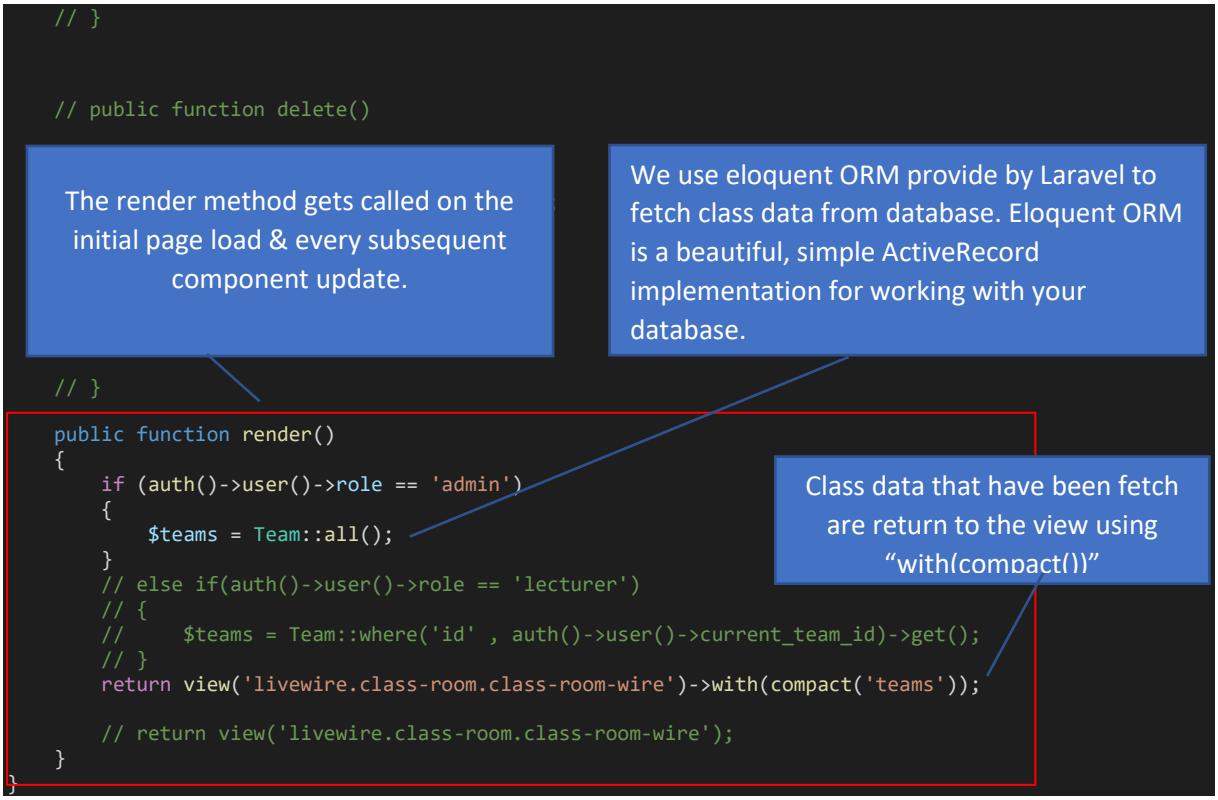


Figure 115: "ClassRoomWire.php" source code

11) Search for "**class-room-wire.blade.php**" and uncomment the section "**DATATABLE CLASS**" like below. This is the table that shows the list of classes.

```
<div>
    {{-- START SECTION - CLASS FORM  --}}
    {{-- @livewire('class-room.class-room-form-wire') --}}
    {{-- END SECTION - CLASS FORM  --}}


{{-- START SECTION - DATATABLE CLASS  --}}
<div class="row">
<div class="col-md-12">
    <div class="card" style="box-shadow: 0 .5rem 1rem rgba(0,0,0,.15)!important;border-radius: 5px;">
        <div class="card-body" style="overflow:scroll">
            <table class="table table-hover" >
                <thead>
                    <tr>
                        <th>Class Name</th>
                        <th>Class Owner</th>
                        <th>Is Personal Class</th>
                        <th>Created At</th>
```

```
                        <th>Updated At</th>
                        <th style='width:40px'>Action</th>
                    </tr>
                </thead>
                @foreach ($teams as $team)
                    <tbody>
                        <tr>
                            <td>{{$team->name}}</td>
                            <td>{{$team->owner ? $team->owner->name : ''}}</td>
                            <td>{{($team->personal_team == 1) ? 'Yes' : 'No'}}</td>
                            <td>{{date('j F Y', strtotime($team->created_at))}}</td>
                            <td>{{date('j F Y', strtotime($team->updated_at))}}</td>
                            <td>
                                <table style="border:none">
                                    <tr>
                                        @if (auth()->user()->role == 'admin')
                                            <td style="border:none">
                                                <button type="button" wire:click="selectItem({{$team->id}} , 'update' )" class="btn btn-sm waves-effect waves-light btn-warning"><i class="far fa-edit"></i></button>
                                            </td>
                                            <td style="border:none">
                                                <button type="button" wire:click="selectItem({{$team->id}} , 'delete' )" class="btn btn-sm waves-effect waves-light btn-danger data-delete" data-form="{{$team->id}}"><i class="fas fa-trash-alt"></i></button>
                                            </td>
                                        @endif

                                    </tr>
                                </table>
                            </td>
                        </tr>
                    </tbody>
                @endforeach
            </table>
        </div>
    </div>
</div>
</div>
 {{-- END SECTION - DATATABLE CLASS  --}}
```

Class data send from controller is use to show all class exist in system.

```
@push('scripts')
{{-- START SECTION - SCRIPT FOR DELETE BUTTON  --}}
{{-- <script>

document.addEventListener('livewire:load', function () {


$(document).on("click", ".data-delete", function (e)
    {
        e.preventDefault();
        swal({
        title: "Are you sure?",
        text: "Once deleted, you will not be able to recover!",
        icon: "warning",
        buttons: true,
        dangerMode: true,
        })
        .then((willDelete) => {
        if (willDelete) {
            e.preventDefault();
            Livewire.emit('delete')
        }
        });
    });
})
</script> --}}
{{-- END SECTION - SCRIPT FOR DELETE BUTTON  --}}
@endpush
</div>
```

Figure 116: "class-room-wire.blade" source code

12) Now open the system and you will see table of class with its data.

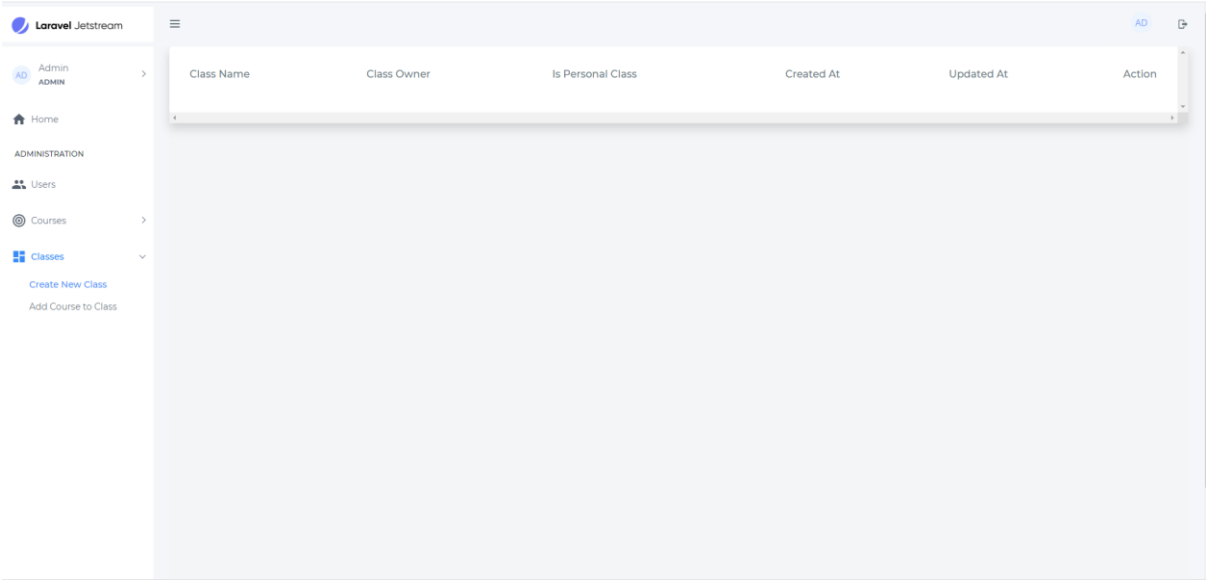


Figure 117: Create New Class page with table of class

Create New Class Room Function

13) Now, we already have view for class table. To enable the create function which means to add class functionality, search for "**ClassRoomFormWire.php**" and uncomment the code like below:

```php
<?php

namespace App\Http\Livewire\ClassRoom;

use Livewire\Component;
use App\Models\Team;
use App\Models\User;

class ClassRoomFormWire extends Component
{
    public $owner_id;
    public $name;
    public $personal_team;
    // public $model_id;

    // protected $listeners = [
    //     'getModelId'
    // ];

    // public function getModelId($model_id)
    // {
    //     $team = Team::find($model_id);

    //     $this->owner_id = $team->user_id;
    //     $this->name = $team->name;
    //     $this->personal_team = $team->personal_team;
    //     $this->model_id = $team->id;
    // }

    public function store()
    {
        // if($this->model_id)
        // {
        //     $this->validate([
        //         'owner_id' => 'required',
        //         'name' => 'required|string|max:255',
        //     ]);
```

Import user and class model so we can use it to fetch the data.

Public properties in Livewire are automatically made available to the view. You need to declare all the properties needed in view(V) here.


```
        //      $update = Team::find($this->model_id);
        //      $update->name = $this->name;
        //      $update->user_id = $this->owner_id;
        //      $update->personal_team = 0;
        //      $update->save();

        //      session()->flash('message', Class successfully updated');
        // }
        // else
        // {
            $this->validate([
                'name' => 'required|string|max:255',
            ]);

            $add = New Team;
            $add->name = $this->name;
            $add->user_id = $this->owner_id;
            $add->personal_team = 0;
            $add->save();

            session()->flash('message', 'New class
                      successfully added');
        // }

        // $this->emit('refreshParent');
    }

    public function render()
    {
        $users = User::where('role' , 'lecturer')->get();
        return view('livewire.class-room.class-room-form-wire')->with(compact('users'));
        // return view('livewire.class-room.class-room-form-wire');
    }
}
```

Figure 118: "ClassRoomFormWire.php"

14) Now go to "class-room-form-wire.blade.php" and uncomment the "CLASS FORM" section

```
    {{-- START SECTION - CLASS FORM  --}}
<div>
    <div class="row">
        <div class="col-lg-4">
            <h3 style="color:black">Class Details</h3>
```

```
            <p class="text-muted">Create a new class by fill up all required field.</p>
        </div>
        <div class="col-lg-12">

            <form wire:submit.prevent="store">

            <div class="card" style="box-shadow: 0 .5rem 1rem rgba(0,0,0,.15)!important;border-
radius: 5px;">
                <div class="card-body" >
                    <div class="row">
                        <div class="col-md-4">
                            <div class="form-group">
                                <label class="control-label" style="font-
weight:500">Class Name</label>
                                <input wire:model="name" type="text" id="name" name="name" class="form
-control" >
                                @error('name') <span class="error">{{ $message }}</span> @enderror
                            </div>
                        </div>
                        <div class="col-md-4">
                            <div class="form-group">
                                <label class="control-label" style="font-
weight:500">Class Owner</label>
                                <select wire:model="owner_id" name="owner_id" id="owner_id" class="for
m-control custom-select" data-placeholder="Choose Class Owner" tabindex="1">
                                    <option value="">-- Choose Class Owner --</option>
                                    @foreach ($users as $user)
                                        <option value="{{$user->id}}">{{$user->name}}</option>
                                    @endforeach
                                </select>
                                @error('owner_id') <span class="error">{{ $message }}</span> @enderror
                            </div>
                        </div>
                    </div>
                </div>

                <div class="card-footer" style="background-color: #f9fafb !important;border-
top: none;">
                    <div class="row">
                        <div class="col-md-12 text-right">
                            @if (session()->has('message'))
                                {{ session('message') }}
```

Laravel livewire use wire:submit.prevent = ”function in controller” to submit form.

Laravel livewire use wire:model to synchronize value of form input and public properties in the controller.

If validation is fail at controller. It will return the error message to view and the message will show up here.

```
                        @endif

                        <button type="submit" class="inline-flex items-center px-4 py-2 bg-gray-
800 border border-transparent rounded-md font-semibold text-xs text-white uppercase tracking-
widest hover:bg-gray-700 active:bg-gray-900 focus:outline-none focus:border-gray-900 focus:shadow-
outline-gray disabled:opacity-25 transition ease-in-out duration-150">

                            Save

                        </button>

                    </div>

                </div>

            </div>

        </form>

        </div>

    </div>

    </div>

    {{-- END SECTION - CLASS FORM  --}}
```

15) To include "**class-room-form-wire.blade.php**" component inside "**class-room-wire.blade.php**", go to "**class-room-wire.blade.php**" and uncomment the "**CLASS FORM**" section.

```
{{-- START SECTION - CLASS FORM  --}}

@livewire('class-room.class-room-form-wire')

{{-- END SECTION - CLASS FORM  --}}
```

Figure 119: "class-room-wire.blade" Class Form section

16) Now go back to system and you will see the create form for class room. You can create class after fill in the class name and choose the class owner which is lecturer. So if you don't have any lecturers in your user database, you can add lecturer at Users Section. After successfully create class room, a massage will appear notified that class has been created. You can see the added data by simply refreshing the page.

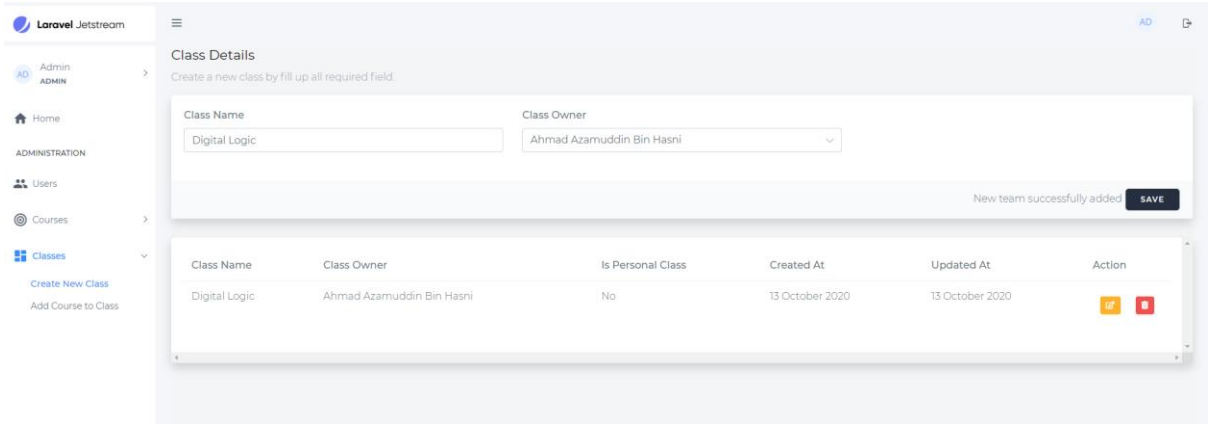


Figure 120: User page with table of user & create user component

17) To have a real-time update on the class room table, go back to "**ClassRoomWire.php**" and uncomment the code other than `// 'delete'` . This is the listener to listen on successful store event of class room and will helps automatically added into table below without refreshing the page.

```
protected $listeners = [

    'refreshParent' => '$refresh',

    // 'delete'
];
```

Figure 121: "ClassRoomWire.php" listener

18) After that go to "**ClassRoomFormWire.php**" and uncomment below code. This code is use to triggered the listener.

```
$this->emit('refreshParent');
```

Figure 122: "ClassRoomFormWire.php" emitting to listener

19) We use listener on store function in "**ClassRoomFormWire.php**" to detect if class have been successfully created. First, we define the listener in "**ClassRoomWire.php**" just like in the code above. Then we call the listener by using **emit('refreshParent').** When the listener is triggered, the **magic keyword** provide by Laravel livewire which is "**$refresh**" will refresh the "**ClassRoomWire.php**" component which is the class table so we will have the latest data that we just created. Go on and try to insert back new class.

Edit the Class Room Function

20) If you noticed we already uncomment some code inside "**class-room-wire.blade**". The code will fire when user click on the button. The **wire:click** event will run the **selectItem** function inside "**ClassRoomWire.php**" to set all the public properties to a specific class that we have choose.

```
<td style="border:none">
       <button type="button" wire:click="selectItem({{$team->id}} , 'update' )" class="btn btn-
sm waves-effect waves-light btn-warning"><i class="far fa-edit"></i></button>
</td>
```

21) Go to "**ClassRoomWire.php**" and uncomment like below code. The "**selectItem**" function will run on "**wire:click**" that we define at "**class-room-wire.blade.php**". We write the logic '**if($action == "update")**' if the parameter "**$action**" pass an "**update**" value, thus it will emit listener name "**getModelId**" with parameter of **class id**.

```
public function selectItem($modelid , $action)
    {
        $class = Team::find($modelid);

        $this->class_id = $modelid;
        $this->name = $class->name;
        $this->action = $action;

        if($action == "update")
        {
            $this->emit('getModelId' , $this->class_id);
        }
    }
```

Figure 123: "ClassRoomWire.php" selectItem function

22) After that, go to "**ClassRoomFormWire.php**" and uncomment below code. This code is to listen to an event call "**getModelID**" from the "**ClassRoomWire.php**". The listener will run the function **getModelId** and set all the data of course. This data will synchronise automatically with the view where we define **wire:model** at "**class-room-wire-form.blade.php**".

```
<?php

namespace App\Http\Livewire\ClassRoom;

use Livewire\Component;
use App\Models\Team;
use App\Models\User;

class ClassRoomFormWire extends Component
{
    public $owner_id;
    public $name;
    public $personal_team;
```

```php
public $model_id;

protected $listeners = [
    'getModelId'
];

public function getModelId($model_id)
{
    $team = Team::find($model_id);

    $this->owner_id = $team->user_id;
    $this->name = $team->name;
    $this->personal_team = $team->personal_team;
    $this->model_id = $team->id;

}
```

Figure 124: "ClassRoomFormWire.php"

23) Now go to the system and click at the edit button and you will see the data that you want to change will automatically fill in the form above.

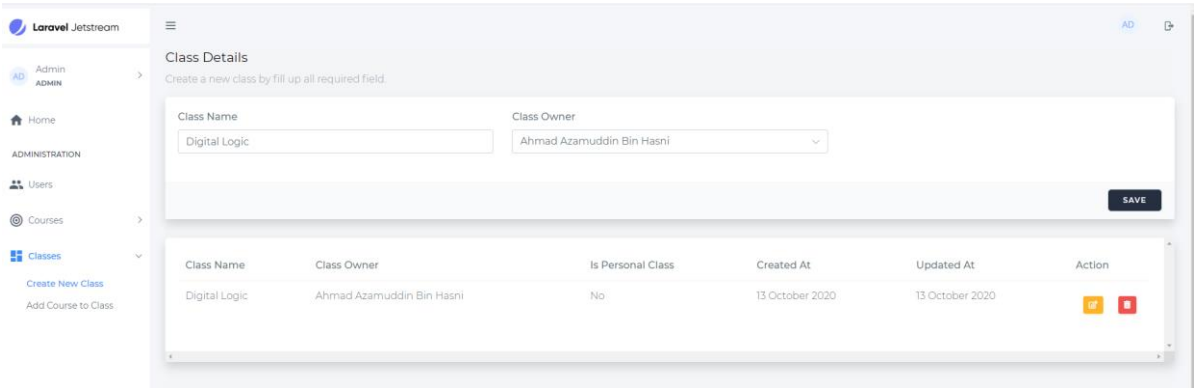


Figure 125: Create New Class page with applied real-time update

24) But If you click on the save button, the data will create a new class. To enable the edit function lets do some logic on the store function. Go back to "ClassRoomFormWire.php" and uncomment all the code located in the store function.

```php
public function store()
{
    if($this->model_id)
    {
        $this->validate([
            'owner_id' => 'required',
            'name' => 'required|string|max:255',
        ]);

        $update = Team::find($this->model_id);
        $update->name = $this->name;
        $update->user_id = $this->owner_id;
```

If this function detect "model_id" has a value, then it will do an update to the existed class data.

```
            $update->personal_team = 0;
            $update->save();

            session()->flash('message', 'Class successfully updated');
        }
        else
        {
            $this->validate([
                'name' => 'required|string|max:255',
            ]);

            $add = New Team;
            $add->name = $this->name;
            $add->user_id = $this->owner_id;
            $add->personal_team = 0;
            $add->save();

            session()->flash('message', 'New class successfully added');
        }

        $this->emit('refreshParent');

    }
```

Figure 126: "ClassRoomFormWire.php" source code

25) Lets jump back to the system and edit some user. The edit function will be successful.

