**1. Introducción**

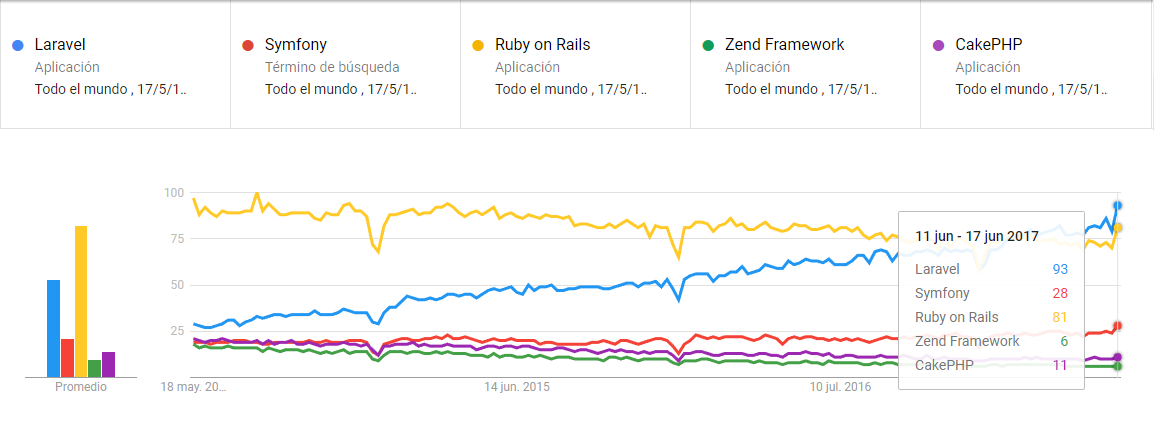
La curiosidad y ganas de aprendizaje sobre el mundo de la programación web me ha llevado a la creación de este trabajo.

El objetivo de este trabajo es la creación de un sistema gestor de contenidos “CMS“ utilizando el framework de PHP Laravel, el cual nos permitirá crear una estructura para la gestión y administración de contenidos de nuestra página web.

La idea principal de nuestro CMS será la de crear una interfaz que controlará una base de datos la cual almacenará el contenido de nuestra página web.

He escogido el framework Laravel, ya que es uno de los frameworks que más ha crecido en los últimos años.

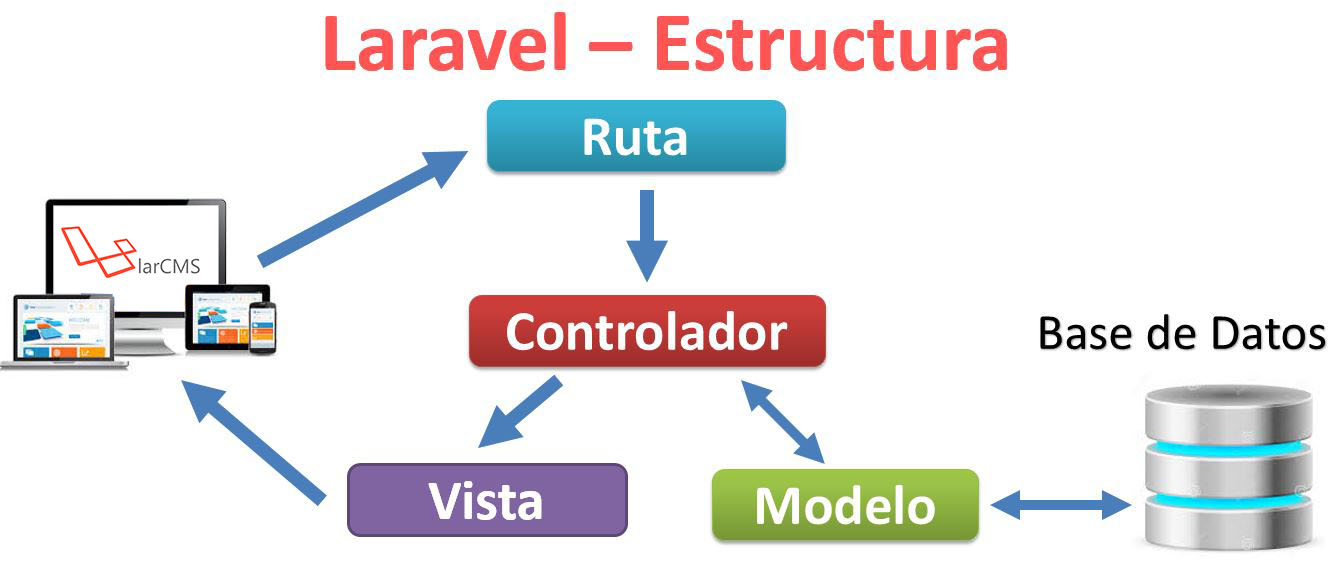
Podemos observar el Interés a lo largo del tiempo de los principales frameworks de PHP y comprobamos que Laravel está en crecimiento, superando actualmente a Ruby on rails, el framework líder de la última década.



