LARAVGRAM

Desarrolado con Laravel.

DESCRIPCIÓN BREVE

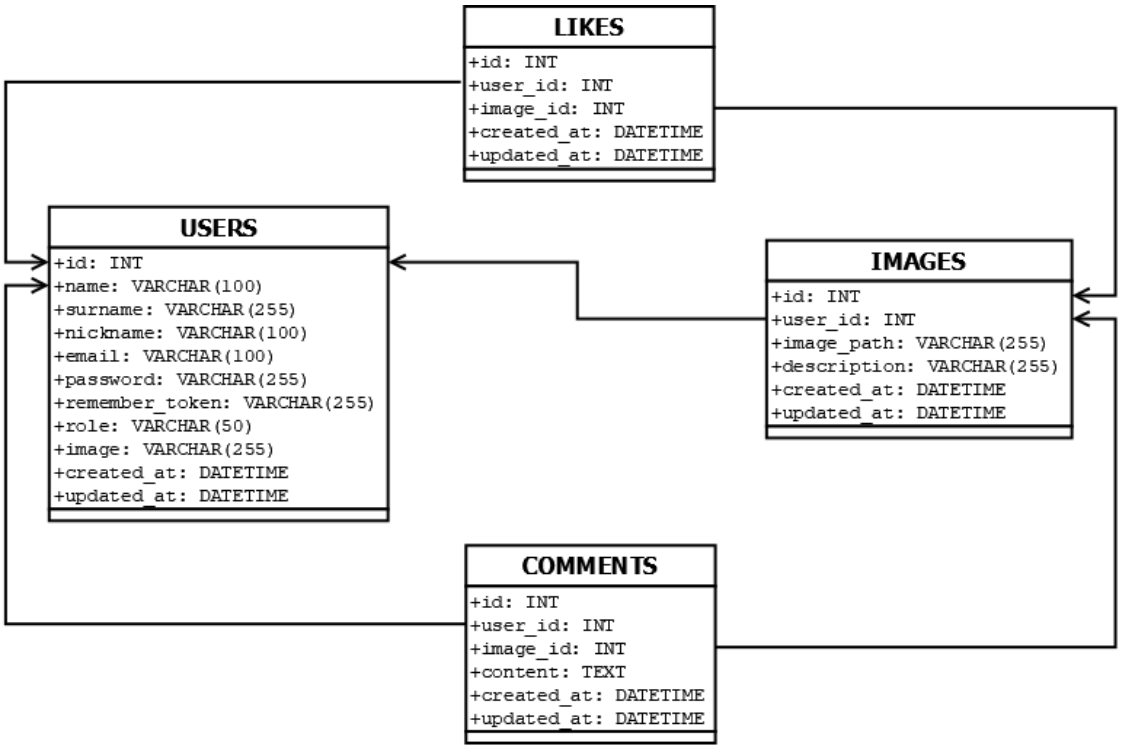
[Capte la atención del lector con un resumen atractivo. Este resumen es una breve descripción del documento. Cuando esté listo para agregar contenido, haga clic aquí y empiece a escribir.]

Fco Marcet

[Título del curso]

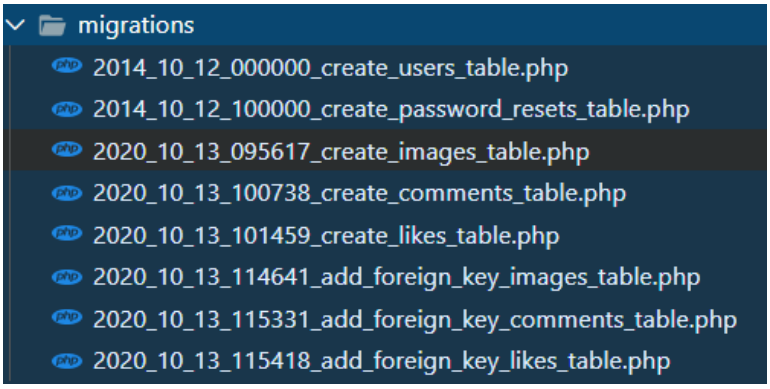
CONTENIDO

1. DISEÑO BASE DE DATOS



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;
        ");
```

```php
public function down()
    {
        Schema::dropIfExists('users');
    }
```

```php
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.

```
$ php artisan make:model Image
Model created successfully.
```

¿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
<?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 com
entarios  */
       //** 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:

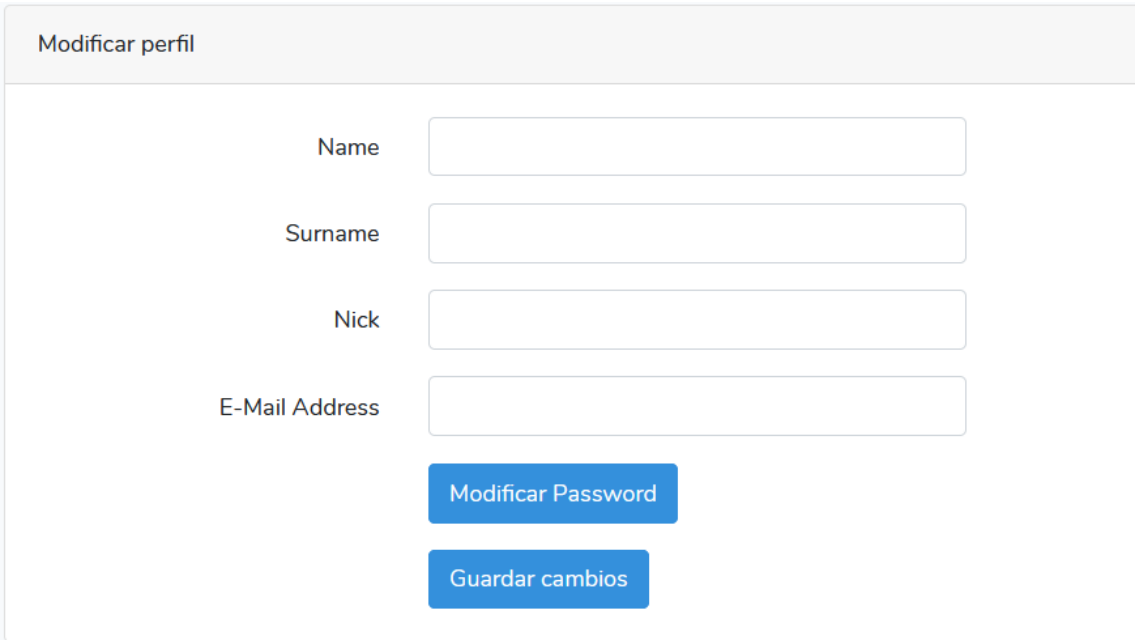
```
Route::get('/configuracion','UserController@config')-> name('config');
```

Controlador:

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

```php
class UserController extends Controller
{
    public function config()
    {
        return view('user.config');
    }
}
```

Vista:



RELLENAR DATOS DEL FORMULARIO

Usaremos el método estatico que tiene laravel:

```php
Auth::user()
```

En el cual podemos acceder a las diferentes propiedades

```php
Auth::user()->nick
```

Podemos usarlos en el campo value del form:

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

Al enviar el formulario ejecutamos el la ruta:

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

RECIBIR DATOS FORMULARIO EN EL CONTROLADOR.

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

```php
 //* 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

```php
//** ALMACENAR IMAGEN AVATAR
//Comprobamos que llega la imagen en formato file
if ($request->hasFile('image')) { //check file is getting or not..
    // Almacenamos la imagen
    $image = $request->file('image');

    //comprobamos que el objeto image no es null
    if ($image) {

        // Obtenemos un nombre de imagen unico
        $image_path_unique = time() . "_" . $image->getClientOriginalName();

        //Obtenemos la imagen con el metodo estatico get() de la clase File
        $image_to_save = File::get($image);

        //Usamos la clase storage y su metodo estatico disk para almacenar la imagen
        // con el metodo put en storage/users/avatar
        Storage::disk('users')->put($image_path_unique, $image_to_save);

        //Setear imagen para el update con el nombre unico
        $user->image = $image_path_unique;
    }
}
//* ejecutar consulta y cambios en la BD
$update = $user->update();
```

```php
if ($update) {
    return redirect()->route('config')
                    ->with(['message-ok' => 'Usuario actualizado correctamente']);
}
else {
    return redirect()->route('config')
                    ->with(['message-error' => ' Error. No se actualizaron '
                          . 'tus datos por favor vuelve a intentarlo']);
}
```

MOSTRAR IMAGEN AVATAR

Crear un controlador para obtener la imagen de storage

```php
//* 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
```

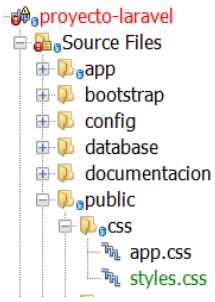
Creamos en includes-> avatar.blade.php

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:

```
//Control solo ususarios identificados
  public function __construct() {
    $this->middleware('auth');
}
```

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":

```
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasic
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'es_mayor_de_edad' => \App\Http\Middleware\MyMiddleware::class,
];
```

```
Route::get('dashboard', ['middleware' => 'es_mayor_de_edad', function ()
    //...
}]);
```

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":

```
Route::get('profile', [
    'middleware' => 'auth',
    'uses' => 'UserController@showProfile'
]);
```

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:

```
Route::get('dashboard', ['middleware' => ['auth', 'es_mayor_de_edad'],fun
    //...
}]);
```

***Middleware* dentro de controladores**

```
class UserController extends Controller
{
    /**
     * Instantiate a new UserController instance.
     *
     * @return void
     */
    public function __construct()
    {
        // Filtrar todos los métodos
        $this->middleware('auth');

        // Filtrar solo estos métodos...
        $this->middleware('log', ['only' => ['fooAction', 'barAction']]);

