|  |  |
| --- | --- |
| Una imagen de una carretera de curvas con árboles  LaraVGram  Desarrolado con Laravel. | Descripción breve  [Capte la atención del lector con un resumen atractivo. Este resumen es una breve descripción del documento. Cuando esté listo para agregar contenido, haga clic aquí y empiece a escribir.]  Fco Marcet  [Título del curso] |

Contenido

[1. DISEÑO BASE DE DATOS 3](#_Toc54943353)

[2. MIGRACION DB. 3](#_Toc54943354)

[3. MODELOS Y RELACIONES ORM ELOQUENT. 5](#_Toc54943355)

[3.1 RELACIONES DE LOS MODELOS O ENTIDADES. 5](#_Toc54943356)

[Formulario modificar datos personales. 8](#_Toc54943357)

[RELLENAR DATOS DEL FORMULARIO 9](#_Toc54943358)

[RECIBIR DATOS FORMULARIO EN EL CONTROLADOR. 9](#_Toc54943359)

[VALIDACION DE DATOS. 10](#_Toc54943360)

[MOSTRAR MENSAJE CON BOOTSTRAP UPDATE OK 11](#_Toc54943361)

[SUBIR IMAGEN AVATAR USUARIO. 12](#_Toc54943362)

[MOSTRAR IMAGEN AVATAR 14](#_Toc54943363)

[AÑADIR HOJA DE ESTILOS. 16](#_Toc54943364)

[ACCESO SOLO USUARIOS IDENTIFICADOS. 16](#_Toc54943365)

[MIDDLEWARES. 16](#_Toc54943366)

[FORMULARIO AÑADIR NUEVA IMAGEN. 18](#_Toc54943367)

[LISTADO DE IMÁGENES PAGINA PRINCIPAL. 21](#_Toc54943368)

[CARD-HEADER CARD(IMG AVATAR+ NICK+FECHA PUBLICACION) 22](#_Toc54943369)

[OBTENER AVATAR 22](#_Toc54943370)

[OBTENER NICK +FECHA 23](#_Toc54943371)

[CARD-BODY (IMAGEN). 23](#_Toc54943372)

[CARD-FOOTER(LIKES+COMENTARIOS). 24](#_Toc54943373)

[PAGINACION ENTRADAS PAGINA HOME. 25](#_Toc54943374)

[DETALLE DE IMAGEN. 25](#_Toc54943375)

[PROVIDERS/helpers EN LARAVEL - FORMATO FECHA. 26](#_Toc54943376)

[COMENTARIOS. 27](#_Toc54943377)

[FORMULARIO + AÑADIR COMENTARIOS. 27](#_Toc54943378)

[LISTAR COMENTARIOS 29](#_Toc54943379)

[ORDENAR COMENTARIOS. 29](#_Toc54943380)

[ELIMINAR COMENTARIO. 30](#_Toc54943381)

[OBTENER OBJETO DEL COMENTARIO 30](#_Toc54943382)

[GESTION DE LIKES. 32](#_Toc54943383)

[METODO LIKE. 32](#_Toc54943384)

[METODO DISLIKE. 34](#_Toc54943385)

[DETECTAR LIKES. 35](#_Toc54943386)

[CARGAR FICHERO JS. 35](#_Toc54943387)

[CAMBIAR COLOR BOTON LIKE. 36](#_Toc54943388)

[PETICIONES AJAX. 38](#_Toc54943389)

[LISTAR LIKES 41](#_Toc54943390)

[INCLUDE image.blade.php 41](#_Toc54943391)

[PERFIL DE USUARIO. 43](#_Toc54943392)

[MOSTRAR DATOS EN EL PERFIL. 44](#_Toc54943393)

[BORRADO Y EDICION DE PUBLICACIONES. 45](#_Toc54943394)

[BOTONES DE BORRADO Y EDICION. 45](#_Toc54943395)

[BORRAR PUBLICACION. 45](#_Toc54943396)

[MODAL ELIMINAR PUBLICACION. 48](#_Toc54943397)

[EDITAR PUBLICACION. 49](#_Toc54943398)

[GENTE Y BUSCADOR. 51](#_Toc54943399)

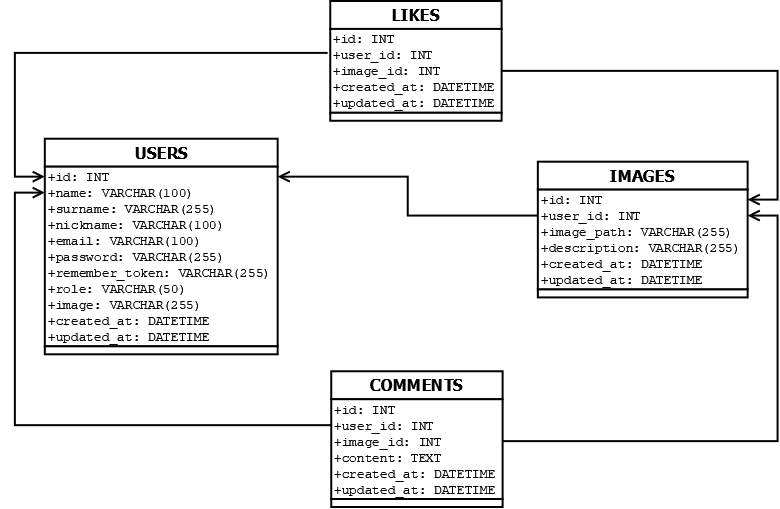
[GENTE. 51](#_Toc54943400)

[BUSCADOR. 52](#_Toc54943401)

[AUTENTICACION EN LARAVEL 6 53](#_Toc54943402)

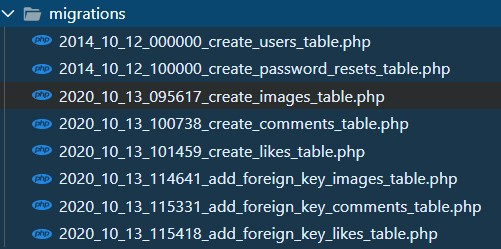
[AUTENTICACION EN LARAVEL 8 53](#_Toc54943403)

# 1. DISEÑO BASE DE DATOS



# 2. MIGRACION DB.

Realizamos un script por cada tabla ,mas sencillo y evitamos errores de claves e índices al ejecutar la migración



Vamos a ver un ejemplo de cada:

 public ***function*** **up**()

    {

*/\*Schema::create('users', function (Blueprint $table) {*

*$table->engine = 'InnoDB';*

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

*$table->string('name');*

*$table->string('surname');*

*$table->string('nickname');*

*$table->string('email')->unique();*

*$table->string('password');*

*$table->string('role');*

*$table->string('image');*

*$table->rememberToken();*

*$table->timestamps();*

*});*

*\*/*

***DB*::statement**("

      CREATE TABLE IF NOT EXISTS users

      (

         id INT AUTO\_INCREMENT NOT NULL PRIMARY KEY,

         name VARCHAR(100) NOT NULL,

         surname VARCHAR(255) NOT NULL,

         nickname VARCHAR(100) NOT NULL,

         password VARCHAR(255) NOT NULL,

         email VARCHAR(100) NOT NULL UNIQUE,

         remember\_token VARCHAR(255),

         role VARCHAR(100),

         image VARCHAR(255),

         created\_at DATETIME NOT NULL DEFAULT CURRENT\_TIMESTAMP,

         updated\_at DATETIME NOT NULL DEFAULT CURRENT\_TIMESTAMP

        )ENGINE=InnoDB;

        ");

public ***function*** **down**()

    {

***Schema*::dropIfExists**('users');

    }

***class*** **AddForeignKeyImagesTable** extends ***Migration***

{

*/\*\**

*\* Run the migrations.*

*\**

*\****@return****void**

*\*/*

    public ***function*** **up**()

    {

***DB*::statement**("

        ALTER TABLE images

        ADD CONSTRAINT fk\_images\_users FOREIGN KEY (user\_id) REFERENCES users (id) ON DELETE NO ACTION;

        ");

    }

*/\*\**

*\* Reverse the migrations.*

*\**

*\****@return****void**

*\*/*

    public ***function*** **down**()

    {

***Schema*::table**('images', ***function*** (***Blueprint*** ***$table***) {

        });

    }

}

# 3. MODELOS Y RELACIONES ORM ELOQUENT.

Vamos a crear los modelos que se almacenan en la carpeta App. Se crean en SINGULAR, ya representan un registro o un único elemento de la BD.



**¿Por qué interesa establecer relaciones entre los modelo?**

Esto va a ser de gran utilidad , ya que si por ejemplo al obtener los datos de una imagen nos interesa sacar de una sola tirada conjuntamente otro objeto como los datos del usuario que ha creado la imagen ahorrándonos realizar consultas.

## 3.1 RELACIONES DE LOS MODELOS O ENTIDADES.

**ENTIDAD IMAGE**

QUE NOS INTERESA OBTENER:

* Todos los comentarios asociados a una imagen
  + Una imagen puede tener muchos comentarios->(One to many) METODO comments()
* Todos los likes asociados a una imagen
  + Una imagen puede tener muchos likes->(One to many) METODO likes()
* Usuario que ha creado la imagen
  + ->Muchas imágenes puedes ser creadas por un usuario(Many to one) METODO

Para ello vamos a implementar los métodos comments() y likes()

Laravel entenderá automáticamente que en la tabla "comments" existirá la clave foránea con el que sea autor (user\_id en la tabla comments) y que en la tabla local (images), la clave primaria se llama "id"

<?php

**namespace** **App**;

**use** Illuminate\Database\Eloquent\***Model***;

***class*** **Image** extends ***Model***

{

*//\*Indicamos con que tabla va a trabajar el modelo \*/*

    protected ***$table*** **=** 'images';

*//\* Una imagen puede tener muchos comentarios 1:N [One To Many]\*/*

   public ***function*** **comments**()

   {

*//\* hasmany(Objeto\_con\_el\_se\_relaciona) \*/*

*//\*\* Median el id\_image de comentario y va a obtener el array de los comentarios  \*/*

*//\*\* Haremos un comments(5) y obtendremos los comentarios cuyo id\_image es 5 \*/*

**return** $this**->hasMany**('App\Comment');

   }

   public ***function*** **likes**()

   {

*//\* Una imagen puede tener muchos likes 1:N [One To Many]\*/*

**return** $this**->hasMany**('App\Like');

   }

   public ***function*** **user**()

   {

*//\* Muchas imagenes puede ser de un unico user N:1 [Many to One]\*/*

**return** $this**->belongsTo**('App\User', 'user\_id');

   }

}

ENTIDAD COMMENT

* Todos los comentarios asociados a un usuario (Many to One)
* Todos los comentarios asociados a una imagen(Many to One)