Delete the Class Room Function

26) If you noticed we already uncomment some code inside "**class-room-wire.blade**". The code will fire when user click on the button. The **wire:click** event will run the **selectItem** function inside "**ClassRoomWire.php**" to set all the public properties to a specific class that we have choose.

```
<td style="border:none">
    <button type="button" wire:click="selectItem({{$team->id}} , 'delete' )" class="btn btn-
sm waves-effect waves-light btn-danger data-delete" data-form="{{$team->id}}"><i class="fas fa-trash-
alt"></i></button>
</td>
```

27) To active the delete button, go to "**ClassRoomWire.php**" and uncomment the below code

```
public $class_id;
public $owner_id;
public $name;
public $personal_team;
public $action;

    protected $listeners = [
        'refreshParent' => '$refresh',
        'delete',
    ];

    public function delete()
    {
        $team = Team::find($this->class_id);
        $team->delete();
    }
```

Because we already set the public properties on wire:click event at step 26. We can use the value to search for the class and delete it.

Figure 127: "ClassRoomWire.php"

28) Then, go to "**class-room-wire.blade.php**" and uncomment section "**SCRIPT FOR DELETE BUTTON**"

```
<script>
document.addEventListener('livewire:load', function () {
$(document).on("click", ".data-delete", function (e)
    {
        e.preventDefault();
        swal({
        title: "Are you sure?",
        text: "Once deleted, you will not be able to recover!",
        icon: "warning",
        buttons: true,
        dangerMode: true,
        })
```

This script will run on user click on tag that has a "data-delete" class. If you see at step 26, we define a "data-delete" class at the delete button.

```
        .then((willDelete) => {
        if (willDelete) {
            e.preventDefault();
            Livewire.emit('delete')
        }
        });
    });
})
</script>
```

This emit function of delete will sent a message to the listener located at "ClassRoomWire.php" and call the function delete.

Figure 128: "class-room-wire.blade.php" Script for Delete Button section

29) After refreshing the page, you can try to delete class room by clicking the delete button as below

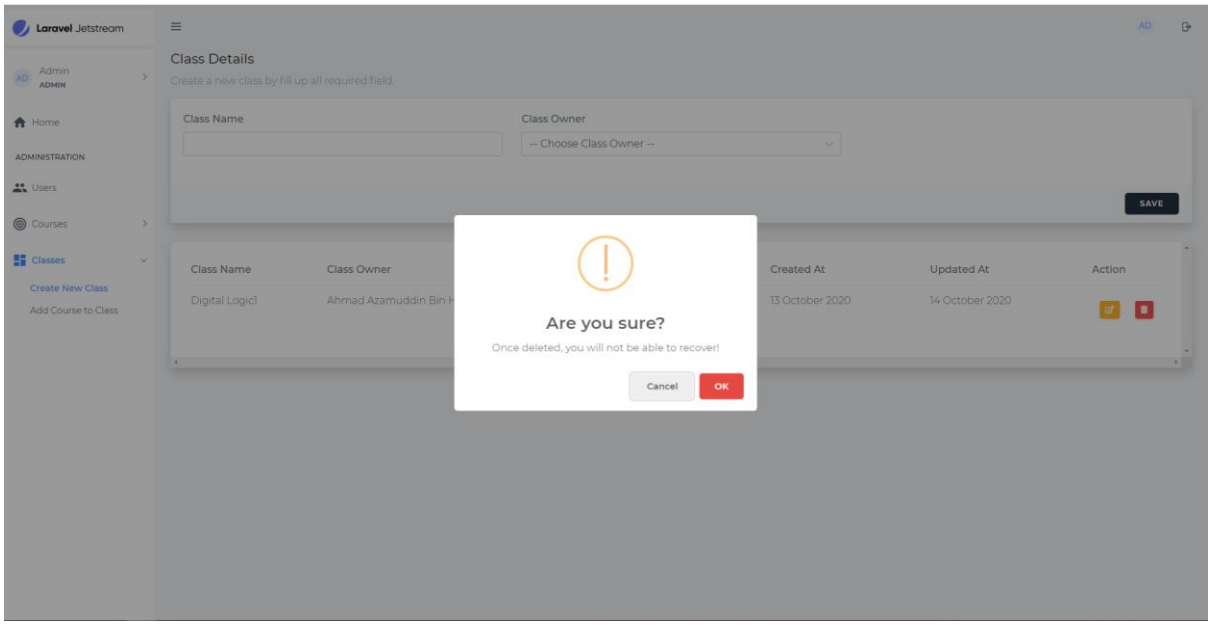


Figure 129: Class Page with delete function

Submodule – Add Course to Class

Submodule Outcome – At the end of this module, you can develop the Submodule Add Course to Class for administration use. This module will include below function:

i) Show the class course data
ii) Creating the class course
iii) Delete the class course

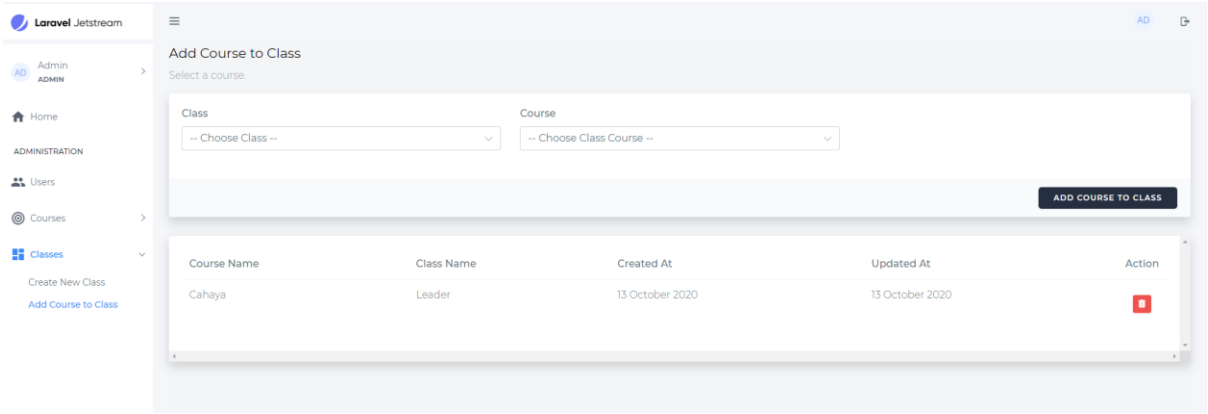


Figure 108: Course assignment into the class page

1) To have like above page. We must have the **Model-View-Controller (MVC)** component.

2) **For Model**, open the "**elearning-source**" folder and navigate to **Model** folder. Copy the **"ClassCourse.php"** file and paste it to "**www\elearning\app\Models\"** in your project folder. This will be the model for class course.

Explanation: We already provide the Model file in the "**elearning-source**" folder to make things easier for you to develop. The original way of generating Class Course model is through Laravel artisan command in terminal which is "**php artisan make:model ClassCourse**". This will create model file inside "**app\Models\ClassCourse.php**".

3) **For Controller**, open the "**elearning-source**" folder and navigate to **Controller** folder. Copy the **"ClassCourse"** folder and paste it to "**laragon\www\elearning\app\Http\ Livewire**\" (project folder)**.** This will be the logical code for this submodule.

4) **For View,** open back "**elearning-source**" folder and navigate to **View** folder. Copy the **"class-course"** folder and paste it to "**laragon\www\elearning\resources\views\ livewire**\" (project folder). This will be the view for this submodule.

Explanation: We also provide the Controller and View file in the "**elearning-source**" folder to make things easier for you to develop. The original way of generating the component is through artisan console command in terminal. The artisan command to generate the component is '**php artisan make:livewire ClassCourse\\ClassCourseWire**'. This will create the Controller file inside "**app\Http\Livewire\ClassCourse\ ClassCourseWire.php**" and View file inside "**resources\views\livewire\class-course\class-course-wire.php**".

Route

5) Route defines the URL pattern that you want to generate. Route will point to a controller. From the controller you can develop process logic or return to a view (V) or redirect to other url. Now open back your project inside the VS Code and lets take a look at the route located at web.php. For quick search, use the hotkey "**Ctrl+P**" and key in "**web.php**".

Uncomment below code. This is route we use to navigate to add resource to class course page. The **url "/classcourse"** will point to "**ClassCourseWire.php**" class where the "**function render()**" inside "**ClassCourseWire.php**" class will be run to return the user to the view (V)

```
Route::get('/classcourse',  ClassCourseWire::class);
```

Figure 130: Insert route to ClassCourseWire class

And you need to import the class so the route can access to the right class that you have assigned.

```
use App\Http\Livewire\ClassCourse\ClassCourseWire;
```

Figure 131: Import class of ClassCourseWire

The "**web.php**" should look like below after you uncomment the above code.

```
<?php

use Illuminate\Support\Facades\Route;

use App\Http\Livewire\User\UserWire;

use App\Http\Livewire\Course\CourseWire;
use App\Http\Livewire\CourseFile\CourseFileWire;

use App\Http\Livewire\ClassRoom\ClassRoomWire;
use App\Http\Livewire\ClassCourse\ClassCourseWire;

// use App\Http\Livewire\MyClassWire;



/*
|--------------------------------------------------------------------------
| Web Routes
|--------------------------------------------------------------------------
|
| Here is where you can register web routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| contains the "web" middleware group. Now create something great!
|
*/

Route::get('/', function () {
    return redirect('login');
});


////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////
//Laravel Livewire
////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////
Route::middleware('auth')->group(function () {

    Route::get('/user', UserWire::class);

    Route::get('/course',   CourseWire::class);
    Route::get('/coursefile',   CourseFileWire::class);
```

```
    Route::get('/class',     ClassRoomWire::class);
    Route::get('/classcourse',  ClassCourseWire::class);

    // Route::get('/myclass',    MyClassWire::class);

});
//////////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////

Route::middleware(['auth:sanctum', 'verified'])->get('/dashboard', function () {
    return view('dashboard');
})->name('dashboard');
```

Figure 132: Route "web.php" source code

6) Now open the system and try insert the **url "/classcourse"** after your local domain. Example: **elearning.test/classcourse** This bring you to the empty class course page.

Explanation: The route pointing to the "Class**CourseWire.php**" class where inside the class the "**function render()**" is run and the function return a view "**class-course-wire.blade.php**". Inside that view you will find the view code to generate below page.

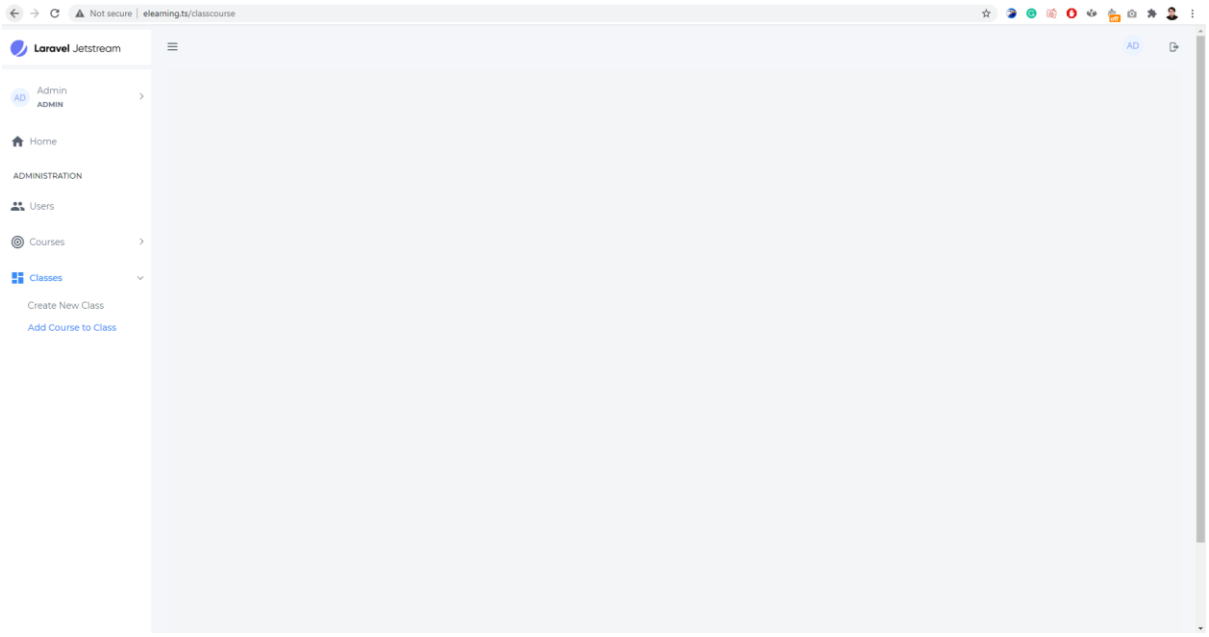


Figure 133: Empty Add Course to class page

Show The Class Course Data Function

7) Search for "**ClassCourseWire.php**" and uncomment the code like below:

```php
<?php

namespace App\Http\Livewire\ClassCourse;

use Livewire\Component;
use App\Models\ClassCourse;

class ClassCourseWire extends Component
{
    // public $id_classcourse;
    // public $id_class;
    // public $id_course;
    // public $action;

    // protected $listeners = [
    //     'refreshParent' => '$refresh',
    //     'delete',
    // ];
    // public function selectItem($modelid , $action)
    // {
    //     $this->id_classcourse = $modelid;
    //     $this->action = $action;

    //     if($action == "update")
    //     {
    //         $this->emit('getModelId' , $this->id_classcourse);
    //     }
    // }

    // public function delete()
    // {
    //     $classcourse = ClassCourse::find($this->id_classcourse);

    //     $classcourse->delete();
    // }
```

Import ClassCourse model so we can use it to fetch class course data.

```php
    public function render()
    {
        $classcourses = ClassCourse::all();
        return view('livewire.class-course.class-course-wire')->with(compact('classcourses'));
        // return view('livewire.class-course.class-course-wire');

    }
}
```

Figure 134: "ClassCourseWire.php" source code

8) Search for "**class-course-wire.blade.php**" and uncomment the section "**DATATABLES CLASS COURSE**" like below. This is a table to show the list of class course assignment.

```
<div>
    {{-- START SECTION - CLASS COURSE FORM  --}}
    {{-- @livewire('class-course.class-course-form-wire') --}}
    {{-- END SECTION - CLASS COURSE FORM  --}}

 {{-- START SECTION - DATATABLE CLASS COURSE  --}}
<div class="row">
@if (auth()->user()->role == 'admin')
        <div class="col-md-12">
    @else
        <div class="col-md-4"></div>
        <div class="col-md-8">
    @endif


        <div class="card" style="box-shadow: 0 .5rem 1rem rgba(0,0,0,.15)!important;border-
radius: 5px;">
            <div class="card-body" style="overflow:scroll">
                <table class="table table-hover" >
                    <thead>
                        <tr>
                            <th>Course Name</th>
                            <th>Class Name</th>
                            <th>Created At</th>
                            <th>Updated At</th>
                            <th style='width:40px'>Action</th>
                        </tr>
                    </thead>
                        @foreach ($classcourses as $classcourse)
```

```
                        <tbody>
                            <tr>
                                <td>{{$classcourse->course->name}}</td>
                                <td>{{$classcourse->class ? $classcourse->class->name : ''}}</td>
                                <td>{{date('j F Y', strtotime($classcourse->created_at))}}</td>
                                <td>{{date('j F Y', strtotime($classcourse->updated_at))}}</td>
                                <td>
                                    <table style="border:none">
                                        <tr>
                                            <td style="border:none">
                                                <button type="button" wire:click="selectItem({{$classc
ourse->id}} , 'delete' )" class="btn btn-sm waves-effect waves-light btn-danger data-delete" data-
form="{{$classcourse->id}}"><i class="fas fa-trash-alt"></i></button>
                                            </td>
                                        </tr>
                                    </table>
                                </td>
                            </tr>
                        </tbody>
                        @endforeach
                </table>
            </div>
        </div>
    </div>
</div>
    {{-- END SECTION - DATATABLE CLASS COURSE   --}}
@push('scripts')
{{-- START SECTION - SCRIPT FOR DELETE BUTTON  --}}


{{-- <script>


  document.addEventListener('livewire:load', function () {


    $(document).on("click", ".data-delete", function (e)
        {
            e.preventDefault();
            swal({
            title: "Are you sure?",
            text: "Once deleted, you will not be able to recover!",
            icon: "warning",
            buttons: true,
```

```
            dangerMode: true,
            })
            .then((willDelete) => {
            if (willDelete) {
                e.preventDefault();
                Livewire.emit('delete')
            }
            });
        });

  })
</script> --}}
{{-- END SECTION - SCRIPT FOR DELETE BUTTON  --}}

@endpush

</div>
```

Figure 135: "class-course-wire.blade" source code

9) Now open the system and you will see empty table of class course

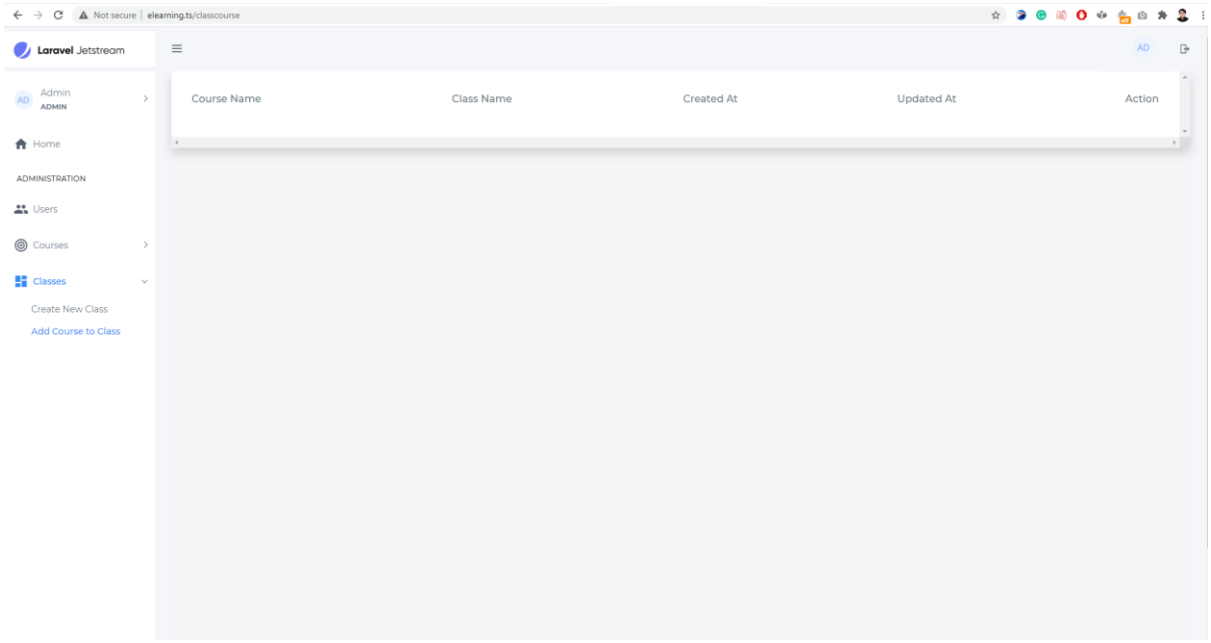


Figure 136: Class Course page with table of course

Create the Class Course Function

10) Now, we already have view for table of class course. To have a create function, search for "**ClassCourseFormWire.php**" and uncomment the code like below:

```php
<?php

namespace App\Http\Livewire\ClassCourse;

use Livewire\Component;
use App\Models\Team;
use App\Models\Course;
use App\Models\ClassCourse;

class ClassCourseFormWire extends Component
{
    public $id_class;
    public $id_course;

    public function store()
    {

            $this->validate([
                'id_course' => 'required',
                'id_class' => 'required',
            ]);

            $add = New ClassCourse;
            $add->id_class = $this->id_class;
            $add->id_course = $this->id_course;
            $add->save();

            session()->flash('message', 'New course successfully added into class');

        // $this->emit('refreshParent');
    }
```

```
public function render()
    {
        if(auth()->user()->role == 'admin')
        {
            $courses = Course::all();
            $classes = Team::all();
        }
        // else
        // {
        //     $courses = Course::where('id_lecturer' , auth()->user()->id)->get();
        // }

        return view('livewire.class-course.class-course-form-wire')-
>with(compact('courses' , 'classes'));

        // return view('livewire.class-course.class-course-form-wire');
    }


}
```

