**Dump and Die 🡪** dd(var); (display var dump and die the page)

**compact(‘var\_name’) ->** will pass a value from $var\_name to var\_name parameter. So you can access it in the view by calling the $var\_name

**Step to start**

* Install via composer
* Wait until download complete.
* Open terminal in the directory and do “php artisan serve”
* Ctrl + c to quit
* Code . to call vs code

**ROUTE**

**Defining route**

Normal route

Route::get(‘/route, function(){

return view(‘view);

});

View Route, can be shortened with

Route::view( ‘/route’, ’view’, [‘key’ => ‘value’] );

Controller route

Route::get( ‘/route’, ‘ControllerName@methodName’ );

Route::get( ‘/route’, [ ControllerName::Class, ’methodName’ ] ); //in version 8

*Method will define to which view the route will go*

Route Name, you can give a route a name by :

Route::get( ‘/route’, ‘ControllerName@methodName’ )->name(‘theRouteName’);

you can use route name to navigate to each page by

{{ route(‘theRouteName’) }};

*If not using route name then you can use ->* {{ url(‘/where’) }}

**Prefix**

Prefix add prefix to the link before the route that is included in the group.

Route::group ( [ ‘prefix’ => ‘page’ ], function(){

Route::get( ‘/home, ‘ControllerName@home )->name(‘index’);

Route::get( ‘/about, ‘ControllerName@about )->name(‘aboutus’);

}); //when the link is clicked or the route is used, the url will show page/home or page/about instead of home and about

//caution! To use prefix you must use Route name in the url .

Directive

@section( ‘thisSection’ )

Section will group a certain block of codes to indicate those codes in their representative sections, e.q : content

@endsection

*@section(‘PageTitles’, ‘The titles’) -> you can also give parameters after it usually to define each views titles, meta description, meta keywords and any section which is unique, remember that section and yield is casesensitive so in this case yield should be @yield(‘PageTitles’).*

@yield( ‘thisSection’ )

Yield will indicate where the codes in @section will be placed.

@component(‘componentFile’)

@slot(‘theVariable’)

text

@endslot

@endComponent

Inside the component you can call your components by wrapping them in their slots. The text inside slot will replace the variable with the same name in the componentFile( {{$theVariable}} ). You can use this to change the attribute of style, text and many others to make them more dynamic.

**Database MIGRATION**

php artisan make:migration => to make migration

php artisan migrate => to migrate

php artisan migrate:rollback

**CHECK DOCUMENTATION => Database -> Migration**

create table

add column

modify column

renaming column

Note: to do task such as renaming or removing column you might require to put

doctrine/dbal = “\*” (default version). under the required-dev in composer.json then do **composer update** in the terminal or just ***require doctrine/dbal.*** But, just if you succeed in adding the dependencies but suffer some errors like

*‘"Doctrine\DBAL\Driver\PDOMySql\Driver" not found* ’ you will need to downgrade the doctrine/dbal = “^2.10“ (version 2)

**Authentication**

**CHECK DOCUMENTATION => Security -> authentication**

Install laravel/ui using ***composer require laravel/ui “^1.0”***

laravel/ui Compatibility

Version Laravel Version

1.x 5.8, 6.x

2.x 7.x

3.x 8.x

Laravel/ui

To edit the view or php files provided by the laravel/ui

Go to vendor>laravel>framework>src/illuminate>foundation>auth

There exist list of methods but you shouldn’t edit them, instead copy the methods to the controller files and override them there.

{{ Auth::user()->name }} means the name of the current authenticated user model’s name by placing it inside {{ }} means to output the user’s name there is also other attribute depends on the model

To edit user :

    public function profileUpdate(Request $request){

        $user = Auth::user();

        $user->name = $request['name'];

        $user->email = $request['email'];

        $user->save();

       return back();

    }

$request will store the data from Request object. Request is where to obtain an instance of the current HTTP request, in this example, is the user’s name and email from the post method

Back() is global method to redirect to previous view

To do validation just put it before the operation, if something went wrong the rest of the code won’t run

public function profileUpdate(Request $request){

        //validation

        $request->validate([

            'name' => 'required|min:2|string|max:255',

            'email' => 'required|email|max:255'

        ]);

        $user = Auth::user();

        $user->name = $request['name'];

        $user->email = $request['email'];

        $user->save();

       return back()->with(‘flashMessage’,’profile updated’);

    }

**Then you can use @error or @if ($error) method in the view to print the error. Look at the documentation in The Basics -> Validation to see how to implement @error , @if and other validation rules available**

Upload Profile

Install intervention image (you can use this package to modify the image and save it to the database).