Comparativa de interès de los diferentes frameworks de PHP [Google Trends]

Que nos proporciona Laravel? [[1]](https://medium.com/@realvjy/why-laravel-is-best-php-framework-98a2784d76dc)

1. **Modularidad**: Posibilidad de separar los componentes. Laravel trabaja con modelo vista controlador [MVC].
2. **Motor de plantillas Blade**: Disponemos de un motor de plantillas con el cual podremos escribir código directamente sobre plantillas para ahorrarnos tiempo y esfuerzo.
3. **Auenticación**: Laravel mejora mucho en la simplicidad de la autenticación de usuarios, siendo esta parte imprescindible para el control de usuarios del gestor de contenidos.
4. **Almacenamiento en caché:** Laravel nos proporciona la posibilidad de gestionar la memoria caché directamente des de un fichero alojado en *config/cache.php*
5. **Laravel Inversion of Control (IoC)**: Herramienta para administrar dependencias de clase, donde una inyección de dependencias es un patrón de diseño orientado a objetos el cual subministra objetos a una clase en lugar de ser la propia clase la que cree el objeto.
6. **Enrutamiento**: Nos proporciona facilidad y un buen control en el enrutamiento.
7. **Controladores Restful**: Separación de la lógica para servir peticiones HTTP GET i POST.
8. **Generador de consultas de DB:** Laravel nos proporciona una interfaz para generar y ejecutar querys.



**2. Idea de la aplicación web**

El objetivo de este trabajo de final de grado, no es únicamente la creación de una aplicación web, más bien es el aprendizaje y conocimiento de una tecnología desconocida personalmente, como es el uso de un framework desconocido, sobre un lenguaje conocido.

La idea principal es la creación de gestor de contenidos reproduciendo el conocido Wordpress, utilizando gestión de publicaciones des de un administrador y reproduciéndolos en el frontend de la web.

Por otro lado, el correcto manejo de usuarios y de sus correspondientes permisos es muy importante en la aplicación.

Con este gestor de contenidos, el usuario principal administrador, podrá crear autores, los cuales podrán acceder al gestor de contenidos y manipular sus propias publicaciones. También podrá crear moderadores que tendrán permisos sobre las publicaciones de todos los autores.

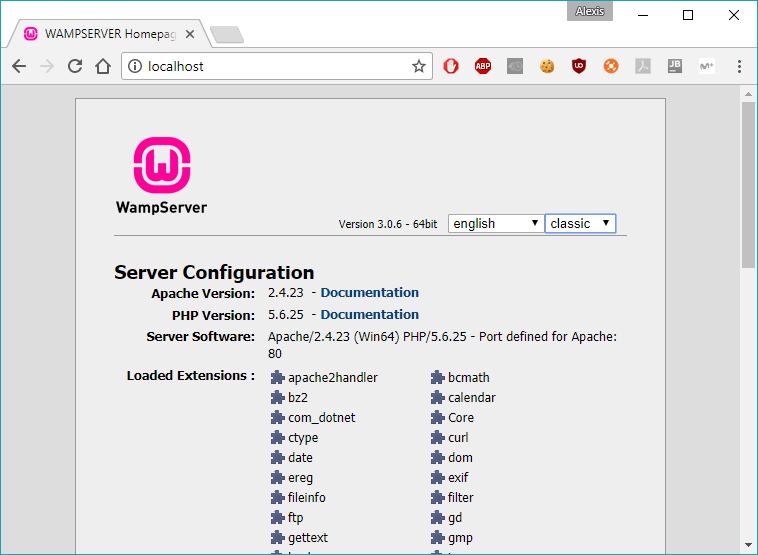
El único usuario que tendrá acceso a la gestión de usuarios será el administrador, evitando así problemas de seguridad.