Figure 137: "ClassCourseFormWire.php"

11) Now go to "**class-course-form-wire.blade.php**" and uncomment the all the lines

```
<div>

    {{-- START SECTION - CLASS COURSE FORM  --}}
    <div class="row">
        <div class="col-lg-4">
            @if (auth()->user()->role == 'admin')
                <h3 style="color:black">Add Course to Class</h3>
                <p class="text-muted">Select a course.</p>
            @else
                <div class="md:col-span-1">
                    <div class="px-4 sm:px-0">
                        <h3 class="text-lg font-medium text-gray-900">Add Course to Class</h3>
                        <p class="mt-1 text-sm text-gray-600">
                            Select a course.
                        </p>
                    </div>
                </div>
            @endif
        </div>
```

```
    </div>
    <div class="row">
        @if (auth()->user()->role == 'admin')
        <div class="col-md-12">
        @else
        <div class="col-md-4"></div>
        <div class="col-md-8">
        @endif
            <form wire:submit.prevent="store">
            <div class="card" style="box-shadow: 0 .5rem 1rem rgba(0,0,0,.15)!important;border-
radius: 5px;">
                <div class="card-body" >
                    <div class="row">
                        <div class="col-md-4">
                            <div class="form-group">
                                <label class="control-label" style="font-weight:500">Class</label>
                                <select wire:model="id_class" name="id_class" id="id_class" class="for
m-control custom-select" data-placeholder="Choose Class Course" tabindex="1">
                                    <option value="">-- Choose Class --</option>
                                    @foreach ($classes as $class)
                                        <option value="{{$class->id}}">{{$class->name}}</option>
                                    @endforeach
                                </select>
                                @error('id_class') <span class="error">{{ $message }}</span> @enderror
                            </div>
                        </div>
                        <div class="col-md-4">
                            <div class="form-group">
                                <label class="control-label" style="font-weight:500">Course</label>
                                <select wire:model="id_course" name="id_course" id="id_course" class="
form-control custom-select" data-placeholder="Choose Class Course" tabindex="1">
                                    <option value="">-- Choose Class Course --</option>
                                    @foreach ($courses as $course)
                                        <option value="{{$course->id}}">{{$course->name}}</option>
                                    @endforeach
                                </select>
                                @error('id_course') <span class="error">{{ $message }}</span>@enderror
                            </div>
                        </div>
                    </div>
                </div>
```

```
            <div class="card-footer" style="background-color: #f9fafb !important;border-top: none;">
                    <div class="row">
                        <div class="col-md-12 text-right">
                            @if (session()->has('message'))
                                {{ session('message') }}
                            @endif
                            <button type="submit" class="inline-flex items-center px-4 py-2 bg-gray-
800 border border-transparent rounded-md font-semibold text-xs text-white uppercase tracking-
widest hover:bg-gray-700 active:bg-gray-900 focus:outline-none focus:border-gray-900 focus:shadow-
outline-gray disabled:opacity-25 transition ease-in-out duration-150">
                                Add Course to Class
                            </button>
                        </div>
                    </div>
                </div>
            </div>
        </form>
        </div>
    </div>
{{-- END SECTION - CLASS COURSE FORM  --}}
    </div>
```

Figure 138: "class-course-form-wire.blade.php" source code

12) To include "**class-course-form.blade.php"** component inside the "**class-course-wire.blade.php**", go to "**class-course-wire.blade.php**" and uncomment the "**CLASS FORM SECTION**"

```
    @livewire('class-course.class-course-form-wire')
```

Figure 139: "class-course-wire.blade" Class Course Form section

13) Now go back to the system and you will see the form to assign course into the class. On successful entry, you will got a message notified that course has been successfully added to the class. You need to refresh the page to see the data you just entered into the table below the form.

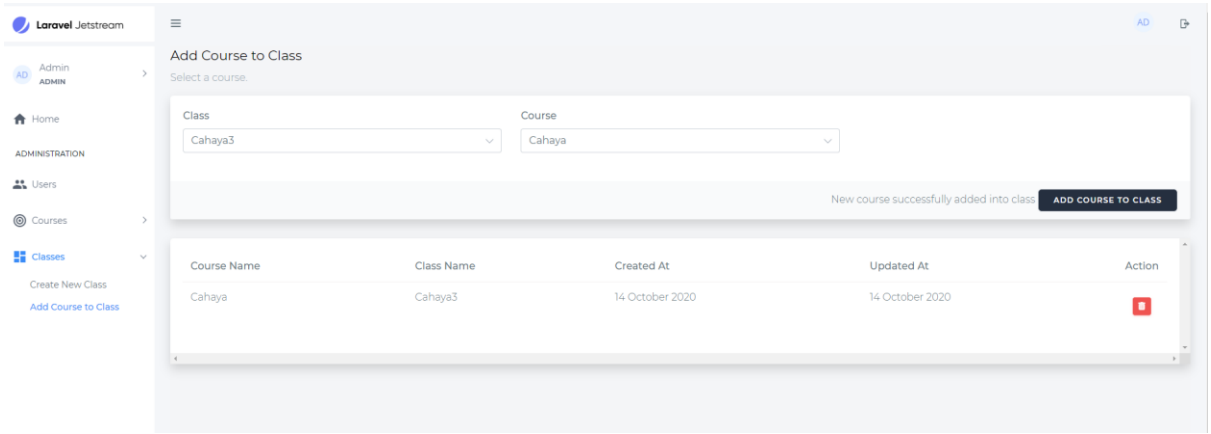


Figure 140: Class Course page with table of course & add course to class component

14) To have a real-time update on the class course table, go back to "**ClassCourseWire.php**" and uncomment the code other than `// 'delete'`. The code is use to trigger the listener.

```
protected $listeners = [

    'refreshParent' => '$refresh',

    // 'delete'

];
```

Figure 141: Listeners

15) After that go to "**ClassCourseFormWire.php**" and uncomment below code. This code is use to triggered the listener.

```
$this->emit('refreshParent');
```

16) We use listener on store function in "**ClassCourseFormWire.php**" to detect if class course have been successfully created. First, we **define the listener** in "**ClassCourseWire.php**" just like in the code above. Then we call the listener by using **emit('refreshParent')**. When the listener is triggered, the **magic keyword** provides by Laravel livewire which is **"$refresh"** will refresh the "**ClassCourseWire.php**" component which is the class course table so we will have the latest data that we just created. Go on and try to insert back new course.

Delete the Class Course Function

17) If you noticed we already uncomment some code inside "**class-course-wire.blade**". The code will fire when user click on the button. The **wire:click** event will run the **selectItem** function inside "**ClassCourseWire.php**" to set all the public properties to a specific class course that we have choose.

```
<td style="border:none">
       <button type="button" wire:click="selectItem({{$classcourse->id}} , 'delete' )" class="btn btn-sm waves-effect waves-light btn-danger data-delete" data-form="{{$classcourse->id}}"><i class="fas fa-trash-alt"></i></button>
</td>
```

18) Beside using wire:click event, to include delete function, go to "**class-course-wire.blade.php**" and uncomment the subsection "**SCRIPT FOR DELETE BUTTON**" like below. . This is the script to popup a message of confirmation on deletion of course.

```
<script>


  document.addEventListener('livewire:load', function () {


    $(document).on("click", ".data-delete", function (e)
        {
            e.preventDefault();
            swal({
            title: "Are you sure?",
            text: "Once deleted, you will not be able to recover!",
            icon: "warning",
            buttons: true,
            dangerMode: true,
            })
            .then((willDelete) => {
            if (willDelete) {
                e.preventDefault();
                Livewire.emit('delete')
            }
            });
        });


  })
</script>
```

Figure 142: "class-course-wire.blade.php" Script for Delete Button section

19) Then go to "**ClassCourseWire.php**", uncomment the codes below

```
public $id_classcourse;
public $id_class;
public $id_course;
public $action;
```

Figure 144: "ClassCourseWire.php"

```
public function selectItem($modelid , $action)
{
    $this->id_classcourse = $modelid;
    $this->action = $action;
}
public function delete()
{
    $classcourse = ClassCourse::find($this->id_classcourse);

    $classcourse->delete();

}
```

Figure 145: "ClassCourseWire.php"

20) Now lets navigate back to system and try the delete function by clicking the delete button as below

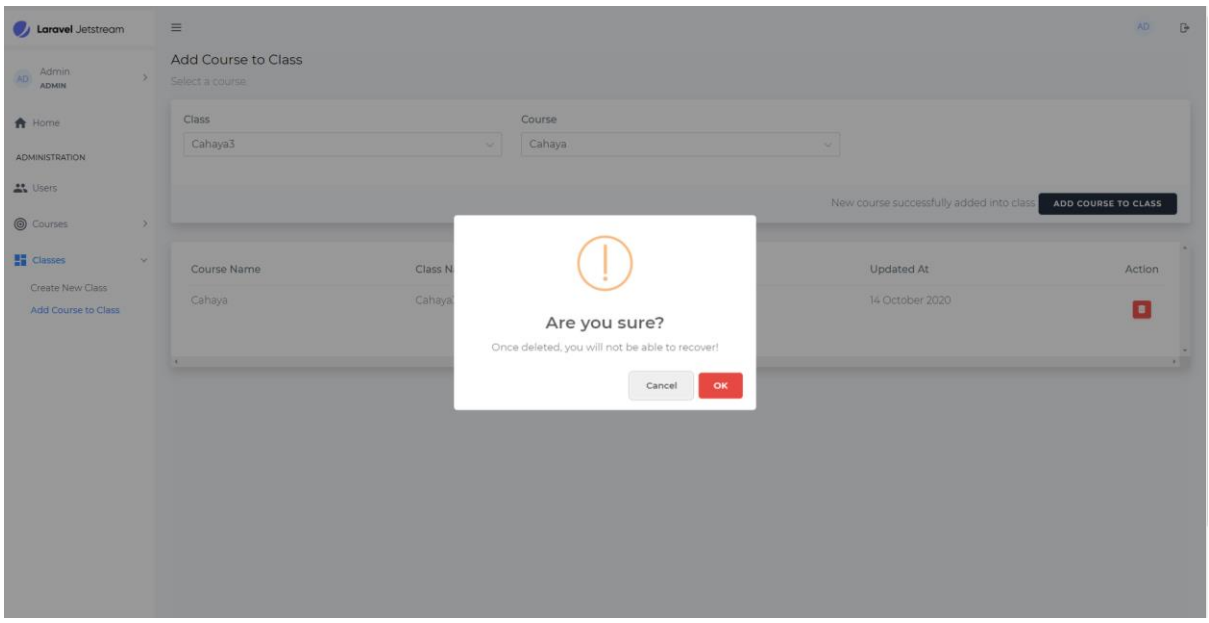


Figure 146:Class Course Page with delete function

Role: Lecturer

User Module

Lecturer will not have privilege to manage user. Only administrator have this privilege. So, let's skip this module and go to next module.

Course Module

Submodule – Create New Course

Module Outcome – At the end of this module, you can develop the Submodule Create New Course for lecturer use. This module will include below function

i) Show the course data for specific lecturer
ii) Creating the course
iii) Edit the course
iv) Delete the course.

Show the course data for specific lecturer

We already develop this submodule for administrator. We will use back all the function for lecturer. For administrator, they can see all course exist in the system but for lecturer, they only can see their course. So, we need to filter some code so the system will react based on user role.

Before that, we need to show course menu at system sidebar for lecturer. Go to "**app.blade.php**" and uncomment the course menu under **lecturer section**.

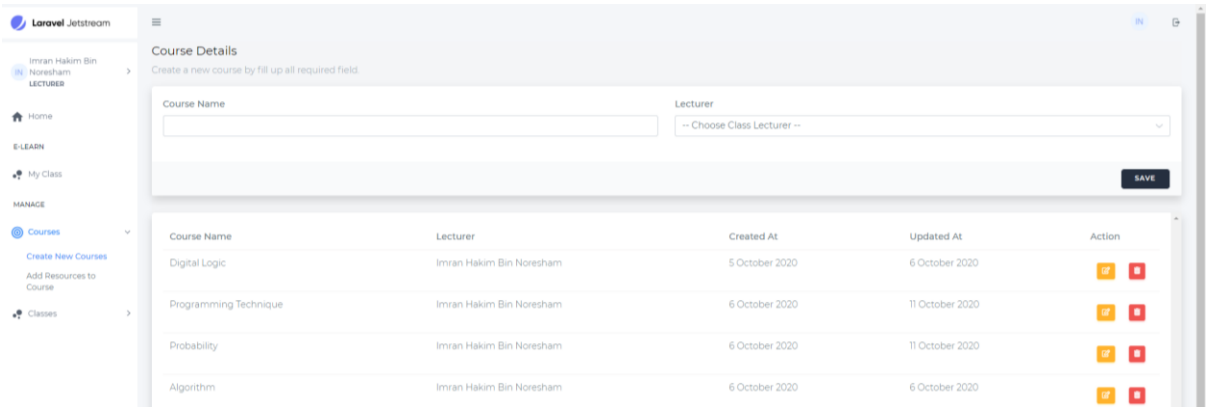
```
{{-------- START SECTION - LECTURER --------}}
@if (auth()->user()->role == 'lecturer')
    {{-- <li class="nav-small-cap">E-LEARN</li>
    <li> <a class="waves-effect waves-dark" href="{{URL::to('myclass')}}" aria-
expanded="false"><i class="mdi mdi-chart-bubble"></i><span class="hide-menu">My Class</span></a>

    </li> --}}

    <li class="nav-small-cap">MANAGE</li>
    <li> <a class="has-arrow waves-effect waves-dark" href="#" aria-
expanded="false"><i class="mdi mdi-bullseye"></i><span class="hide-menu">Courses </span></a>
        <ul aria-expanded="false" class="collapse">


            <li><a href="{{URL::to('course')}}">Create New Courses</a></li>


            <li><a href="{{URL::to('coursefile')}}">Add Resources to Course</a></li>

        </ul>
    </li>

    {{-- <li> <a class="has-arrow waves-effect waves-dark" href="#" aria-
expanded="false"><i class="mdi mdi-chart-bubble"></i><span class="hide-menu">Classes </span></a>
        <ul aria-expanded="false" class="collapse">


            <li><a href="{{URL::to('teams/create')}}">Create New Class</a></li>

            @if (Auth::user()->current_team_id)
                <li><a href={{ route('teams.show', Auth::user()->currentTeam-
>id) }}>Class Setting</a></li>
            @endif
        </ul>
    </li> --}}

@endif
{{-------- END SECTION - LECTURER --------}}
```

Go to "**CourseWire.php**" and follow the code like below.

```php
public function render()
    {
        if (auth()->user()->role == 'admin') {
            $courses = Course::all();
        }
        else if (auth()->user()->role == 'lecturer'){
            $courses = Course::where('id_lecturer',auth()->user()->id)->get();
        }
        return view('livewire.course.course-wire')->with(compact('courses'));

        // return view('livewire.course.course-wire');
    }
```

This is the data we use to show at table of course. We need to filter so that lecturer will only see their course.

Login as lecturer and go to **Create New Course page**. You will see that the course table only show data for that lecturer only.

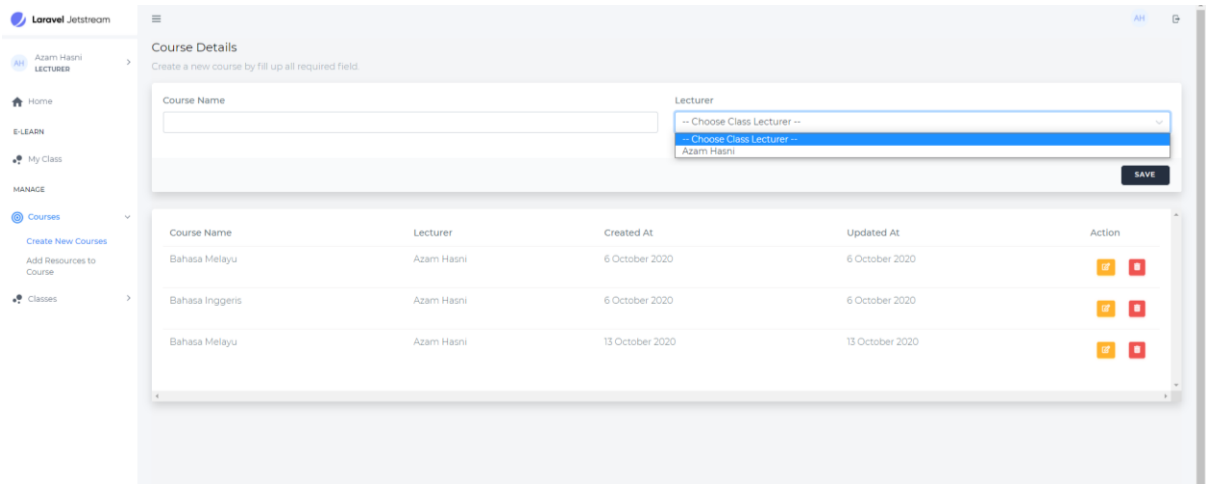


Creating the course & Editing the course

Other things that we need to inform is when lecturer want to create new course. If administration, they can create course and assign to any lecturer exist in the system. But for lecturer they can create course on their behalf only. If you noticed, we already uncomment code inside "**course-form-wire.blade.php**". This is where the system filter to show only specific lecturer based on user role.

```php
@if (auth()->user()->role == 'admin')
        <select wire:model="id_lecturer" name="id_lecturer" id="id_lecturer" class="form-
control custom-select" data-placeholder="Choose Class Owner" tabindex="1">
                <option value="">-- Choose Class Lecturer --</option>
                @foreach ($lecturers as $lecturer)
                        <option value="{{$lecturer->id}}">{{$lecturer->name}}</option>
                @endforeach
        </select>
@else
        <select wire:model="id_lecturer" name="id_lecturer" id="id_lecturer" class="form-
control custom-select" data-placeholder="Choose Class Owner" tabindex="1">
                <option value="">-- Choose Class Lecturer --</option>
                <option value="{{auth()->user()->id}}">{{auth()->user()->name}}</option>
        </select>
@endif
```

Login as lecturer and go to **Create New Course page**. You will see that the dropdown to select lecturer already filter to them only.



Deleting Course

No customization needs when delete course for lecture. They will follow the same process as admin.

Submodule – Add Resources to Course

Submodule Outcome – At the end of this module, you can develop the Submodule Add Resources to Course for lecturer use. This module will include below function:

i) Show the resources data for current login lecturer
ii) Add the resources
iii) Delete the resources

Show the resources data for current login lecturer

We already develop this submodule for administrator. We will use back all the function for lecturer. For administrator, they can see all course file exist in the system but for lecturer, they can only see their course file. So, we need to filter some code so the system will react based on user role.

Go to "**CourseFileWire.php**" and follow the code like below.