Image::make($avatar)->resize(250,250)->save(public\_path("/img/avatar/".$filename));

Image is the name of the class created, see the documentation for details in installation page

After you installed intervention image, to use the GD library, please uncomment the GD extension in php.ini file, after that reset your server

MIDDLE WARE

    /\*\*

     \* Create a new controller instance.

     \*

     \* @return void

     \*/

    public function \_\_construct()

    {

        $this->middleware(['auth','verified']);

    }

Some example on the use in above, means to access that controller, the user must be authenticated and has their email verified, if no they will be redirected to login form, where to redirect can be found in

Illuminate\Auth\Middleware\Authenticate  where it defaultly redirect to login

<?php

namespace App\Http\Middleware;

use Illuminate\Auth\Middleware\Authenticate as Middleware;

class Authenticate extends Middleware

{

    /\*\*

     \* Get the path the user should be redirected to when they are not authenticated.

     \*

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

     \* @return string|null

     \*/

    protected function redirectTo($request)

    {

        if (! $request->expectsJson()) {

            return route('login');

        }

    }

}

To Make Email Verification there is a site named Mailtrap to help this

To use simply updated your .env files of MAIL by copying the configuration the site provide

**ELOQUENT ORM**

php artisan make:model model\_name –m //// dash m means to automatically make migration to create the table for that model

Relationship

One to one

* hasOne() and belongsTo()
* public function phone() {
* return $this->hasOne('App\phone');
* }
* public function user(){
* return $this->belongsTo('App\User');
* }

one to many

* hasMany() and belongsTo()
* public function jobs(){
* return $this->hasMany('App\Job','client\_id');
* } // by sppecifying client\_id in the second arguments, this overrides the default which is user\_id as reference
* public function client(){
* return $this->belongsTo(User::class,'client\_id');
* }

**Quick Tip on making table with foreign key**

**Schema::create('role\_user', function (Blueprint $table) {**

**$table->bigIncrements('id');**

**$table->unsignedBigInteger('user\_id');**

**$table->foreign('user\_id')->references('id')->on('users')->onDelete('cascade');**

**$table->unsignedBigInteger('role\_id');**

**$table->foreign('role\_id')->references('id')->on('roles')->onDelete('cascade');**

**$table->timestamps();**

**});**

Many to Many

    public function roles(){

        return $this->belongsToMany(Role::class,'role\_user');

    }

    public function users(){

        return $this->belongsToMany(User::class,'role\_user');

    }

Many to many will result in 1 new table that will manage the relationship named combined from both table example, user and role will produce role\_user, which placed on alphabetical order

**DATA TESTING**

Faker (php library), if you want to see the full details what kind of data faker serve just visit the documentation in the github fzaninotto/faker

To generate fake data for testing, Laravel has user factory (under database section) for sample code to show you how to do it. After done you can call the factory you created in the database seeder and run it by php artisan db seed

php artisan make:factory JobFactory 🡪 to create factory

in the factory : (remember to change the (use App\model to use App\Job) or your model name

$factory->define(Job::class, function (Faker $faker) {

    return [

        'client\_id' => rand(1,3),

        'job\_title' => $faker->sentence(rand(5,7)),

        'job\_type'  => $faker->word(rand(5,7)),

        'job\_description' => $faker->paragraphs(rand(5,7),true),

        'salary' => $faker->numberBetween($min =1000, $max = 9000),

        'slug' => $faker->unique()->slug,

        'created\_at' => now(),

        'upadated\_at'=>now()

    ];