        // Filtrar todos los métodos excepto...
        $this->middleware('subscribed', ['except' => ['fooAction', 'barAction']]);
    }
}
```

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:</stro
ng></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

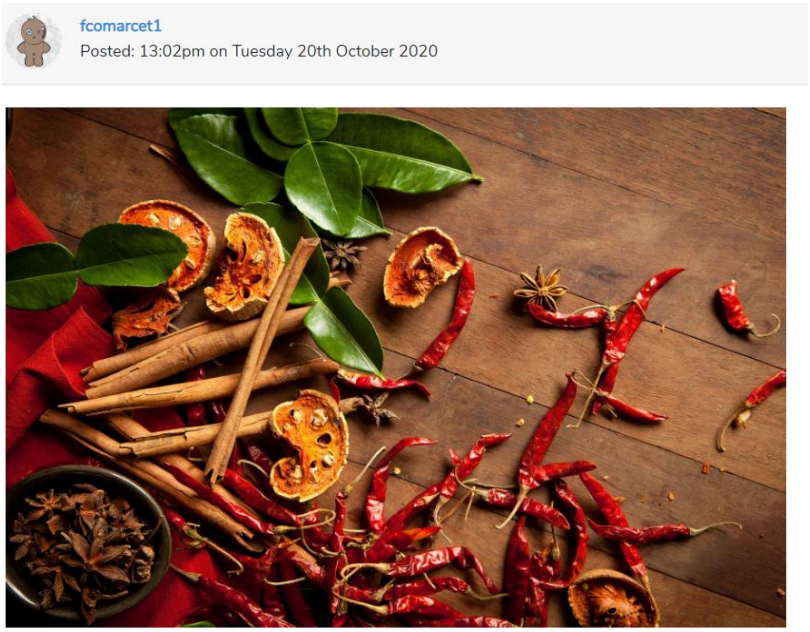
```
use App\Image;
use Illuminate\Support\Facades\Storage;
use Illuminate\Support\Facades\File;
```

```php
public function save(Request $request){

    //Validación
    $validate = $this->validate($request, [
        'description' => 'required',
        'image_path'  => 'required|image'
    ]);

    // Recoger datos
    $image_path = $request->file('image_path');
    $description = $request->input('description');

    // Asignar valores nuevo objeto
    $user = \Auth::user();
    $image = new Image();
    $image->user_id = $user->id;
    $image->description = $description;

    // Subir fichero
    if($image_path){
        $image_path_name = time().$image_path->getClientOriginalName();
        Storage::disk('images')->put($image_path_name, File::get($image_path));
        $image->image_path = $image_path_name;
    }

    $image->save();

    return redirect()->route('home')
                     ->with(['message' => 'La foto ha sido subida correctamente!!'
    ]);
}
```

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.

```php
public function index()
{

    $images = Image::orderBy('id', 'desc')->get();

    return view('home', array(
        'images' => $images
    ));
}
```

Ya podemos usar la variable $images en la vista 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

```php
//* Muchas imagenes puede ser de un unico user N:1 [Many to One]*/
public function user()
{
    return $this->belongsTo('App\User', 'user_id');
}
```

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

```
{{$image->user->id}}
```

```
{{$image->user->name}}
```

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

```
<!-- Image list -->
@foreach($images as $image)
<div class="card promoting-card">
    <div class="card-header d-flex flex-row">
        @if($image->user->image)
            <img src="{{route('user.avatar',['filename'=>$image->user->image])}}"
                 id="preview" class="rounded-circle mr-3" height="50px" width="50px" alt="avatar"/>
            <div class="mask rgba-white-slight"></div>
        @endif
        <div>
            <a href class="card-title font-weight-bold mb-2">{{$image->user->nick}}</a>
            <p class="card-text">Posted: {{$image->created_at->format('G:ia  \o\n l jS F Y')}}</p>
        </div>
    </div>
    <div class="card-body">
        <!-- Card image -->
        <div class="view overlay">
            <img class="card-img-top rounded-0" src="https://mdbootstrap.com/img/Photos/Horizontal/Fo
            <a href="#!">
                <div class="mask rgba-white-slight"></div>
            </a>
        </div>
        {{$image->image_path}}
    </div>
    <div class="card-footer">
        Likes
    </div>
</div>
@endforeach
<!-- end image list -->
```

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

```
//* Obtener la imagen del usuario(usada para avatar)
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);

}
```

OBTENER NICK +FECHA

```
<div>
    <a href class="card-title font-weight-bold mb-2">{{$image->user->nick}}</a>
    <p class="card-text">Posted: {{$image->created_at->format('G:ia  \o\n l jS F Y')}}</p>
</div>
```

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

```
use Illuminate\Http\Response;
```

```
//* Obtener la imagenes publicadss del usuario(usada para lista img home)
public function getImage($filename) {

    //* obtenememos la imagen del disco storage
    $file = Storage::disk('images')->get($filename);

    // devolvemos el resultado en una response
    return new Response($file, 200);

}
```

Ruta:

```
// Obtener listado imagenes en home
Route::get('image/file/{filename}', 'ImageController@getImage')->name('image.file');
```

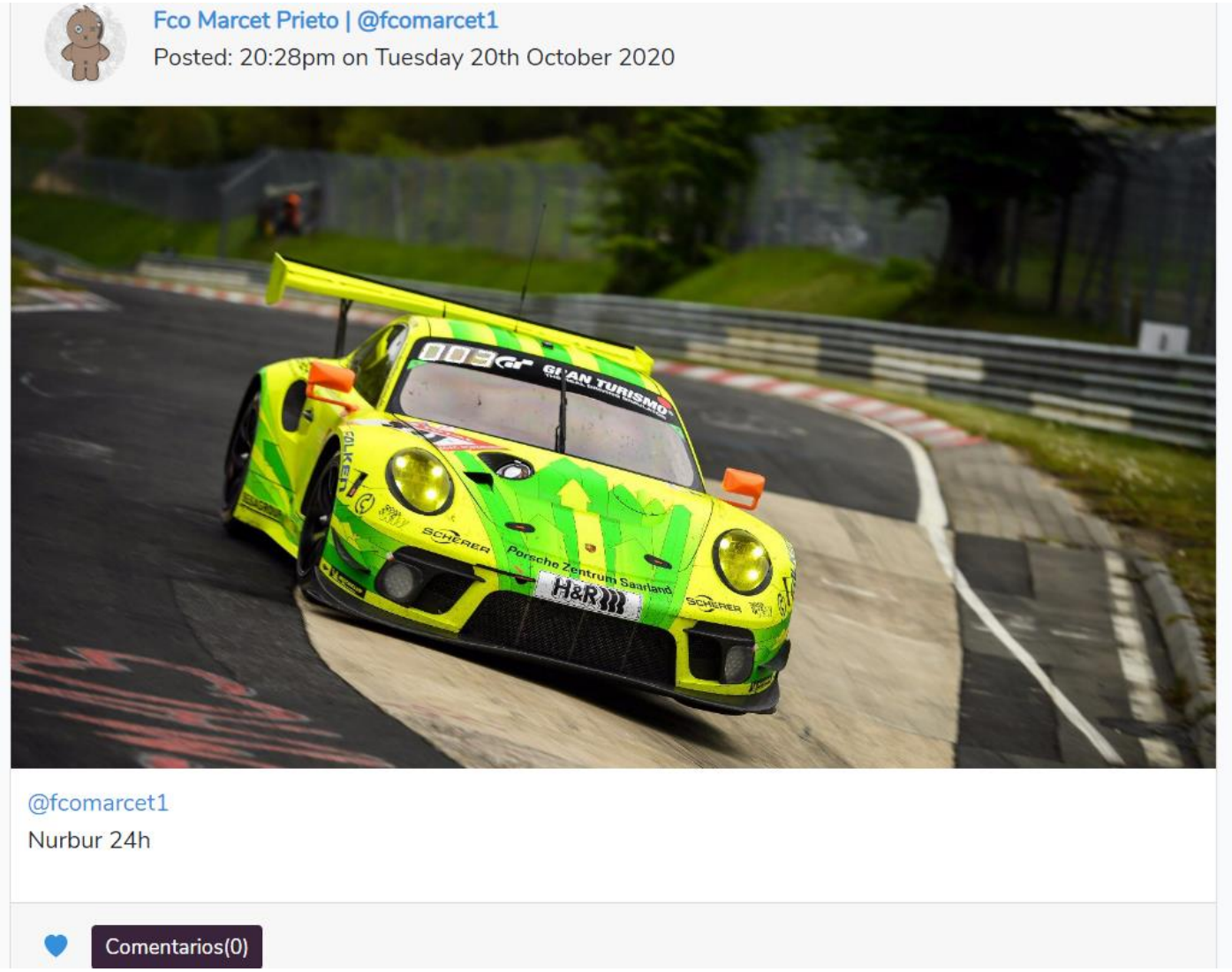
Ahora solo falta imprimir la imagen en la vista

```
<img class="card-img-top rounded-0" src="{{route('image.file',['filename' => $image->image_path])}}"  alt="Card
```

CARD-FOOTER(LIKES+COMENTARIOS).

```
<div class="card-footer">
    <a href="">
        <svg width="1em" height="1em" viewBox="0 0 16 16" class="bi bi-heart-fill" fill="currentCol
          <path fill-rule="evenodd" d="M8 1.314C12.438-3.248 23.534 4.735 8 15-7.534 4.736 3.562-3.
        </svg>
    </a>
    <button type="button" class="btn btn-warning btn-sm btn-comments">
        Comentarios({{count($image->comments)}})
    </button>
```

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.

```php
public function index() {

    $images = Image::orderBy('id', 'desc')->paginate(5);

    return view('home', array(
        'images' => $images
    ));
}
```

Ahora en la vista de home.blade.php añadimos el método link()

```php
<!-- Pagination -->
<div class="clearfix"></div>
<div class="pagination">
    {{ $images->links() }}
</div>
<!-- end Pagination -->
```

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

```php
//* Obtener detalle imagen
public function detail($id) {

    $image = Image::find($id);

    return view('images.detail', array(
        'image' => $image
    ));

}
```

Ruta:

```php
// Obtener detalle imagen
Route::get('image/{id}', 'ImageController@detail')->name('image.detail');
```

Vista(detail.blade.php):

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

```php
<!-- Card image -->
<a href="{{route('image.detail',['id'=>$image->id])}}">
    <div class="image-container">
        <img class="card-img-top rounded-0" src="{{route('image.file',['filename' => $image->image_path])}}"  alt=
        <a href="#!">
            <div class="mask rgba-white-slight"></div>
        </a>
    </div>
</a>
```

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/

Paso 2. Crear el provider, ejecutando el comando:

```
php artisan make:provider FormatTimeServiceProvider
```

Paso 3. Incluir el método register en el provider:

```php
public function register()
{
    require_once app_path() . '/Helpers/FormatTime.php';
}
```

Paso 4. Entrar al directorio config/app.php y añadir el provider al array de providers:

```
App\Providers\FormatTimeServiceProvider::class,
```

Y añadir un alias de nuestro helper:

```
'FormatTime' => App\Helpers\FormatTime::class,
```

Ya podemos usar nuestro helper en cualquier parte de nuestro código, por ejemplo en una vista hariamos algo así:

```
{{ \FormatTime::LongTimeFilter($entrada->created_at) }}
```

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:

```
<form method="POST" action="{{route('comment.save')}}">
    @csrf
    <input type="hidden" name="image_id" value="{{$image->id}}" />