This is the data we use to show at table of course file. We need to filter so that lecturer will only see their course file.

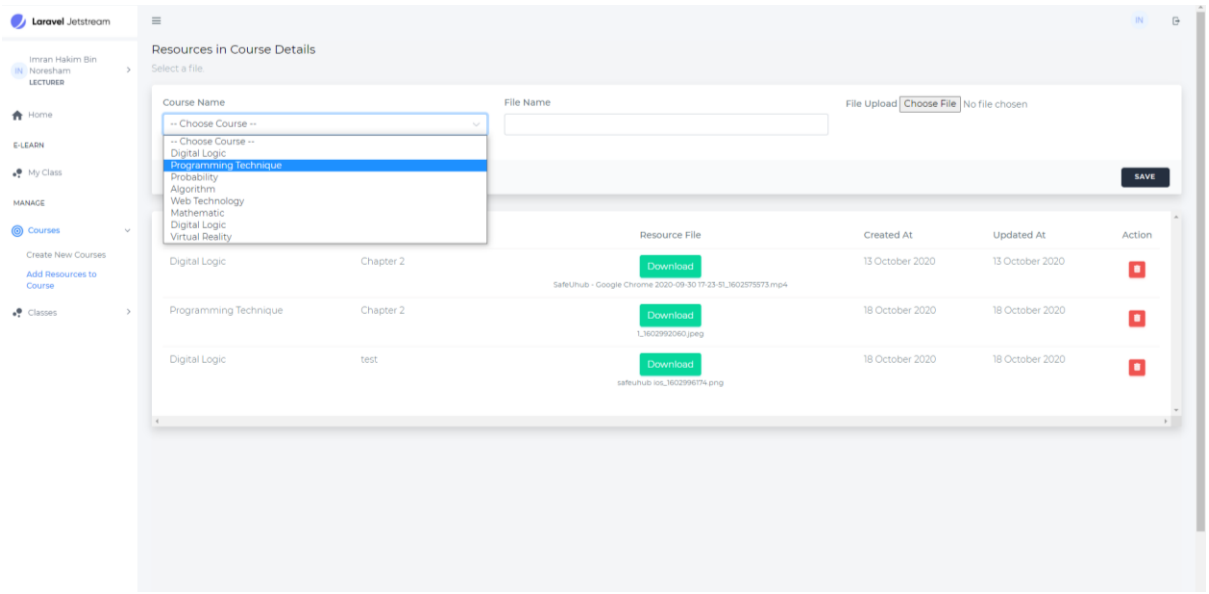
```
public function render()
    {
        if (auth()->user()->role == "admin") {
            $coursefiles = CourseFile::all();
        } else if(auth()->user()->role == "lecturer"){
            $coursefiles = CourseFile::whereHas('course' , function ($query) {
                $query->where('id_lecturer' , auth()->user()->id);
            })->get();
        }

        return view('livewire.course-file.course-file-wire')->with(compact('coursefiles'));

        // return view('livewire.course-file.course-file-wire');
    }
```

Add the resources

Other things that we need to inform is when lecturer want to add resource to course. If administration, they can add resource to any course exist in the system. But for lecturer they can add resource for their course file only. So, we need to filter some code. Go to "**CourseFileFormWire .php**" and follow like below code.

```
public function render()
    {
        if (auth()->user()->role == "admin") {
            $courses = Course::all();
        } else if(auth()->user()->role == "lecturer") {
            $courses = Course::where('id_lecturer',auth()->user()->id)->get();
        }

        return view('livewire.course-file.course-file-form-wire')->with(compact('courses'));

        // return view('livewire.course-file.course-file-form-wire');
    }
```

Login as lecturer and go to **Add Resource to Course** page. You will see that the dropdown to select course already filter to course to lecturer that currently login only.



Delete the resources

No customization needs when delete course file for lecture. They will follow the same process as admin.

Class Module

Module Outcome – At the end of this module, you can develop the Class Module for lecturer use. This module will include below function:

i) Show information of current access class
ii) Create new class
iii) Add member to class
iv) Edit the class
v) Delete the class

For class module we used the Laravel Jetstream package that include the team management function. We just change and modify the team name to class and we`re good to go.

Show information of current access class

To have my class page, we need to have the controller and view component.

1) For Controller, open the "**elearning-source**" folder and navigate to **Controller** folder. Copy the **"MyClassWire.php"** file and paste it to "**laragon\www\elearning\app\Http\Livewire**" (project folder)**.** This will be the logical code for this module.

2) **For View,** open back "**elearning-source**" folder and navigate to **View** folder. Copy the **"my-class-wire.blade.php"** folder and paste it to "**laragon\www\elearning\resources\views\livewire**" (project folder). This will be the view for this submodule.

Explanation: We already provide the Controller and View file in the "**elearning-source**" folder to make things easier for you to develop. The original way of generating the component is through artisan console command in terminal. The artisan command to generate the component is '**php artisan make:livewire MyClassWire**'. This will create the Controller file inside "**app\Http\Livewire\MyClassWire.php**" and View file inside "**resources\views\livewire\my-class-wire.php**".

Route

3) Go to "web.php" and uncomment below code. This is route we use to navigate to MyClass page. The **url "/myclass"** will point to **"MyClassWire.php"** class where the "**function render()**" inside "**MyClassWire.php**" class will be run to return the user to the view (V).

```
Route::get('/myclass',  MyClassWire::class);
```

And you need to import the class so the route can access to the right class that you have assigned.

```
use App\Http\Livewire\MyClassWire;
```

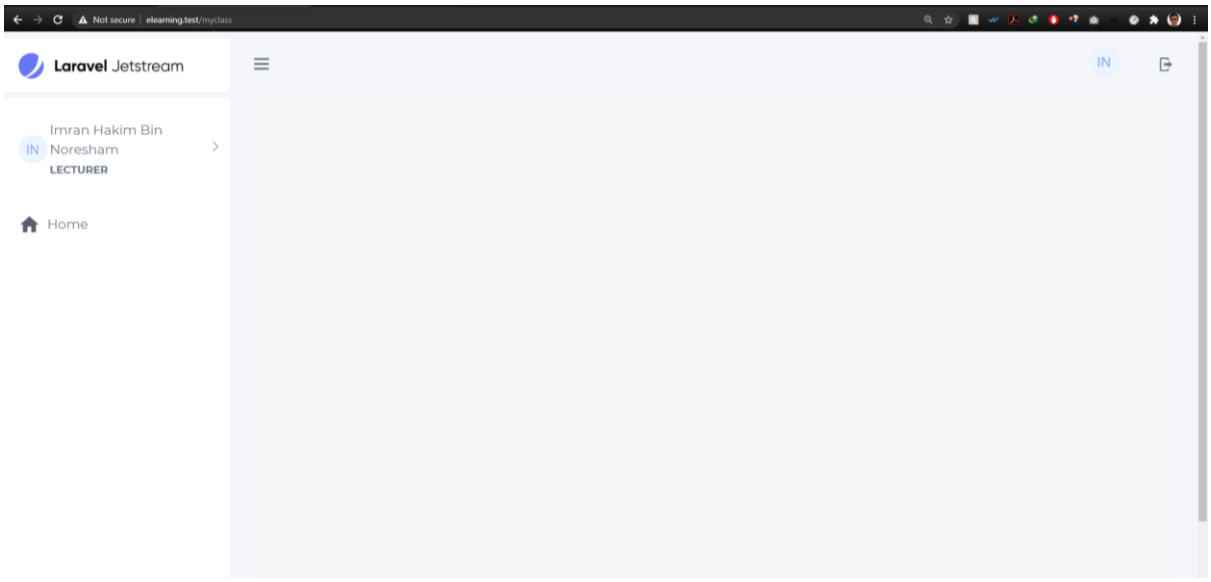
The "**web.php**" should look like below after you uncomment the above code.

```
<?php

use Illuminate\Support\Facades\Route;

use App\Http\Livewire\User\UserWire;

use App\Http\Livewire\Course\CourseWire;
use App\Http\Livewire\CourseFile\CourseFileWire;
```

```php
use App\Http\Livewire\ClassRoom\ClassRoomWire;
use App\Http\Livewire\ClassCourse\ClassCourseWire;

use App\Http\Livewire\MyClassWire;


/*
|--------------------------------------------------------------------------
| Web Routes
|--------------------------------------------------------------------------
|
| Here is where you can register web routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| contains the "web" middleware group. Now create something great!
|
*/

Route::get('/', function () {
    return redirect('login');
});


////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////
//Laravel Livewire
////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////
Route::middleware('auth')->group(function () {

    Route::get('/user', UserWire::class);

    Route::get('/course',   CourseWire::class);
    Route::get('/coursefile',   CourseFileWire::class);

    Route::get('/class',    ClassRoomWire::class);
    Route::get('/classcourse',  ClassCourseWire::class);

    Route::get('/myclass',  MyClassWire::class) ;

});
////////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////

Route::middleware(['auth:sanctum', 'verified'])->get('/dashboard', function () {
    return view('dashboard');
})->name('dashboard');
```

4) Now open the system and try insert the **url "/myclass"** after your local domain. Example: **elearning.test/myclass.** This will bring you to the empty my class page.

Explanation: The route pointing to the "**MyClassWire.php**" class where inside the class the "**function render()**" is run and the function return a view "**my-class-wire.blade.php**". Inside that view you will find the view code to generate below page.

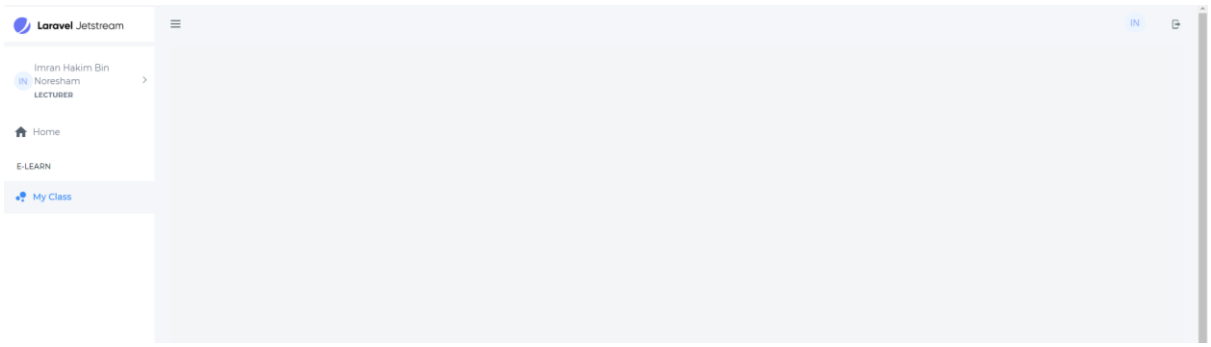


5) By using **Ctrl+p** search for "**app.blade.php**" and open it. We need to have My Class menu at the sidebar. To have those menu, uncomment the My Class menu under lecturer section. This will display the My Class menu at system sidebar. We use <a> tag as a link to the My Class page when user click on the My Class menu.

```
{{-------- START SECTION - LECTURER --------}}
@if (auth()->user()->role == 'lecturer')
    <li class="nav-small-cap">E-LEARN</li>
    <li> <a class="waves-effect waves-dark" href="{{URL::to('myclass')}}" aria-
expanded="false"><i class="mdi mdi-chart-bubble"></i><span class="hide-menu">My Class</span></a>

    </li>

    <li class="nav-small-cap">MANAGE</li>
    <li> <a class="has-arrow waves-effect waves-dark" href="#" aria-
expanded="false"><i class="mdi mdi-bullseye"></i><span class="hide-menu">Courses </span></a>
        <ul aria-expanded="false" class="collapse">


            <li><a href="{{URL::to('course')}}">Create New Courses</a></li>


            <li><a href="{{URL::to('coursefile')}}">Add Resources to Course</a></li>

        </ul>
    </li>

    {{-- <li> <a class="has-arrow waves-effect waves-dark" href="#" aria-
expanded="false"><i class="mdi mdi-chart-bubble"></i><span class="hide-menu">Classes </span></a>
        <ul aria-expanded="false" class="collapse">


            <li><a href="{{URL::to('teams/create')}}">Create New Class</a></li>

            @if (Auth::user()->current_team_id)
```

```
                <li><a href={{ route('teams.show', Auth::user()->currentTeam-
>id) }}>Class Setting</a></li>
            @endif
        </ul>
    </li> --}}

@endif
{{-------- END SECTION - LECTURER --------}}
```

6) Now open back the system and you can see My Class menu has been display at the sidebar but with an empty content.



7) To have the content, go to “MyClassWire.php” and follow like below.

```php
<?php

namespace App\Http\Livewire;

use Livewire\Component;
use App\Models\Course;
use App\Models\ClassCourse;

class MyClassWire extends Component
{
    public $course;
    public $action;

    public function selectItem($modelid , $action)
    {
        $this->course = Course::find($modelid);

        $this->action = $action;

        if($action == "showCourseContent")
        {
            $this->dispatchBrowserEvent('openModal_showCourseContent');
        }

    }
     public function render(){

        if (auth()->user()->currentTeam) {
            $classcourses = ClassCourse::where('id_class' , auth()->user()->currentTeam->id)->get();
            return view('livewire.my-class-wire')->with(compact('classcourses'));
        }
        else {
            $classcourses = [];
            return view('livewire.my-class-wire')->with(compact('classcourses'));
        }
```

```
        // return view('livewire.my-class-wire');
    }
```

8) After that, go to "**my-class-wire.blade.php**" and uncomment all the code inside "**SECTION - MY CLASS**".

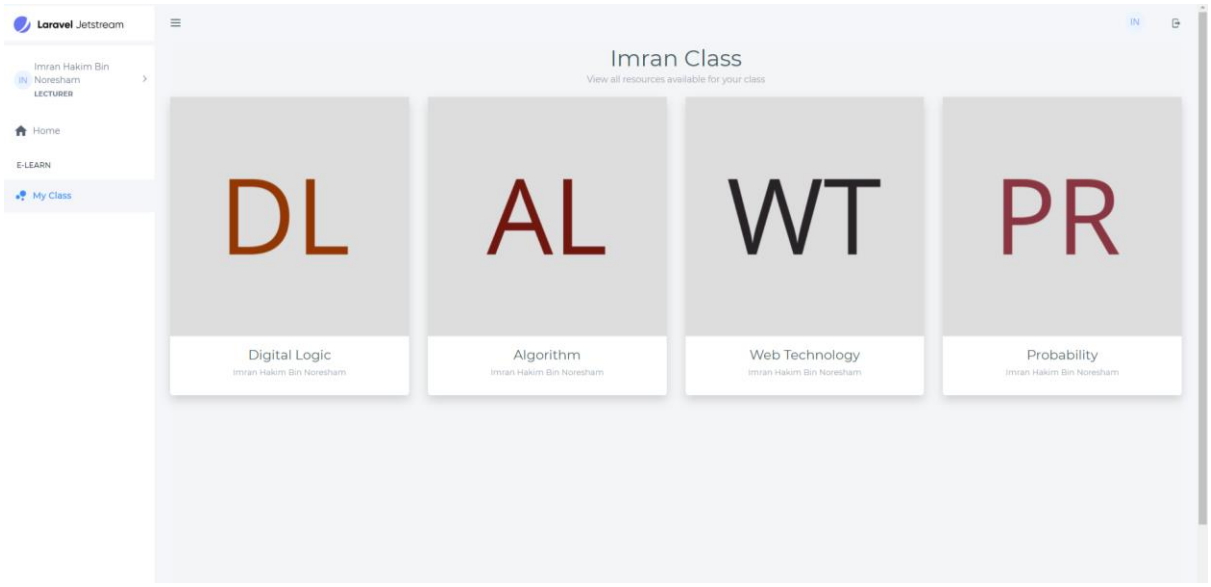
```
{{-- START SECTION - MY CLASS --}}
@if (auth()->user()->currentClass)

    <div class="row el-element-overlay">
        <div class="col-md-12 text-center">
            <h1 class="card-title">{{auth()->user()->currentClass ? auth()->user()->currentClass-
>name : 'Not In Any Class'}}</h1>
            <h6 class="card-subtitle m-b-20 text-
muted">View all resources available for your class</h6>
        </div>

        @foreach ($classcourses as $classcourse)
            <div class="col-lg-3 col-md-6" style="padding-bottom: 30px;">
                <a wire:click="selectItem({{$classcourse->course->id}} , 'showCourseContent')">
                <div class="card zoom2" style="box-
shadow: 0 .5rem 1rem rgba(0,0,0,.15)!important;border-radius: 5px;">
                    <div class="el-card-item">

                            <div class="el-card-avatar el-overlay-1">
                                <img src="https://ui-avatars.com/api/?size=512&name={{$classcourse-
>course ? $classcourse->course-
>name : 'Undefined'}}&color={{mt_rand( 0, 0xFFFFF )}}" alt="user" style="cursor: pointer;"/>

                            </div>

                        <div class="el-card-content" style="cursor: pointer;">
                        <h3 class="box-title">{{$classcourse->course ? $classcourse->course-
>name : 'Undefined'}}</h3>
                        <small class="text-muted">{{$classcourse->course ? ($classcourse->course-
>lecturer ? $classcourse->course->lecturer->name : 'undefined') : 'undefined'}}</small>
                        <br/> </div>
                    </div>
                </div>
            </a>
            </div>
        @endforeach
    </div>

@else

    <!-- Tell user if no data in My Class -->
    <table style="width:100%;height:80vh">
        <tr>
            <td>
```

```
                <div class="row">
                    <div class="jumbotron jumbotron-fluid" style=" margin: auto;width: 50%;">
                        <div class="container">
                            <h2 class="display-3 text-center">You`re currently not in any class.</h2>
                            <p class="lead">This is a modified jumbotron that occupies the entire hori
zontal space of its parent.</p>
                        </div>
                    </div>
                </div>
            </td>
        </tr>
    </table>

@endif


<!-- Modal - Show My Class Data When User Click on Subject -->
<div id="showCourseContentModal" class="modal fade" tabindex="-1" role="dialog" aria-
labelledby="showCourseContentModal" aria-hidden="true" style="display: none;">
    <div class="modal-dialog modal-xl">
        <div class="modal-content">
            <div class="modal-header text-center">
                <table style="border:none;width:100%">
                    <tr>
                        <td style="border:none;width:95%"><h3 class="modal-title font-weight-
bold" id="myLargeModalLabel">{{$course ? $course->name : ''}}</h3></td>
                        <td style="border:none;width:5%"><button type="button" class="close" data-
dismiss="modal" aria-hidden="true">×</button></td>
                    </tr>
                </table>
            </div>
            <div class="modal-body">
                <div class="row">
                    <div class="col-md-12">
                        <div class="card" style="box-
shadow: 0 .5rem 1rem rgba(0,0,0,.15)!important;border-radius: 5px;">
                            <div class="card-body" style="overflow:scroll">
                                <table class="table table-hover" >
                                    <thead>
                                        <tr>
                                            <th>Resource Name</th>
                                            <th class="text-center">Resource File</th>
                                            <th>Created At</th>
                                            <th>Updated At</th>
                                        </tr>
                                    </thead>
                                    @if ($course)
                                        @foreach ($course->coursefile as $coursefile)
```

```
                                    <tbody>
                                        <tr>
                                            <td>{{$coursefile->name}}</td>
                                            <td class="text-center">
                                                @if ($coursefile->file_type == "360file")
                                                    <a href="{{URL::to('/video_360/'.$coursefile-
>id.'')}}" target="_blank" class="btn btn-success">View 360° Video</a>
                                                @elseif ($coursefile->file_type == "3dfile")
                                                    <a href="{{URL::to('/3d_model_view/'.$coursefi
le->id.'')}}" target="_blank" class="btn btn-success">View in VR</a>
                                                    <a href="{{URL::to('/ar_view/'.$coursefile-
>id.'')}}" target="_blank" class="btn btn-success">View in AR</a>
                                                @else
                                                    <a href="{{URL::to(''.$coursefile-
>file_path.'')}}" target="_blank" class="btn btn-success">Download/View File</a>
                                                @endif

                                                <br>
                                                <small>{{$coursefile->file_name}}</small>
                                            </td>
                                            <td>{{date('j F Y', strtotime($coursefile-
>created_at))}}</td>
                                            <td>{{date('j F Y', strtotime($coursefile-
>updated_at))}}</td>

                                        </tr>
                                    </tbody>
                                    @endforeach
                                @endif
                            </table>
                        </div>
                    </div>
                </div>
            </div>
            <div class="modal-footer">
                <button type="button" class="btn btn-danger waves-effect text-left" data-
dismiss="modal">Close</button>
            </div>
        </div>
        <!-- /.modal-content -->
    </div>
    <!-- /.modal-dialog -->
</div>
<!-- /.modal -->


<!-- Loading Animation When Click on Subject -->
```

```
<div wire:loading style="position: absolute;top: 0%;width: 100%;height: 100%;z-index: 9999;background-color:#d2d6dc66">
    <table style="width:100%;height:100%">
        <tr>
            <td style="text-align:center;vertical-align:middle">
                <i class="mdi mdi-loading mdi-spin mdi-48px"></i>
            </td>
        </tr>
    </table>
</div>

@push('scripts')

<script>
  document.addEventListener('livewire:load', function () {

        window.addEventListener('openModal_showCourseContent', event => {
            $('#showCourseContentModal').modal('show')
        })

  })
</script>

@endpush --}}
{{-- END SECTION - MY CLASS --}}
```

9) Now open the system and you will see the content. This content will only show for the current class that the lecturer access. To see for other class, lecturer need to switch their class at the topbar. If there is no data, the page will inform to user that there is no learning material for that class.