});

php artisan make:seeder JobsTableSeeder 🡪 to create seeder

in the JobsTableSeeder call the JobFactory function (remember to use App\Job whenever you call the class

factory(Job::class,20)->create();

then in the DatabaseSeeder: put this line

$this->call(JobsTableSeeder::class);

Then you can execute it in the terminal with:

Php artisan db:seed –class=JobsTableSeeder

**Route Model Binding**

To pass a data to a method and route you can use this in the controller :

     Public function show($id){

         $job = Job::find($id);

         return view('jobs.show',compact('job'));

     }

And then the route will looks like this :

Route::get('/jobs/{id}','JobController@show')->name('jobs.show');

To access the route you can either use this for Route Naming:

{{route('jobs.show',$job->id)}}

Or this for url

{{url('jobs/',$job->id)}}

But those above are just the basics.

Route Model Binding allows you to write the code like this:

    Public function show(Job $job){

         return view('jobs.show',compact('job'));

     }

Where you don’t need to specify the job var anymore, and laravel will do the works for you.

Then change the route into

Route::get('/jobs/{job}','JobController@show')->name('jobs.show');

In case you don’t want to use the default parameter id you, can specify the column to use like :

Route::get('/jobs/{job:slug}','JobController@show')->name('jobs.show');

Other method to use is to override the **getRouteKeyName()** method, by calling it in the Job class and return the preferred column like :

class Job extends Model

{

    //Get the client that owns this job

    public function client(){

        return $this->belongsTo(User::class,'client\_id');

    }

    public function getRouteKeyName(){

        return 'slug';

    }

}

So after that the route will use ‘slug’ column as the parameter, example: the red is value of slug column from the selected job <http://127.0.0.1:8000/jobs/nemo-beatae-voluptas-distinctio-sequi>

Date formatting

$job->created\_at->diffForHumans()

Will use the diffForHumans() method for the date in created\_at column of jobs table

diffForHumans() is a method provided by carbon API, and already installed in laravel 5 +

diffForHumans means difference for humans which will shows the format easier for human to read like 1 hour ago, 1 months ago, etc.

for other method or use view <https://carbon.nesbot.com/docs/>

**CRUD**

Index

Displaying list

    public function index()

    {

        //

        $jobs = Job::where('client\_id',Auth()->id())->get();

        return view ('dashboard.index',compact('jobs'));

    }

To find more about the query, go see the documentation Laravel Database Query Builder

**Create**

Mass Assignment

When you create new data, you will assign more than 1 column/field of data you will need to specify the field of your model that are fillable or guarded to proceed the action, so that laravel know which is okay to fill and which is not okay to fill. To achieve that you can place either of this code in the model (e.q. : Job )

*While you can only choose one of this, there is no restriction and it is on your own preference*

$fillable

protected $fillable=['job\_title','job\_type','job\_description','salary','slug'];

$guarded (the reverse of fillable, by assigning an empty array means all field is fillable

protected $guarded=[];

<form action="{{route('dashboard.store') }}" method="POST">

                    @csrf

                    <div class="form-group">

                        <label for="job\_title"><strong>Job Title:</strong></label>

                        <input type="text" class="form-control" name="job\_title" id="job\_title">

                    </div>

                    <div class="form-group">

                        <label for="job\_type"><strong>Job Type:</strong></label>

                        <input type="text" class="form-control" name="job\_type" id="job\_type">

                    </div>

                    <div class="form-group">

                        <label for="salary"><strong>Salary:</strong></label>

                        <input type="text" class="form-control" name="salary" id="salary">

                    </div>

                    <div class="form-group">

                        <label for="slug"><strong>Slug:</strong></label>

                        <input type="text" class="form-control" name="slug" id="slug">

                    </div>

                    <div class="form-group">

                        <label for="job\_description"><strong>Job Description:</strong></label>

                        <textarea row="10" class="form-control" id="job\_description" name="job\_description"></textarea>

                    </div>

                    <button class="btn btn-success" type="submit">Save</button>

                    </form>

Store method

This method is used to store the data from create view to the database.

 public function store(Request $request)