        <p>
            <textarea class="form-control {{ $errors->has('content') ? 'is-invalid' : '' }}" name="content"></text
            @if($errors->has('content'))
            <span class="invalid-feedback" role="alert">
                <strong>{{ $errors->first('content') }}</strong>
            </span>
            @endif
        </p>

    <input type="submit" class="btn btn-sm btn-success" value="Enviar"/>
</form>
```

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

```
PS C:\wamp64\www\proyecto-laravel> php artisan make:controller CommentController
Controller created successfully.
```

Ruta:

```
//* COMENTARIOS
// Enviar nuevo comentario
Route::post('/comment/save','CommentController@save')->name('comment.save');
```

En controlador CommentController, método save()

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

```
use App\Comment;
```

```php
    public function save(Request $request) {

        $method = $request->method();
        $image_id = $request->input('image_id'); // cast to Int

        if ($request->isMethod('POST') && $method == 'POST') {

            // Validación
            $validate = $this->validate($request, [

                'image_id' => 'integer|required',
                'content' => 'required|string'
            ]);

            //recoger datos
            $user = \Auth::user();
            $user_id = $user->id;
            $content = $request->input('content');
            $image_id =(int) $request->input('image_id');

            //var_dump($user_id, $content, $image_id); die();

            //instanciamos nuevo objeto
            $comment = new Comment();

            // Seteamos objeto
            $comment->user_id = $user_id;
            $comment->content = $content;
            $comment->image_id = $image_id;

            //var_dump($comment); die();

            $save = $comment->save();

            if ($save) {
                return redirect()->route('image.detail', ['id' => $image_id])
                                 ->with([
                                     'message' => 'Tu comentario se ha publicado correctamente'
                                 ]);
            }
            else{
                // error al guardar comentario
                return redirect()->route('image.detail', ['id' => $image_id])
                                 ->with([
                                     'message-error' => 'Error.Tu comentario no se ha publicado, por favor vuelve a
                                 ]);
            }

        }
        else {

            //no llegan parametros por post del form, error.redirecc al detalle de imagen
            return redirect()->route('image.detail', ['id' => $image_id])
                             ->with(['message-error' => 'Error. No se pudo enviar el comentario, por favor vuelve a
        }

    }
}
```

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

```
//* Una imagen puede tener muchos comentarios 1:N [One To Many]*/
//** Obtener todos los comentarios asociados a una imagen  */
public function comments() {
    //** 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');
}
```

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()

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

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>";
}
```

En la vista quedaría:

```
<!-- Lista de comentarios -->
<?php $comments = $image->comments ?>
@foreach($comments as $comment)
<div class="">
    <a href class="card-title font-weight-bold mb-2">
        {{$comment->user->name." ".$comment->user->surname." | @".$comment->user->nick}}</a>
    <p class="card-text">Posted: {{ \FormatTime::LongTimeFilter($comment->created_at) }}</p>
    <p>{{$comment->content}}</p>
</div>
<hr/>
@endforeach
```

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

```
public function comments() {
    //** 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')->orderBy('id', 'desc');
}
```

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)

```
$comment->user_id == $user->id
```

Propietario de la imagen publicada:

- Id(Usuario logueado) == user_id(Imagen)

```
$comment->image->user_id == $user->id)
```

OBTENER OBJETO DEL COMENTARIO

```
$comment = Comment::find($id);
```

Si hacemos un var_dump($comment)

```
protected 'attributes' =>
  array (size=6)
    'id' => int 6
    'user_id' => int 7
    'image_id' => int 8
    'content' => string 'MIiiiiiiiiiiiiiic CHECKKKKKKKKKKKKKKKKKKKK.' (length=46)
    'created_at' => string '2020-10-22 09:18:17' (length=19)
    'updated_at' => string '2020-10-22 09:18:17' (length=19)
protected 'original' =>
```

$comment

Id=6, user_id= 7, image_id = 8 …

$comment->user

Id=7, name=Fco, surname=Marcet, email= fcomarect@mail.es …

$comment->image

Id=8, user_id= 7, path=image.png, …

Asi que la condición quedaría:

```
// Comprobar si estamos logueado y ademas (si soy el dueño del comentario o de la publicación)
if($user && ($comment->user_id == $user->id || $comment->image->user_id == $user->id)){
```

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

```php
return redirect()->route('image.detail', ['id' => $comment->image->id])
                 ->with([
                    'message' => 'Comentario eliminado correctamente!!'
                 ]);
```

Metodo delete() del controlador CommentController

```php
public function delete($id) {

    // Conseguir datos del usuario logueado
    $user = \Auth::user();
    $user_id = $user->id;

    // Conseguir objeto del comentario
    $comment = Comment::find($id);

    //var_dump($comment); die();
    //var_dump($comment->id); die();
    //var_dump($comment->user); die();
    //var_dump($comment->user->name); die();
    //var_dump($comment->image); die();
    //var_dump($comment->image->image_path); die();


    // Comprobar si estamos logueado y ademas (si soy el dueño del comentario o de la publicación)
    if($user && ($comment->user_id == $user->id || $comment->image->user_id == $user->id)){

        $delete = $comment->delete();

        if ($delete) {
            return redirect()->route('image.detail', ['id' => $comment->image->id])
                             ->with(['message' => 'El comentario se ha eliminado correctamente ']);

        }
        else{
            // error delete
            return redirect()->route('image.detail', ['id' => $comment->image->id])
                             ->with(['message-error' => 'Error.El comentario no se pudo eliminar']);
        }
    }
    else {
        //el usuario no es dueño del comentario ni de la publicacion
        return redirect()->route('image.detail', ['id' => $comment->image->id])
                         ->with(['message-error' => 'Error.El comentario no se pudo eliminar debes ser el propi

    }

}
```