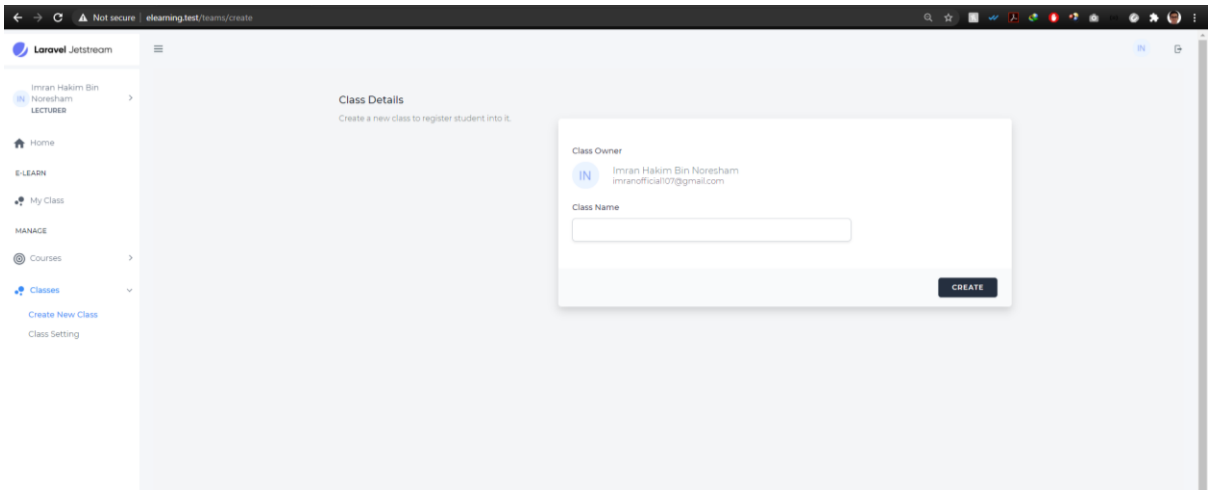
Create New Class

We must first have access to class menu for lecturer. By using **Ctrl+p** search for "**app.blade.php**" and open it. We need to have Class menu at the sidebar. To have those menu, uncomment the Class menu under lecturer section. This will display the Class menu at system sidebar. We use <a> tag as a link to the Class page when user click on the Class menu.

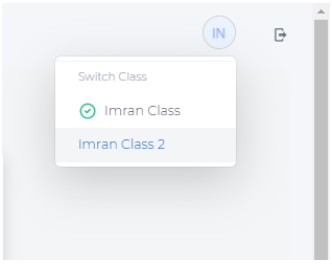
```
{{-------- START SECTION - LECTURER --------}}
@if (auth()->user()->role == 'lecturer')
    <li class="nav-small-cap">E-LEARN</li>
    <li> <a class="waves-effect waves-dark" href="{{URL::to('myclass')}}" aria-
expanded="false"><i class="mdi mdi-chart-bubble"></i><span class="hide-menu">My Class</span></a>

    </li>

    <li class="nav-small-cap">MANAGE</li>
    <li> <a class="has-arrow waves-effect waves-dark" href="#" aria-
expanded="false"><i class="mdi mdi-bullseye"></i><span class="hide-menu">Courses </span></a>
        <ul aria-expanded="false" class="collapse">


            <li><a href="{{URL::to('course')}}">Create New Courses</a></li>


            <li><a href="{{URL::to('coursefile')}}">Add Resources to Course</a></li>

        </ul>
    </li>

    <li> <a class="has-arrow waves-effect waves-dark" href="#" aria-
expanded="false"><i class="mdi mdi-chart-bubble"></i><span class="hide-menu">Classes </span></a>
```

```
        <ul aria-expanded="false" class="collapse">


            <li><a href="{{URL::to('teams/create')}}">Create New Class</a></li>

            @if (Auth::user()->current_team_id)
                <li><a href={{ route('teams.show', Auth::user()->currentTeam-
>id) }}>Class Setting</a></li>
            @endif
        </ul>
    </li>

@endif
{{-------- END SECTION - LECTURER --------}}
```

Go to "**Create New Class**" page and you will see the page is different from administrator page. This is because we use the Laravel Jetstream package that already come with the team management function and for lecturer they only can create class for themselves.

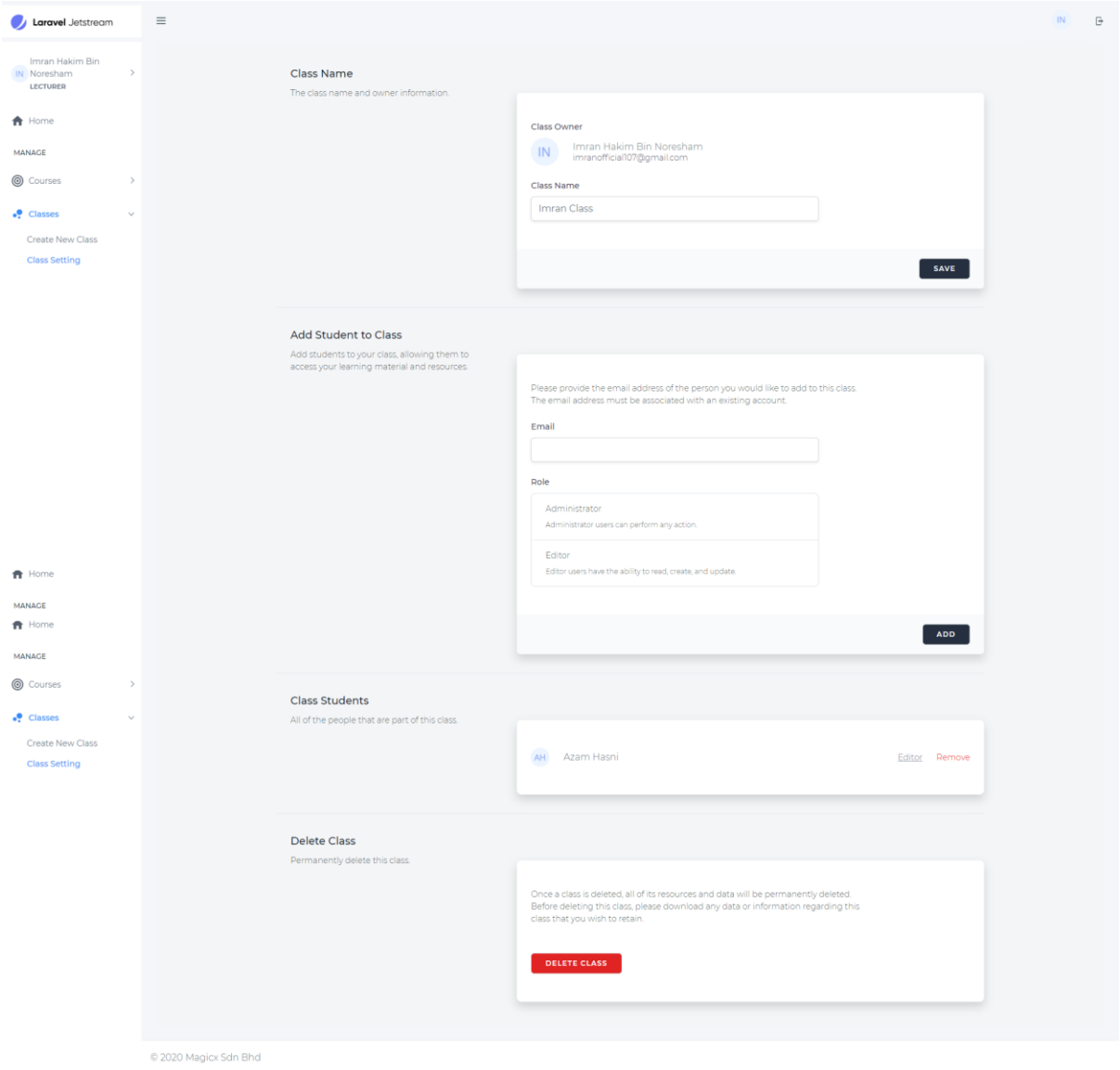


If lecturer have many classes, they can switch between the class from the top bar menu. And the system will only show for the current class that they access.



Add member to class, Edit the class & Delete the class

To manage class, go to “Class Setting” page. This page is also provided by Laravel Jetstream and we just change the name of team to class. This is where lecturer can edit their class, add student to class and delete the class.



Role: Student

User Module

Student will not have privilege to access user module. Only administrator have this privilege. So, lets skip this module and go to next module.

Course Module

Student will not have privilege to access course module. Only administrator and lecturer have this privilege. So, lets skip this module and go to next module.

Class Module

For class module, student only have access to My Class page as they cannot manage or create any class. That privilege only applicable to administrator and lecturer.

To have this menu go to "**app.blade.php**" and uncomment the section for student like below.

```
{{-------- START SECTION - STUDENT --------}}
@if (auth()->user()->role == 'student')
    <li class="nav-small-cap">E-LEARN</li>
    <li> <a class="waves-effect waves-dark" href="{{URL::to('myclass')}}" aria-
expanded="false"><i class="mdi mdi-chart-bubble"></i><span class="hide-menu">My Class</span></a>

    </li>
@endif
{{-------- END SECTION - STUDENT --------}}
```

Login as student and go to My Class page. You will see the same view as lecturer view for my class. If student is assigned to many classes by their lecturer. They can switch between the class to see the learning material at my class page.

HOW TO SETUP VIRTUAL REALITY (VR) SCENE?

Introduction

What is VR?

Virtual reality (VR) is a technology that uses head-mounted headsets with displays to generate the realistic images, sounds, and other sensations to put users into an immersive virtual environment. VR allows us to create unbounded worlds that people can walk around and interact with using their hands, to feel as if they were transported to another place.

What is WebXR?

WebXR is a JavaScript API for creating immersive 3D, VR experiences, and augmented reality (AR) in your browser. Or simply put, allows VR and AR in the browser over the Web. The XR stands for "Extended Reality".

What is A-Frame?

A-Frame is an easy and powerful web framework for building VR experiences. A-Frame is based on top of HTML, making it simple to get started. But A-Frame is not just a 3D scene graph or a markup language; the core is a powerful entity-component framework that provides a declarative, extensible, and composable structure to three.js.

A-Frame supports most mobile phones and VR headsets such as Vive, Rift, Windows Mixed Reality, Daydream, GearVR, Cardboard, Oculus Go, and can even be used for AR. Although A-Frame supports the whole spectrum, A-Frame aims to define fully immersive interactive VR experiences that go beyond basic 360° content, making full use of positional tracking and controllers.

Stage 1 – HTML & primitives

Installation of A-Frame using CDN

To include A-Frame in an HTML file in the local server, you can drop a `<script>` tag pointing to the CDN build:

```
<head>
    ...
    <script src="https://aframe.io/releases/1.0.4/aframe.min.js"></script>
</head>
```

And that is it!

Basic A-Frame primitives

1) Let us start by building a basic A-Frame scene. This is a HTML starter structure:

```
<html>
    <head>
        <script src="https://aframe.io/releases/1.0.4/aframe.min.js"></script>
    </head>
    <body>
        <a-scene>
        </a-scene>
    </body>
</html>
```

2) The `<a-scene>` will contain every entity in our scene, such as camera, lights, 3D models, etc.
3) Within the `<a-scene>`, we attach 3D entities using one of A-Frame's standard primitives `<a-box>`. We can use `<a-box>` just like a normal HTML element, defining the tag and using HTML attributes to customize it. Some other examples of primitives that come with A-Frame include `<a-cylinder>`, `<a-plane>`, or `<a-sphere>`.

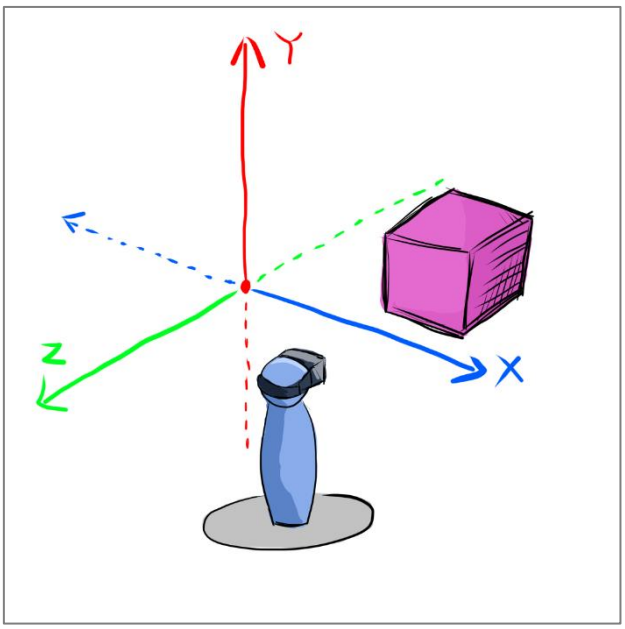
```
<a-scene>
    <a-box color="red"></a-box>
</a-scene>
```

4) Besides that, primitives are A-Frame's easy-to-use HTML elements that wrap the underlying entity-component assembly. They can be convenient, but underneath `<a-box>` is `<a-entity>` with the geometry and material components:

```
<a-entity id="box" geometry="primitive: box" material="color: red"></a-entity>
```

Transformation: position, rotation, scale

1) The transformation tools help us to move the entities, rotate them and change their size. The 3D space is illustrated as follows:



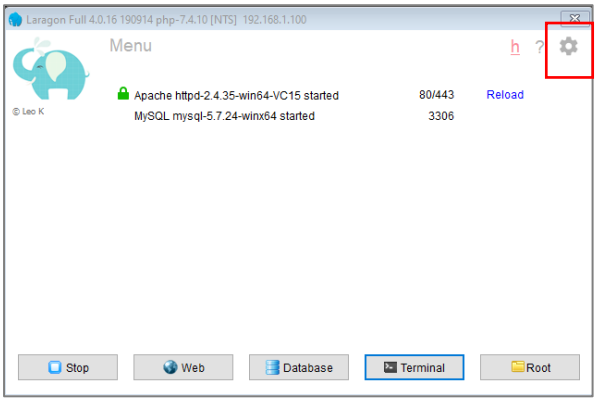
Note: The position of the object in 3D scene is defined in the XYZ-axes.

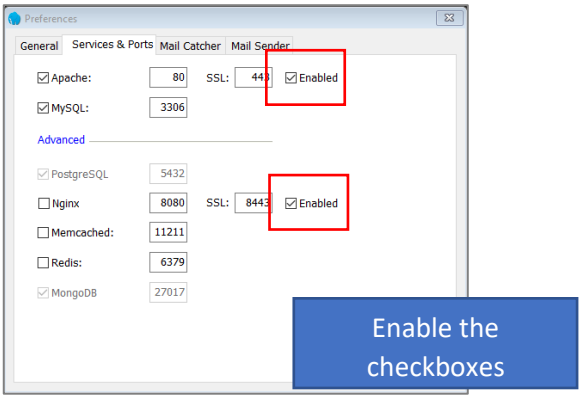
2) We can move the box back *5 meters* on the negative Z-axis with the position component. We also move it up *2 meters* on the positive Y-axis. Next, the box is rotate at an angle and double its size:

```
<a-entity id="box" geometry="primitive: box" material="color: red"
position="0 2 -5" rotation="0 45 45" scale="2 2 2"></a-entity>
```

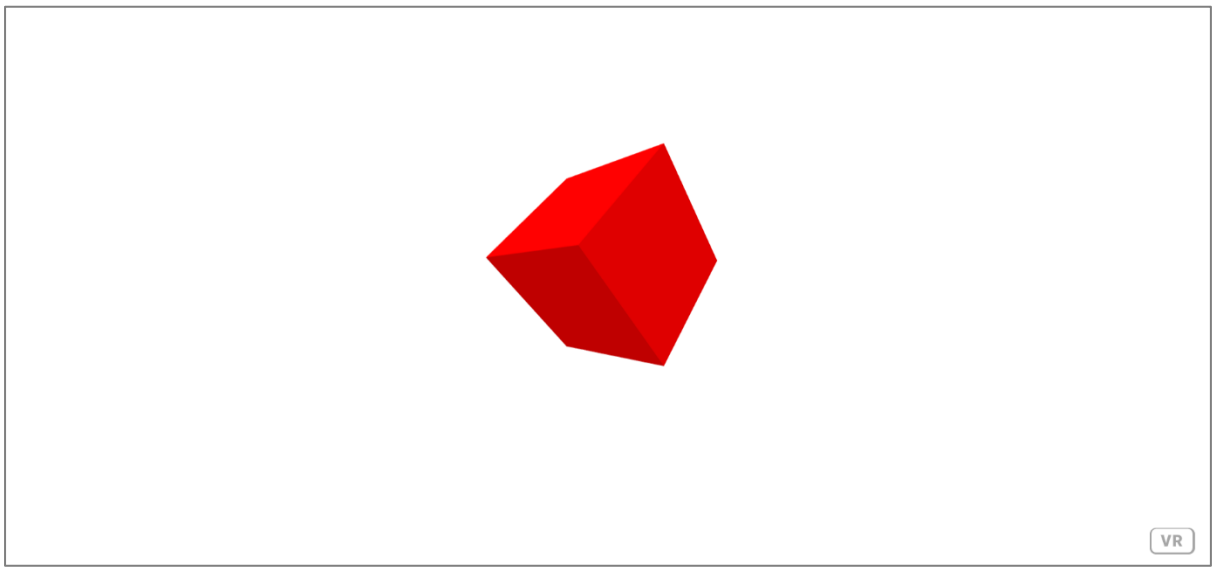
Note: `position` component moves the entity; `rotation` component rotates the entities; `scale` component changes the size of the entities.

3) To run this HTML file, make sure you are accessing the site over HTTPS to enable enter VR mode and grant access to the device sensors. These are the steps to enable HTTPS by using Laragon.





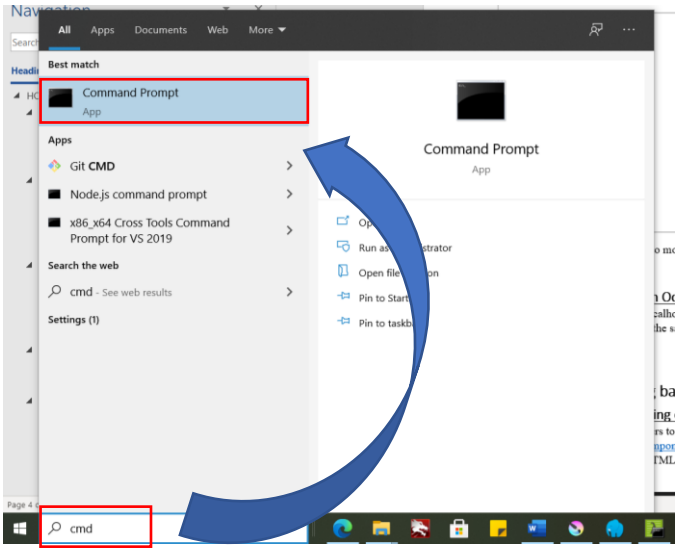
4) Type in the address bar in your browser: "**https://<your-project-name>.test**". Now we can see the box!