    {

        //

        $request->validate([

            'job\_title' => 'required',

            'slug' => 'required|unique:jobs',

            'job\_type' => 'required',

            'salary' => 'required|numeric',

            'job\_description' => 'required|min:250'

        ]);

        $request->user()->jobs()->create($request->all());

        return redirect('dashboard')->with('message','New job post uploaded');

    }

Quick note , so $request->user() is a method that returns the current authenticated user, and since the User has relation with Job by the method jobs(), so it gets the User Id, and will assign it to the client\_id of jobs table while also store the data from forms to the jobs table

Bonus script to retype slug

    <script>

        $('#job\_title').on('keyup',function(){

            var theTitle = this.value.toLowerCase().trim();

            slugInput = $('#slug');

            theSlug = theTitle.replace(/&/g,'-and-').replace(/[^a-z0-9-]+/g,'-').replace(/\-\-+/g,'-').replace(/^-+|-+&/g, '');

            slugInput.val(theSlug);

 });

    </script>

**Edit**

To do edit you need the usual @csrf method to let you modify the database in laravel

To do edit you can use between put or patch method but need to be specified specifically using blade template because HTML doesn’t recognize those methods like this:

<form action="{{ route('dashboard.update',$job->id)}}" method="patch">

        @method('PATCH')

        @csrf

                    <div class="form-group">

                        <label for="job\_title"><strong>Job Title:</strong></label>

                        <input type="text" class="form-control" name="job\_title" value="{{ $job->job\_title }}">

                    </div>

                    <div class="form-group">

                        <label for="job\_type"><strong>Job Type:</strong></label>

                        <input type="text" class="form-control" name="job\_type" value="{{ $job->job\_type }}">

                    </div>

                    <div class="form-group">

                        <label for="Salary"><strong>Salary:</strong></label>

                        <input type="text" class="form-control" name="Salary" value="{{ $job->salary }}">

                    </div>

                    <div class="form-group">

                        <label for="slug"><strong>Slug:</strong></label>

                        <input type="text" class="form-control" name="slug" value="{{ $job->slug }}">

                    </div>

                    <div class="form-group">

                        <label for="job\_description"><strong>Job Description:</strong></label>

                        <textarea rows="10" class="form-control" id="job\_description" name="job\_description" >{{ $job->job\_description }}</textarea>

                    </div>

                    <button class="btn btn-success" type="submit">Save</button>

                    </form>

**Update Method**

  public function update(Request $request, $id)

    {

        //

        $request->validate([

            'job\_title' => 'required',

            'slug' => 'required',

            'job\_type' => 'required',

            'salary' => 'required',

            'job\_description' => 'max:1500'

        ]);

        Job::find($id)->update($request->all());

        return redirect()-> route('dashboard.index')->with('message','Job updated succesfully');

    }

REDIRECT method

Redirect -> to route\_name

Back -> to previous view

**Destroy**

 <td>

    <a href="#" class="btn btn-danger btn-xs"

    onclick="event.preventDefault();

    document.getElementById('deleteJobPostForm').submit();" >Delete</a>

  </td>

</tr>

@endforeach

</tbody>

</table>

<form action="{{ route('dashboard.destroy',$job->id) }}" method="post" id="deleteJobPostForm">

 @method('DELETE')

  @csrf

</form>

So, just like what previewed in the cut of codes above, if you make delete button from a link, you can make javascript to trigger the submit of the form delete somewhere, or you can just make a button submit inside the form like this

 <td>

<form action="{{ route('dashboard.destroy',$job->id) }}" method="post" id="deleteJobPostForm">

 @method('DELETE')

  @csrf

 <button class="btn btn-danger btn-xs">Delete</button>

</form>

    <a href="#" class="btn btn-danger btn-xs">Delete</a>

  </td>

</tr>

@endforeach

</tbody>

</table>

Destroy method

   public function destroy($id)

    {

        $job = Job::find($id);

        $job->delete();

        return redirect('/dashboard')->with('message','Job post deleted succesfully');

    }

***Class::find will retrieve object based on the primary key***

Using free table name

protected $table = "country";

Laravel automatically believe the table of the model is the plural version, (+s) but if it is not you must declare or override the table var manually.