

A dark blue vertical bar on the left side of the page. A blue arrow points to the right from the bar, containing the year 2021.

2021

# PROYECTO LARAVEL

Proyecto de red social usando el  
framework Laravel.

Several thin, curved lines in dark blue and light gray originate from the bottom left corner and curve upwards and to the right.

Miguel A. M. Rastau

# Contenido 1

1 – Introducción al proyecto.....	3
2 – Base de datos y entidades del proyecto .....	4
2.1 – Diseño de la Base de Datos.....	4
2.2 – Generar un nuevo Proyecto de Laravel .....	5
2.3 – Crear la base de datos .....	6
2.4 – Conexión a la Base de Datos.....	7
2.5 – Crear los modelos .....	7
2.6 – Configurar modelos y relaciones .....	8
2.7 – Probando el ORM .....	10
3 – Login y registro de usuarios en Laravel .....	12
3.1 – Login y registro de usuarios (Laravel UI) .....	12
*****IMPORTANTE***** .....	13
3.2 – Modificando el registro de usuarios.....	13
3.3 – Elementos del menú .....	15
4 – Página de Configuración del Usuario.....	17
4.1 – Formulario de configuración .....	17
4.2 – Recibir datos del formulario de configuración .....	17
4.3 – Validar el formulario de configuración.....	18
4.4 – Actualizar el usuario .....	18
4.5 – Subir imagen de usuario .....	19
4.6 – Mostrar avatar .....	20
4.7 – Avatar en el menú.....	21
4.8 – Solo para usuarios autenticados .....	21
5 – Imágenes de la aplicación (clon Instagram) .....	22
5.1 – Formulario para subir imágenes.....	22
5.2 – Subir imagen .....	23
5.3 – Listado de imágenes .....	24
5.4 – Paginación en Laravel .....	26
5.5 – Maquetación de likes.....	26

5.6 – Número de comentarios.....	27
5.7 – Detalle de la imagen .....	27
5.8 – Formatear fechas.....	28
6 – Sistema de comentarios .....	28
6.1 – Formulario de comentarios .....	28
6.2 – Validar formulario.....	28
6.3 – Guardar los comentarios .....	29
6.4 – Listar comentarios .....	30
6.5 – Borrar comentarios.....	30
7 – Sistemas de Likes .....	33
7.1 – Método Like .....	33
7.2 – Método dislike .....	34
7.3 – Detectar likes .....	34
7.3 – Cargar archivos JS y cambiar el color del like .....	35
7.4 – Peticiones AJAX.....	36
7.5 – Like en el detalle .....	38
7.6 – Listar likes .....	38
8 – Borrado y modificación de imágenes .....	39
8.1 – Botones de las imágenes .....	39
8.2 – Eliminar imagen .....	39
8.3 – Modal en Bootstrap 4 .....	41
8.4 – Formulario de edición de imágenes .....	42
8.5 – Actualizar imágenes.....	42
9 – Gente y buscador.....	44
9.1 – Página de gente .....	44
9.2 – Buscador de gente .....	44
9.3 – Formulario del buscador.....	45
Documentación .....	46

## 1 – Introducción al proyecto

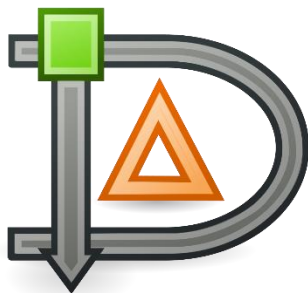
---

En este proyecto usaremos el framework Laravel para crear una página web imitando a Instagram, donde contaremos con una base de datos para que los usuarios se puedan registrar, subir fotos, dar ‘me gusta’ a esas fotos, comentarlas, etc...

Para ello haremos uso algunos programas para realizar dicho proyecto.

Para diseñar la base de datos usaremos el creador de diagramas **Dia**. Usaremos, como hemos mencionado, **Laravel** como framework. También usaremos **git** para crear un repositorio en **GitHub** donde guardaremos el proyecto para que dicho código pueda ser visto por todo el mundo. Usaremos **Cygwin64** como terminal de comandos por comodidad del creador. También usaremos **wampserve64** como servidor local para poder usar bases de datos, servidor web y php. Y finalmente también haremos uso de **Apache NetBeans** como IDE (Integrated Development Environment) o Entorno de Desarrollo Integrado.

También estarán integrados en este proyecto el ORM (Object Relational Mappings) o, Mapeadores de Objetos Relacionales, en concreto usaremos **Eloquent** que viene integrado en Laravel.



## 2 – Base de datos y entidades del proyecto

### 2.1 – Diseño de la Base de Datos

Crearemos con Dia un diagrama de la base de datos usando la definición UML que viene en el programa. Es importante que los atributos de las tablas tengan la sintaxis correcta porque Laravel es muy estricto con respecto al ORM ya que nos generará ciertas cosas que veremos más adelante de forma automática.

Vamos a crear una tabla USERS (haremos la base de datos en inglés) con los siguientes atributos que, repetimos, es necesario que siga una correcta sintaxis:

(En negrita se van a señalar los campos obligatorios que necesariamente deben estar escritos de esa forma para que Laravel no tenga problemas.)

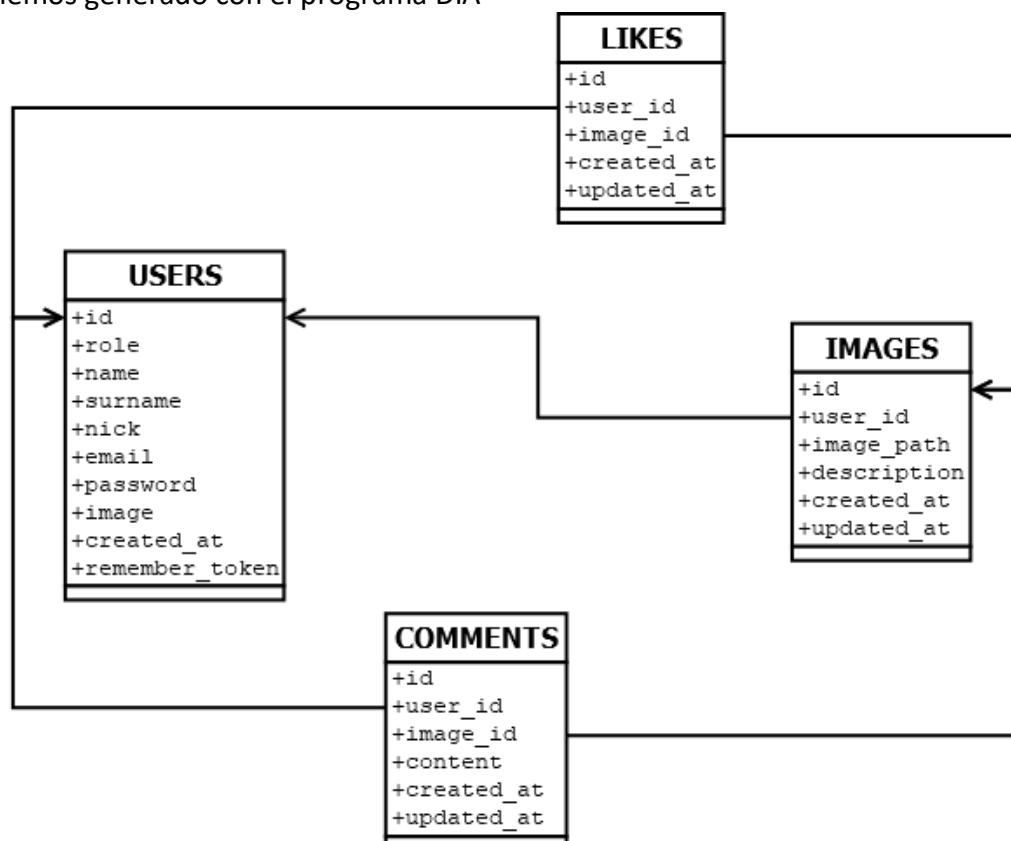
Tabla USERS -> **id**, **role**, name, surname, nick, email, password, image, **created\_at**, **updated\_at**, **remember\_token**