You may drag around the screen to move the camera view.

Extra: View the VR scene in Oculus Quest and mobile phone

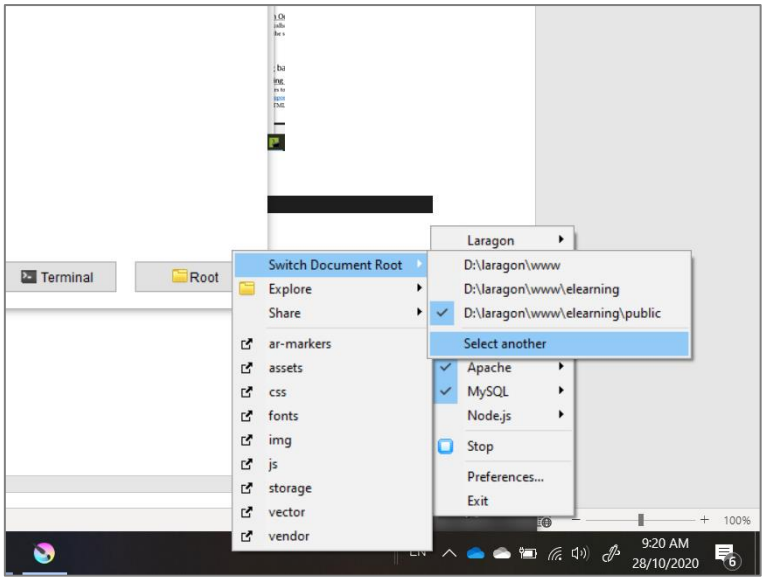
1) To open the website in localhost by using Oculus Quest or mobile phone, the laptop (localhost) and Oculus Quest need to connect to the same network.
2) Then, type in "cmd" in your Windows search bar and open the command prompt:



3) Type in the command prompt:

```
ipconfig
```

4) Next, we need to add our path into Laragon's settings. Open Laragon at the bottom menu bar by right click and select the second option (the name will change depend on your root folder). Then, select "Switch Document Root" and click "Select another" to open a file explorer.



5) Using the file explorer, search for the “elearning” > “public” folder.
6) Search for the “IPv4 Address”. Then, open the browser in Oculus Quest and type in the address: **https://<IPv4 Address>/**

Stage 2 – Building basic scene

Using external library: adding environment

1) A-Frame allows developers to create and share reusable components for others to easily use. @feiss‘s environment component procedurally generates a variety of entire environments for us with a single line of HTML. To include the environment component, we add a script tag after A-Frame:

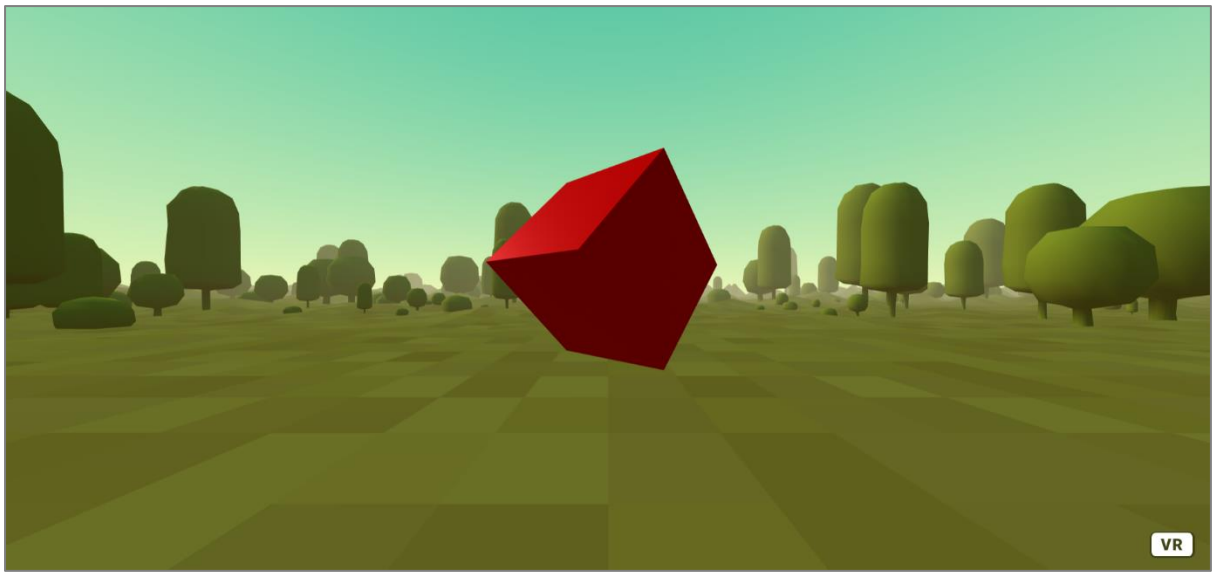
```
<head>
    <script src="https://aframe.io/releases/1.0.4/aframe.min.js"></script>
    <script src="https://unpkg.com/aframe-environment-component/dist/aframe-
environment-component.min.js"></script>
</head>
```

2) Then within the scene, add an entity with the environment component attached. We can specify a preset (e.g., forest) with along many other parameters (e.g., 200 trees):

```
<a-scene>
    <a-entity id="box" geometry="primitive: box" material="color: red"
position="0 2 -5" rotation="0 45 45" scale="2 2 2"></a-entity>

    <a-entity environment="preset: forest; dressingAmount: 500"></a-entity>
</a-scene>
```

You can try with these preset values: none, default, contact, egypt, checkerboard, forest, goaland, yavapai, goldmine, threetowers, poison, arches, tron, japan, dream, volcano, starry, osiris

3) The result:



Adding audio

1) Now, we would like to add a sound effect to the scene.
2) A-Frame provides a nice asset management system for performance. The asset management system makes it easier for the browser to cache assets (e.g., images, videos, models) and A-Frame will make sure all the assets are fetched before rendering.
3) To use the asset management system for an audio:

```
<a-scene>
    <a-assets>
        <audio id="backgroundnoise" src=https://cdn.aframe.io/basic-guide/audio/backgroundnoise.wav autoplay preload></audio>

    </a-assets>

    <!-- ... -->
    <a-sound src="#backgroundnoise"></a-sound>
</a-scene>
```

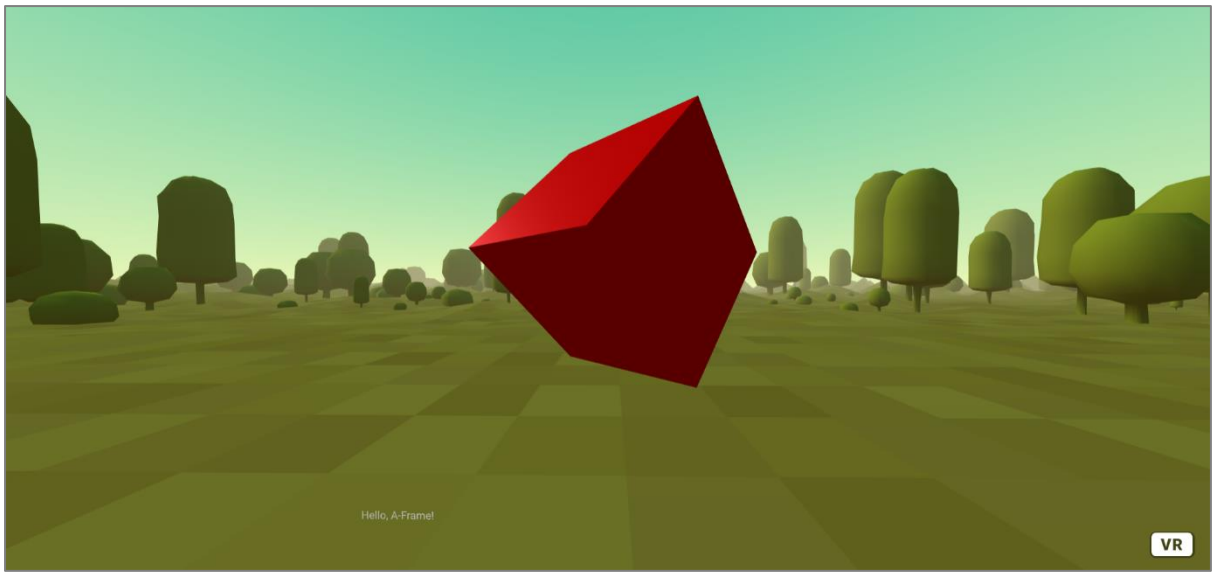
Note: Do not copy and paste the whole code. Just copy and paste the new lines in your text editor. The `<!-- ... -->` represents the other codes that you written before.

Adding text

1) A-Frame also comes with a text component:

```
<a-scene>

    <!-- ... -->
    <a-entity text="value: Hello, A-Frame!; color: #BBB" position="-0.9 0.2 -3" scale="1.5 1.5 1.5"></a-entity>
</a-scene>
```

2) The result:



Import 3D model

1) A-Frame provides components for loading 3D models. We recommend using gltf or glb if possible as glb and glTF gain adoption as the standard for transmitting 3D models over the Web.
2) In this example, we will import a city model into the scene with animation.
3) Create a new folder named “assets” in your root folder. Then, create another folder “3d_models” inside the “assets” folder.
4) Next, copy the “VC.glb” 3D model into “3d_models” folder.
5) Use the asset management system to import the 3D model:

```
<a-scene>
    <a-assets>

        <!-- ... -->
        <a-asset-item id="cityModel" src="assets/3d_models/VC.glb"></a-asset-item>
    </a-assets>

    <!-- ... -->

    <a-entity id="gltfModel" gltf-model="#cityModel" position="0 0.1 0"></a-entity>

</a-scene>
```

Use the `gltf-model` component to link to the id of the 3D model asset.

6) Some author provides the animation of the 3D model. The animation-mixer component, part of aframe-extras by Don McCurdy, provides controls for playing animations in three.js (.json) and gltf models.
7) To use the animation-mixer component, we need to add the script tag in `<head>` first:

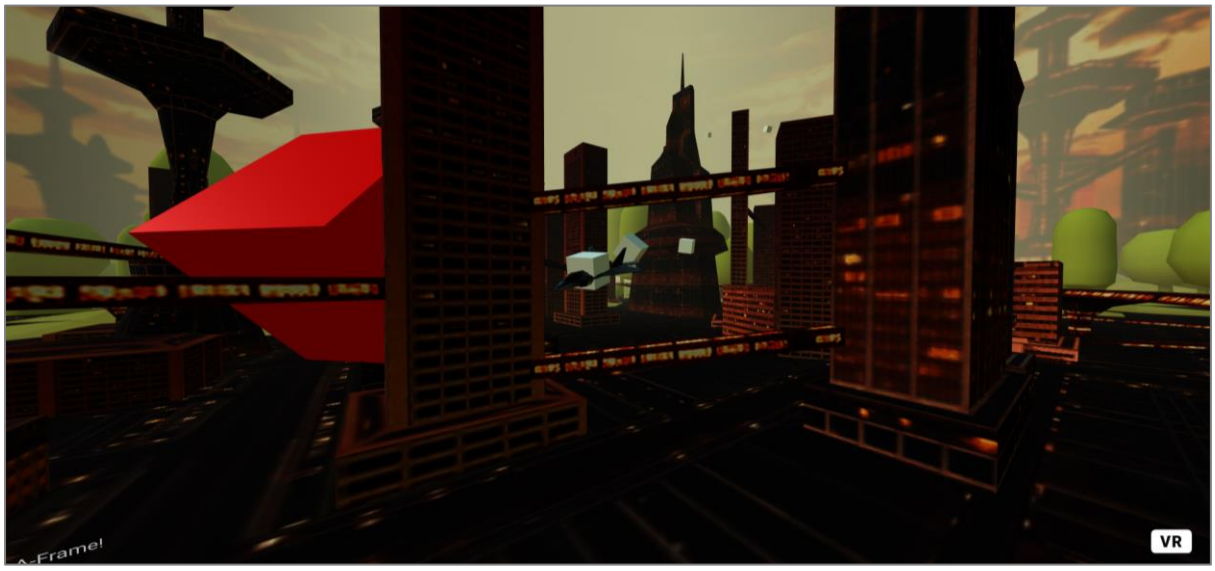
```
<head>
```

```
    <!-- ... -->
    <script src="https://unpkg.com/aframe-extras.animation-
mixer@3.4.0/dist/aframe-extras.animation-mixer.js"></script>
</head>
```

8) Then, just add `animation-mixer` in your 3D model entity:

```
<a-scene>

    <!-- ... -->
    <a-entity id="gltfModel" gltf-
model="#cityModel" position="0 0.1 0" animation-mixer></a-entity>

</a-scene>
```

9) Result (3D model with animation):



Stage 3 – User interaction

Gaze-based cursor pointer

1) Let's add interaction with the box: when we look at the box, we'll increase the size of the box.
2) Firstly, we will focus this section on using basic **mobile** and **desktop** inputs with the built-in cursor component. The cursor component by default provides the ability to "click" on entities by staring or gazing at them on mobile, or on desktop, looking at an entity and click the mouse.
3) To have a visible cursor fixed to the camera, we place the cursor as a child of the camera. Child entities inherit transformations (i.e., position, rotation, and scale) from their parent. Let's define `<a-camera>` that contain `<a-cursor>`:

```
<a-camera id="head" camera wasd-controls look-controls>
    <a-cursor id="cursor"></a-cursor>
</a-camera>
```

Note: By default, A-Frame automatically included an `<a-camera>` for us if we didn't specifically define a camera in the scene.

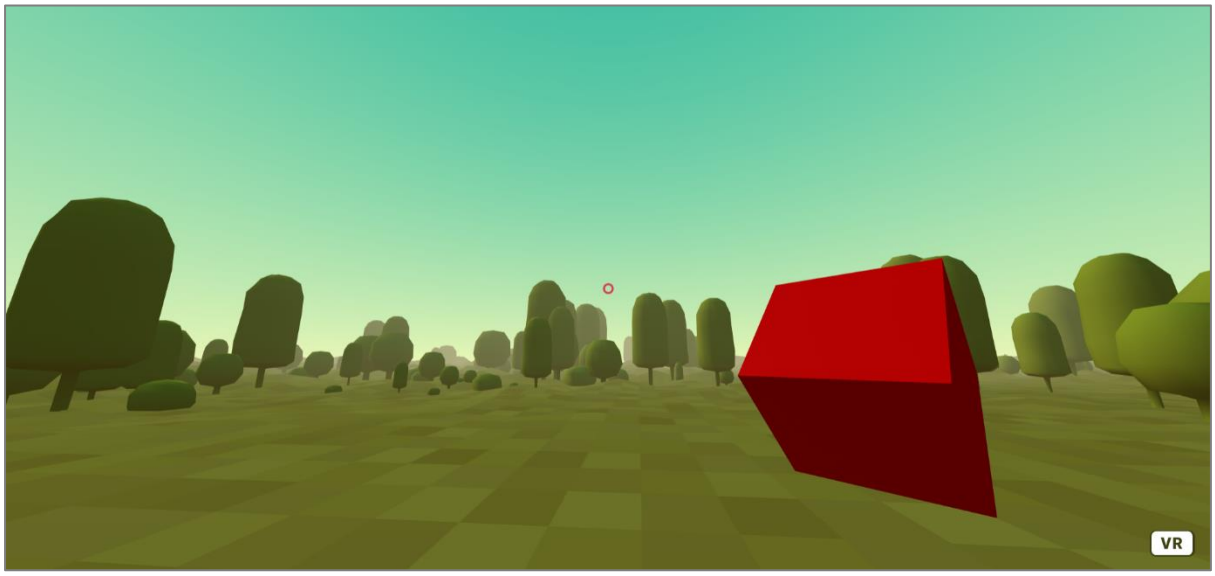
Note: The `wasd-controls` component allows the user to move around the scene using WASD keys; the `look-controls` component allows the user to look around the scene by dragging using the mouse (desktop) or move user's head (mobile VR and HMD).

4) We can also stylize our cursor. To add visual feedback to the cursor to show when the cursor is clicking or fusing, we can use the animation component. When the mouse is clicked, it will emit an event on the `<a-cursor>`:

```
<a-camera id="head" camera wasd-controls look-controls>
    <a-cursor id="cursor"
        animation__click="property: scale; startEvents: click; from: 0.1 0.1 0
.1; to: 1 1 1; dur: 150"
        animation__fusing="property: fusing; startEvents: fusing; from: 1 1 1;
 to: 0.1 0.1 0.1; dur: 1500"
        material="color: crimson; shader: flat"></a-cursor>
</a-camera>
```

We also change our cursor color into crimson by using `material` component.

5) The result:



JavaScript interaction using A-Frame component registration

1) We can use the A-Frame component registration by using JavaScript to handle the cursor events, such as click. Now, we want to scale the box when the cursor hovered over it. Use

the following code to register new `scale-on-mouseenter` component to the box entity when `mouseenter` event (cursor hover to the target box) is detected:

```
<head>

    <!-- ... -->
    <script>
        AFRAME.registerComponent('scale-on-mouseenter', {
            schema: {
                to: {default: '3 3 3', type: 'vec3'}
            },

            init: function () {
                var data = this.data;
                var el = this.el;
                this.el.addEventListener('mouseenter', function () {
                    el.object3D.scale.copy(data.to);
                });
            }
        });
    </script>
</head>

<body>
    <a-scene>

        <!-- ... -->
        <a-
entity id="box" geometry="primitive: box" material="color: red" position="3 2
-5" rotation="0 45 45" scale="2 2 2" scale-on-mouseenter></a-entity>

        <!-- ... -->
    </a-scene>
</body>
```

Note: The `scale-on-mouseenter` component is attached to the box entity.

2) Play the scene and hover your cursor to the box, notice that the box will be scaled up.

3) To change the size of box back to its origin when the cursor left the box, let's add a `mouseleave` event to the `scale-on-mouseenter` component registration:

```
<script>
        AFRAME.registerComponent('scale-on-mouseenter', {
            schema: {
                to: {default: '3 3 3', type: 'vec3'},
                origin: {default: '2 2 2', type: 'vec3'}
            },

            init: function () {

                /* ... */
                this.el.addEventListener('mouseleave', function() {
                    el.object3D.scale.copy(data.origin);
                })
            }
        });
    </script>
```

4) There are many cursor events we can explore and utilize to make our VR application more interactive.