En la vista :

```blade
<!-- Mostrar boton borrar si somos propietarios del comentario o de la publicacion -->
@if((Auth::check() && ($comment->user_id == Auth::user()->id || $comment->image->user_id == Auth::user()->id)))
    <div class="btn-delete">
        <a href="{{route('comment.delete', ['id' => $comment->id])}}" class="btn btn-sm btn-danger">Borrar</a>
    </div>
@endif
```

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:

```php
// Almacenar nuevo like
Route::get('/like/{image_id}', 'LikeController@like')->name('like.save');
```

Metodo -> like($image_id):

```php
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:

```php
//Dislike
Route::get('/dislike/{image_id}', 'LikeController@dislike')->name('dislike.save');
```

Método:

```php
    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:

```
$image->likes;
```

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

```
<div class="likes">

    <!--COMPROBAR SI EL USUARIO DIO LIKE -->
    <?php $user_like = false; ?>

    <!-- Recorremos los likes y comprobamos si coincide con el
         ususario identificado -->
    @foreach($image->likes as $like)
        @if($like->user->id == Auth::user()->id)
            <?php $user_like = true; ?>
        @endif
    @endforeach

    {{count($image->likes)}}
    @if($user_like)
        <img src="{{asset('img/heart-red.png')}}"  class="btn-like"/>
    @else
        <img src="{{asset('img/heart-black.png')}}"  class="btn-dislike"/>

    @endif

</div>
```

CARGAR FICHERO JS.

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

```
<script src="{{ asset('js/main.js') }}" ></script>
```

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:

```
@if($user_like)
    <img src="{{asset('img/heart-red.png')}}"  class="btn-like"/>
@else
    <img src="{{asset('img/heart-black.png')}}"  class="btn-dislike"/>

@endif
```

Vamos a comentar que hace el código JQuery:

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

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

2. 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

```
$('.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', 'img/heart-red.png');
```

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

```
//CAMBIAR PUNTERO CUANDO PASEMOS POR ENCIMA EL RATON
$('.btn-like').css('cursor', 'pointer');
$('.btn-dislike').css('cursor', 'pointer');
```

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

```
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', 'img/heart-red.png');

            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', 'img/heart-black.png');

            like();

        })
    }

    dislike();

});
```

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

PETICIONES AJAX.

Añadimos la petición AJAX

Like()

```
//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');
        }
    }
});
```

Para el dislike es similar pero cambiamos la ruta

```
//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');
        }
    }
});
```

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:

```
@foreach ($likes as $like)
    User id:{{$like->user_id}}<br/>

    image_id:{{$like->image_id}}<br/>
    <hr>
@endforeach
```

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

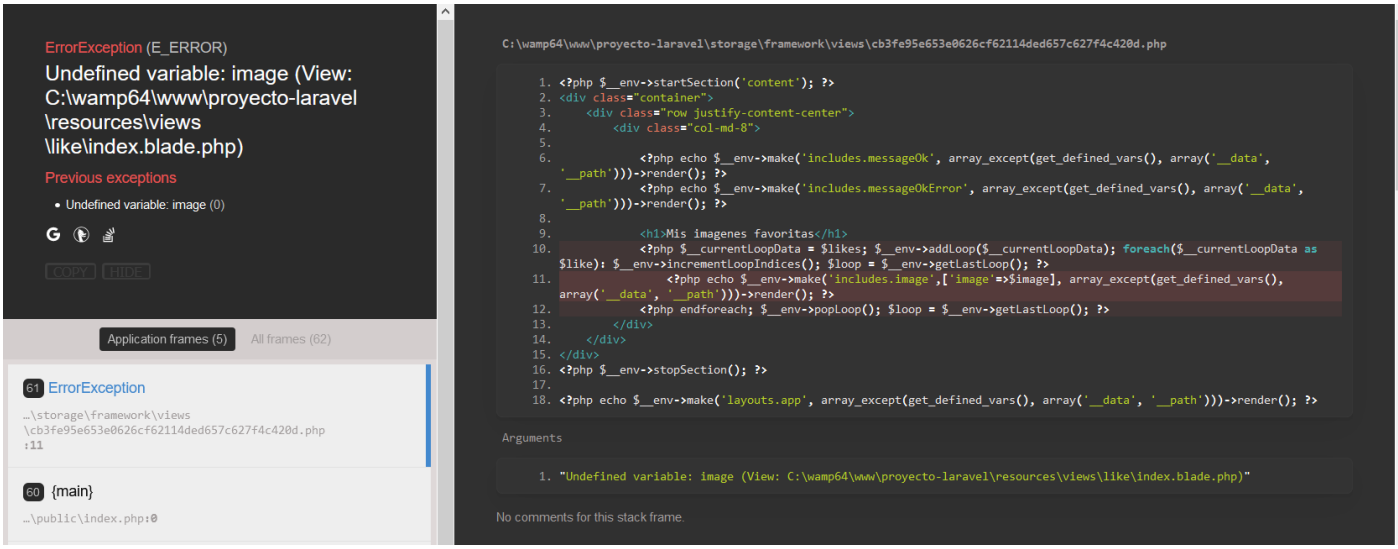
```
<!-- Image list -->
@foreach($images as $image)
    @include('includes.image')
@endforeach
<!-- end image list -->
```