**3. Entorno de trabajo**

1. **WampServer**

La aplicación correrá sobre Windows i se utilizará el entorno de desarrollo WampServer[[2]](http://www.wampserver.com/en/) que nos permite crear aplicaciones web con Apache2, PHP y MySQL.

Una vez instalado WampServer ya podremos acceder a nuestro navegador e ingresar la dirección localhost:



En Wamp la ruta donde añadimos nuestros proyectos es: wamp\www

1. **Composer**

Composer[[3]](https://getcomposer.org/) es el gestor de paquetes que nos proporcionará un formato estándar para gestionar las dependencias del software PHP y sus librerías, muy similar al “npm” de Node.js. Lo descargamos de la web y solo hay que añadirle la ruta del ejecutable del php.

Al finalizar la instalación, podremos comprobar la versión de composer de una manera sencilla:



1. **Laravel**

La actual versión de Laravel[[4]](https://laravel.com/docs/5.4) en la creación de este trabajo es la 5.4.

Laravel utiliza composer para gestionar sus dependencias, así que instalaremos Laravel utilizando Composer:

composer global require “laravel/installer”

Esta comanda nos instala laravel y nos genera los ficheros adecuados en el directorio

C:\Users\Alexis\AppData\Roaming\Composer

Entre los ficheros generados se encuentra el json adecuado con el siguiente contenido:

{

“require”:{ “laravel/installer”:”^1.3”}

}

Podemos comprobar nuestra versión de Laravel instalada.



1. **Crear proyecto**

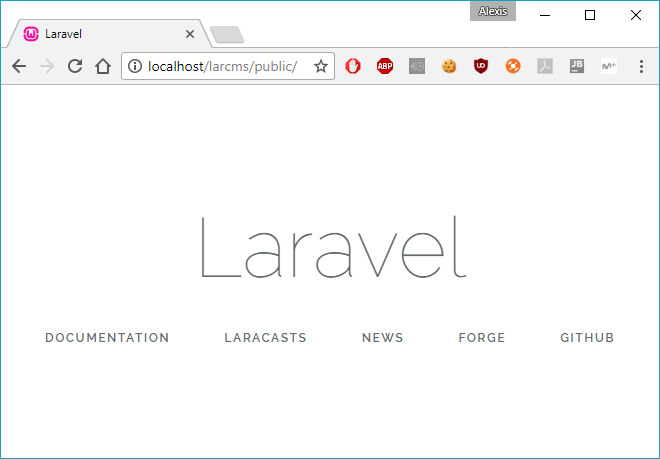
Crearemos un proyecto en la carpeta wamp\www de la siguiente manera:

laravel new larcms

Ejecución del proyecto

php artisan serve

A continuación ya podremos acceder des del navegador a la carpeta pública de nuestro proyecto: “localhost/larcms/public” o bien “localhost:8000”

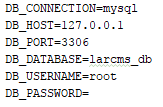


1. **Creación de la base de datos**

Utilizaremos MySQL[[5]](https://dev.mysql.com/downloads/mysql/) para gestionar la base de datos, con el usuario root y sin contraseña podremos acceder a la consola de MySQL.

CREATE DATABASE larcms\_db

Y en el fichero “.env” situado en la raíz de nuestro proyecto seleccionaremos la base de datos como la base de datos a utilizar.



**4. Estructura de la base de datos**

**4.2 Database: Migrations**

Las migraciones en Laravel equivalen al control de versiones para la base de datos, pudiendo modificar el esquema de la base de datos de la aplicación, nos permitirá crear y manipular tablas en todos los sistemas de base de datos compatibles con Laravel.

Todas nuestras tablas seguirán el patrón “lc\_nombreTabla” siguiendo el formato de WordPress con sus tablas “wp\_...”.