ENTIDAD LIKE

* Todos los likes asociados a un usuario (Many to One)
* Todos los likes asociados a una imagen(Many to One)

ENTIDAD USUARIO

* Un usuario puede tener muchas imagenes (One to many)

TESTEO ORM

Vamos a obtener diferentes datos de las imágenes

**use** App\***Image***;

*//\*antes hemos utilizado en el ej fruteria*

*//\*DB::table('images')->get();*

*//es similar a lo que ahora haremos*

*//\*OBTENER TODAS LAS IMAGENES*

***$images*** **=** ***Image*::all**();

**foreach** (***$images*** **as** ***$image***) {

*//var\_dump($image);*

**echo** "<h3>Nombre: ***$image*->*image\_path***</h3>";

**echo** "<h3>Descripcion: ***$image*->*description***</h3>";

*//\*OBTENER USUARIO QUE HA CREADO LA FOTO(Nos valemos de las*

*relaciones que implementamos*

*//\* en el ORM).Los metodos que hemos creado en las relaciones  ahora podemos acceder como propiedades del objeto*

*$image->user->name*

*//\**

*//var\_dump($image->user);*

**echo** "<h3>Creada por:"**.*$image*->*user*->*name*.**

" "**.*$image*->*user*->*surname*.** "</h3>";

*//\*OBTENER LOS COMENTARIOS D CADA IMAGEN*

**if**( **sizeof**(***$image*->*comments***) **>=** 1 ){

***$comentarios*** **=** ***$image*->*comments***;

**echo** "<strong>Comentarios</strong></br>";

**foreach** (***$comentarios*** **as** ***$comment***) {

**echo** ***$comment*->*content*** **.**"</br>";

*//\* OBTENER QUE USUARIO HIZO EL COMENTARIO*

**echo** "<h3>comentado por:"**.*$comment*->*user*->*name*.**

" "**.*$comment*->*user*->*surname*.** "</h3>";

       }

     }

*//\* LIKES*

**echo**"LIKES:"**.count**(***$image*->*likes***);

**echo** "<hr/>";

Como hemos podido observar hemos accedido a un monton de datos sin ejecutar consultas haciendo uso del ORM y sus relaciones.

Accediendo a los métodos que hemos creado como si fueran propiedades de este:

***$image*->*user*->*name***

Recordemos que implementamos un método user() con relación Many to One (Un usuario muchas imágenes)

De manera análoga los comentarios asociados a cada imagen:

**foreach** (***$comentarios*** **as** ***$comment***) {

**echo** ***$comment*->*content*** **.**"</br>";

*//\* OBTENER QUE USUARIO HIZO EL COMENTARIO*

**echo** "<h3>comentado por:"**.*$comment*->*user*->*name*.**

" "**.*$comment*->*user*->*surname*.** "</h3>";

 }

# Formulario modificar datos personales.

Una vez creados los enlaces en la vista app.blade.php

<a class="dropdown-item" href="">

**{{** **\_\_**('Mi perfil') **}}**

 </a>

<a class="dropdown-item" href="**{{** **route**('config') **}}**">

**{{** **\_\_**('Configuracion') **}}**

</a>

**Ruta:**

***Route*::get**('/configuracion','UserController@config')**-> name**('config');

**Controlador:**

Creamos el controlador -> UserController, mediante: **php artisan make:controller UserController**

***class*** **UserController** extends ***Controller***

{

    public ***function*** **config**()

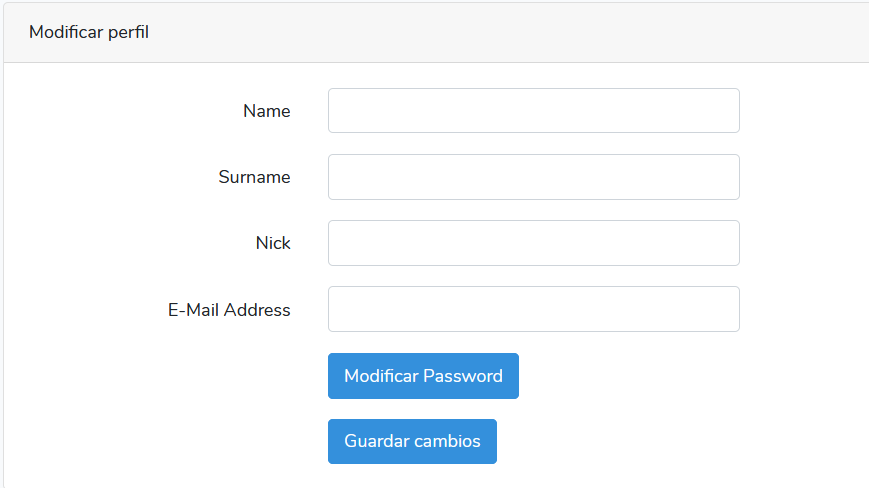
    {

**return** **view**('user.config');

    }

}

**Vista:**



# RELLENAR DATOS DEL FORMULARIO

Usaremos el método estatico que tiene laravel:

**Auth::user**()

En el cual podemos acceder a las diferentes propiedades

**Auth::user**()**->*nick***

Podemos usarlos en el campo value del form:

value="**{{Auth::user**()**->*nick*** **}}**"

Al enviar el formulario ejecutamos el la ruta:

***Route*::post**('user/update','UserController@update')**->name**('user.update');

# RECIBIR DATOS FORMULARIO EN EL CONTROLADOR.

Vamos a crear el método update(Request, $request):

*//\* actualizar datos del usuario*

    public ***function*** **update**(***Request*** ***$request***)

    {

*//\* Recogemos datos en variables*

***$id*** **=** ***Auth*::user**()**->*id***;

***$name*** **=** ***$request*->input**('name');

**var\_dump**(***$id***, ***$name***);**die**();

    }

**NOTA**

Puede dar error si no ponemos \Auth::user()

# VALIDACION DE DATOS.

*//\* actualizar datos del usuario*

public ***function*** **update**(***Request*** ***$request***)

 {

***$id*** **=** \***Auth*::user**()**->*id***;

*//\* Validacion de campos*

***$validate*** **=** $this**->validate**(***$request***, [

     'name' **=>** 'required|string|max:255',

     'surname' **=>** 'required|string|max:255',

     'nick' **=>** 'required|string|max:255|unique:users,nick,'**.*$id***,

 'email' **=>** 'required|string|email|max:255|unique:users,email,'**.*$id***

      ]);

Mediante unique:users indicamos que sea único en la tabla usuarios, esto funciona correctamente en la creación de un nuevo usuario pero al actualizar los datos hemos de comprobar si el Nick coincide con ese id actual del usuario para ello .

'nick' **=>** 'required|string|max:255|unique:users,nick'**.*$id***,

Resumiendo comprobamos que sea único en la tabla users pero con la excepción que el Nick sea el mismo que el del id, permitiendo meter el mismo Nick, lo mismo para el email.

CODIGO FINAL UPDATE USER

*//\* Actualizar datos usuario*

public ***function*** **update**(***Request*** ***$request***)

  {

*//\* conseguir usuario identificado*

***$user*** **=** \***Auth*::user**(); *//$id = \Auth::user()->id;*

***$id*** **=** ***$user*->*id***;

*//\* Validacion de campos*

***$validate*** **=** $this**->validate**(***$request***, [

       'name' **=>** 'required|string|max:255',

       'surname' **=>** 'required|string|max:255',

     'nick' **=>** 'required|string|max:255|unique:users,nick,'**.*$id***,

 'email' **=>** 'required|string|email|max:255|unique:users,email,'**.*$id***

          ]);

*//\* Recogemos datos del form*

*// si no usamos \Auth::user() con la barra error no encuentra la clase*

***$name*** **=** ***$request*->input**('name');

***$surname*** **=** ***$request*->input**('surname');

***$nick*** **=** ***$request*->input**('nick');

***$email*** **=** ***$request*->input**('email');

*//\* Asignar nuevos valores al objeto del usuario, al ser propiedades publicas* *podemos asignarles directamente el valor.*

***$user*->*name*** **=** ***$name***;

***$user*->*surname*** **=** ***$surname***;

***$user*->*nick*** **=** ***$nick***;

***$user*->*email*** **=** ***$email***;

*//\* ejecutar consulta y cambios en la BD*

***$user*->update**();

**return** **redirect**()

**->route**('config')

**->with**([

'message' **=>**'Usuario Actualizado correctamente'

]);

    }

# MOSTRAR MENSAJE CON BOOTSTRAP UPDATE OK

En la vista vamos a mostrar un mensaje

<div class="container">

        <div class="row justify-content-center">

            <div class="col-md-8">

*<!-- Messages Ok-errors -->*

**@if**(**session**('message-ok'))

                 <div class="alert alert-success">

**{{session**('message-ok')**}}**

                 </div>

**@elseif**(**session**('message-error'))

                 <div class="alert alert-danger">

**{{session**('message-error')**}}**

                 </div>

**@endif**

                <div class="card">

                    <div class="card-header">**{{** **\_\_**('Modificar datos personales: ') **}}**</div>

....

# SUBIR IMAGEN AVATAR USUARIO.

Vamos a añadir un nuevo campo al formulario , y el campo enctype:

<form method="POST" enctype="multipart/form-data"

action="**{{** **route**('user.update') **}}**"

. . .