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

```
<!-- Image list -->
@foreach($images as $image)
    @include('includes.image',['image' => $image])
@endforeach
<!-- end image list -->
```

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

```
<h1>Mis imagenes favoritas</h1>
@foreach ($likes as $like)
    @include('includes.image',['image'=>$like->image])
@endforeach
```

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.

```
<!-- Image list -->
@foreach($images as $image)
    <!-- Mostrar tarjeta de 1 imagen -->
    @include('includes.image',['image' => $image])
@endforeach
<!-- end image list -->
```

```
@foreach ($likes as $like)
    @include('includes.image',['image'=>$like->image])
@endforeach
```

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,

```php
//Perfil de usuario
public function profile() {
    $user = \Auth::user();

    return view('user.profile',[

        'user' => $user
    ]);

}
```

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:

```php
//Perfil de usuario
public function profile($id) {

    $user = User::find($id)->orderBy('created_at');

    return view('user.profile',[

        'user' => $user
    ]);

}
```

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

```
<a class="dropdown-item" href="{{route('profile',['id' => \Auth::user()->id]) }}">{{ __('Mi perfil') }}</a>
```

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

```
<a href="{{route('profile',[$image->user->id])}}" class="card-title font-weight-bold mb-2">
    {{$image->user->name." ".$image->user->surname." | @".$image->user->nick}}
</a>
```

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

```
<h1>Mi Perfil</h1>
<?php // no llega desde el controller $user ?>

<!-- avatar  falta obtener de storage -->
{{$user->image}}

<!-- Nombre -->
<p>{{$user->name}}</p>

<!-- Nick -->
<p>{{$user->nick}}</p>

<!-- se unió: -->
<p>{{$user->created_at}}</p>


<!-- listado imagenes falta obtener de storage usar method  -->
@foreach($user->images as $image)
    <p>{{$image->image_path}}</p>
@endforeach
```

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:

```
@if( (\Auth::user()) && (\Auth::user()->id == $image->user->id) )
    <div class="actions">
        <a href="" class="btn btn-sm btn-primary">Editar publicacion</a>
        <a href="" class="btn btn-sm btn-danger">Eliminar publicacion</a>
    </div>
@endif
```

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

```
//1º MANERA -> No me funciona
$comments = Comment::where('image_id', $id)->get(); // No me funciona
$likes = Like::where('image_id', $id)->get();
```

Una vez que tenemos los comentarios y likes los recorremos;

```
if($comments && sizeof($comments) >= 1){
    foreach ($comments as $comment) {
        $comment->delete();
    }
}
```

```
if($likes && sizeof($likes) >= 1){
    foreach ($likes as $like) {
        $like->delete();
    }
}
```

```
// Eliminar ficheros de imagen
Storage::disk('images')->delete($image->image_path);

// Eliminar registro imagen
$image->delete();
```

2º METODO

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

```php
// ELIMINAR COMENTARIOS
$matchTheseComments = ['image_id' => $id];
$comments = DB::table('comments')->where($matchTheseComments)->count();

//var_dump($comments); die();
if ($comments && $comments >= 1 ) {
    $comments_delete = DB::table('comments')->where($matchTheseComments)->delete();
    if ($comments_delete == false) {

        //error al delete comments
        $message = array('message-error' => 'Error. Comments not delete.');
        return redirect()->route('home')->with($message);
    }


}
```

```php
// ELIMINAR LIKES
$matchTheseLikes = ['image_id' => $id];
$likes = DB::table('likes')->where($matchTheseLikes)->count();

if ($likes && $likes >= 1 ) {
    $likes_delete = DB::table('likes')->where($matchTheseComments)->delete();

    if ($likes_delete == false) {

        //error al delete likes
        $message = array('message-error' => 'Error.Likes not deleted.');
        return redirect()->route('home')->with($message);
    }

}
```

```php
// Eliminar ficheros de imagen
Storage::disk('images')->delete($image->image_path);

// Eliminar registro imagen
$image->delete();
```

3ºMETODO (No es necesario un foreach)

```
3º METODO
$comments = $image->comments();
$likes = $image->likes();
```

```
$comments = $image->comments();
if($comments && sizeof($comments) >= 1){
   $comments->delete();
}
```

```
$likes = $image->likes();
if($likes && sizeof($likes) >= 1){
   $likes->delete();
}
```

```
// Eliminar ficheros de imagen
Storage::disk('images')->delete($image->image_path);

// Eliminar registro imagen
$image->delete();
```

4º METODO (Modificar modelo Image)

Añadimos un nuevo método al modelo Image

```
//delete publicacion, con sus likes y comments asociados
public static function boot() {
    parent::boot();

    static::deleting(function ($images) {
        $images->likes()->each(function ($likes) {
            $likes->delete();
        });
        $images->comments()->each(function ($comments) {
            $comments->delete();
        });
        Storage::disk('images')->delete($image->image_path);
    });
}
```

MODAL ELIMINAR PUBLICACION.

```
<button type="button" class="btn btn-sm btn-danger" data-toggle="modal" data-target="#myModa
    Eliminar publicacion
</button>
<!-- The Modal -->
<div class="modal" id="myModal">
    <div class="modal-dialog">
        <div class="modal-content">

            <!-- Modal Header -->
            <div class="modal-header">
                <h4 class="modal-title">Eliminar publicacion</h4>
                <button type="button" class="close" data-dismiss="modal">&times;</button>
            </div>

            <!-- Modal body -->
            <div class="modal-body">
                ¿Deseas borrar definivamente la publicacion?
            </div>

            <!-- Modal footer -->
            <div class="modal-footer">
                <button type="button" class="btn btn-primary" data-dismiss="modal">
                    Cancelar
                </button>
                <a href="{{route('image.delete', ['id'=> $image->id])}}" class="btn btn-dang
                    Eliminar publicacion
                </a>
            </div>

        </div>
    </div>
</div>
<!--  end Modal -->
```

EDITAR PUBLICACION.