Tabla IMAGES -> **id**, **user\_id**, image\_path, description, **created\_at**, **updated\_at**

Tabla COMMENTS -> **id**, **user\_id**, **image\_id**, content, **created\_at**, **updated\_at**

Tabla LIKES -> **id**, **user\_id**, **image\_id**, **created\_at**, **updated\_at**

Estas tablas conformarán la base de datos. A continuación, podemos ver el diagrama que hemos generado con el programa DIA



## 2.2 – Generar un nuevo Proyecto de Laravel

En primer lugar, crearemos un nuevo proyecto de Laravel desde la consola de comandos con el comando `composer` en el directorio `C:\wamp64\www\master-php` y llamaremos al proyecto `proyecto-laravel`.

```
mmras@DESKTOP-UG00QBB /cygdrive/c/wamp64/www/master-php
$ composer create-project laravel/laravel proyecto-laravel --prefer-dist
```

A continuación, vamos a crear un host virtual para que sea más cómodo trabajar con el proyecto. Para ello nos vamos a editar el fichero

**C:\wamp64\bin\apache\apache2.4.46\conf\extra\httpd-vhosts.conf**

como vemos a continuación:

```
# VHOST PROYECTO LARAVEL
<VirtualHost *:80>
    DocumentRoot "${INSTALL_DIR}/www/master-php/proyecto-laravel/public"
    ServerName proyecto-laravel.com.devel
    ServerAlias www.proyecto-laravel.com.devel
    <Directory "${INSTALL_DIR}/www/master-php/proyecto-laravel/public">
        Options Indexes FollowSymLinks
        AllowOverride All
        Order Deny,Allow
        Allow from all
    </Directory>
</VirtualHost>
```

Y finalmente editamos el fichero **C:\Windows\System32\drivers\etc\hosts** como vemos a continuación:

```
127.0.0.1 localhost::1 localhost
127.0.0.1 localhost

127.0.0.1 aprendiendo-laravel.com.devel
127.0.0.1 proyecto-laravel.com.devel
|
::1 localhost
```

## 2.3 – Crear la base de datos

En la raíz del proyecto vamos a crear un nuevo archivo que llamaremos **database.sql** donde crearemos nuestro código SQL para crear la Base de Datos **laravel\_master**.

\*\*\*\*\*IMPORTANTE\*\*\*\*\*

Como estamos usando Apache NetBeans debemos configurar el IDE para que pueda ejecutar el código SQL desde el IDE a la base de datos. El enlace de descarga está en la sección de [Documentación](#)

\*\*\*\*\*

A continuación, podemos ver el código SQL generado según el diseño que pudimos ver en [2.1 – Diseño de la Base de Datos](#).

```
CREATE DATABASE IF NOT EXISTS
laravel_master;
USE laravel_master;

CREATE TABLE IF NOT EXISTS users(
id          int(255) auto_increment not null,
role        varchar(20),
name        varchar(100),
surname     varchar(200),
nick        varchar(100),
email       varchar(255),
password    varchar(255),
image       varchar(255),
created_at  datetime,
updated_at  datetime,
remember_token varchar(255),
CONSTRAINT pk_users PRIMARY KEY(id)
)ENGINE=InnoDB;

CREATE TABLE IF NOT EXISTS images(
id          int(255) auto_increment not null,
user_id     int(255),
image_path  varchar(255),
description text,
created_at  datetime,
updated_at  datetime,
CONSTRAINT pk_images PRIMARY KEY(id),
CONSTRAINT fk_images_users FOREIGN
KEY(user_id) REFERENCES users(id)
)ENGINE=InnoDB;
```

```
CREATE TABLE IF NOT EXISTS comments(
id          int(255) auto_increment not null,
user_id     int(255),
image_id    int(255),
content     text,
created_at  datetime,
updated_at  datetime,
CONSTRAINT pk_comments PRIMARY
KEY(id),
CONSTRAINT fk_comments_users FOREIGN
KEY(user_id) REFERENCES users(id),
CONSTRAINT fk_comments_images FOREIGN
KEY(image_id) REFERENCES images(id)
)ENGINE=InnoDB;
```

```
CREATE TABLE IF NOT EXISTS likes(
id          int(255) auto_increment not null,
user_id     int(255),
image_id    int(255),
created_at  datetime,
updated_at  datetime,
CONSTRAINT pk_likes PRIMARY KEY(id),
CONSTRAINT fk_likes_users FOREIGN
KEY(user_id) REFERENCES users(id),
CONSTRAINT fk_likes_images FOREIGN
KEY(image_id) REFERENCES images(id)
)ENGINE=InnoDB;
```

## 2.4 – Conexión a la Base de Datos

Vamos a configurar el proyecto de Laravel para realizar la conexión a la base de datos, vamos a configurar el fichero .env

Primero vamos a cambiar APP\_URL por la URL del proyecto.

```
APP_URL=http://proyecto-laravel.com.devel/
```

Y la configuración de la base de datos la dejaremos como vemos a continuación:

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

Esta configuración cambiaría en el caso de que fuese un servidor remoto, donde el DB\_HOST sería la IP del servidor y el DB\_PORT el puerto, pero por lo general suele ser el 3306 para base de datos. El DB\_USERNAME por lo general también es root pero podría cambiar, así como la contraseña que pondríamos en DB\_PASSWORD.

Pero para este proyecto, en servidor local, esta configuración es suficiente.

## 2.5 – Crear los modelos

Una de las partes más importantes de la arquitectura MVC como ya sabemos son los modelos y en Laravel definir modelos o entidades es algo muy importante.

Para eso Laravel nos trae un ORM el cual nos permite abstraernos completamente de la base de datos y tener modelos o entidades y ellos se encargan de hacer las operaciones en la base de datos o con la base de datos.

Las entidades o modelos de Laravel representan los registros de la base de datos, por ejemplo, la entidad users representará el registro de la base de datos de users. Eso quiere decir que una entidad es una clase en la cual instanciaremos un objeto y cada uno de estos objetos representará un registro en la base de datos.

Los modelos se generan dentro de la carpeta app/Models. Por defecto Laravel nos genera un modelo de User. Este modelo se representa en singular porque se refiere a un único usuario.

A continuación, vamos a generar el resto de modelos que necesitamos.



Los modelos **siempre** serán descritos en singular porque serán un objeto que representará un registro concreto de la base de datos.

Para generar un modelo usaremos la consola de comandos con los comandos de artisan con la siguiente sintaxis:

```
$ php artisan make:model Nombre_en_singular
```

Este comando nos generará un modelo nuevo y con la configuración básica en la carpeta app/Models.

## 2.6 – Configurar modelos y relaciones

Vamos a configurar nuestros modelos para que funcionen de la mejor manera posible y también vamos a configurar las relaciones entre ellos.

Vamos a configurar el modelo Image.