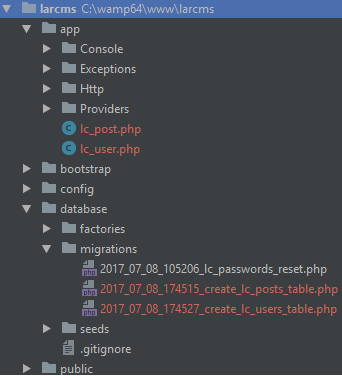
Mostraremos como creando un modelo y una migración podremos crear tablas en nuestra base de datos

Creación de la tabla que almacenará los posts de nuestro CMS.

Para crear una tabla, crearemos un modelo por comada que automáticamente creará el fichero migration. Dentro del fichero migration podremos crear los campos de nuestra tabla. La opción –m al crear un modelo, nos associará una migration.

Para las tablas que no necesiten modelo, podremos hacer “make:migration” en vez de make:model

php artisan make:model lc\_post -m



Podemos ver como nos ha creado los modelos “lc\_post y lc\_users” y también la comanda nos creará la migración identificada por la fecha de creación, y todas las migraciones quedaran almacenadas en la carpeta “migrations”.

Para añadir los campos a una tabla dentro de la migration lo haremos de la siguiente manera:

public function up()  
{  
 Schema::create('lc\_posts', function(Blueprint $table){  
 $table->increments('id');  
 $table->integer('autor\_id')->unsigned();  
 $table->foreign('autor\_id')->references('id')  
 ->on('lc\_users’)->onDelete('restrict');  
 $table->string('title');  
 $table->string('slug',250)->unique();  
 $table->text('excerpt');  
 $table->text('body');  
 $table->string('imagen')->nullable();  
 $table->timestamps();  
 });  
}

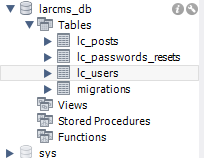
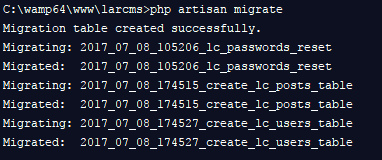
Un Post estará identificado por una ID i hará referencia a la id de un usuario como clave foránea. Tendrá un título, descripción e imagen de cabecera.

El término “Slug” hace referencia al título de una noticia, es un término muy importante para el posicionamiento SEO, cambia espacios por guiones y elimina caracteres especiales.

Por otro lado encontramos el campo excerpt, que de la misma manera que WordPress, permite a los usuarios mostrar más contenido en menos espacio, para mostrar contenido en la portada con enlaces a artículos completos.

Finalmente para realizar la migración, ejecutaremos el siguiente comando:

php artisan migrate

 Podremos comprobar el resultado creando usuarios, passwords\_resets y posts.

**Crear una migración para añadir un campo a la tabla lc\_posts.**

Si nos olvidamos de un campo de nuestra tabla, lo podemos añadir fácilmente. Como las migraciones consisten en un control de versiones, crearemos una nueva migración la cual asociaremos a la tabla ‘lc\_posts’ i automáticamente nos generará el schema necesario para modificar nuestra tabla. De la siguiente manera:

php artisan anadir\_fecha\_publicacion\_columna –table=lc\_posts

Y a continuación solo tendremos que añadir el campo fecha de publicación a nuestra tabla de posts.

public function up()  
{  
 Schema::table('lc\_posts', function (Blueprint $table) {  
 $table->timestamp('published\_at')->nullable();  
 });  
}

De la misma manera en la función down podemos hacer que elimine la tabla si queremos hacer un revert.

public function down()  
{  
 Schema::table('lc\_posts', function (Blueprint $table) {  
 $table->dropColumn('published\_at');  
 });  
}

Finalmente ya podemos ejecutar la migration.

php artisan migrate

**4.3 Database: Seeding**

Laravel incluye un método simple para inicializar los valores de una base de datos, pudiendo inicializar nuestras tablas.