</form>

*<!--Avatar -->*

<div class="form-group row">

    <label for="image" class="col-md-4 col-form-label text-md-

right">{{ \_\_('Avatar') }}

</label>

    <div class="col-md-6">

      <input type="file" name="image" class="form-control-file"

id="image"

accept="image/png, image/jpeg, image/jpg, image/gif ">

    </div>

    @if($errors->has('image'))

        <span class="invalid-feedback" role="alert">

             <strong>{{ $errors->first('image') }}</strong>

        </span>

    @endif

</div>

<div class="form-group row">

    <label for="image"

         class="col-md-4 col-form-label text-md-right">{{ \_\_(' Preview avatar') }}</label>

    <div class="col-md-6">

        <div class="ml-2 col-sm-6">

            <img src="https://placehold.it/80x80"

id="preview" class="img-thumbnail"></img>

        </div>

    </div>

</div>

Para almacenar archivos en Laravel hacemos uso del storage disk, para ello vamos a crear 2 nuevos discos:

* Users -> Almacenar ficheros de los usuarios
* Images -> Almacenar imágenes del proyecto

En el fichero **config/filesystems.php** añadimos los 2 discos:

*/\* Almacenar imagenes avatar users\*/*

        'users' **=>** [

            'driver' **=>** 'local',

            'root' **=>** **storage\_path**('app/users/avatar'),

            'url' **=>** **env**('APP\_URL')**.**'/storage',

            'visibility' **=>** 'public',

        ],

*/\* Almacenar imagenes del proyecto\*/*

        'images' **=>** [

            'driver' **=>** 'local',

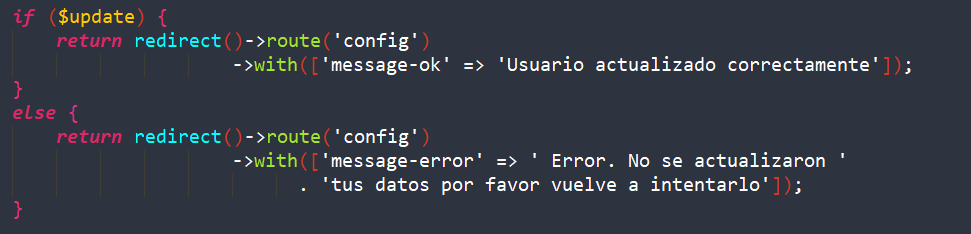
            'root' **=>** **storage\_path**('app/images'),

            'url' **=>** **env**('APP\_URL')**.**'/storage',

            'visibility' **=>** 'public',

        ],

Justo antes de ejecutar el update a la BD en el método update() del controlador de usuario, vamos a verificar y validar el almacenamiento de la imagen



# MOSTRAR IMAGEN AVATAR

Crear un controlador para obtener la imagen de storage

*//\* Obtener la imagen del usuario*

    public ***function*** **getImage**(***$filename***) {

*//\* obtenememos la imagen del disco storage*

***$file*** **=** ***Storage*::disk**('users')**->get**(***$filename***);

*// devolvemos el resultado en una response*

**return** **new** ***Response***(***$file***, 200);

    }

Crear una ruta

*// Obtener imagen avatar*

***Route*::get**('user/avatar/{filename}','UserController@getImage')**->name**('user.avatar');

Vamos a mostrar un div cuando exista la imagen del usuario

*<!-- Preview Avatar -->*

**@if**(**Auth::user**()**->*image***)

<div class="form-group row">

   <label for="image" class="col-md-4 col-form-label text-md-right">

**{{** **\_\_**(' Preview avatar') **}}**</label>

   <div class="col-md-6">

       <div class="ml-2 col-sm-6">

        <img src="**{{route**('user.avatar',['filename'**=>Auth::user**()-**>*image*** ])**}}**"

id="preview" class="img-thumbnail"/>

        </div>

    </div>

</div>

**@endif**

Podemos incluir en un include la imagen del avatar para cuando queramos mostrarla en mas sitios. Para ello en views creamos una carpeta -> includes

NOTA

Finalmente he decidido no crear el include ya que tiene demasiados divs con clases de Bootstrap y no es eficiente para luego mostrar una miniatura del avatar

Si fuera algo asi mas simple:

**@if**(**Auth::user**()**->*image***)

   <div class="ml-2 col-sm-6">

       <img src="**{{route**('user.avatar',['filename'**=>Auth::user**()-**>*image*** ])**}}**"

id="preview" class="img-thumbnail"/>

   </div>

**@endif**

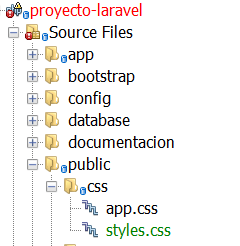
Creamos en includes-> avatar.blade.php

Y lo insertamos en la vista:

**@include(**"includes.avatar">**)**

# AÑADIR HOJA DE ESTILOS.

En la carpeta public vamos a crear un fichero css



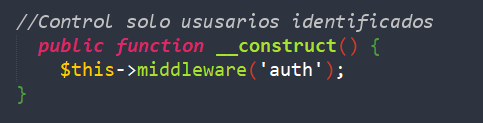
Vamos a añadir en la vista de layout -> **app.blade.php** el enlace al fichero .css

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

# ACCESO SOLO USUARIOS IDENTIFICADOS.

Vamos a utilizar el mismo método que usa laravel en su HomeController para evitar el acceso de usuarios no identificados mediante un middleware en el contructor de la clase.

Asi en nuestro UserController, añadimos dicho middleware:



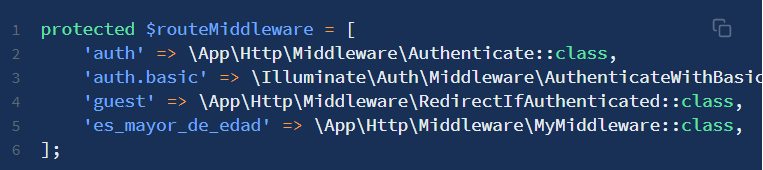
## MIDDLEWARES.

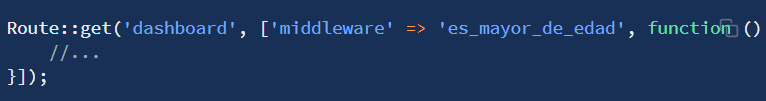
NOTA

Tambien podríamos haber implementado el middleware en la ruta

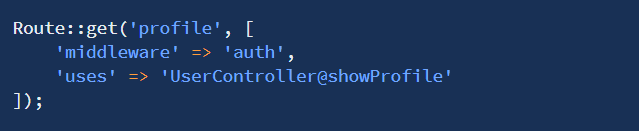
En el caso de querer que nuestro *middleware* se ejecute solo cuando se llame a una ruta o a un grupo de rutas también tendremos que registrarlo en el fichero app/Http/Kernel.php, pero en el array $routeMiddleware. Al añadirlo a este array además tendremos que asignarle un nombre o clave, que será el que después utilizaremos asociarlo con una ruta.

En primer lugar añadimos nuestro filtro al array y le asignamos el nombre "es\_mayor\_de\_edad":

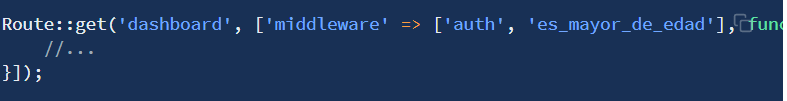




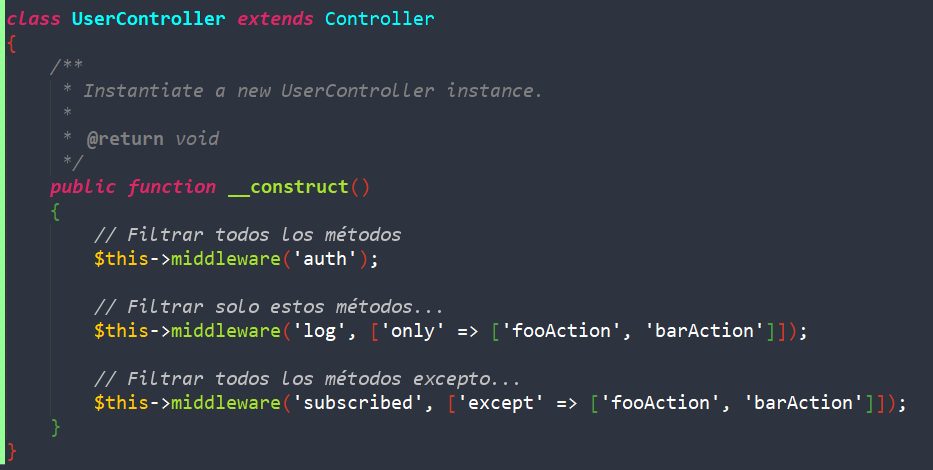
Para asociar un filtro con una ruta que utiliza un método de un controlador se realizaría de la misma manera pero indicando la acción mediante la clave "uses":



Si queremos asociar varios *middleware* con una ruta simplemente tenemos que añadir un array con las claves. Los filtros se ejecutarán en el orden indicado en dicho array:



***Middleware* dentro de controladores**



# FORMULARIO AÑADIR NUEVA IMAGEN.

Vamos a crear una vista, un controlador y una ruta para ello:

Ruta

*//form añadir imagen*

Route*::*get('/image-upload','ImageController@create')*->*name('image.create');

Vista:

@extends('layouts.app')

@section('content')

<div class="container">

    <div class="row justify-content-center">

        <div class="col-md-9">

            <div class="card">

                <div class="card-header">

                    <strong>{{ \_\_('Añadir nueva imagen') }}</strong>

                </div>

                <div class="card-body">

                    <form action="{{route('image.save')}}" method="POST"

enctype="multipart/form-data">

                        @csrf

                        <label>Seleccionar Imagen</label>

                        <div class="custom-file mb-3">

                            <input type="file" class="custom-file- input" id="customFile" name="filename">

                            <label class="custom-file-label" for="customFile">Seleccionar imagen</label>

                        </div>

*<!-- Preview image -->*

                        <div class="form-group">

                            <label><strong>Preview imagen seleccionada:</strong></label>

                            <div class="image-preview">

                                <img class="image-prev" id="image-prev" src="http://placehold.it/180" alt="Tu imagen" />

                            </div>

                        </div>

*<!-- -->*

                        <div class="form-group">

                            <label for="description"><strong>Descripcion:</strong></label>

                            <textarea class="form-control" name="description" id="description" rows="4"></textarea>

                        </div>

                        <div class="form-group">

                            <input type="submit" class="btn btn-primary" value="Subir imagen" name="submit\_image"/>

                        </div>

                    </form>

                </div>

            </div>

        </div>

    </div>

</div>

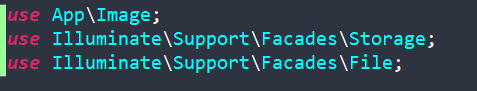
@endsection

Controlador:

Vamos a necesitar cargar el modelo de imagen, asi de esta manera no tenemos que cargar el namespace cuando usemos la clase image para setear los datos por ejemplo:

*use* App\Image;

Ademas vamos a cargar para poder subir la imagen al servidor

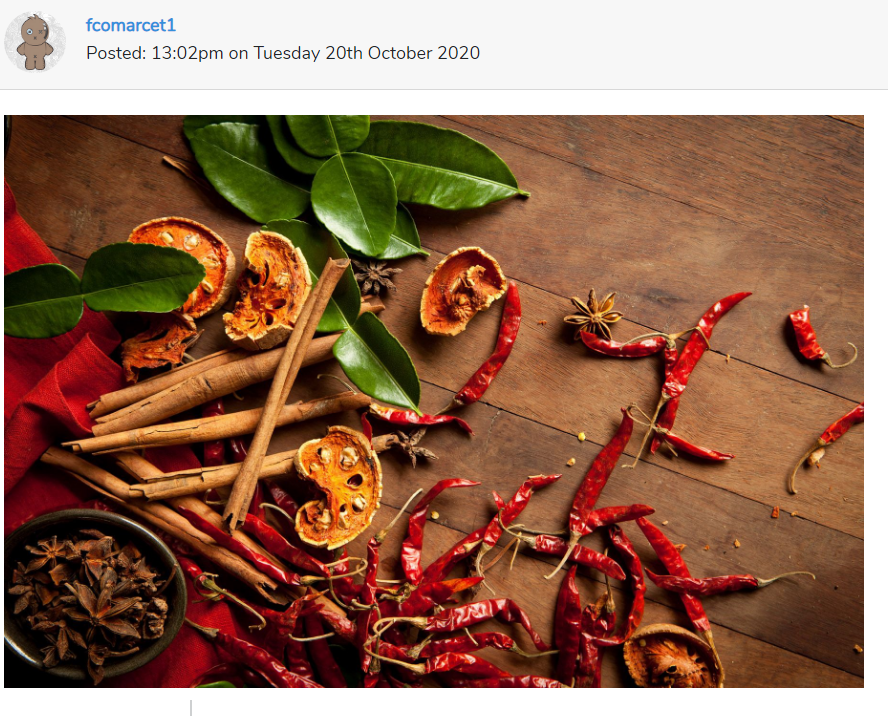




# LISTADO DE IMÁGENES PAGINA PRINCIPAL.

Vamos a listar todas las imágenes de los usuarios en la pagina home de la app.