En primer lugar, usaremos la propiedad protegida \$table para indicarle que tabla de la base de datos va a modificar.

```
class Image extends Model
{
    use HasFactory;

    protected $table = 'images';
}
```

A continuación, vamos a realizar de relación One to Many, es decir de uno a varios debido a que una imagen puede tener uno o más comentarios, así como uno o más likes.

```
//Relacion One to Many (De uno a varios)
public function comments() {
    return $this->hasMany('App\Models\Comment');
}

//Relacion One to Many (De uno a varios)
public function likes() {
    return $this->hasMany('App\Models\Like');
}
```

También crearemos una relación Many to One, de muchos a uno, pues varias imágenes pueden pertenecer a un único usuario que las ha subido. En esta función deberemos indicar el namespace de user, así como la clave user\_id para indicar a que usuario pertenecen las imágenes.

```
//Relacion Many to One (De muchos a uno)
public function user() {
    return $this->belongsTo('App\Models\User', 'user_id');
}
```

---

Por último, configuramos las relaciones del resto de modelos.

```
class Comment extends Model
{
    use HasFactory;

    protected $table = 'comments';

    //Relacion Many to One (De muchos a uno)
    public function user(){
        return $this->belongsTo('App\Models\User', 'user_id');
    }

    //Relacion Many to One (De muchos a uno)
    public function image(){
        return $this->belongsTo('App\Models\Image', 'image_id');
    }
}

class Like extends Model
{
    use HasFactory;

    protected $table = 'likes';

    //Relacion Many to One (De muchos a uno)
    public function user(){
        return $this->belongsTo('App\Models\User', 'user_id');
    }

    //Relacion Many to One (De muchos a uno)
    public function image(){
        return $this->belongsTo('App\Models\Image', 'image_id');
    }
}
```

---

A la entidad usuario nos interesa añadirle un método que liste todas las imágenes que ha creado.

```
//Relacion One to Many (De uno a varios)
public function images() {
    return $this->hasMany('App\Models\Image');
}
```

## 2.7 – Probando el ORM

Una vez rellenada la base de datos con algunos datos para hacer pruebas podemos continuar.

Ahora vamos a realizar las pruebas y para ellos nos vamos a el archivo de rutas. En primer lugar deberemos indicar el uso de los modelos indicando use y su namespace en el archivo.

Con el método all() del objeto Image podemos sacar todos los objetos que tenemos en la base de datos.

```
use App\Models\Image;

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

    $images = Image::all();
    foreach($images as $image){
        echo $image->image_path."<br/>";
        echo $image->description."<br/>";
        echo "<hr/>";
    }

    die();
}
```



test.jpg
descripcion de prueba 1

playa.jpg
descripcion de prueba 2

arena.jpg
descripcion de prueba 3

familia.jpg
descripcion de prueba 4

Ahora trataremos de listar el usuario que ha creado una imagen y los comentarios asociados a la imagen. Gracias a ORM tenemos propiedades en cada objeto creadas virtualmente a las que podemos acceder gracias a las relaciones creadas previamente. De este modo en pocas líneas que hemos añadido al código anterior obtenemos la misma información que con una consulta SQL mucho más compleja.

test.jpg  
descripcion de prueba 1  
Mikel Over  
Comentarios:  
==> Buena foto de PLAYA!!

---

playa.jpg  
descripcion de prueba 2  
Mikel Over  
Comentarios:

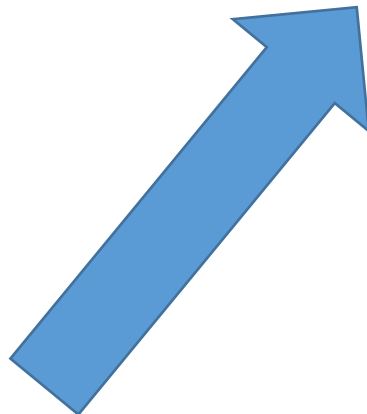
---

arena.jpg  
descripcion de prueba 3  
Mikel Over  
Comentarios:

---

familia.jpg  
descripcion de prueba 4  
Manolo Garcia  
Comentarios:  
==> Buena foto de familia!!  
==> que bueno!!

---



```
echo $image->user->name.' '. $image->user->surname."<br/>";  
echo "Comentarios: <br/>";  
foreach($image->comments as $comment) {  
    echo "==> ". $comment->content."<br/>";  
}
```

Y del mismo modo lo podríamos hacer para contar los likes de cada foto.

## 3 – Login y registro de usuarios en Laravel

---

### 3.1 – Login y registro de usuarios (Laravel UI)

Vamos a realizar el Login y el registro en la aplicación. Dentro de Laravel es muy sencillo porque incluye una serie de clases y métodos que implementan esta funcionalidad con solo ejecutar unos comandos porque es algo tan común que Laravel ya lo ha hecho.

Como ya tenemos instalado un proyecto de Laravel el primer paso para la autenticación sería instalar Laravel UI (el paquete de interfaz de usuario de Laravel). Instalar Laravel UI para el scaffolding de autenticación en Laravel 8. Ejecutar el siguiente comando de composer para ello:

```
$ composer require laravel/ui
```

Después de la instalación de Laravel UI. Ahora podemos hacer scaffold a nuestra autenticación con Bootstrap, Vue, React, etc. Si queremos generar scaffold con Vue, entonces tenemos que ejecutar el comando como se muestra a continuación.

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

Automáticamente se nos añadirán las siguientes líneas al archivo de rutas web.php

```
Auth::routes();  
Route::get('/home', [App\Http\Controllers\HomeController::class, 'index'])->name('home');
```

En este paso, tenemos que instalar todas nuestras dependencias de NPM, para ello debemos tener instalado node.js primero. A continuación, para instalar las dependencias de NPM, ejecutar el comando que se indica a continuación:

```
$ npm install
```

Luego compilamos con el siguiente comando:

```
$ npm run dev
```

Tras ejecutar estos comandos podremos comprobar que en la ruta <http://proyecto-laravel.com.devel/login> y <http://proyecto-laravel.com.devel/register> tenemos lo necesario para autenticar y registrar usuarios de manera funcional, además de otros añadidos como haber olvidado la contraseña para recuperarla.

\*\*\*\*\***IMPORTANTE**\*\*\*\*\*

Han salido una serie de errores que vamos a ver ahora:

Tras instalar node.js hay que comprobar que este en el PATH.

Si da error al compilar run dev, se debe usar el siguiente comando:

**npm i vue-loader**

Si volvemos a compilar debería funcionar correctamente.

\*\*\*\*\*

Las vistas generadas vienen por defecto en inglés, pero modificar estas vistas si nos vamos a resources donde podemos encontrar el estilo, los textos dentro de lang y las views.

En el siguiente capítulo veremos cómo hacer modificaciones.

## 3.2 – Modificando el registro de usuarios

En primer lugar vamos a añadirle algunos campos al formulario de registro para rellenar todos los campos de la table users de la base de datos. Para ello nos dirigimos a resources/views/auth/register.blade.php esta vista es la correspondiente a la página de registro de usuarios del proyecto.

```
<div class="form-group row">
  <label for="name" class="col-md-4 col-form-label text-md-right">{{ __('Name
  <div class="col-md-6">
    <input id="name" type="text" class="form-control @error('name') is-inva
    @error('name')
      <span class="invalid-feedback" role="alert">
        <strong>{{ $message }}</strong>
      </span>
    @enderror
  </div>
</div>

<div class="form-group row">
```

Como podemos ver se está usando bootstrap para darle forma a la página, Laravel ya incluye validaciones propias para estos campos, pero eso lo veremos más adelante.

Por ahora vamos a copiar el campo del formulario del nombre para hacer el del apellido o *surname* y para el nickname.

```
protected $fillable = [  
    'role',  
    'name',  
    'surname',  
    'nick',  
    'email',  
    'password',  
];
```

A continuación, también debemos modificar el modelo User para añadir algunos campos de relleno masivo. Esto lo encontramos, como vimos anteriormente, en app/Models.

Añadimos el campo role porque se rellenará desde otro lugar aunque no sea desde el formulario como veremos a continuación.

El próximo paso es modificar los controladores, en este caso vamos a modificar el controlador de app/Http/Controllers/Auth/RegisterController para modificar nuestro **Validator** para darle la misma validación al surname y al Nick que al name:

```
*/  
protected function validator(array $data)  
{  
    return Validator::make($data, [  
        'name' => ['required', 'string', 'max:255'],  
        'surname' => ['required', 'string', 'max:255'],  
        'nick' => ['required', 'string', 'max:255'],  
        'email' => ['required', 'string', 'email', 'max:255', 'unique:users'],  
        'password' => ['required', 'string', 'min:8', 'confirmed'],  
    ]);  
}
```

Luego deberemos modificar igualmente el **create**, como vemos a continuación:

```
*/  
protected function create(array $data)  
{  
    return User::create([  
        'role' => 'user',  
        'name' => $data['name'],  
        'surname' => $data['surname'],  
        'nick' => $data['nick'],  
        'email' => $data['email'],  
        'password' => Hash::make($data['password']),  
    ]);  
}
```

Como podemos ver, hemos indicado que el campo role se ponga siempre como *user* en lugar de recoger el dato del formulario.