Dentro del fichero DatabaseSeeder se hacen todas las llamadas a los seeders.

public function run()  
{  
 $this->call(UsersSeeder::*class*);  
 $this->call(PostsSeeder::*class*);  
}

Un seed será un fichero php que almacenarà los datos de inicialización de la tabla, un ejemplo es la creación de un usuario inicial:

public function run()  
{  
 // Reseteamos la tabla de usuarios  
 DB::table('lc\_users')->truncate();  
  
 // Generamos el usuario inicial  
 DB::table('lc\_users')->insert(  
 [  
 'name' => 'Alexis',  
 'email' => 'Alexis@udg.com',  
 'password' => bcrypt('alexis')  
 ]  
 );  
}

Inicializamos tanto usuarios como posts utilizando seeds, nos ayudamos a la generación de post con la librería para crear frases y palabras aleatorias “Faker/Factory” ya incluida en el composer.

Cuando ya hemos creado los ficheros seeds, procedemos a su ejecución:

php artisan db:seed

Tabla Users;

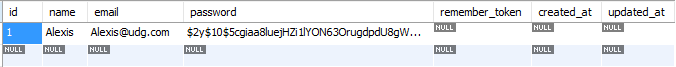
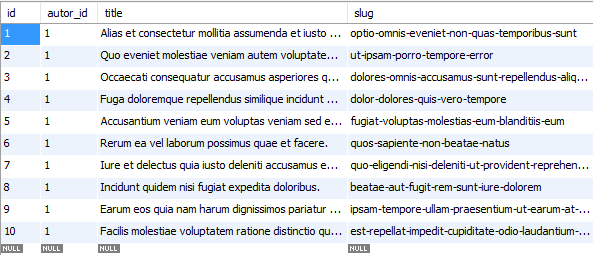


Tabla Posts:



**5. Elementos Utilizados de Laravel**

En los siguientes puntos se explicará el uso de elementos de laravel y como han sido aplicados al proyecto, no están todas las características de laravel, pero si los utilizados en nuestra aplicación.

**5.1 Rutas**

Las rutas de Laravel las encontramos en routes/web.php i nos permiten realizar acciones según la url.

La ruta recibe una URL y sabe a qué método del controlador llamar, el cual decidirá si necesita llamar a una vista, o enviar datos al modelo.

Un ejemplo de una ruta simple es el siguiente:

Route::get('/admin', 'Backend\HomeController@index')->name('admin');

Cuando la aplicación recibe la ruta /admin sabrá que tiene que acceder al controlador Backend\HomeController i buscar el método index que nos devolverá la vista del backend.

En laravel podemos ver el listado de rutas utilizando la comanda:

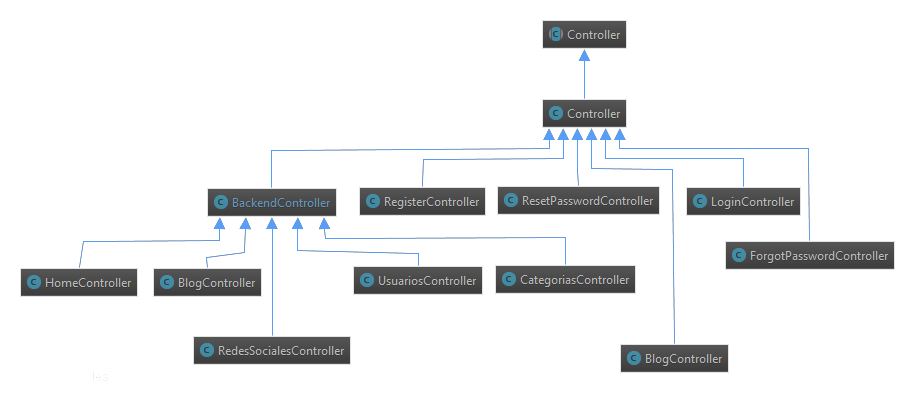
php artisan route:list

|  |  |  |
| --- | --- | --- |
| Rutas del frontend | | |
| Metodo | **Ruta** | **Controlador@Acción** |
| GET | / | BlogController@**index** |
| GET | blog/{post} | BlogController@**muestraPost** |
| GET | categoria/{categoría} | BlogController@**categoria** |
| GET | autor/{autor} | BlogController@**autor** |

|  |  |  |
| --- | --- | --- |
| Rutas de autenticación (auth) | | |
| Metodo | **Ruta** | **Controlador@Acción** |
| POST | login | Auth\LoginController@**login** |
| GET | login | Auth\LoginController@**showLoginForm** |
| POST | logout | Auth\LoginController@**logout** |
| GET | logout | Auth\LoginController@**logout** |
| POST | password/email | Auth\ForgotPasswordController@**sendResetLinkEmail** |
| POST | password/reset | Auth\ResetPasswordController@**reset** |
| GET | password/reset | Auth\ForgotPasswordController@**showLinkRequestForm** |
| GET | password/reset/{token} | Auth\ResetPasswordController@**showResetForm** |
| GET | register | Auth\RegisterController@**showRegistrationForm** |
| POST | register | Auth\RegisterController@**register** |