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

```
<a href="{{route('image.edit', ['id' => $image->id])}}" class="btn btn-sm btn-primary">
    Editar publicacion
</a>
```

```
//Editar imagen
Route::get('/image/edit/{id}', 'ImageController@edit')->name('image.edit');
```

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

```
//* EDITAR PUBLICACION
public function edit($id) {

    //Usuario identificado
    $user = \Auth::user();
    $user_id = $user->id;

    //Obtener publicacion a editar
    $image = Image::find($id);

    // Validamos que existe el ususario identificado, la imagen buscada, y ademas
    // sea el propietario de la publicacion
    if ($user && $image && ($image->user->id == $user->id)) {

        return view('image.edit', [
            'image' => $image
        ]);
    }
    else {
        return redirect()->route('home');
    }
}
```

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:

```
//Actualizar publicacion/Imagen
Route::post('/image/update','ImageController@update')->name('image.update');
```

El formulario llamara al método update() de ImageController.

```php
public function update(Request $request) {
    $method = $request->method();
    if (($request->isMethod('POST')) && ($method == 'POST')) {
        //VALIDACION
        $validate = $this->validate($request, [
            'description' => 'required',
            'filename' => 'image'
        ]);

        //* OBTENER DATOS DEL FORM
        $description = $request->input('description');
        $image_id = (int) $request->input('image_id');
        $image_path = $request->file('filename');

        //*OBTENER IMAGEN PARA UPDATE
        $image = Image::find($image_id);
        $image->description = $description;

        //* Subir fichero si la img nos llega not null
        if ($image_path) {

            // eliminar imagen del storage
            Storage::disk('images')->delete($image->image_path);

            // nombre unico
            $image_path_name = time() . "_" . $image_path->getClientOriginalName();

            // almacenar storage
            Storage::disk('images')->put($image_path_name, File::get($image_path));

            // setear objeto con el nuevo path
            $image->image_path = $image_path_name;
        }
```

```php
        //* Actualizar registro
        $update = $image->update();

        //Validar update
        if ($update) {
            return redirect()->route('image.detail', ['id' => $image_id])
                             ->with(['message' => 'Publicacion actualizada con exito']);
        }
        else {
            // error
            return redirect()->route('image.detail', ['id' => $image_id])
                             ->with(['message-error' => 'Error. La publicacion no pudo actualiza
        }
    }
    else {
        // error al enviar form
        return redirect()->route('image.detail', ['id' => $image_id])
                         ->with(['message-error' => 'Error. La publicacion no pudo actualizarse,
    }
}
```

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

```
<li class="nav-item">
    <a  class="nav-link" href="{{route('user.index')}}">Gente</a>
</li>
```

Ruta:

```
// listado de usuarios de la aplicacion
Route::get('/gente','UserController@index')->name('user.index');
```

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

```
//Listado de usuarios de la aplicacion
public function index() {

    // Obtener todos los usuarios
    $users = User::orderBy('id', 'desc')->paginate(5);

    return view('user.index', [
        'users' => $users
    ]);
}
```

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.

```
Route::get('/gente/{search?}','UserController@index')->name('user.index');
```

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.

```
//Listado de usuarios de la aplicacion + Buscador
public function index($search = null) {

    if (!empty($search)) {
        $users = User::where('nick','LIKE', '%'.$search.'%')
                        ->orWhere('name', 'LIKE', '%'.$search.'%')
                        ->orWhere('surname', 'LIKE', '%'.$search.'%' )
                        ->orderBy('id', 'desc')
                        ->paginate(5);
    }
    else{
        // Obtener todos los usuarios
        $users = User::orderBy('id', 'desc')->paginate(5);
    }

    return view('user.index', [
        'users' => $users
    ]);
}
```

Solo nos falta implementar un formulario para el buscador

```
<form method="GET" action="{{route('user.index')}}">
    <div class="input-group mb-3">
        <input type="text" name="search" class="form-control" placeholder="Buscar usuarios">
        <div class="input-group-append">
            <button class="btn btn-success" type="submit">Buscar</button>
        </div>
    </div>
</form>
```

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 :

```
composer create-project --prefer-dist laravel/laravel test-laravel6 "6.*"
```

Creamos una tabla users:

id	role	name	surname	nick	email	password	image	created_at	updated_at	remember_token

Por ultimo configuramos fichero .env

CAMBIOS RESPECTO LA VERSION 5.8

- Instalacion laravel ui

```
composer require laravel/ui "^1.0" --dev
```

```
php artisan ui vue --auth
```

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

Para acabar de visualizar bien los estilos:

AUTENTICACION EN LARAVEL 8