Esto es porque es un dato fijo que siempre

vamos a tener en la base de datos ya que es en este método, en **create**, donde se ingresan los registros en la base de datos.

### 3.3 – Elementos del menú

Esta aplicación requiere autenticación en todo momento, será como una red social privada. Y va a requerir autenticación en todo momento porque lo estamos indicando con un middleware en el constructor del controlador como vemos a continuación:

```
public function __construct()  
{  
    $this->middleware('auth');  
}
```

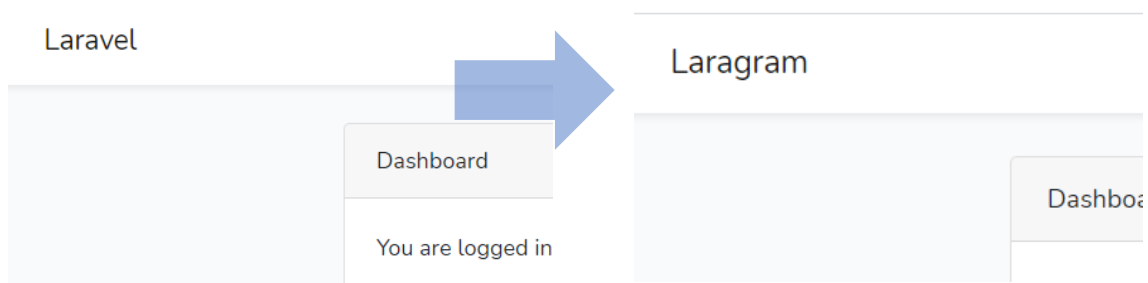
Esto bloquea el acceso al resto de métodos que tengamos en el caso de que no estar logueado. Solo podrán los usuarios autenticados.

Ahora vamos a añadir algunos elementos al menú, para eso lo que vamos a hacer es abrir la plantilla, vamos a abrir el **resources/views/layouts/app.blade.php**

Primero nos dirigimos al navbar y modificamos el nombre por defecto por un nombre nuevo para nuestra aplicación.

```
<a class="navbar-brand" href="{{ url('/') }}">  
    {{ config('app.name', 'Laravel') }}  
</a>  
  
<a class="navbar-brand" href="{{ url('/') }}">  
    Laragram  
</a>
```

Y aquí podemos ver el resultado.



A continuación, vamos a cambiar más cosas, pero solo nos fijaremos en el código.



Añadiremos en la barra de navegación derecha una serie de links que más adelante haremos que sean funcionales. Como vemos a continuación:

```
@else
    <li class="nav-item">
        <a class="nav-link" href=" " >{{ __('Inicio') }}</a>
    </li>

    <li class="nav-item">
        <a class="nav-link" href=" " >{{ __('Subir foto') }}</a>
    </li>

    <li class="nav-item dropdown">
        <a id="navbarDropdown" class="nav-link dropdown-toggle" href=
            {{ Auth::user()->name }}
        </a>

        <div class="dropdown-menu dropdown-menu-right" aria-labelledby=
            <a class="dropdown-item" href=" " >
                {{ __('Mi Perfil') }}
            </a>

            <a class="dropdown-item" href=" " >
                {{ __('Configuración') }}
            </a>

            <a class="dropdown-item" href="{{ route('logout') }}"
                onclick="event.preventDefault();
```

## 4 – Página de Configuración del Usuario

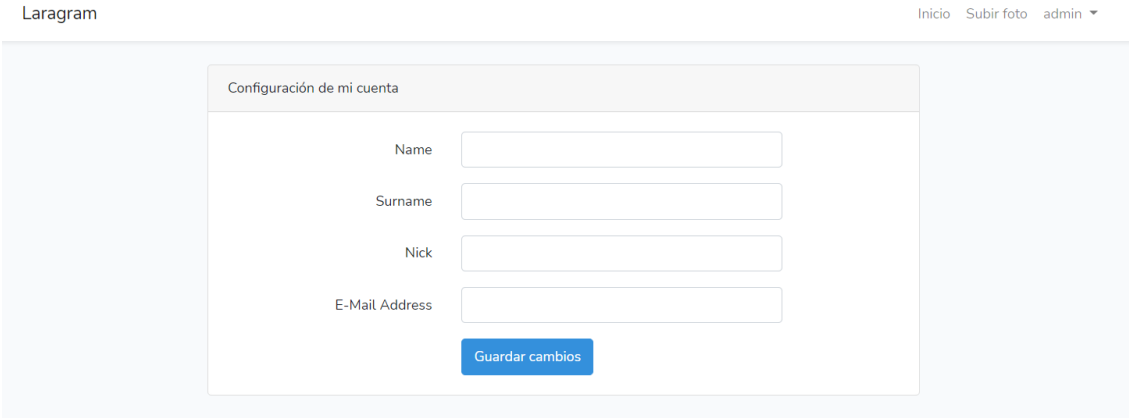
### 4.1 – Formulario de configuración

Vamos a realizar un formulario que modifique la información del usuario.

Para ello vamos a generar un nuevo controlador.

Para la vista iremos copiando y editando el código de `register.blade.php` para tener el mismo estilo.

El cambio de contraseña lo haremos en un formulario distinto.



### 4.2 – Recibir datos del formulario de configuración

Vamos a hacer que el formulario se rellene con los datos correspondientes del usuario identificado. Laravel nos facilita esto poniendo simplemente el método `email` del objeto `auth::user` como vemos a continuación para este caso.

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

Y así con todos los campos.

El siguiente paso es crear un método en el controlador que reciba los datos del formulario y tras eso podemos hacer un `update` de una forma muy sencilla.

```
public function update(Request $request) {  
    $id = \Auth::user()->id;  
    $name = $request->input('name');  
    $surname = $request->input('surname');  
    $nick = $request->input('nick');  
    $email = $request->input('email');
```

```
Route::post('/user/update', [App\Http\Controllers\UserController::class, 'update'])->name('user.update');
```

## 4.3 – Validar el formulario de configuración

Ahora vamos a proceder a validar el formulario con un método que nos proporciona Laravel, incluyendo los campos a validar y las reglas de validación:

```
$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]
]);
```

La construcción `'unique:users,nick, '.$id` que vemos en el campo de Nick, igual que la de email, es una particularidad extra que nos brinda Laravel. Gracias a esta construcción hace más de una comprobación SQL, comprueba que el Nick no existe y que además si existe que sea solo una vez y que coincida con el id de usuario por si se da el caso de que quiera cambiar el nombre, pero no el Nick o el email.

## 4.4 – Actualizar el usuario

A continuación, podemos ver el código completo.

```
public function update(Request $request){
    //Conseguir el usuario identificado
    $user = \Auth::user();
    $id = $user->id;

    //Validacion del formulario
    $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]
    ]);

    //Recoger los datos del formulario
    $name = $request->input('name');
    $surname = $request->input('surname');
    $nick = $request->input('nick');
    $email = $request->input('email');

    //Asignar nuevos valores al objeto del usuario
    $user->name = $name;
    $user->surname = $surname;
    $user->nick = $nick;
    $user->email = $email;

    //Ejecutar consulta y cambios en la base de datos
    $user->update();

    return redirect()->route('config')
        ->with(['message'=>'Usuario actualizado correctamente']);
}
```

## 4.5 – Subir imagen de usuario

Ahora aprenderemos a subir archivos al servidor, en este caso una imagen de usuario que se mostrará en el avatar del mismo.