Seguiremos un formato como el de Instagram



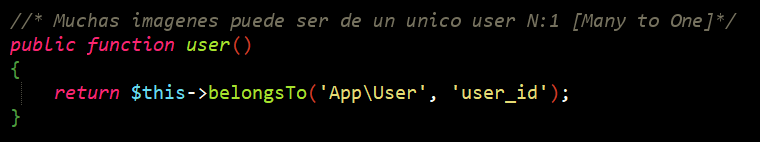
En el controlador de HomeController en el método index vamos a realizar una consulta con el ORM, para obtener todos los registros de la tabla imágenes ordenados por id.



Ya podemos usar la variable $images en la vista [home.blade.php](http://home.blade.php) y recorrerla para obtener las imágenes

**NOTA**

Recordemos que cuando establecimos la relaciones en el ORM creamos un método para obtener los datos del usuario asociados a la imagen en cuestión



Podemos acceder al método user() de la clase image llamándolos como propiedades , y obtener cualquier dato del usuario





## CARD-HEADER CARD(IMG AVATAR+ NICK+FECHA PUBLICACION)

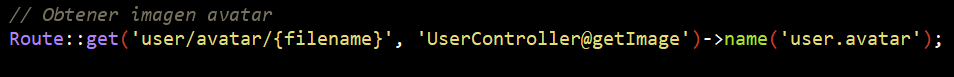
### OBTENER AVATAR

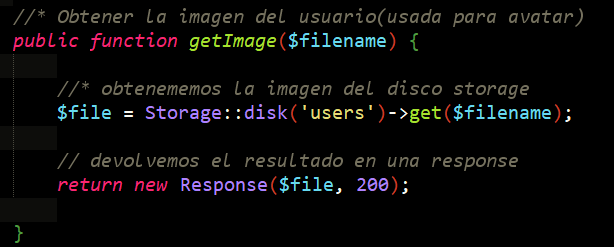
El único problema aparente es mostrar la imagen del avatar la cual la hemos de leer del storage->Disk(image), como hicimos para previsualizar el avatar cuando actualizábamos los datos de un usuario. Pero esta vez en vez de obtener la imagen del usuario identificado , vamos a obtener directamente la imagen del objeto



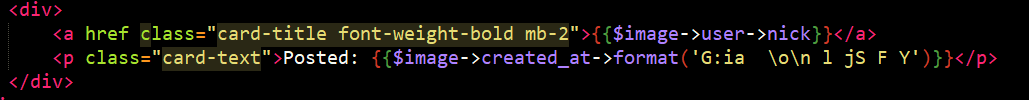
Con esto ya estaría, pero vamos a recordar que hacia la ruta y a que método llamaba NO VOLVER A IMPLEMENTAR

Recordemos que la ruta user.avatar llamaba al método getImage, y este obtenia la imagen de storage y del disk(users)





### OBTENER NICK +FECHA



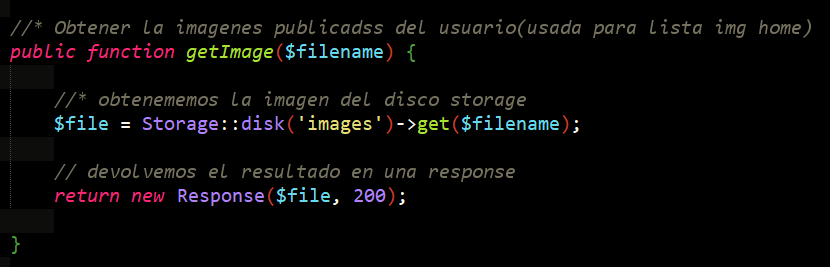
## CARD-BODY (IMAGEN).

Aquí vamos a mostrar la imagen, para ello vamos a crear un método en ImageController para obtener la imagen de storage -> disk(images)

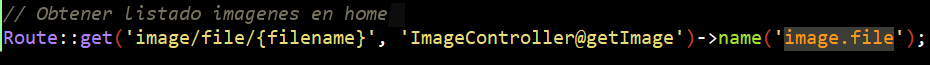
Metodo en ImageController

Vamos a importar el modulo de response





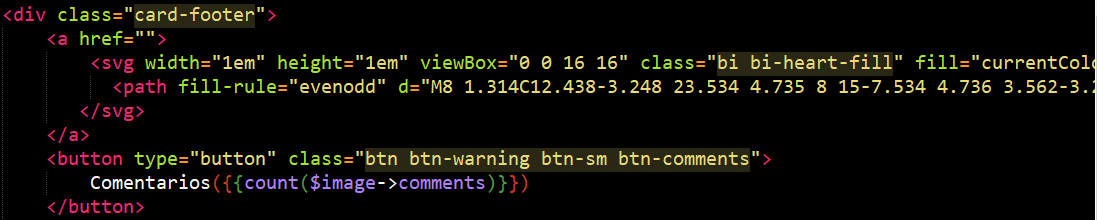
Ruta:



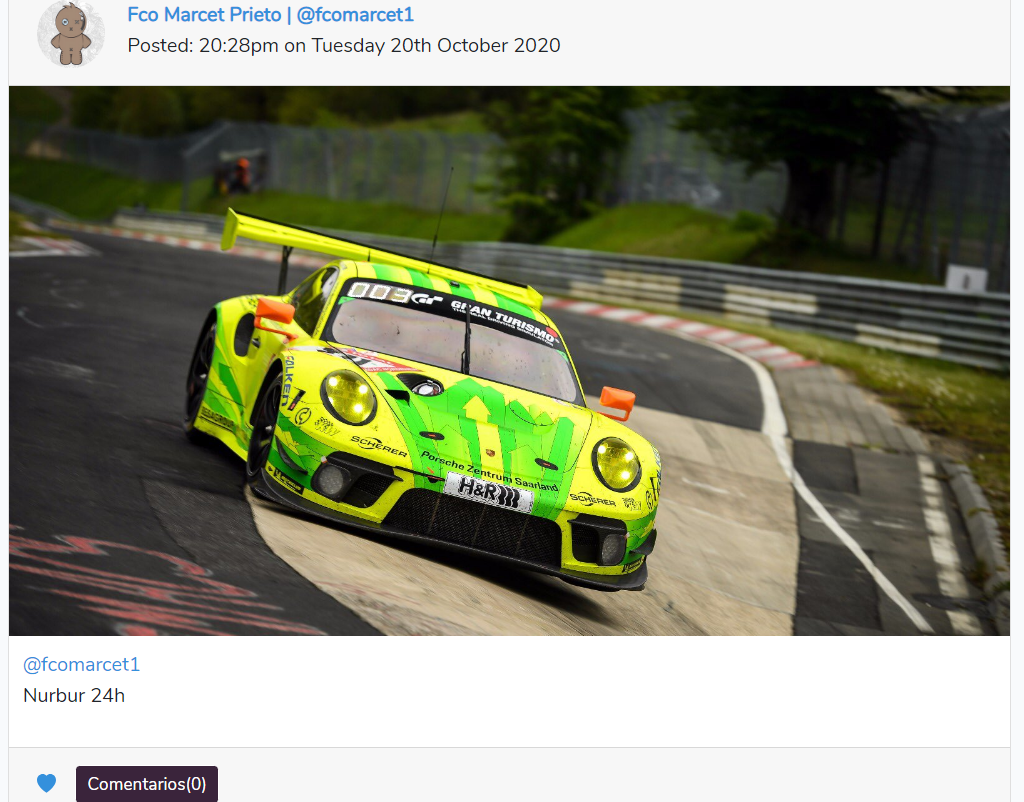
Ahora solo falta imprimir la imagen en la vista



## CARD-FOOTER(LIKES+COMENTARIOS).



Asi quedaría:



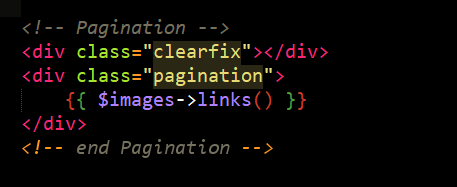
# PAGINACION ENTRADAS PAGINA HOME.

Es muy sencillo en laravel implementar paginaciones, para ello nos vamos al controlador de Home que es donde tenemos el método para obtener el listado de imágenes.

Para ello modificamos el método index(), y cambiamos el método get(), por paginate(N), N-> Nº elementos por pagina.



Ahora en la vista de [home.blade.php](http://home.blade.php) añadimos el método link()



# DETALLE DE IMAGEN.

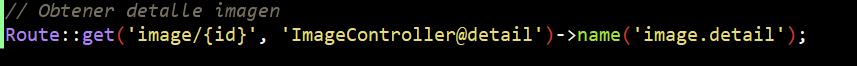
Al pulsar sobre una imagen vamos a ver la imagen en mas detalle asi como sus comentarios asociados. Para ello vamos a crear un nuevo método en imageController -> detail($id) , el cual recibirá el id de de imagen.