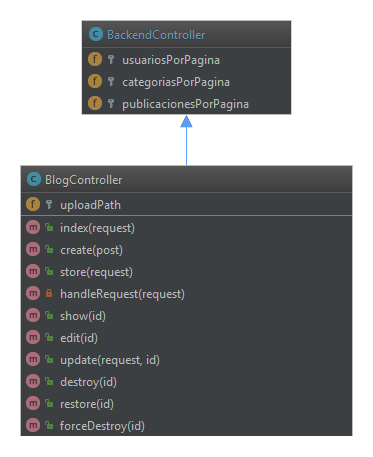
|  |  |  |
| --- | --- | --- |
| Rutas del backend (con autenticación) | | |
| Metodo | **Ruta** | **Controlador@Acción** |
| GET | backend/blog | BlogController@**index** |
| POST | backend/blog | Backend\BlogController@**store** |
| GET | backend/blog/create | Backend\BlogController@**create** |
| DELETE | backend/blog/force-destroy/{blog} | Backend\BlogController@**forceDestroy** |
| PUT | backend/blog/restore/{blog} | Backend\BlogController@**restore** |
| GET | backend/blog/{blog} | Backend\BlogController@**show** |
| DELETE | backend/blog/{blog} | Backend\BlogController@**destroy** |
| PUT | backend/blog/{blog} | Backend\BlogController@**update** |
| POST | backend/categorias | Backend\CategoriasController@**store** |
| GET | backend/categorias | Backend\CategoriasController@**index** |
| GET | backend/categorias/create | Backend\CategoriasController@**create** |
| GET | backend/categorias/{categoria} | Backend\CategoriasController@**show** |
| PUT | backend/categorias/{categoria} | Backend\CategoriasController@**update** |
| DELETE | backend/categorias/{categoria} | Backend\CategoriasController@**destroy** |
| GET | backend/categorias/{categoria}/edit | Backend\CategoriasController@**edit** |
| POST | backend/usuarios | Backend\UsuariosController@**store** |
| GET | backend/usuarios | Backend\UsuariosController@**index** |
| GET | backend/usuarios/confirmar/{users} | Backend\UsuariosController@**confirm** |
| GET | backend/usuarios/create | Backend\UsuariosController@**create** |
| DELETE | backend/usuarios{usuario} | Backend\UsuariosController@**destroy** |
| GET | backend/usuarios{usuario} | Backend\UsuariosController@**show** |
| PUT | backend/usuarios{usuario} | Backend\UsuariosController@**update** |
| GET | backend/usuarios{usuario}/edit | Backend\UsuariosController@**edit** |
| GET | editar-perfil | Backend\HomeController@**edit** |
| PUT | editar-perfil | Backend\HomeController@**update** |
| GET | backend/redes-sociales/{redes\_sociale} | Backend\RedesSocialesController@**show** |
| DELETE | backend/redes-sociales/{redes\_sociale} | Backend\RedesSocialesController@**destroy** |
| PUT | backend/redes-sociales/{redes\_sociale} | Backend\RedesSocialesController@**update** |
| GET | backend/redes-sociales/{redes\_sociale}/edit | Backend\RedesSocialesController@**edit** |

**5.2 Controllers**

Los controllers nos ayudan a organizar el comportamiento de la lógica de rutas. En nuestra aplicación tenemos 3 claras vertientes, los controladores de backend, los controladores de autenticación y los controladores del frontend.

|  |  |  |
| --- | --- | --- |
| Frontend | Backend | Auth |
| BlogController | HomeController | RegisterController |
|  | BlogController | ResetPasswordController |
|  | UsuariosController | LoginController |
|  | CategoriasController | ForgotPasswordController |
|  | RedesSocialesController |  |

Los métodos de un controller van relacionados directamente con los métodos de las rutas. Por lo tanto la estructura de un controller habitualmente es la siguiente:



* Index: Metodo encargado de recuperar datos del modelo y mostrar-los por pantalla.
* Create: Creamos una instancia del modelo.
* Store: A partir de la instancia podemos almacenar los datos en la base de datos a través de un request.
* Edit: Recupera los datos a través de un identificador para su futura actualización en el método update.
* Update: Recibe una request y actualiza los datos del identificador recibido.
* Destroy: Recibe un identificador y una request y elimina la entrada.