Primero añadimos un campo al formulario para poder subir un archivo, usando el tipo file. Posteriormente deberemos añadir un método a nuestro controlador que nos permita subir imágenes. En Laravel para guardar imágenes no podemos guardarlas de una manera directa en el sistema de archivos, sino que hay que utilizar una especie de discos virtuales que nos permiten generar Laravel para tener mucho más protegidos los archivos que subimos al servidor y para tenerlo también mucho más organizado entonces tenemos que usar el sistema que tiene esto y el storage que nos provee para guardar las imágenes.

Para ello primero debemos configurar el archivo config/filesystems.php y vamos a agregar unos nuevos discos virtuales que estarán enlazados a unas carpetas que hemos creado dentro de storage/app.

```
'users' => [
    'driver' => 'local',
    'root' => storage_path('app/users'),
    'url' => env('APP_URL').'/storage',
    'visibility' => 'public',
],

'images' => [
    'driver' => 'local',
    'root' => storage_path('app/images'),
    'url' => env('APP_URL').'/storage',
    'visibility' => 'public',
],
```

Modificaremos la siguiente línea del formulario para que se puedan subir archivos tal y como vemos:

```
<div class="card-body">
  <form method="POST" action="{{ route('user.update') }}" enctype="multipart/form-data">
    @csrf
```

Y a continuación modificaremos el controlador.

Primero añadiremos estos dos objetos:

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

Ahora configuramos el Controlador:

```
//Subir la imagen
$image_path = $request->file('image_path');
if($image_path){
    //Asignar a la imagen un nombre unico
    $image_path_name = time().$image_path->getClientOriginalName();

    //Guardar la imagen en la carpeta storage/app/users
    Storage::disk('users')->put($image_path_name, File::get($image_path));

    //Seteo el nombre de la imagen en el objeto user
    $user->image = $image_path_name;
}
```

Con esto ya se sube la imagen.

## 4.6 – Mostrar avatar

Para mostrar el avatar deberemos crear un método en nuestro controlador que nos saque la imagen del storage.

```
public function getImage($filename){
    $file = Storage::disk('users')->get($filename);
    return new Response($file, 200);
}
```

Y en el formulario mostraremos la imagen si existe:

```
<label for="image_path" class="col-md-4 col-form-label text-md-right">{{ __('Avatar') }}</label>
@if(Auth::user()->image)

@endif
```

## 4.7 – Avatar en el menú

Hemos puesto el código de la imagen en una view dentro de una carpeta llamada Include para hacer el código más limpio. Donde queramos tener la imagen solo deberemos realizar un include de este código. También vamos a crear una hoja de estilos para el avatar de forma que cambie según donde se imprima.

Para poner el avatar en el menú nos dirigimos a app.blade.php y modificamos este archivo añadiendo un nuevo <li>

## 4.8 – Solo para usuarios autenticados

Si intentamos entrar en configuración sin estar logueados nos devolverá una página con error, pero nosotros no queremos que esto suceda queremos que el middleware de autenticación no deje entrar a ninguna parte hasta que estemos identificados.

Para conseguir esto debemos incluir este middleware en los controladores como vemos a continuación:

```
class UserController extends Controller
{
    public function __construct()
    {
        $this->middleware('auth');
    }
}
```

## 5 – Imágenes de la aplicación (clon Instagram)

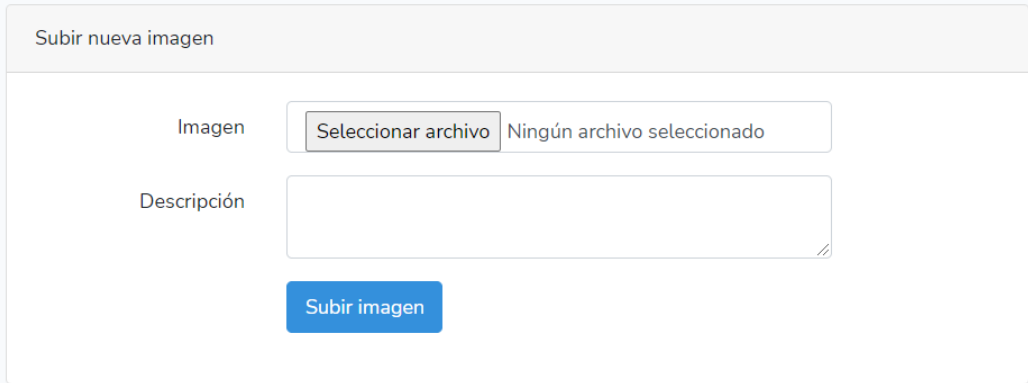
---

### 5.1 – Formulario para subir imágenes

Vamos a crear el controlador de imágenes, pero antes pondremos los enlaces del navbar correctamente para que Inicio dirija al Home, Subir imágenes a la vista que le vamos a crear, etc.

Una vez creado el controlador ImageController lo primer que vamos a hacer es restringir el acceso con el middleware como hicimos en el UserController.

Creamos el método create que nos devolverá una vista con el formulario para subir imágenes, muy parecido al que vimos en configuración del usuario.



Subir nueva imagen

Imagen  Ningún archivo seleccionado

Descripción

## 5.2 – Subir imagen

Este sería el controlador listo para subir las imagenes

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Storage;
use Illuminate\Support\Facades\File;
use App\Models\Image;

class ImageController extends Controller
{
    public function __construct()
    {
        $this->middleware('auth');
    }

    public function create(){
        return view('image.create');
    }

    public function save(Request $request){

        // Validacion del formulario
        $validate = $this->validate($request, [
            'description' => ['required'],
            'image_path' => ['required', 'image']
        ]);

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

        // Asignar valores la 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;
        }

        //Guardar objeto en la base de datos
        $image->save();

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



## 5.3 – Listado de imágenes

Vamos a listar todas las imágenes que están en la aplicación en la página de inicio paginándolas, esto gracias al ORM es una tarea muy sencilla.

En el controlador de HomeController incluiremos el espacio de nombre de Images para poder acceder a la base de datos de las imágenes.

```
use App\Models\Image;
```

En el controlador haremos que mediante los métodos ORM del objeto Image se pasen los datos de las imágenes a la vista de home.

```
public function index()
{
    $images = Image::orderBy('id', 'desc')->get();

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

En la vista home haremos un foreach que recorra el array que hemos mandado por el controlador para que liste las imágenes y las incluya dentro de la card para mantener el formato que queremos.

```
@foreach($images as $image)
    <div class="card pub_image">
        <div class="card-header">
            @if($image->user->image)
                <div class="container-avatar">
                    
                </div>
            @endif
            <div class="data-user">
                {{ $image->user->name.' '.$image->user->surname | @.$image->user->nick }}
            </div>
        </div>

        <div class="card-body">

        </div>
    </div>
@endforeach
```

Ahora vamos a mostrar la imagen en el body de tarjeta (las tarjetas son los div de clase card, esto viene con bootstrap), para ello debemos crear un método que devuelva las imágenes del storage en el ImageController.

Debemos incluir el nombre de espacio `Illuminate\Http\Response`; y también debemos crear la ruta en el web.php

```
Route::get('image/file/{filename}', [App\Http\Controllers\ImageController::class, 'getImage'])
    ->name('image.file');
```

En el controlador de ImageController para mostrar la imagen debemos crear el siguiente método:

```
public function getImage($filename) {
    $file = Storage::disk('images')->get($filename);
    return new Response($file, 200);
}
```

Y en el home deberemos indicar la siguiente ruta para que se muestre la imagen:

```
<div class="card-body">
    <div class="image-container">
        
    </div>
</div>
```

Hemos modificado el css para cambiar el estilo y dejarlo mejor, pero eso ya entra dentro de los términos de maquetación que están fuera de este proyecto.

## 5.4 – Paginación en Laravel

Para ello solo debemos sustituir el método `get` por el método `simplePaginate` en el controlador del `home` e indicar el número de elementos por página.