**NOTA**

Vamos a utilizar el método find(), al cual le podemos pasar un id , y nos devuelve un objeto cuya imagen sea la referente al id indicado



**Ruta:**



**Vista(detail.blade.php):**

Vamos a crear un enlace, para cuando pulsemos sobre la imagen nos lleve al detalle de esta :

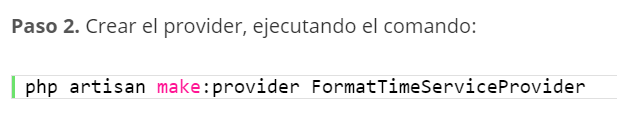


# PROVIDERS/helpers EN LARAVEL - FORMATO FECHA.

PASO 1:

En app vamos a crear una carpeta -> Helpers->FormatTime.php

El código lo tenemos en : <https://victorroblesweb.es/2018/01/18/crear-helpers-en-laravel-5/>







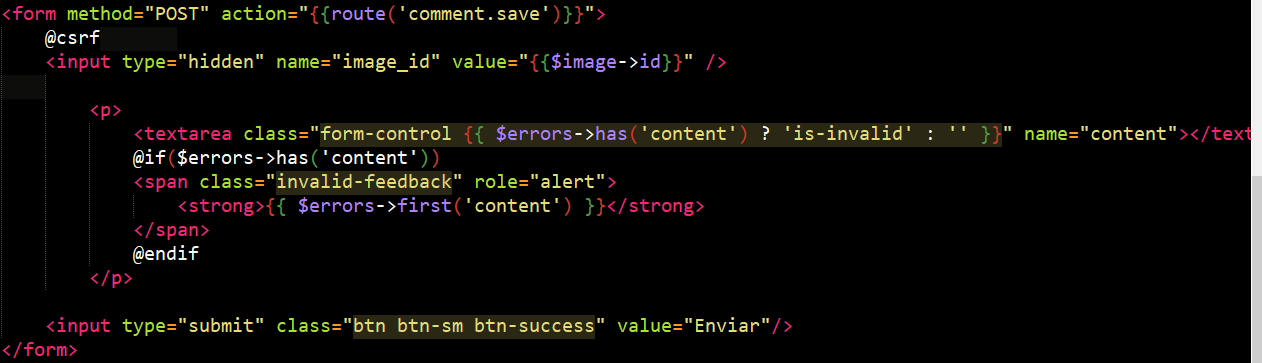
# COMENTARIOS.

## FORMULARIO + AÑADIR COMENTARIOS.

Vamos a insertar un simple formulario con los siguientes campos:

* Campo type hidden para el imagen\_id
* Textarea para añadir el comentario.

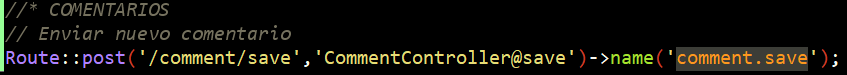
En el textarea meter también mostrar error para cuando este vacio por ej:



Como se trata de una entidad diferente a las anteriores vamos a crear un nuevo controlador:



**Ruta:**



En controlador CommentController, método save()

Para poder usar la entidad Comment vamos a cargar la clase:

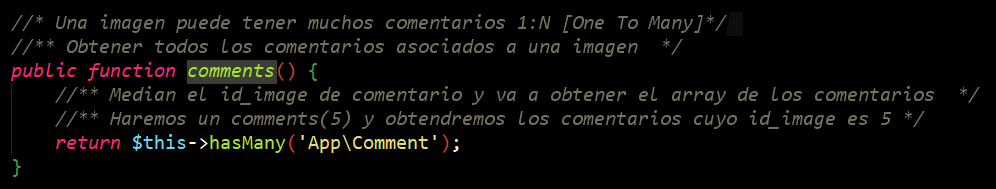


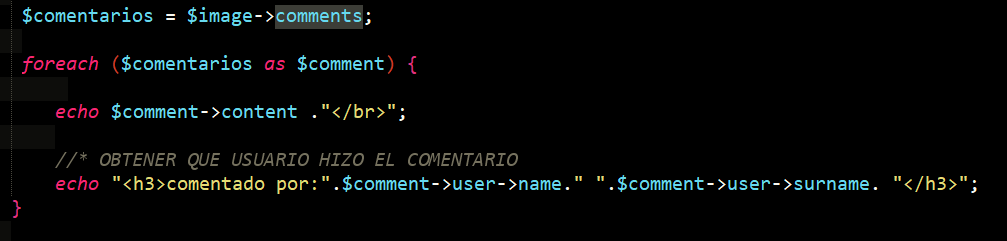
## LISTAR COMENTARIOS

Listaremos los comentarios de momento en el detalle de la imagen ->views/image/detail.blade.php

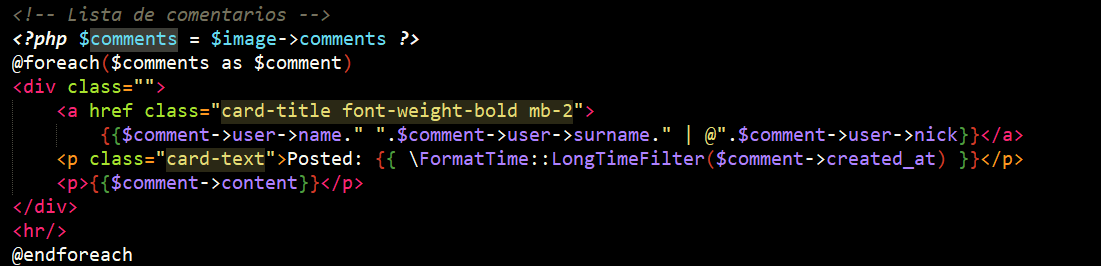
Vamos a usar el método que definimos en el ORM de la entidad Image Mediante el cual relacionamos la entidad comment con el id\_imagen para obtener todos los comentarios de una determinada imagen



Recordemos que podemos acceder al método como si fuera una propiedad, dado que devuelve un array con varios result set hemos de recorrerlos en la vista()

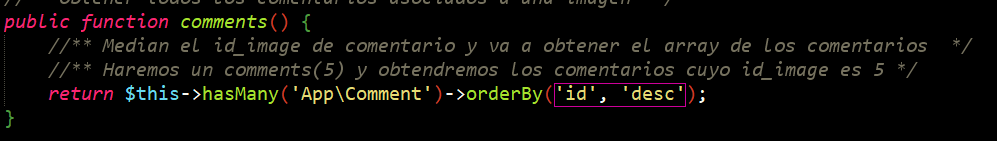


En la vista quedaría:



## ORDENAR COMENTARIOS.

Nos interesa que los comentarios estén ordenados, mostrando los mas recientes los primeros. Para ello vamos a modificar el modelo de Image, en la relación que tenemos con comentarios meter un order by



## ELIMINAR COMENTARIO.

Vamos a poder eliminar comentarios si somos el usuario que ha publicado el comentario o si somos el propietario de la imagen publicada.

Para ello vamos a crear un método en CommentController llamado **delete($id).**

Hemos de tener en cuenta que solo podremos eliminar mensajes si somos los propietarios del comentario o de la publicación

**Propietario del comentario:**

* Id(Usuario logueado) == user\_id(comentario)



**Propietario de la imagen publicada:**

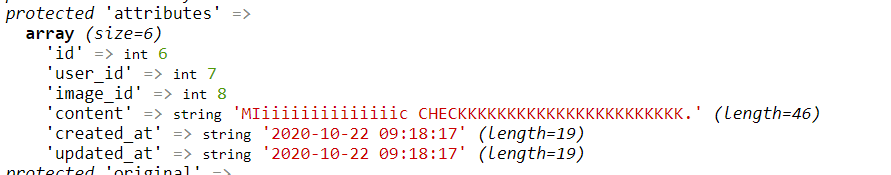
* Id(Usuario logueado) == user\_id(Imagen)



### OBTENER OBJETO DEL COMENTARIO



Si hacemos un var\_dump($comment)



$comment

Id=6, user\_id= 7, image\_id = 8 …

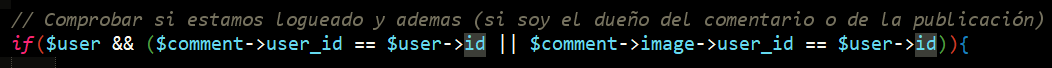
$comment->user

Id=7, name=Fco, surname=Marcet, email= [fcomarect@mail.es](mailto:fcomarect@mail.es) …

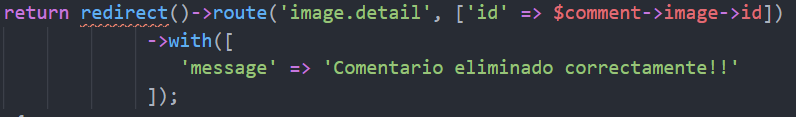
$comment->image

Id=8, user\_id= 7, path=image.png, …

Asi que la condición quedaría:



Si se cumplen ejecutamos el método **delete()**, y una redirección y si no un redirección al detalle de la imagen pasándole el id de la imagen a través del método **image()** de la clase Comment

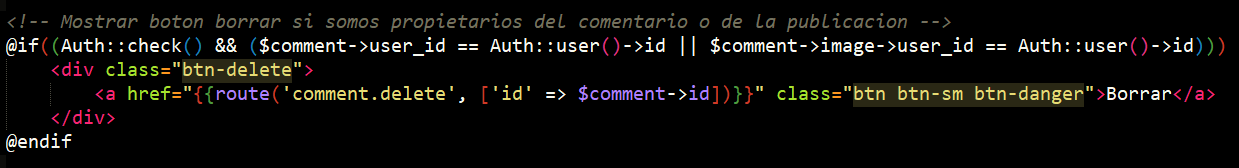


Metodo delete() del controlador CommentController





En la vista :



# GESTION DE LIKES.

**NOTA**

En esta parte para que el icono de like cambie de color vamos a tratarlo con AJAX para que cambie de color etc…, asi que devolveremos tras ejecutar la lógica del controlador una respuesta con un Json para su posterior manipulación.

## METODO LIKE.

Al pulsar sobre el icono de like vamos a llamar a la acción del LikeController -> like($image\_id), este se encargará:

* Obtener al usuario que pulsa Like
* Validar que no le ha dado ya like a esa publicación.
* Almacenar en la BD los registros en la tabla Likes
* Devolver una response con un Json para su posterior manipulación con AJAX

**Ruta:**



**Metodo -> like($image\_id):**

public *function* like($image\_id) {

        //\*Obtener datos del usuario

        $user = \*Auth*::user();

        $user\_id = $user->id;

        //\*Validar que el like no existe en la BD.

        //\* Consulta tbl likes where id\_user=$id\_user AND image\_id=$image\_id

        $isset\_like = *Like*::where('user\_id', '=', $user\_id)

                ->where('image\_id', '=', $image\_id)

                ->count();

        if ($isset\_like == 0) {

            //Setear valores obj

            $like = new *Like*();

            $like->user\_id = $user\_id;

            $like->image\_id = (*int*)$image\_id;

            //guardar datos

            $save = $like->save();

            //var\_dump($like); die();

            if ($save) {

                // insert Ok devolvemos Json para manipularlo con AJAX

                return response()->json([

                    'like' => $like,

                    'message' => 'Has dado like!!'