Stage 4 – Integration with Laravel project

Create a simple scene to display a 3D model

1) We can integrate the VR application with our elearning Laravel project. Create a new folder named "3d_model_view" in your "resources" > "views" folder. Then, copy and paste your "index.html" into "3d_model_view" folder. Rename "index.html" into "index.blade.php".
2) Create a new route in "web.php" in "routes" folder:

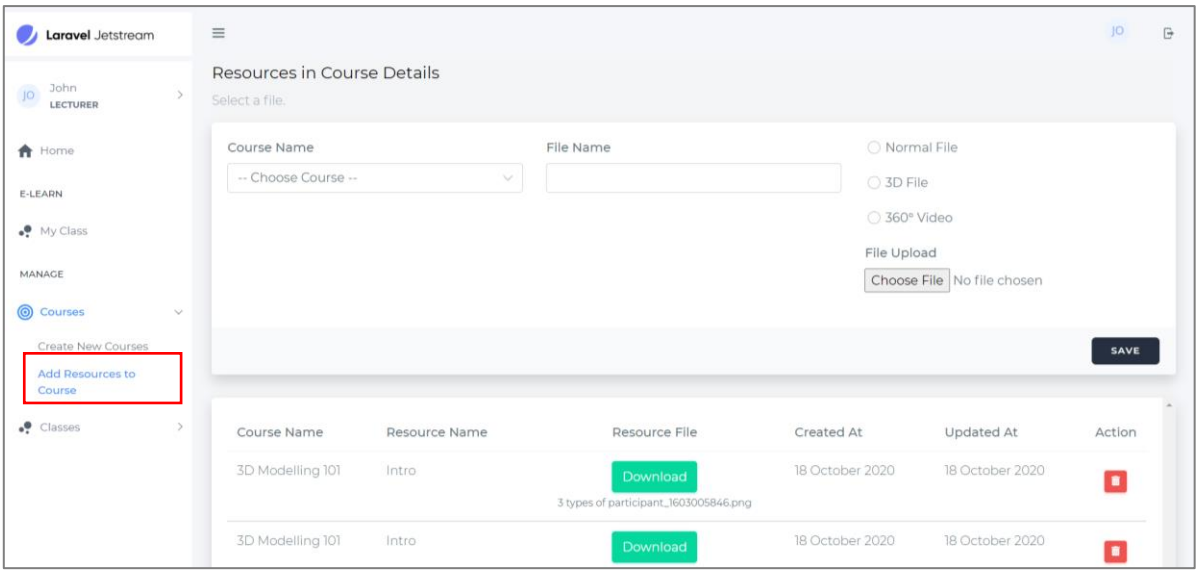
```
Route::get('/3d_model_view', function() {
        return view('3d_model_view.index');
})->name('3d_model_view');
```

3) Type in the address bar in your browser: "**https://<your-project-name>.test /3d_model_view**".

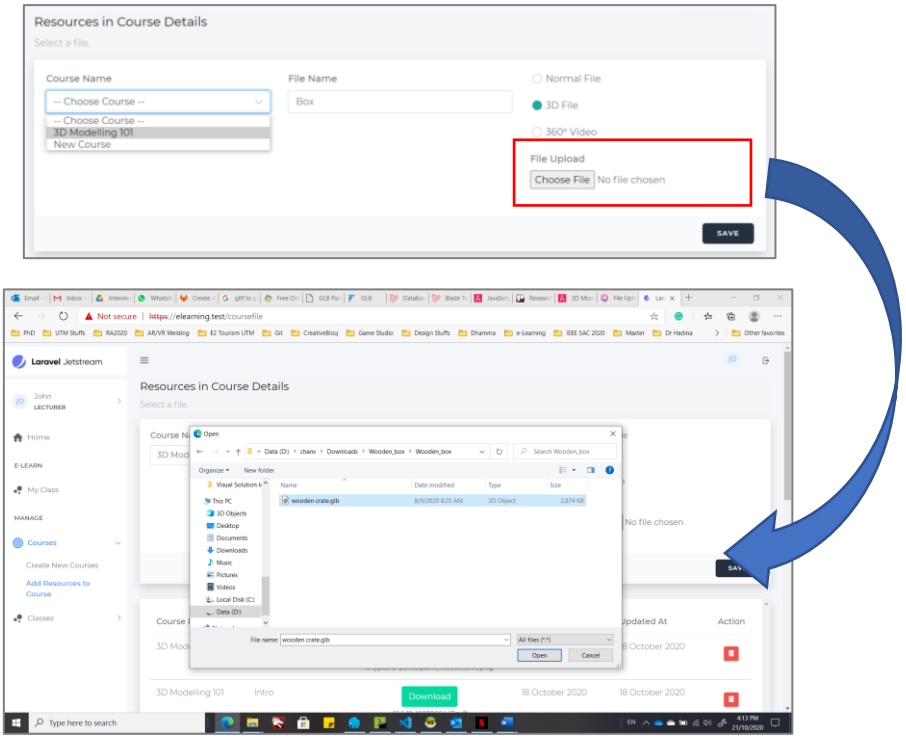
Upload the 3D model

1) Firstly, we need to make sure that we have the correct 3D model format. The elearning only accepts glb file format. These are the websites that can convert your 3D model files into glb:
   - Aspose.3D conversion: https://products.aspose.app/3d/conversion/gltf-to-glb

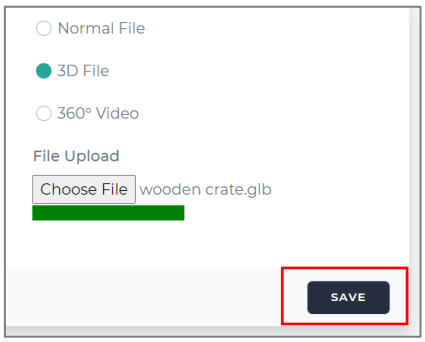
2) Go to “Add Resources to Courses” page.

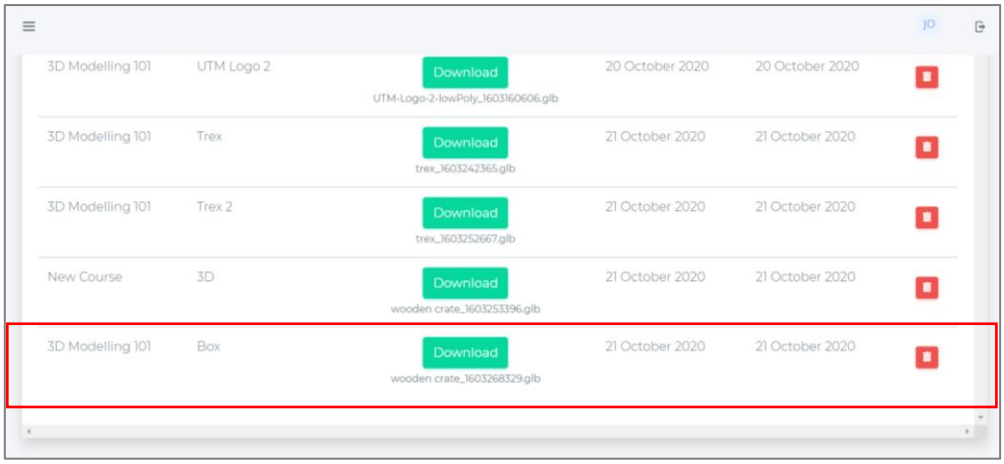


3) To assign the 3D model to a course, you need select a course, enter file name, choose the file type (3D file) and click on the “Choose File” button to search for the 3D model.



4) Wait for the upload to complete and click “SAVE” button to continue. You may scroll to find your uploaded 3D model.





Create function to display 3D model dynamically

1) Since our elearning project consists of the 3D model upload function, we can view the 3D model VR view in the elearning based on the model ID dynamically.
2) Update the “web.php” to accept the model ID and get the 3D model based on the ID:

```
Route::get('/3d_model_view/{modelId}', function($modelId) {
        $model = App\Models\CourseFile::find($modelId);

        return view('3d_model_view.index', [
            'modelSrc' => $model
        ]);
})->name('3d_model_view');
```

3) Update the asset source in the “index.blade.php” to link the 3D model to the chosen 3D model directory.

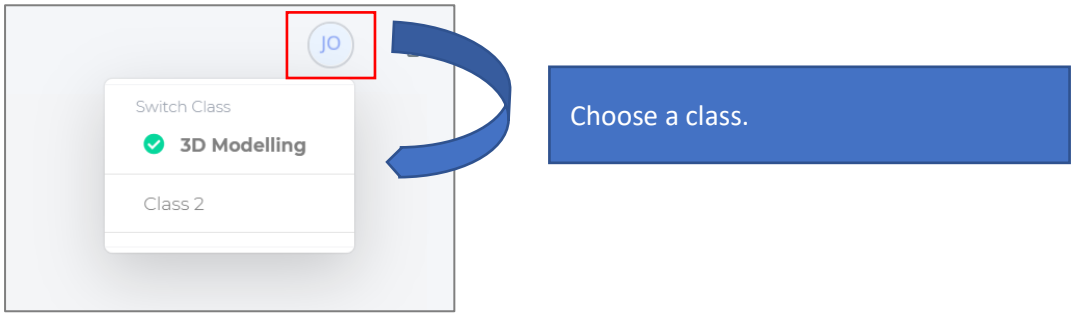
```
<a-assets>
    <a-asset-item id="cityModel"
        src="{{URL::to('').$modelSrc->file_path.'')}}"></a-asset-item>
</a-assets>
```

So, whenever the model ID changed, the scene updates the 3D model according to the ID.

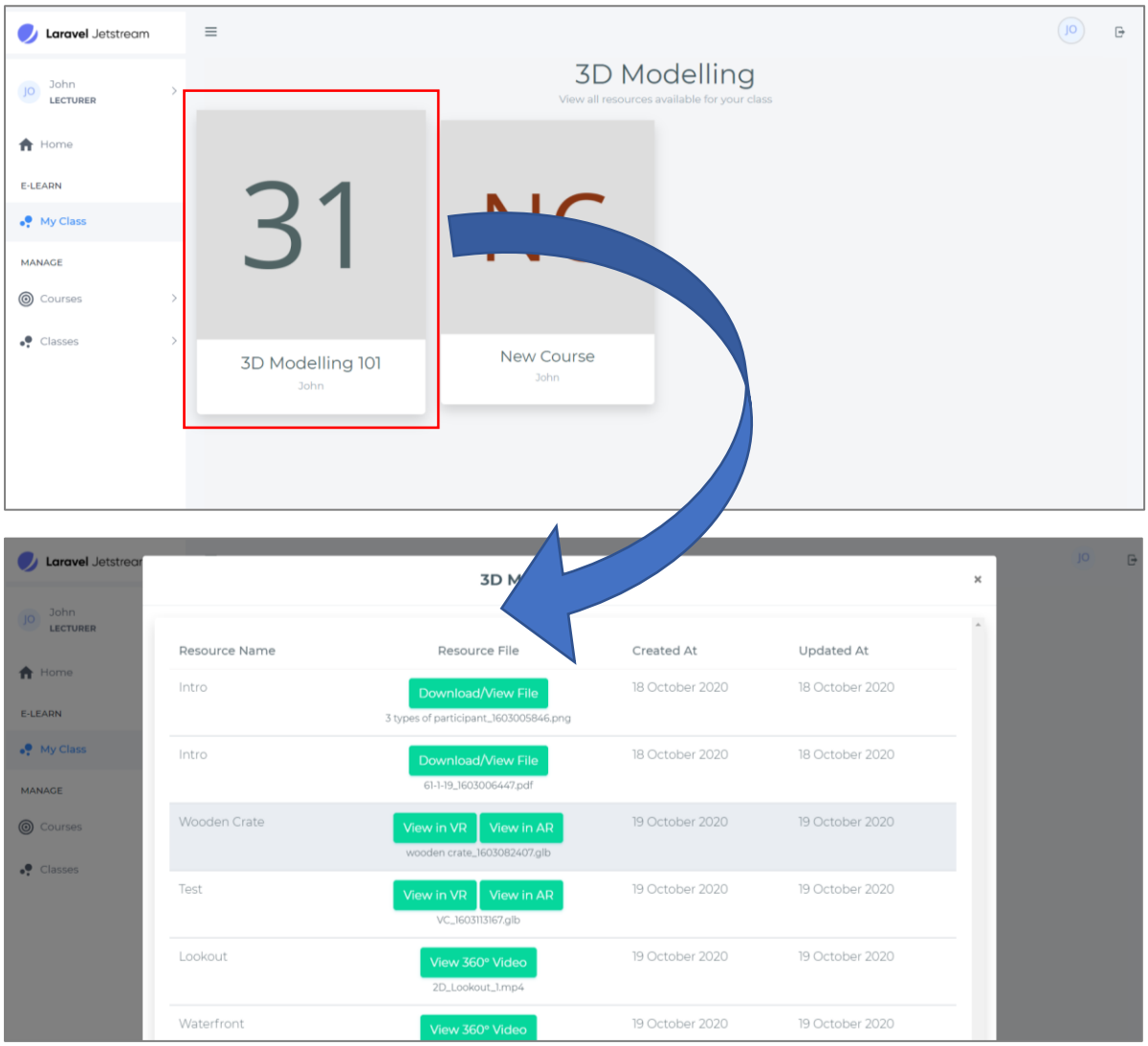
4) Go to the class webpage and open the 3D model scene.

View the 3D model

1) Choose a class by clicking your username icon at the top right corner.



2) In the “My Class” page, choose a course you want to view, which will open a modal dialog box.



3) Search for the 3D model you want to view. You may notice that only 3D model asset has 2 buttons: "**View in VR**" button and "**View in AR**" button. Click on the "**View in VR**" button to open the VR scene.

More user interaction using Oculus Touch Controller

1) In this section, we want to add 3D buttons into the scene to let the user change the size of the 3D model.
2) We can extend the current web application into HMD VR application by adding tracked controller for a more immersive interaction. Adding HTC Vive or Oculus Touch tracked controller is easy:

```
<!-- Vive. -->
<a-entity vive-controls="hand: left"></a-entity>
<a-entity vive-controls="hand: right"></a-entity>

<!-- Or Rift. -->
<a-entity oculus-touch-controls="hand: left"></a-entity>
<a-entity oculus-touch-controls="hand: right"></a-entity>
```

3) To add the tracked controller input, we can place it as the child of the camera entity. When moving or rotating the camera relative to the scene, use a camera rig. By doing so, the camera's height offset can be updated by room-scale devices, while still allowing the tracked area to be moved independently around the scene, which is applicable to the HMD devices such as HTC Vive and Oculus Quest. Therefore, change your `<a-camera>` entity into:

```
<a-entity id="cameraRig" position="0 0.6 0">
    <a-camera id="head" camera wasd-controls look-controls>
        <a-cursor id="cursor"></a-cursor>
    </a-camera>
</a-entity>
```

4) Next, we add a tracked controller into the scene with a laser pointer.

```
<a-entity id="cameraRig" position="0 0.6 0">

    <!-- ... -->
    <a-entity id="rightHand" oculus-touch-controls="hand: right" laser-
controls></a-entity>
</a-entity>
```

5) Then, we use the box entity for button and text entity for the button's text. Button ID `btn-minus` is to reduce the size of the 3D model, while button `btn-plus` is to increase the size of the 3D model.

```
<a-scene>

      <!-- ... -->
```

```
      <a-entity position="-2 4 0" rotation="0 15 0">
        <a-entity id="btn-minus"
          class="link"
          geometry="primitive: box"
          material="color: blue"
          scale="2 1 0.5"
          addValue="-0.05"
          cursor-listener></a-entity>
        <a-entity text="value: - Reduce size; width: 3; color: white;"
          position="1 0 0.25"></a-entity>
      </a-entity>

      <a-entity position="2 4 0" rotation="0 -15 0">
        <a-entity id="btn-plus"
          class="link"
          geometry="primitive: box"
          material="color: green"
          scale="2 1 0.5"
          addValue="0.05"
          cursor-listener>
        </a-entity>
        <a-entity text="value: + Add size; width: 3; color: black;"
          position="1 0 0.25"></a-entity>
      </a-entity>
</a-scene>
```

6) Add an object (with class named `link`) to the `raycaster` component to the cursor entity so that the cursor can be clicked on the `link` button classes.

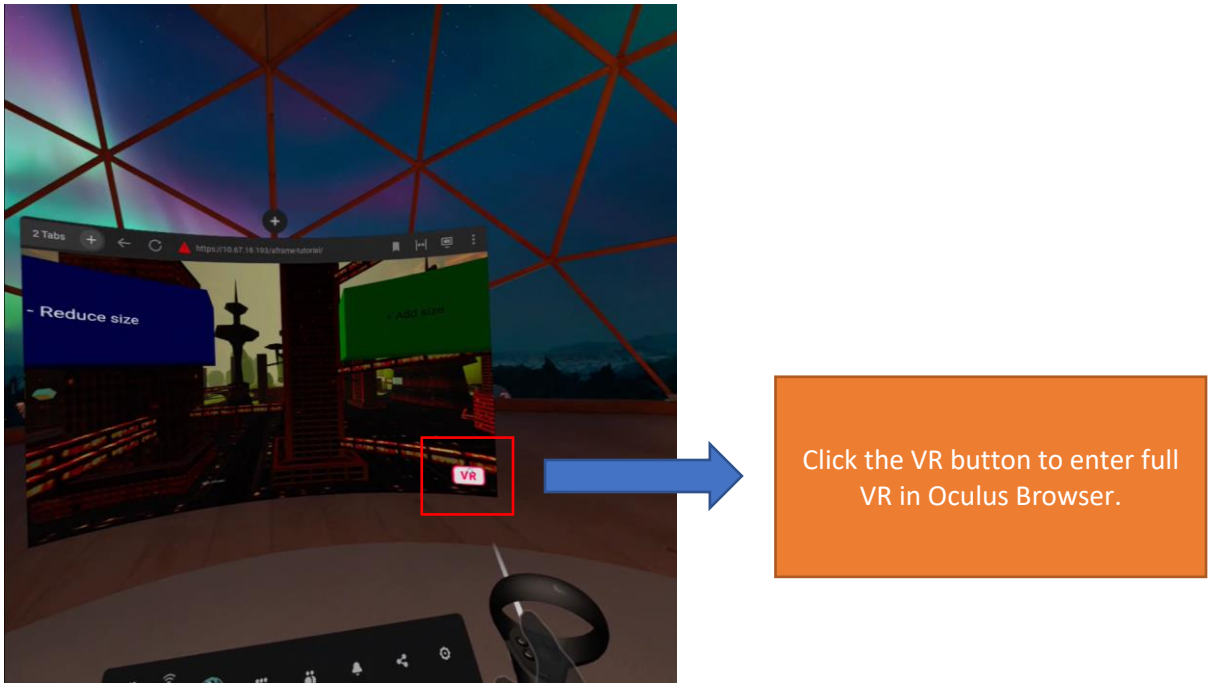
```
<a-scene>
      <a-entity id="cameraRig" position="0 .6 4">
        <a-camera id="head" camera wasd-controls look-controls>
          <a-cursor id="cursor" raycaster="objects: .link"></a-cursor>
        </a-camera>

        <!-- ... -->
      </a-entity>
</a-scene>
```

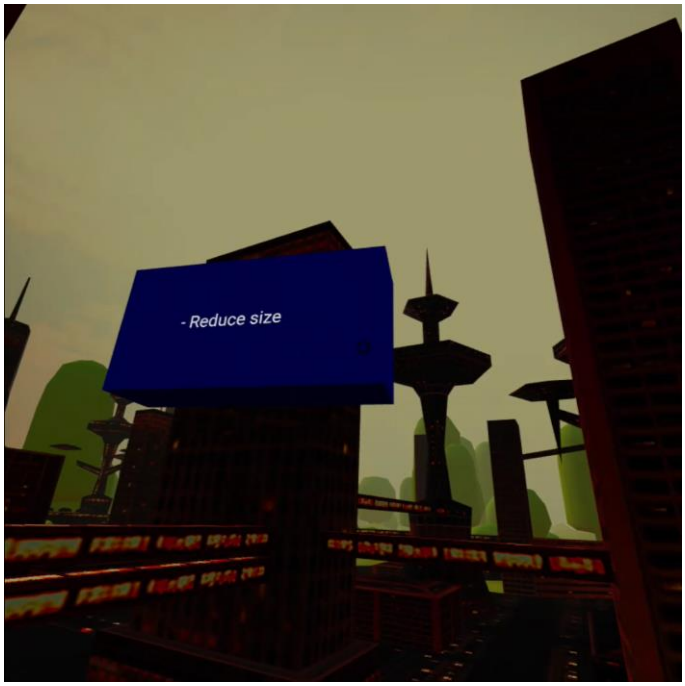
7) Let's add the `cursor-listener` component registration to increase or reduce the size of the model when the plus button or minus button is clicked:

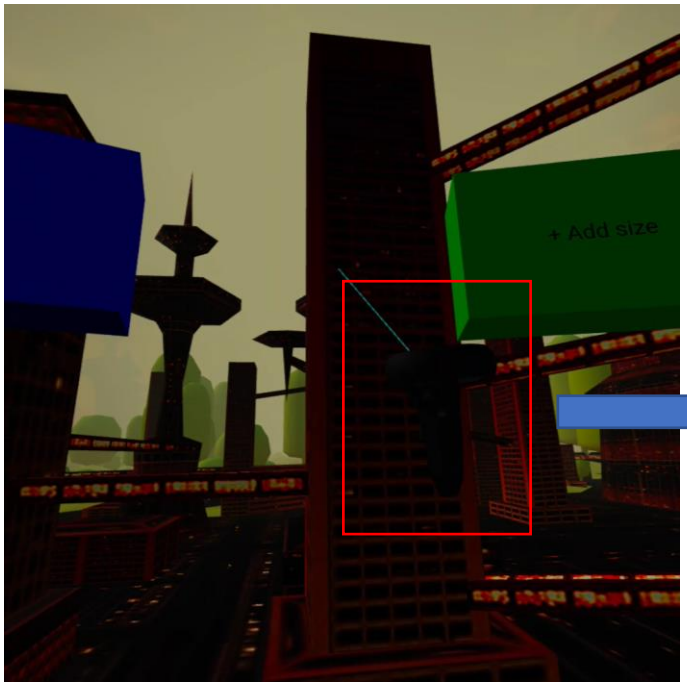
```
<script>

      <!-- ... -->
      AFRAME.registerComponent('cursor-listener', {
        init: function () {
```

```
          this.el.addEventListener('click', function (evt) {
            var btnEl = evt.target;
            var addValue = parseFloat(btnEl.getAttribute('addValue'));
            var targetModel = document.querySelector('#gltfModel');
            var scale = targetModel.getAttribute('scale');
            targetModel.setAttribute('scale', {
              x: scale.x + addValue,
              y: scale.y + addValue,
              z: scale.z + addValue
            });
          });
        }
      });
</script>
```

8) Play the scene. Direct the cursor to the minus button and use mouse click to see the change of 3D model's size, while the plus button increases the 3D model's size. If you are using Oculus Quest VR, you can enter the VR scene and use the controller to click on the buttons to see the effect. You can enter the Laravel project by following the step in Extra: View the VR scene in Oculus Quest and type in the address: **https://<IPv4 Address>/**







Oculus Touch Controller

HOW TO SETUP 360-DEGREE VIDEO?