```
public function index()
{
    $images = Image::orderBy('id', 'desc')->simplePaginate(5);

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

Y para mostrar los links para cambiar de página solo debemos añadir esto donde nos interese que aparezca:

```
<!-- PAGINACION -->
<div class="clearfix"></div>
{{ $images->links() }}
```

En este caso lo hemos puesto al final del listado.

## 5.5 – Maquetación de likes

Hemos descargado unas imágenes de corazones para los likes, luego dentro de la carpeta `public` hemos creado la carpeta `img` donde los hemos guardado y para acceder a esta imagen lo hacemos de la siguiente manera:

```
<div class="likes">
    
</div>
```

Hemos modificado su estilo con `css` para dejarlo más bonito.

## 5.6 – Número de comentarios

Vamos a añadir el número de comentarios de cada publicación de una forma muy sencilla como vemos a continuación:

```
<div class="comments">
    <a href="" class="btn btn-warning btn-sm btn-comments">
        Comentarios ({{ count($image->comments) }})
    </a>
</div>
```

## 5.7 – Detalle de la imagen

Para mostrar el detalle de la imagen vamos a crear un método en nuestro ImageController, este método recibirá un id por la url de la imagen que queremos mostrar en detalle, este método nos llevará a una vista donde se mostrarán los detalles, y esta vista será muy parecida a la de home pero sin foreach.

```
public function detail($id){
    $image = Image::find($id);

    return view('image.detail', [
        'image' => $image
    ]);
}
```

Crearemos la ruta en web.php y crearemos un link para acceder a esta página como podemos ver a continuación:

```
<div class="data-user">
    <a href="{{ route('image.detail', ['id'=>$image->id]) }}">
        {{ $image->user->name.' '.$image->user->surname }}
        <span class="nickname">{{ ' | @'.$image->user->nick }}</span>
    </a>
</div>
```

## 5.8 – Formatear fechas

Laravel nos trae un formato de fecha tipo de las redes sociales por defecto, si queremos formatear la fecha de tal modo que nos indique hace cuando de esa fecha, por ejemplo hace cuanto que se subió dicha foto tenemos la siguiente forma de hacerlo por Blade.

**`diffForHumans(null, false, false, 1) }}`**

Pero si queremos que sea aun más profesional debería usarse de formar local para ello se puede mejorar lo anterior con el siguiente método.

**`locale('es_ES')->diffForHumans(null, false, false, 1)`**

Vemos a continuación como queda en nuestro proyecto:

```
<span class="nickname date">
  {{ ' | '.$image->created_at->locale('es_ES')->diffForHumans(null, false, false, 1) }}
</span>
```

Y aquí podemos ver el resultado:

@LuviLover | hace 57 minutos

(Para más información consultar la sección de Documentación)

## 6 – Sistema de comentarios

---

### 6.1 – Formulario de comentarios

En la página de detalles vamos a crear un formulario de comentarios y a mostrarlos. Es necesario que en este formulario tengamos un input tipo hidden con la id de la imagen.

### 6.2 – Validar formulario

Los comentarios son una entidad diferente por lo que vamos a crear un controlador nuevo llamado CommentsController. Y validamos este formulario como hemos hecho con los anteriores.

## 6.3 – Guardar los comentarios

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Models\Comment;

class CommentController extends Controller
{
    public function __construct()
    {
        $this->middleware('auth');
    }

    public function save(Request $request) {

        // Validacion del formulario
        $validate = $this->validate($request, [
            'image_id' => ['integer', 'required'],
            'content' => ['string', 'required']
        ]);

        // Recoger datos
        $user = \Auth::user();
        $image_id = $request->input('image_id');
        $content = $request->input('content');

        //Asigno los valores a el nuevo objeto
        $comment = new Comment();
        $comment->user_id = $user->id;
        $comment->image_id = $image_id;
        $comment->content = $content;

        //Guardamos los datos asignados al objeto en la base de datos
        $comment->save();

        //Redirección
        return redirect()->route('image.detail', ['id' => $image_id])
            ->with([
                'message' => "Has publicado tu comentario correctamente"
            ]);
    }
}
```

## 6.4 – Listar comentarios

Para listar los comentarios, al ser estos un objeto del que tenemos ya un array con todos los comentarios solo deberemos hacer lo siguiente:

```
@foreach($image->comments as $comment)
<div class="comment">
    <span class="nickname">{{ '@' . $comment->user->nick }}</span>
    <span class="nickname date">
        {{ ' | ' . $comment->created_at->locale('es_ES')->diffForHumans(null, false, false, 1) }}
    </span>
    <p>{{ $comment->content }}</p>
</div>
@endforeach
```

Además, si queremos que el orden de los comentarios sea de mas nuevo a más antiguo empezando por arriba debemos cambiar la relación de los comentarios en el modelo de imagen de la siguiente forma:

```
//Relacion One to Many (De uno a varios)
public function comments() {
    return $this->hasMany('App\Models\Comment')->orderBy('id', 'desc');
}
```

## 6.5 – Borrar comentarios

Ahora crearemos una forma de borrar los comentarios en el caso de que seamos dueño de dicho comentario o de la imagen donde esta ese comentario.

Lo primero que debemos hacer es crear en nuestro CommentController un nuevo método, esa acción se va a encargar de borrar el comentario. Este método se compone de varias partes.

**Primero:**

```
// Conseguir datos del usuario logueado
$user = \Auth::user();
```

En la variable **\$user** guardamos al usuario que esta autenticado en la sesión, esto es para poder comparar luego su id con la del creador del comentario y la del creador de la imagen y ver si coinciden.

**Segundo:**

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

Gracias al metodo find del objeto Comment podemos obtener las propiedades de un objeto en concreto, en este caso la del comentario que queremos borrar, dándole su \$id ya que es única. De este modo podremos comprobar mas adelante si coincide con la id de quien quiere borrar el comentario.

**Tercero:**

```
// Comprobar si soy el dueño del comentario o de la foto
if($user && ($comment->user_id == $user->id || $comment->image->user_id == $user->id)){
}
}
```

Mediante esta condición haremos que el comentario solo pueda ser borrado si el usuario esta autenticado y además si se cumple la condición de que el id del usuario que publicó el comentario es igual al id del usuario autenticado o bien si el id de la imagen relacionada con dicho comentario es igual a la id del usuario autenticado.

**Cuarto:**

```
$comment->delete();
```

Gracias al ORM tenemos una relación directa entre el objeto y la base de datos por lo cual al usar el método delete sobre el objeto \$comment que queremos borrar también se borra automáticamente el registro de la base de datos.



**Quinto:**

```

//Redirección
return redirect()->route('image.detail', ['id' => $comment->image->id])
    ->with([
        'message' => "Comentario eliminado correctamente"
    ]);
}else{
    //Redirección
    return redirect()->route('image.detail', ['id' => $comment->image->id])
        ->with([
            'message' => "Comentario no se ha eliminado"
        ]);
}

```

Por último, vamos a redireccionar al detalle, según si ha finalizado correctamente o no ira con un mensaje distinto. Es importante indicar el id de la imagen relacionado con el comentario de la forma **\$comment->image->id**

Ahora solo nos queda crear la ruta para este método y crear un botón para eliminar el comentario en el listado de comentarios solo cuando el usuario autenticado sea el dueño de la imagen o el dueño del comentario, básicamente la misma condición que tenemos en el controlador.

En Blade deberemos modificar algunas cosas, por ejemplo para comprobar si el usuario esta autenticado usaremos el **Auth::check()** que nos devolverá true si el usuario esta autenticado y en lugar del objeto \$user que teníamos en el controlador creado en este caso deberemos usar el método **Auth::user()** que nos devolverá el objeto del usuario autenticado. Quedaría de este modo

```

<p>{{ $comment->content }}<br>
@if(Auth::check() &&
    ($comment->user_id == Auth::user()->id ||
    $comment->image->user_id == Auth::user()->id))
    <a href="{{ route('comment.delete', ['id' => $comment->id]) }}"
        class="btn btn-sm btn-danger">
        Eliminar
    </a>
@endif
</p>

```

## 7 – Sistemas de Likes

---

### 7.1 – Método Like

Vamos a crear unos métodos para tener la funcionalidad de dar like o dislike y que cambie el icono también, para ello necesitaremos un nuevo controlador llamado LikeController. A continuación, crearemos los métodos.