                ]);

            }

            else {

                //error al guardar datos en la bd

                return redirect()->route('image.detail',['id' => $image\_id]);

            }

        }

        else {

            //Ya existe el like en la BD devolvemos Json

            return response()->json([

                    'message' => 'El like ya existe!!'

                ]);

        }

    }

## METODO DISLIKE.

PASOS:

* Obtener usuario que pulsa dislike.
* Consulta a la BD para obtener el registro
* Eliminar registro
* Devolver response con json.

**Ruta:**



**Método:**

    public *function* dislike($image\_id) {

        //\*Obtener datos del usuario

        $user = \*Auth*::user();

        $user\_id = $user->id;

        //Obtener registro del like

        $like = *Like*::where('user\_id', '=', $user\_id)

                ->where('image\_id', '=', $image\_id)

                ->first();

        if ($like) {

            //eliminamos registro like de la BD

            $delete = $like->delete();

            if ($delete) {

                //Delete OK

                return response()->json([

                    'dislike' => $like,

                    'message' => 'Has dado dislike!!!'

                ]);

            }

            else{

                // Error al efectuar delete() en la BD

                return redirect()->route('image.detail',['id' => $image\_id]);

            }

        }

        else{

            //No exixte registro en la BD

            return response()->json([

                'message' => 'El like no existe'

            ]);

        }

 }

## DETECTAR LIKES.

Cuando demos like a una publicación esta ha de tener el icono del corazón en rojo. Para ello tenemos que comprobar si existe el like y si además coincide con el usuario identificado.

Podemos acceder a los likes mediante:



Vamos recorrer el objeto y verificar cuando exista y sea del usuario identificado en cuestión.



## CARGAR FICHERO JS.

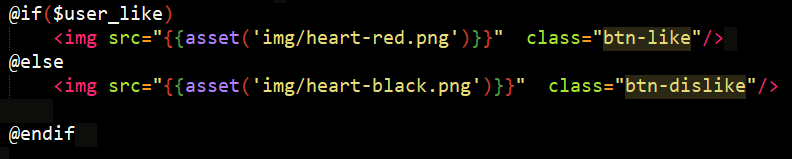
Vamos a añadir un nuevo fichero js en app.blade.php



## CAMBIAR COLOR BOTON LIKE.

Vamos a utilizar JQuery para cambiar el botón de like al pulsar, además que ya viene cargado en Bootstrap de laravel, vamos a crear el fichero main.js

Estos son los botones que deseamos cambiar:

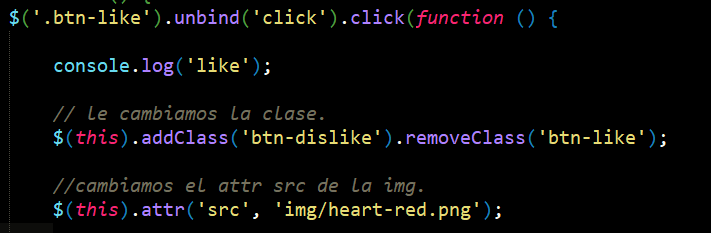


Vamos a comentar que hace el código JQuery:

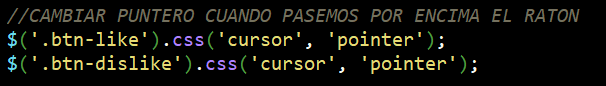
1. Añadimos en window un evento load, mediante una función de callback.



1. Seleccionamos el botón con class **btn-like**, y añadimos el evento onclick, en el cual vamos a cambiarle la clase y la ruta de la imagen. Con **unbind** evitamos que se acumulen los eventos al pulsar click



1. Vamos cambiar el puntero del raton al pasar por encima



Hasta aquí si pulsamos el botón cambia de color pero necesitamos que al volver a pulsar pueda volver al estado inicial(negro), para ello necesitamos volver a detectar el DOM y volver a ejecutar el código para que pueda leer los nuevos cambios en las clases.

Vamos a meter el código en 2 funciones, para que se carge el código al iniciarse la pagina.

Para que al volver a pulsar cambie otra vez hemos de añadir la función contraria





Por ultimo necesitamos que se realice la petición AJAX para almacenar el like en la BD;

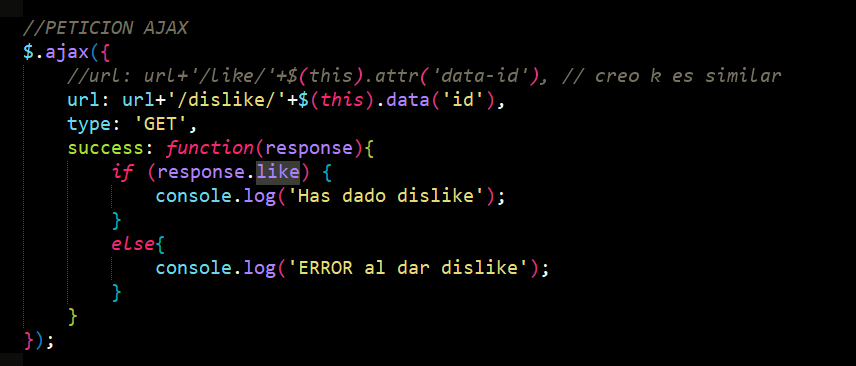
## PETICIONES AJAX.

Añadimos la petición AJAX

Like()



Para el dislike es similar pero cambiamos la ruta



Asi el código final quedaría:

*var* url = 'http://proyecto-laravel.com.devel';

window.addEventListener("load", *function* () {

    //test Jquery

    //alert("La pagina esta completamente cargada");

    //$('body').css('background','red');

    //CAMBIAR PUNTERO CUANDO PASEMOS POR ENCIMA EL RATON

    $('.btn-like').css('cursor', 'pointer');

    $('.btn-dislike').css('cursor', 'pointer');

    //BOTON LIKE

*function* like() {

        $('.btn-like').unbind('click').click(*function* () {

            console.log('like');

            // le cambiamos la clase.

            $(this).addClass('btn-dislike').removeClass('btn-like');

            //cambiamos el attr src de la img.

            $(this).attr('src', url+'/img/heart-red.png');

            //PETICION AJAX

            $.ajax({

                //url: url+'/like/'+$(this).attr('data-id'), // creo k es similar

                url: url+'/like/'+$(this).data('id'),

                type: 'GET',

                success: *function*(*response*){

                    if (response.like) {

                        console.log('Has dado like');

                    }

                    else{

                        console.log('ERROR al dar like');

                    }

                }

            });

            dislike();

        })

    }

    like();

    //BOTON DISLIKE

*function* dislike() {

        $('.btn-dislike').unbind('click').click(*function* () {

            console.log('dislike');

            // le cambiamos la clase.

            $(this).addClass('btn-like').removeClass('btn-dislike');

            //cambiamos el attr src de la img.

            $(this).attr('src', url+'/img/heart-black.png');

            //PETICION AJAX

            $.ajax({

                //url: url+'/like/'+$(this).attr('data-id'), // creo k es similar

                url: url+'/dislike/'+$(this).data('id'),

                type: 'GET',

                success: *function*(*response*){

                    if (response.like) {

                        console.log('Has dado dislike');

                    }

                    else{

                        console.log('ERROR al dar dislike');

                    }

                }

            });

            like();

        })

    }

    dislike();

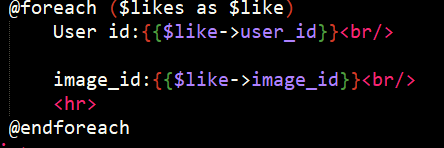
});

## LISTAR LIKES

Vamnos a tener un listado con las publicaciones a las cuales les hemos dado like.

Para ello vamos a crear un nuevo método en LikeController -> index();

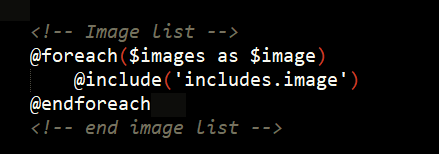
Este método será muy parecido a cuando listábamos todas las imágenes en home:



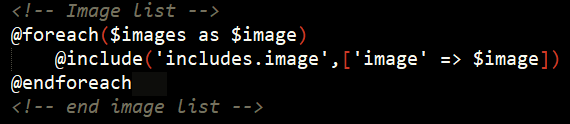
Como el código se va a repetir y es similar a home donde listábamos las imágenes vamos a crear un include, asi poder usarlo en cualquier listado

Asi que cogemos el código de la card donde se muestra la imagen y lo separamos en un include

### INCLUDE image.blade.php

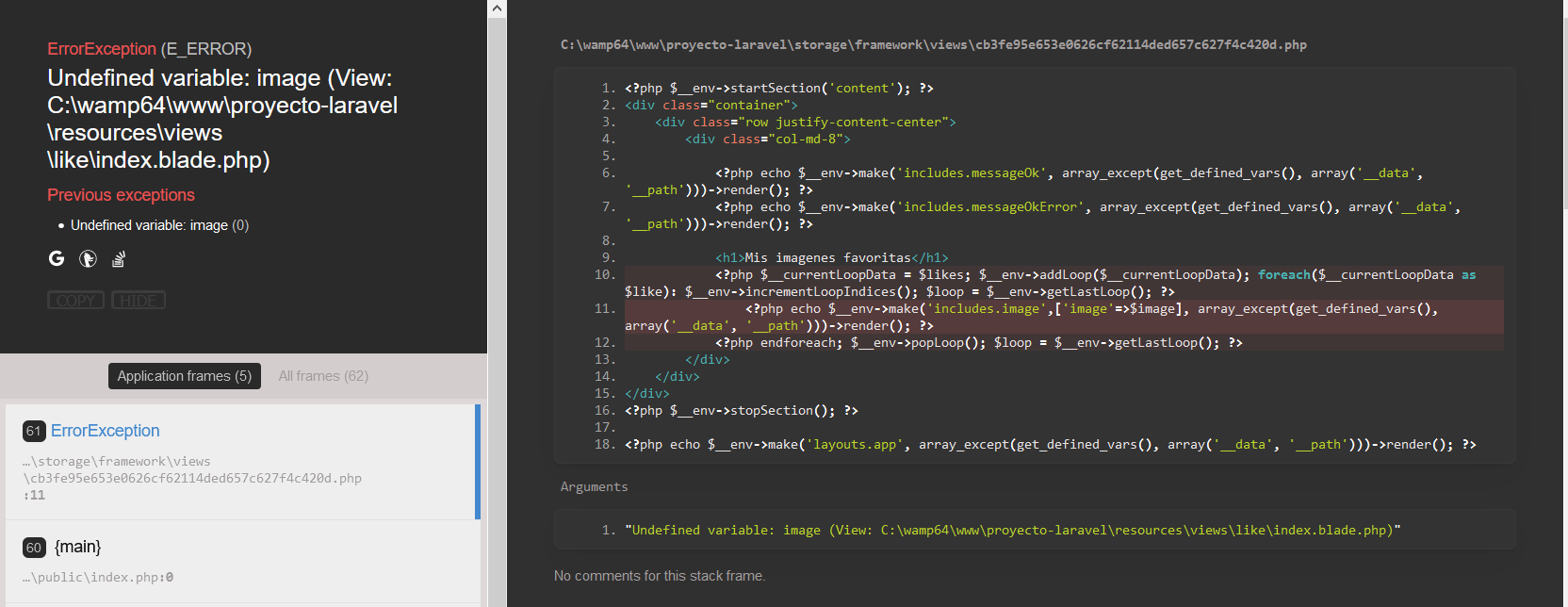


OJO SI HACEMOS ESTO ASI -> ERROR, ya que necesitamos pasarle al include $image para que funcione correctamente.