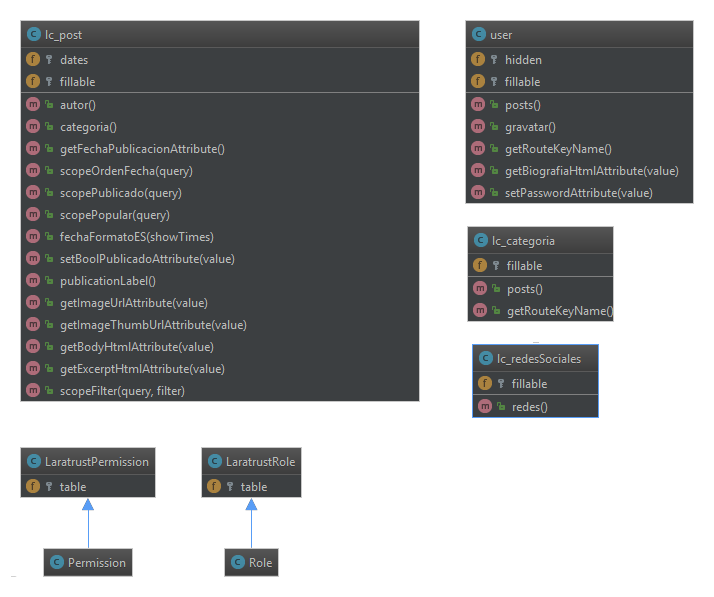
Extra:

* ForceDestroy: Este método es solamente utilizado al eliminar una publicación, ya que al igual que en wordpress, cuando eliminas una publicación està pasa a la papelera (destroy) y es eliminada completamente des de la papelera(forceDestroy).

**5.3 Modelos**

En Laravel existe Eloquent que es un ORM(Object-Relational mapping) el cual nospermite mapear los datos que se encuentran almacenados en la base de datos para mostrarlos en php y viceversa.

Eloquent utiliza los Modelos para enviar o recibir la información a la base de datos.



Todo modelo que necesite enviar información al controlador para ser actualizada en base de datos necesita el atributo “fillable” el cual es una lista de atributos que van a ser modificados en la tabla de la base de datos.

**Eloquent:**

Laravel nos permite jugar con los valores de una manera simple y efectiva utilizando herramientas propias del lenguaje.

* **Accessors:** los accessors nos permiten acceder a los nombre de columnas, por ejemplo si queremos acceder al atributo first\_name simplemente tendremos que crear una función “getFirstNameAttribute” para devolvernos el campo.
* **Mutators:** Los mutators nos permiten modificar campos de la base de datos de la misma manera que los accessors, por otra parte también nos dan la posibilidad de tratar los datos de una manera diferente, es el caso de las fechas, si nosotros marcamos que un campo es una fecha, el mutator nos formatará el campo para que sea una fecha.
* **Fillable:** Eloquent también nos proporciona el campo fillable, que nos obliga a añadir en una lista los campos de nuestra base de datos que van a ser modificados.
* **Scope:** Los scopes reciben una query por parámetro la interpretan y devuelven el resultado. Por ejemplo si creamos el scope **scopePublicado** simplemente tendremos que llamar en el modelo la función **publicado** y nos recogerá las publicaciones cuya fecha de publicación es anterior a la fecha actual.
* **Modelo lc\_Categoria:**

**fillable**: El título y el slug serán elementos modificables al crear una categoria.

**posts():** El modelo categoría tiene como principal función devolver las publicaciones que pertenecen a una categoría.

**getRouteKeyName():** Para poder filtrar la categoria por slug recibido por la url.

* **Modelo lc\_redesSociales:**

**fillable**: Todos los campos de las redes sociales pueden ser modificados, por lo tanto todos ellos aparecerán en la lista de atributos “fillables”.