```
public function like($image_id){
    //Recoger datos de usuario y la imagen
    $user = \Auth::user();

    // Condicion para ver si ya existe el like y no duplicarlo
    $isset_like = Like::where('user_id', $user->id)
                        ->where('image_id', $image_id)
                        ->count();

    if($isset_like == 0){
        $like = new Like();
        $like->user_id = $user->id;
        $like->image_id = (int)$image_id;

        // Guardar
        $like->save();

        return response()->json([
            'like' => $like
        ]);
    }else{
        return response()->json([
            'message' => 'El like ya existe'
        ]);
    }
}
```

Como vemos este método no tiene redirección porque lo vamos a llamar mediante AJAX. Además hemos casteado el **image\_id** porque debe ser numérico y nos lo guardaba como string. También hemos comprobado que el Like no existe en la base de datos antes de guardarlo y de ser así que no se guarde, en ambos casos se devuelve un objeto JSON con diferentes contenidos.

## 7.2 – Método dislike

El método de dislike es muy parecido al anterior pero con la diferencia que en la consulta pondremos **first()** para que nos aparezca solo el primer resultado y de existir dicho resultado lo borraremos, obteniendo en cualquier caso un objeto JSON con la información.

```
public function dislike($image_id) {
    //Recoger datos de usuario y la imagen
    $user = \Auth::user();

    // Condicion para ver si ya existe el like y no duplicarlo
    $like = Like::where('user_id', $user->id)
                ->where('image_id', $image_id)
                ->first();

    if($like){
        // Eliminar like
        $like->delete();

        return response()->json([
            'like' => $like,
            'message' => 'Has dado dislike correctamente'
        ]);
    }else{
        return response()->json([
            'message' => 'El like no existe'
        ]);
    }
}
```

## 7.3 – Detectar likes

Ahora vamos a detectar si hemos dado un like a una publicación y de ser así cambiar el color del corazón.

```
<div class="likes">

    <!-- Comprobar si el usuario le dio like a la imagen -->
    <?php $user_like = false; ?>
    @foreach($image->likes as $like)
        @if($like->user->id == Auth::user()->id )
            <?php $user_like = true; ?>
        @endif
    @endforeach

    @if($user_like)
        
    @else
        
    @endif

    <span class="number_likes">{{count($image->likes)}}</span>
</div>
```

## 7.3 – Cargar archivos JS y cambiar el color del like

Vamos a incluir un archivo javascript en nuestro proyecto para hacer la funcionalidad de los likes. En la carpeta public/js añadiremos un archivo javascript nuevo.

Cargaremos el archivo llamado main.js en app.blade.php

```
<!-- Scripts -->
<script src="{{ asset('js/app.js') }}" defer></script>
<script src="{{ asset('js/main.js') }}" ></script>
```

Ahora en main.js usaremos jquery para poder cambiar el color del corazón del like, que ya viene cargada esta librería en Laravel.

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

    // Cambiamos el cursor al pasar por encima del boton
    $('.btn-like').css('cursor', 'pointer');
    $('.btn-dislike').css('cursor', 'pointer');

    // Boton de like
    function like() {
        $('.btn-like').unbind('click').click(function() {
            console.log('like');
            $(this).addClass('btn-dislike').removeClass('btn-like');
            $(this).attr('src', 'img/hearts-red.png');
            dislike();
        });
    }
    like();

    // Boton de dislike
    function dislike() {
        $('.btn-dislike').unbind('click').click(function() {
            console.log('dislike');
            $(this).addClass('btn-like').removeClass('btn-dislike');
            $(this).attr('src', 'img/hearts-black.png');
            like();
        });
    }
    dislike();
});
```

En primer lugar, tener que se carga el script tras cargar la página. Luego actualizamos el cursor cada vez que pasemos por encima. Posteriormente haremos que los recursos del botón cambien al contrario al darle click, añadimos el **unbind('click')** para que no se acumulen los bindeos si no cada vez que demos click se irán acumulando las llamadas de las peticiones. Al final del click llamamos a la función contraria para volver a cargar el DOM y que vuelva a funcionar en el siguiente click.

## 7.4 – Peticiones AJAX

Ahora vamos a modificar el javascript anterior para añadir una petición AJAX que modifique la base de datos al darle like/dislike. Primero añadimos una variable global con la url de nuestro proyecto:

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

También añadimos al botón de like un data-id:

```
id }}"
```

Y añadimos lo siguiente en el contador de me gustas

```
<span class="number-likes" data-id="{{ $image->id }}">{{ count($image->likes )}}</span>
```

También hemos añadido al objeto json generado por el controlador lo siguiente:

```
if($like){
    // Eliminar like
    $like->delete();

    $numberlikes = Like::where('image_id', $image_id)->count();

    return response()->json([
        'like' => $like,
        'message' => 'Has dado dislike correctamente',
        'numberlikes' => $numberlikes
    ]);
}
```

Por último aquí tenemos el código JavaScript con Ajax completo

```
// Boton de like
function like() {
    $('#btn-like').unbind('click').click(function() {
        console.log('like');
        $(this).addClass('btn-dislike').removeClass('btn-like');
        $(this).attr('src', 'img/hearts-red.png');
        var id = $(this).data('id');

        $.ajax({
            url: url+'/like/'+$(this).data('id'),
            type: 'GET',
            success: function(response) {
                console.log(response);
                var numberlikes = response['numberlikes'];
                $('#number-likes[data-id='+id+']').html(numberlikes);
            }
        })

        dislike();
    });
}
like();

//Boton de dislike
function dislike() {
    $('#btn-dislike').unbind('click').click(function() {
        console.log('dislike');
        $(this).addClass('btn-like').removeClass('btn-dislike');
        $(this).attr('src', 'img/hearts-black.png');
        var id = $(this).data('id');

        $.ajax({
            url: url+'/dislike/'+$(this).data('id'),
            type: 'GET',
            success: function(response) {
                console.log(response);
                var numberlikes = response['numberlikes'];
                $('#number-likes[data-id='+id+']').html(numberlikes);
            }
        })

        like();
    });
}
dislike();
```

## 7.5 – Like en el detalle

Copiamos el div de los likes del home y nos lo llevamos al apartado del detalle.

Pero como nos da un fallo por la url debemos modificar el código JavaScript como vemos a continuación:

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

## 7.6 – Listar likes

Vamos a crear una página donde se listen todas las publicaciones a las que el usuario autenticado le ha dado like.

Primero vamos a limpiar el código del programa haciendo un include de las tarjetas. Cada tarjeta es una de las imágenes que se muestran por lo que nos hemos llevado ese código a **Includes/image.blade.php** y luego hacemos un include donde esté ese código, tanto en home como en likes/index, como podemos ver a continuación.

**En home:**

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

**En likes/index:**

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

En el controlador de LikeController hemos dejado el método index como podemos ver a continuación, muy parecido a otros listados que hemos hecho anteriormente:

```
public function index() {
    $user = \Auth::user();
    $likes = Like::where('user_id', $user->id)->orderBy('id', 'desc')
        ->simplePaginate(5);

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

## 8 – Borrado y modificación de imágenes

---

### 8.1 – Botones de las imágenes

Vamos a añadir unos botones en el detalle de la imagen de actualizar y borrar, pero estos botones solo pueden aparecer cuando ese detalle lo esté viendo el dueño de la imagen, lo haremos de la siguiente forma:

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

Ahora crearemos los métodos y los enlaces necesarios para que estos botones funcionen.

### 8.2 – Eliminar imagen

Creamos un método de borrado en el controlador de ImageController. En primer lugar vamos a cargar los modelos de Images(que ya está), Comments y Likes:

```
use App\Models\Image;
use App\Models\Comment;
use App\Models\Like;
```

Ahora en el método de delete vamos a buscar al usuario autenticado, el objeto de la imagen según el \$id que vamos a recibir por la url y los comentarios y likes asociados a esta imagen mediante el \$id que recibimos.