OJO

Ahora en la vista index() de likes si añadimos el include directamente tendremos un error, ya que no encuentra la el objeto imagen.

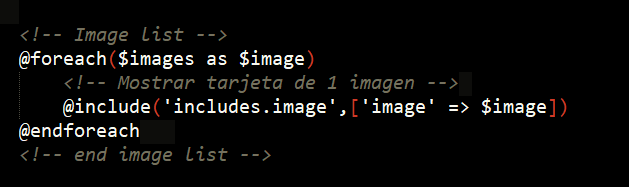


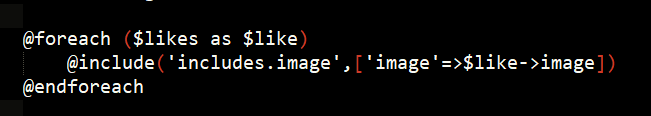
Para solucionar esto hemos de modificar el parámetro k le pasamos por el include



De esta manera se queda mucho mas limpio el código, en home recorremos $images y listamos las imágenes y en index(likes)

Recorremos los likes y mostramos las imágenes.



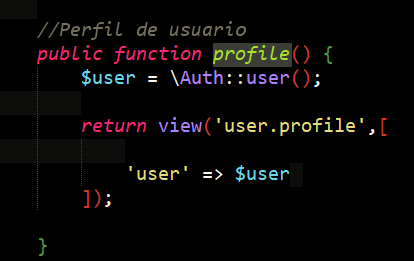


# PERFIL DE USUARIO.

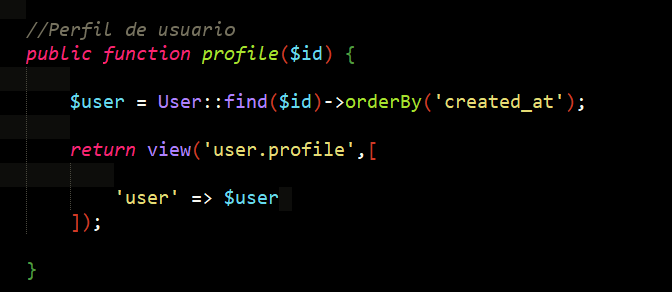
Vamos a crear un nuevo método en nuestro controlador UserController -> profile()

Nota

En principio si solo quisiéramos mostrar sus datos accediendo a \Auth::user() podríamos obtenerlo,



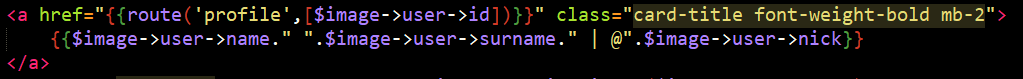
pero como además queremos listar sus publicaciones vamos a realizar una consulta pasándole un $id por la URL(GET), y por ultimo devolvemos el objeto a la vista:



Cambiamos los enlaces de las demás pagina para acceder al perfil en views/layout/app.blade.php :



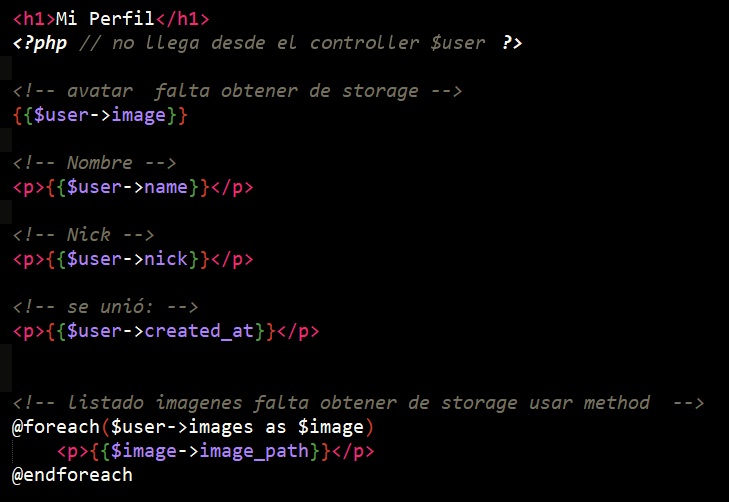
De la misma manera en el includes/image.blade.php, para cuando pulsemos sobre el nombre



## MOSTRAR DATOS EN EL PERFIL.

La pagina será similar a Home pero mostrando las imágenes del usuario, además de los datos del usuario

A la vista nos llega el objeto $user



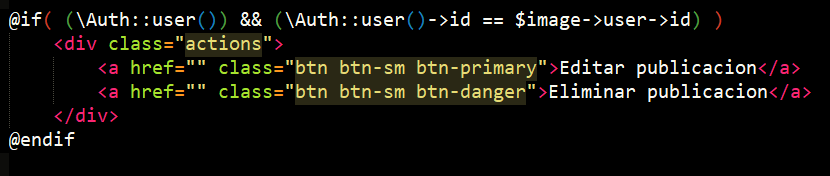
Asi quedaría la vista:

<https://gitlab.com/fcomarcet1/proyecto-laravel/-/blob/master/resources/views/user/profile.blade.php>

# BORRADO Y EDICION DE PUBLICACIONES.

## BOTONES DE BORRADO Y EDICION.

Vamos a añadir en el detalle de imagen para eliminar y editar un publicación. Estos botones solo podrán ver si si somos el autor de la publicación:



## BORRAR PUBLICACION.

Añadimos al ImageController un nuevo método delete($id).

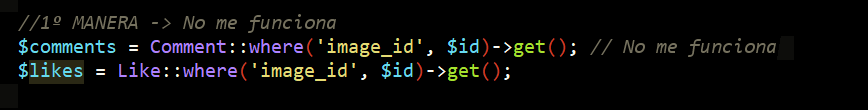
Para eliminar una publicación también hemos de en primer lugar eliminar los comentarios y los likes asociados a la publicación ya que por la integridad referencial no podremos eliminar directamente la imagen.

Esto se podría realizar de varias formas:

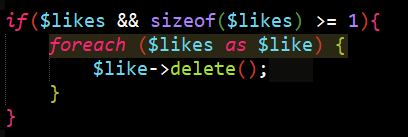
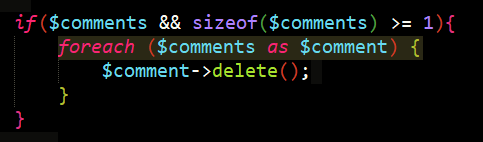
* Obtener los registros mediante consultas mediante el ORM y luego recorrer cada objeto obtenido de esas consultas y ejecutar delete().
* Usando directamente funciones del ORM.
* Aplicando esta función desde el Modelo Image para que en cuanto se borre la imagen en cascada se borre lo demás

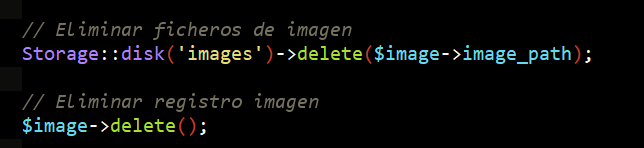
1º METODO:

Este método me devuelve un objeto vacio NO ME FUNCIONA BIEN



Una vez que tenemos los comentarios y likes los recorremos;



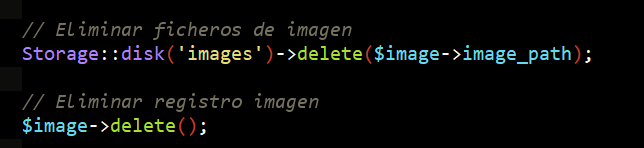


2º METODO

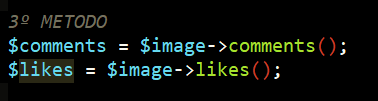
Si Usamos el Query builder no podemos acceder al método delete(), hemos de usar el delete en la misma query

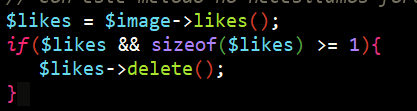
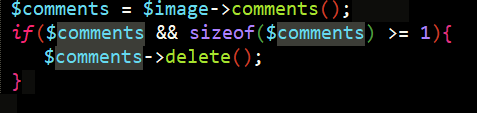


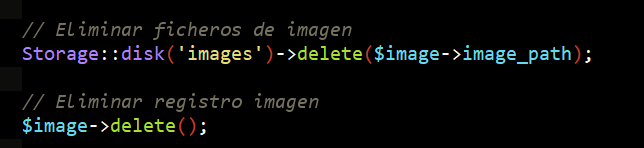




3ºMETODO (No es necesario un foreach)