Introduction

We can use the videosphere primitive to play 360° videos in the background of the scene. Videospheres are a large sphere with the video texture mapped to the inside.

Stage 1 – Setup 360-degree video

Create 360-degree video scene using A-Frame

1) First, include the A-Frame in the HTML:

```
<head>
    <script src="https://aframe.io/releases/1.0.4/aframe.min.js"></script>
</head>
```

2) Use the asset management system to fetch the 360-degree video to the scene. The videosphere entity is defined as `<a-videosphere>`:

```
<body>
    <a-scene>
        <a-assets>
            <video id="video" autoplay loop="true" src="assets/videos/2D_Waterfront.mp4"></video>
        </a-assets>

        <a-videosphere src="#video"></a-videosphere>
    </a-scene>
</body>
```

3) Play the scene and look around:

Stage 2 – Integration with Laravel Project

Create a new route

1) We can integrate the 360-degree video application with our elearning Laravel project. Create a new folder named “video_360” in your “resources” > “views” folder. Then, copy and paste your “index.html” into “video_360” folder. Rename “index.html” into “index.blade.php”.
2) Create a new route in “web.php” in “routes” folder:

```
Route::get('/video_360', function() {
        return view('video_360.index');
})->name('video_360');
```

3) Type in the address bar in your browser: “**https://<your-project-name>.test /video_360**”.

Create function to display 360-degree video based on the requested video ID

1) Since our elearning project consists of the 360-degree video upload function, we can view the video in VR view in the elearning based on the video ID dynamically.
2) Update the “web.php” to accept the video ID and get the 360-degree video based on the ID:

```
Route::get('/video_360/{videoId}', function($videoId) {
        $video = App\Models\CourseFile::find($videoId);

        return view('video_360.index', [
            'videoSrc' => $video
        ]);
})->name('video_360');
```

View 360-degree video

1) Open the 360-degree video scene as in the View the 3D model section.
2) Search for the 360-degree video you want to view, which has a “**View 360° Video**” button. Click on the “**View 360° Video**” button to open the VR scene.

HOW TO SETUP AUGMENTED REALITY (AR) SCENE?

Introduction

What is AR?

Augmented reality (AR) is the technology that makes possible to add overlayed content on the real world. It can be provided for several type of devices: hand-held (like mobile phones), headsets, desktop displays, and so on.

For hand-held devices (more in general, for video-see-through devices) the 'reality' is captured from one or more cameras and then shown on the device display, adding content on top of it.

What is AR.js?

AR.js is a lightweight library for AR on the Web, coming with features like Image Tracking, Location based AR and Marker tracking.

Source: Photo by Patrick Schneider on Unsplash

Stage 1 – Marker tracking

Installation of AR.js and A-Frame using CDN

To include AR.js in an HTML file in the local server, you can drop a <script> tag pointing to the CDN build:

```
<script src="https://raw.githack.com/AR-js-org/AR.js/master/aframe/build/aframe-ar-nft.js"></script>
```

Besides that, we also need to include A-Frame library for the 3D functionality. The A-Frame is working well with the AR.js to create a rich AR experience.

```
<head>
```

```
    <script src="https://aframe.io/releases/1.0.4/aframe.min.js"></script>
    <script src="https://raw.githack.com/AR-js-
org/AR.js/master/aframe/build/aframe-ar-nft.js"></script>
    <script src="https://unpkg.com/aframe-extras.animation-
mixer@3.4.0/dist/aframe-extras.animation-mixer.js"></script>
</head>
```

Adding AR.js entity in A-Frame

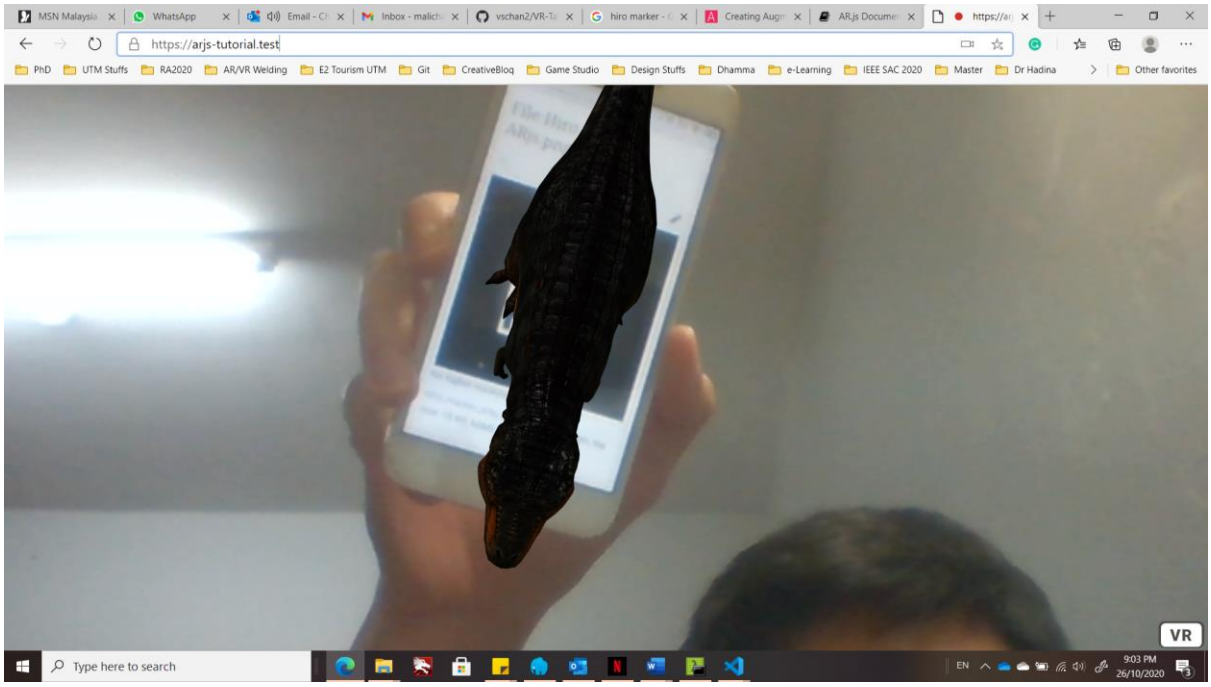
1) In this section, we will focus on the **Marker Tracking**. The `<a-marker>` contains a preset Hiro marker to activate our AR scene.

```
<body style="margin : 0px; overflow: hidden;">
    <a-scene embedded arjs>
      <a-assets>
            <a-asset-item id="3dmodel" src="https://arjs-cors-
proxy.herokuapp.com/https://raw.githack.com/AR-js-
org/AR.js/master/aframe/examples/image-tracking/nft/trex/scene.gltf"></a-
asset-item>
            </a-assets>
        </a-assets>

        <a-marker preset="hiro">
            <a-entity
            id="gltfModel"
            position="0 0 0"
            scale="0.05 0.05 0.05"
            gltf-model="#3dmodel"
            animation-mixer
            ></a-entity>
        </a-marker>
        <a-entity camera></a-entity>
    </a-scene>
</body>
```

We place a 3D model entity inside the `<a-marker>` entity.

2) The Hiro marker.

3) Type in the address bar in your browser: "**https://<your-project-name>.test**". Place your Hiro marker in front of your laptop/desktop camera. Now we can see the T-rex!



Stage 2 – Integration with Laravel project

Create a new route

1) We can integrate the AR application with our elearning Laravel project. Create a new folder named "ar_view" in your "resources" > "views" folder. Then, copy and paste your "index.html" into "ar_view" folder. Rename "index.html" into "index.blade.php".
2) Create a new route in "web.php" in "routes" folder:

```
Route::get('/ar_view', function() {
        return view('ar_view.index');
})->name('ar_view');
```

3) Type in the address bar in your browser: "**https://<your-project-name>.test /ar_view**".

Create function to display 3D model based on the requested model ID

1) Since our elearning project consists of the 3D model upload function, we can view the 3D model in AR view in the elearning based on the model ID dynamically.
2) Update the "web.php" to accept the model ID and get the 3D model based on the ID:

```
Route::get('/ar_view/{modelId}', function($modelId) {
        $model = App\Models\CourseFile::find($modelId);

        return view('ar_view.index', [
            'modelSrc' => $model
        ]);
```

```
})->name('ar_view');
```

3) Update the asset source in the “index.blade.php” to link the 3D model to the chosen 3D model directory.

```
<a-asset-item id="3dmodel"
    src="{{ URL::to('').$modelSrc->file_path.'') }}"></a-asset-item>
</a-assets>
```

So, whenever the model ID changed, the scene updates the 3D model according to the ID.

4) Go to the class webpage and open the 3D model scene.

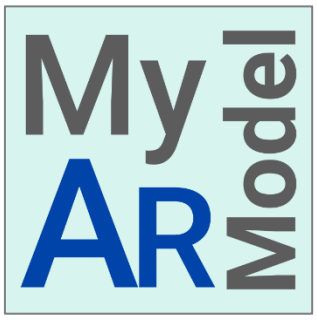
View the 3D model in AR

1) Open the 360-degree video scene as in the View the 3D model section.
2) Search for the 3D model you want to view. You may notice that only 3D model asset has 2 buttons: “**View in VR**” button and “**View in AR**” button. Click on the “**View in AR**” button to open the AR scene.

Stage 3 – Create your own marker

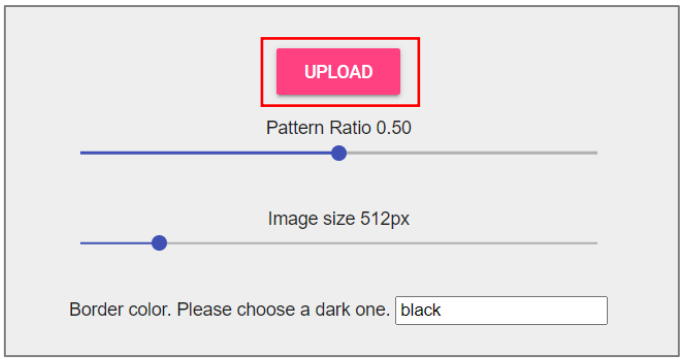
Design and create own marker for elearning

1) Firstly, you need to create a square image by using any image processing program, such as Adobe Photoshop, Adobe Illustrator, GIMP, Inkscape, etc. Example:

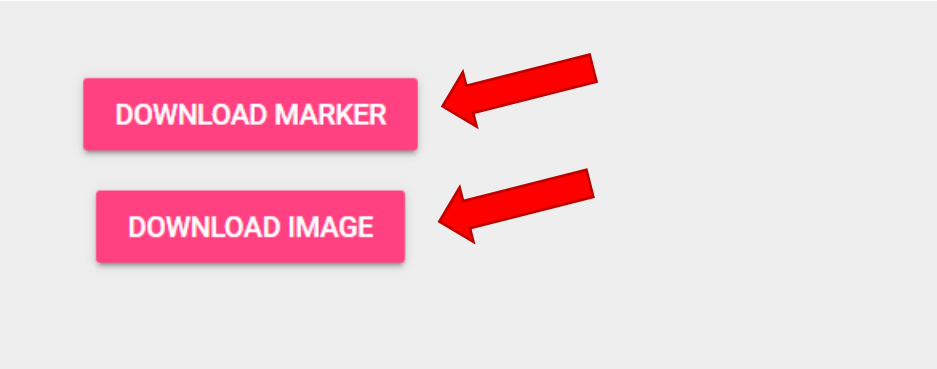


2) Then, you need to upload in this AR.js marker generator website: https://jeromeetienne.github.io/AR.js/three.js/examples/marker-training/examples/generator.html

3) Click on the “upload” button to select your image marker.

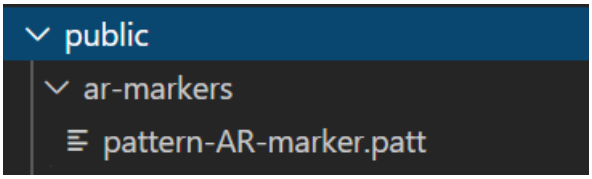


4) Click on the “DOWNLOAD MARKER” button to let the system generate your marker. It will download as a PATT file format.
5) Click on the “DOWNLOAD IMAGE” button to download the image marker.



Import to Laravel project

1) Create a new folder “ar-markers” in “public” folder.
2) Copy the PATT file and paste into your Laravel project in the “ar-markers” folder.



Using the new marker

Then, change your AR marker URL in “index.blade.php” and add a new `type` component in the `<a-marker>` entity which is assigned to `pattern`.

```
<a-marker type="pattern" url="{{ asset('ar-markers/pattern-AR-
marker.patt') }}">
        <a-entity
            id="gltfModel"
            position="0 0 0"
            scale="0.05 0.05 0.05"
            gltf-model="#3dmodel"
            animation-mixer
```

```
            ></a-entity>
</a-marker>
```

Stage 3 – User interaction in AR scene

Create a custom button

1) Same as the topic in VR, we would like to create 2 buttons to change the size of the 3D model. Let's create 2 buttons in the AR scene:

```
<a-scene embedded arjs>

        <!-- ... -->
        <div>
            <button class="change-size btn-
primary" style="left: 0; top: 0" addvalue="-0.05" cursor-
listener>- Reduce size</button>
            <button class="change-size btn-
primary" style="left: 100px; top: 0" addvalue="0.05" cursor-
listener>+ Add size</button>
        </div>

        <!-- ... -->
    </a-scene>
```

2) Then, we use custom CSS to sylize our buttons:

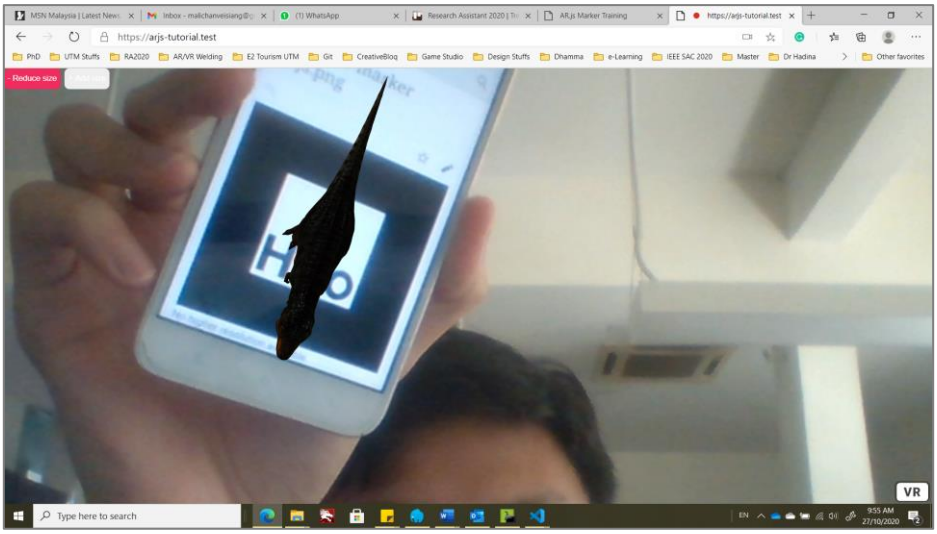
```
<head>

    <!-- ... -->
    <style>
        .btn-primary {
          background-size: 90% 90%;
          border: 0;
          top: 0;
          color: #fff;
          cursor: pointer;
          min-width: 58px;
          min-height: 34px;
          outline: 0;
          position: fixed;
          left: 0;
          transition: background-color .05s ease;
          user-select: none;
          -webkit-transition: background-color .05s ease;
          z-index: 9999;
          border-radius: 8px;
          touch-action: manipulation;
```

```
        }

        .btn-primary:active, .btn-primary:hover {
          background-color: #ef2d5e;
        }
    </style>
</head>
```

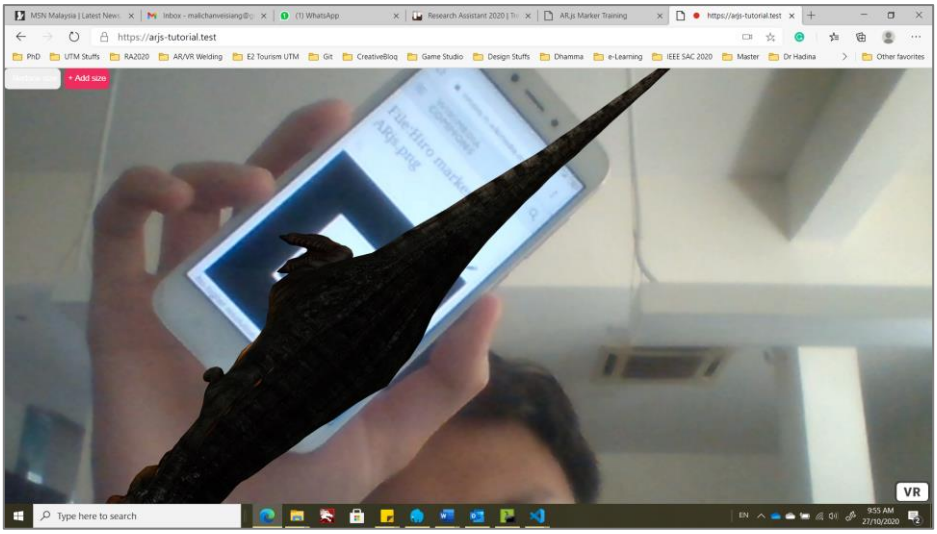
3) Add button listener in JavaScript to reduce or increase the 3D model's size based on the button:

```
<script>
      var buttons = document.querySelectorAll('.change-size');

      for(var i = 0; i < buttons.length; i++) {
        buttons[i].addEventListener('click', function(evt) {
          var btnEl = evt.target;
          var addValue = parseFloat(btnEl.getAttribute('addvalue'));

          var targetModel = document.querySelector('#gltfModel');
          var scale = targetModel.getAttribute('scale');
          targetModel.setAttribute('scale', {
            x: scale.x + addValue,
            y: scale.y + addValue,
            z: scale.z + addValue
          });
        });
      }
</script>
```

4) Open the scene and see the result.



Reduce the size of 3D model.



Increase the size of 3D model.