```
public function delete($id){
    $user = \Auth::user();
    $image = Image::find($id);
    $comments = Comment::where('image_id', $id)->get();
    $likes = Like::where('image_id', $id)->get();
```

Ahora vamos a realizar una condición para que solo el dueño de la imagen pueda borrar las imágenes.

```
if($user && $image->user->id == $user->id){
```

Si se cumple esta condición pasaremos al borrado.



En primer lugar, borraremos los comentarios asociados a la imagen, seguido de borrar los likes asociados a la imagen, posteriormente se eliminar el fichero de imagen (Es decir la imagen en si misma) del directorio del storage y por último se borrará el registro de la base de datos.

```
// Eliminar los comentarios
if($comments && count($comments) >= 1){
    foreach($comments as $comment){
        $comment->delete();
    }
}
```

Para borrar los comentarios primero hago una condición donde veo si existen comentarios y de ser así si el número es igual o mayor a uno. Si

se cumple esta condición se recorre con un foreach el array de **\$comments** borrando cada uno de los **\$comment** con el método **delete()**;

Luego haremos exactamente lo mismo con los likes.

Ahora tenemos que acceder al storage para borrar del disco la imagen asociada.

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

Usaremos la propiedad **disk()** del objeto **Storage** (que ya teníamos cargado) para acceder al disco virtual **images**. Como lo tenemos seleccionado podemos usar el método **delete()** que trae el objeto **Storage** para borrar la imagen dándole el **image\_path** que es la dirección donde se guarda la imagen y este registro está en la base de datos asociada al objeto.

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

Y por último borramos el registro de la base de datos de la imagen con el método **delete()**

Ahora indicaremos un mensaje según se haya eliminado correctamente o no tras el **delete()**

```
$message = array('message'=> "La imagen se ha borrado ");
}else{
    $message = array('message'=> "La imagen no se ha borrado ");
}
```

Y para terminar haremos una redirección con una sesión flash que lleve el mensaje:

```
return redirect()->route('home')->with($message);
```

## 8.3 – Modal en Bootstrap 4

Vamos a utilizar las funcionalidades de bootstrap para que aparezca un modal cuando se quiera borrar una imagen para confirmar o no la elección.

Hemos modificado el código de muestra que nos viene w3schools y lo hemos dejado de la siguiente forma:

```
<!-- 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">¿Estas seguro?</h4>
        <button type="button" class="close" data-dismiss="modal">&times;</button>
      </div>

      <!-- Modal body -->
      <div class="modal-body">
        Si eliminas esta imagen nunca podras recuperarla. ¿Estas seguro de querer borrarla?
      </div>

      <!-- Modal footer -->
      <div class="modal-footer">
        <a href="{{ route('image.delete', ['id' => $image->id]) }}" class="btn btn-success">
          Borrar definitivamente
        </a>
        <button type="button" class="btn btn-danger" data-dismiss="modal">Cancelar</button>
      </div>
    </div>
  </div>
</div>
```

## 8.4 – Formulario de edición de imágenes

Ahora vamos a crear un nuevo método en el controlador para poder acceder al formulario de editar la información de una imagen, pero solo podrá acceder a dichos datos el dueño de la misma. Lo haremos del siguiente modo:

```
public function edit($id) {
    $user = \Auth::user();
    $image = Image::find($id);

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

---

Ahora vamos a crear el formulario en la ruta **image/edit.blade.php** ya que desde el método creado se nos manda a dicha ruta con la información de una imagen en concreto. Este método es muy parecido al de subir imágenes, lo copiaremos y modificaremos un poco.

## 8.5 – Actualizar imágenes

Ahora vamos a crear un método para actualizar los datos de la imagen con lo que mandamos por el formulario.

```
public function update(Request $request) {
    // Validacion del formulario
    $validate = $this->validate($request, [
        'description' => ['required'],
        'image_path' => ['image']
    ]);

    // Indicamos que la variable es falsa por si no se envía una nueva foto
    // y solo se esta cambiando la descripción
    $image_path = false;

    //Recogemos los datos
    $image_id = $request->input('image_id');
    $image_path = $request->file('image_path');
    $description = $request->input('description');

    //Conseguir objeto image
    $image = Image::find($image_id);

    //Setear descripcion
    $image->description = $description;

    //Subir fichero y setear el path
    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;
    }

    //Actualizar registro
    $image->update();

    return redirect()->route('image.detail', ['id' => $image_id])
        ->with(['message' => "Imagen actualizada"]);
}
```

## 9 – Gente y buscador

---

### 9.1 – Página de gente

Ahora vamos a crear una página nueva donde se listarán todos los usuarios de la plataforma para poder visitar sus perfiles.

Primero creamos un controlador index que liste a los usuarios en orden

```
public function index(){
    $users = User::orderBy('id', 'desc')->simplePaginate(5);

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

Y ahora vamos a crear la vista.

### 9.2 – Buscador de gente

Vamos a crear primero el método del buscador y luego a realizar un formulario de búsqueda.

Vamos a modificar levemente el método index para incluir una búsqueda.

```
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')
            ->simplePaginate(5);
    }else{
        $users = User::orderBy('id', 'desc')->simplePaginate(5);
    }

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

Al tener el parámetro search como opcional en la url puede servir tanto como para listar todos los usuarios como para listar una búsqueda según lo que enviemos por el search en el formulario de búsqueda que vamos a hacer a continuación.

## 9.3 – Formulario del buscador

Vamos a crear un pequeño formulario de buscador en la página de gente y también algo de código javascript para que cuando hagamos submit nos añada la búsqueda a la url de gente ya que es el funcionamiento del método que hemos hecho.

El formulario que tenemos es este:

```
<form method="GET" action="{{ route('user.index') }}" id="buscador">
  <div class="row">
    <div class="form-group col">
      <input type="text" id="search" name="search" class="form-control" />
    </div>
    <div class="form-group col btn-search">
      <input type="submit" value="Buscar" class="btn btn-success" />
    </div>
  </div>
</form>
```

Y en el archivo de javascript main.js añadimos esta funcionalidad:

```
$('#buscador').submit(function(){
  $(this).attr('action', url+'/gente/'+$('#buscador #search').val());
});
```

Lo que estamos haciendo es seleccionar el id del formulario y tendremos una función de callback con la cual modificaremos el valor del action del buscador según el contenido del buscador para ello tenemos la url original concatenada con '/gente' y además concatenamos el valor del input con el id search del formulario.

Y con esto finalizamos el proyecto de Laravel.

## Documentación

---

¿Qué es ORM?

<https://www2.deloitte.com/es/es/pages/technology/articles/que-es-orm.html>

Documentación Eloquent

<https://laravel.com/docs/8.x/eloquent>

Descargar DIA

<https://sourceforge.net/projects/dia-installer/>

Descargar conector de base de datos J para Apache NetBeans

<https://dev.mysql.com/downloads/file/?id=506032>

Autenticación y registros de usuarios en Laravel 8.x

<https://laravelarticle.com/laravel-8-authentication-tutorial>

Problemas con node.js y npm

<https://github.com/vuejs/vue-loader/issues/1859>

Página para descargar íconos

<https://www.iconsdb.com/>

Formateo de fechas en Laravel

<https://es.stackoverflow.com/questions/229376/uso-de-m%C3%A9todo-diff-for-humans-en-laravel>

Código de modal de Bootstrap 4

[https://www.w3schools.com/bootstrap4/bootstrap\\_modal.asp](https://www.w3schools.com/bootstrap4/bootstrap_modal.asp)