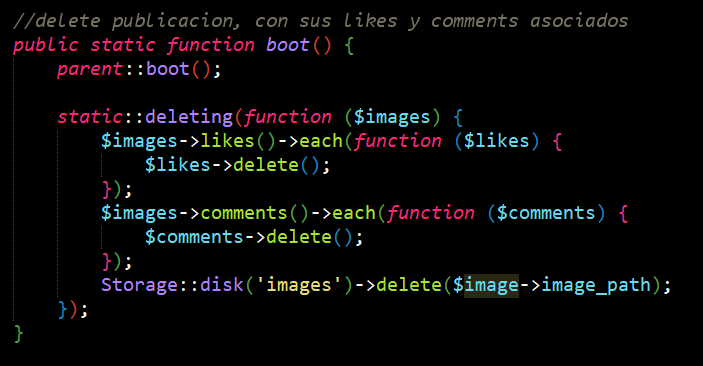




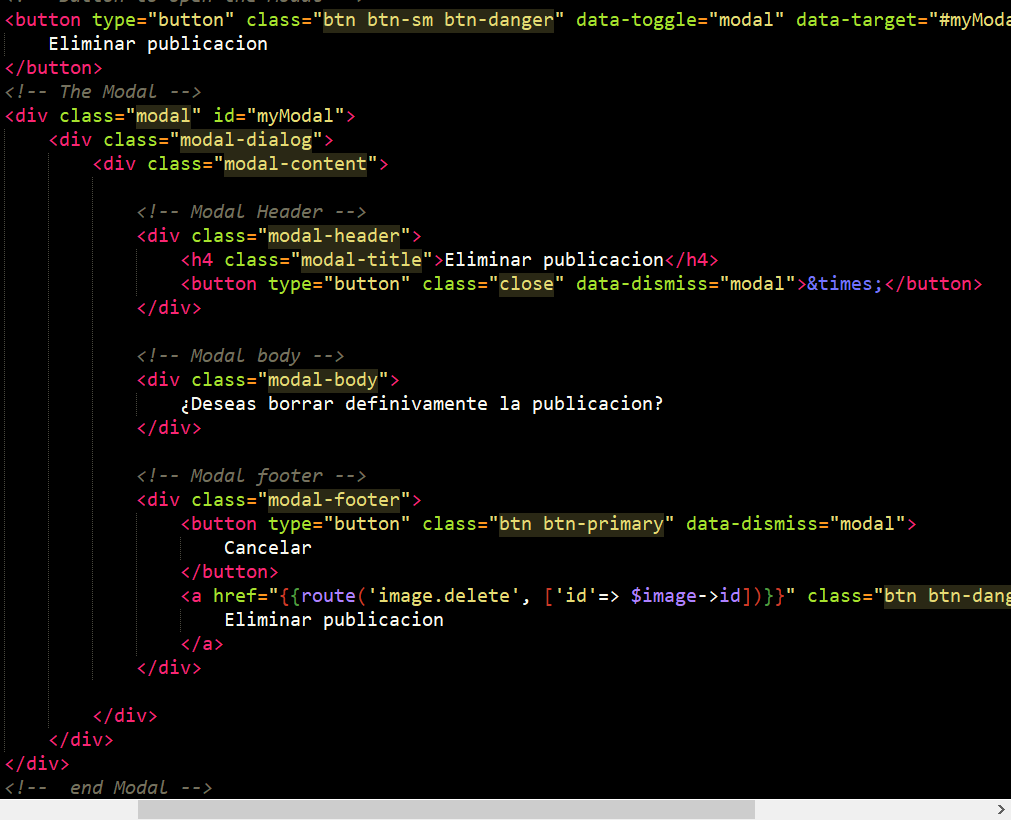


4º METODO (Modificar modelo Image)

Añadimos un nuevo método al modelo Image

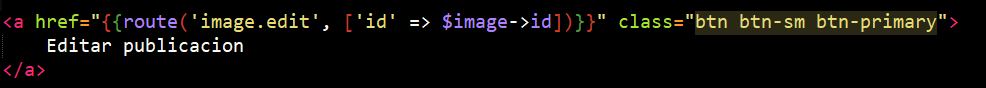


# MODAL ELIMINAR PUBLICACION.



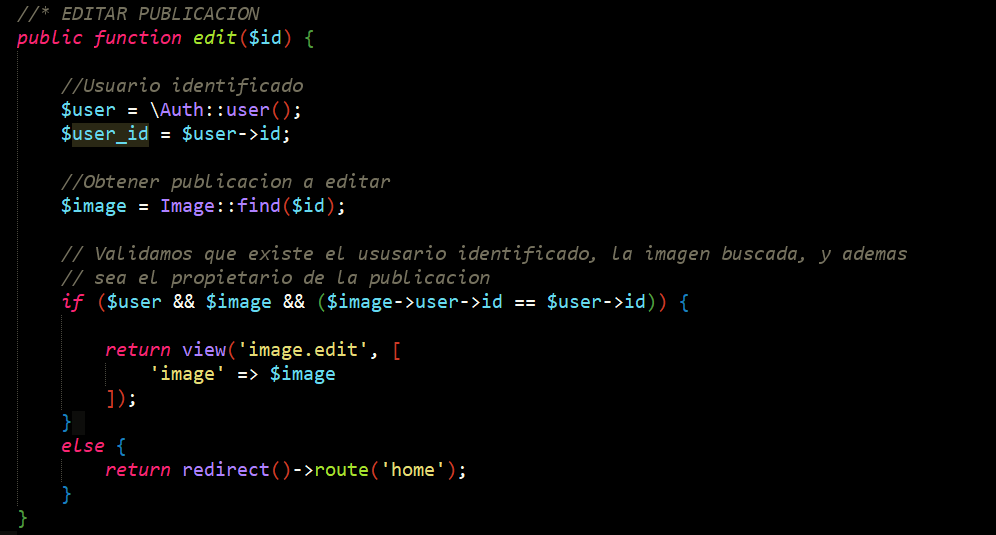
## EDITAR PUBLICACION.

Vamos a crear un nueva ruta que apuntara el botón de editar publicación





Añadimos un nuevo método a ImageController ->edit($id), en el cual obtenemos la imagen indicada y se la pasamos a la vista:



La vista será un simple formulario para añadir una imagen , descripción y una preview de la foto:

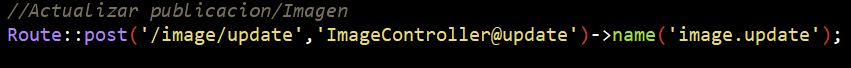
<https://gitlab.com/fcomarcet1/proyecto-laravel/-/blob/master/resources/views/image/edit.blade.php>

Por ultimo necesitamos hacer un update en la BD , en el cual necesitamos:

* image\_id -> Campo type hidden en el formulario.
* image\_path
* description

Hemos de tener en cuenta que no es obligatorio enviar la imagen por el formulario asi que podría llegar con valor null, en cualquier caso vamos a realizar una consulta mediante un find,

Ruta:



El formulario llamara al método update() de ImageController.



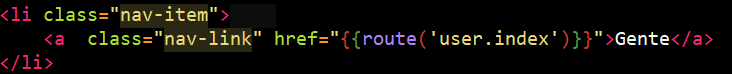


# GENTE Y BUSCADOR.

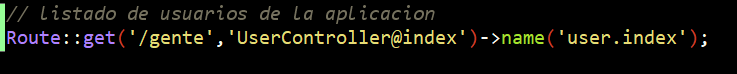
## GENTE.

Vamos a poder listar todos los perfiles de los usuarios de la aplicación.

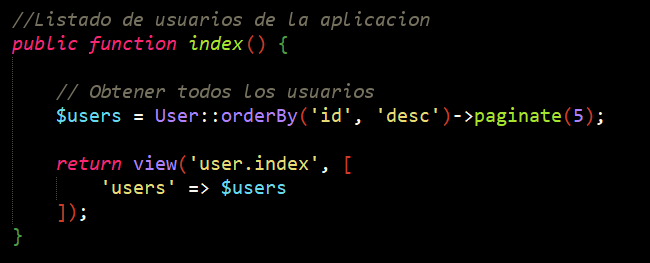
Añadimos el enlace en views/layout/app.blade.php



Ruta:



En el controlador añadimos el método **index()** realizamos una consulta para obtener todos los usuarios y los devolvemos a la vista



Por ultimo en la vista simplemente hemos de recorrer el array con el objeto:

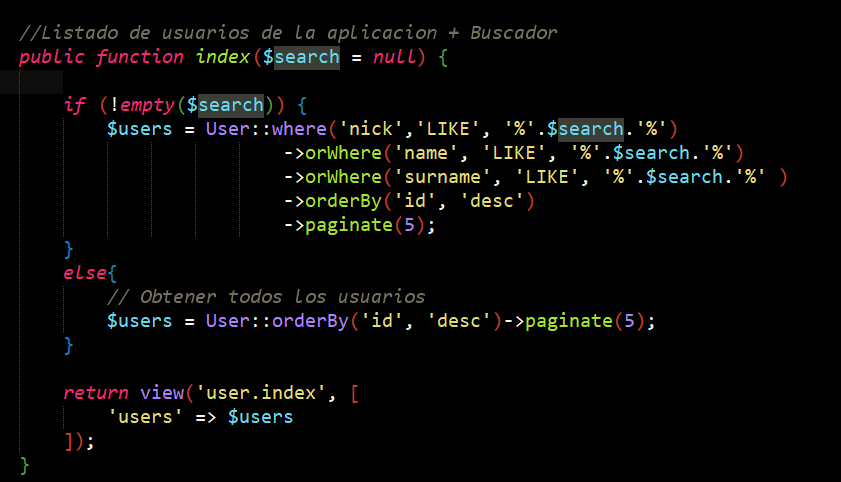
Ver-> views/user/index.blade.php

## BUSCADOR.

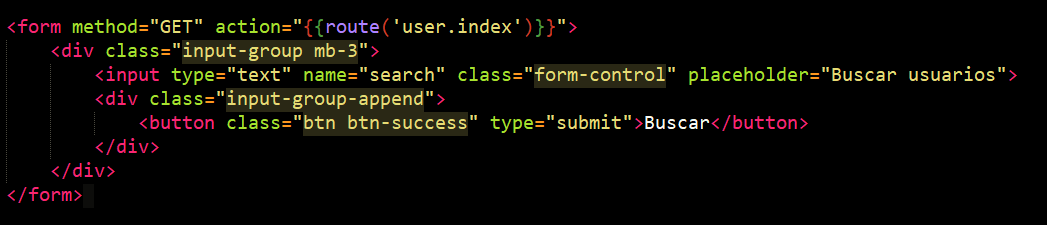
Vamos a implementar un buscador de perfiles en la aplicación. Para ello vamo a modificar la ruta de ver todos los perfiles de la aplicación añadiendo un parámetro opcional search.



Este parámetro search también se lo pasamos al método index como parámetro opcional search = null, en función de este parámetro pues ejecutaremos una consulta u otra.



Solo nos falta implementar un formulario para el buscador



Tambien vamos a necesitar que cuando enviemos la información por el buscador esta de añada a la url para que le llege al controlador para ello vamos a usar Javascript

Nuestro form tiene un id=buscador, el cual usaremos

Añadimos en main.js

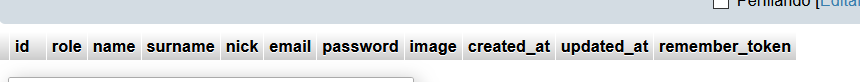
Lo he intentado con js pero no me coge bien la url asi que he implementado otro método que recibe el campo del buscador por poSTy ya efectua la consulta y enviamos los datos a la nueva vista search.blade.php

# AUTENTICACION EN LARAVEL 6

Vamos a crear un nuevo proyecto en laravel 6 :



Creamos una tabla users:



Por ultimo configuramos fichero .env

CAMBIOS RESPECTO LA VERSION 5.8

* Instalacion laravel ui





Tambien vamos a instalar node.js para el gestor de paquete de npm

Para acabar de visualizar bien los estilos:

# AUTENTICACION EN LARAVEL 8