**Redes():**Devuelve el objeto redes sociales que incluye los links de facebook, twitter, instagram y google plus.

* **Modelo user:**

**fillable**: Se deben guardar los campos, nombre, email, password, slug y biografia de un usuario.

**Hidden:** Los campos password y remember\_token deben estar protegidos y no pueden ser mostrados.

**Posts():** Devuelve la lista de publicaciones de un usuario.

**Gravatar():** Función que devuelve la imagen de un usuario.

**getBiografiaHtmlAttribute():** Devuelve la biografia de un usuario en formato html.

**setPasswordAttribute(value):**Encripta el value y lo assigna como contrasenya del usuario.

* **Modelos Role y Permission:**

Estos modelos son generados automáticamente al instalar LaraTrust (explicado mas adelante).

* **Modelo lc\_post:**

**fillable**: Los campos que se actualizan són: titulo, slug, categoria\_id, body, published\_at, image.

**Dates:** cuando una columna es considerada una fecha, si la incluimos dentro del atributo protected dates nos la associará a un formato de fecha correcto. Por lo tanto incluimos published\_at como una fecha.

**Autor():** Devuelve el autor de una publicación.

**categoria():** Devuelve la categoría de una publicación.

**getFechaPublicaciónAttribute():** Devuelve la fecha de una publicación en formato adecuado.

**scopeOrdenFecha():** Devuelve las publicaciones ordenadas por fecha.

**scopePublicado():**Devuelve las publicaciones cuya fecha de publicación es inferior a la fecha actual.

**scopePopular():** Devuelve las publicaciones ordenadas por número de visualizaciones.

**fechaFormatoES():** Devuelve la fecha de la publicación en formato europeo.

**5.4 Vistas**

Las vistas contienen el código html y separan la lógica de la aplicación de la lógica de presentación. Las vistas utilizan el gestor de plantillas blade que nos permite ejecutar código en nuestro código html.

Podemos incluir código de la siguiente manera @extends(‘content’) ,generar condicionales y bucles entre otros.

**Backend**

Para realizar el backend se ha utilizado la plantilla gratuita AdminLTE Control Panel Template descargada de la siguiente dirección: <https://adminlte.io/>

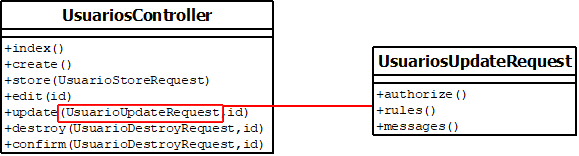
El backend contiene una vista principal que es **backend.blade.php** separada en 3 partes (cuerpo, navbar, sidebar).

**5.5 Requests**

Cuando tratamos directamente con la base de datos, necesitamos validar previamente los datos, para no añadir campos erróneos, o eliminar datos que no debemos.

Los Requests nos permiten tratar dicha validación de una forma simple y sencilla de validar nuestros formularios antes de enviarlos. Solo los utilizaremos en el backend, ya que es el único apartado de nuestra web donde almacenaremos en base de datos.

Si queremos asociar una request a un formulario tenemos que pasarle el objeto request por parámetro a las funciones de nuestro controller y cuando la función update se ejecute, antes comprobará el request.



Se han utilizado 3 metodos en los objetos de request:

* Authorize: Determina si el usuario está autorizado a hacer el request.
* Rules: Genera una lista input-opción, donde por cada input podremos escoger que necesita validación, podremos escoger si el campo es obligatorio, si no puede estar repetido.
* Mensajes: Por cada opción podremos personalizar el mensaje de error que genera el envío del formulario.

**Validación de categorías**

* **CategoriaDestroyRequest**

Para no tener problemas de publicaciones sin categoría, por defecto nuestra aplicación web tendrá la opción de escoger la categoría llamada “Sin Categoría” por lo tanto cuando vayamos a eliminar una categoría tendremos que asegurarnos que la categoría no tiene la id de la categoría por defecto.

Para eso en el request: **CategoriaDestroyRequest** sólo autorizaremos al usuario a borrar la categoría si la categoría no es la categoría por defecto.

* **CategoriaStoreRequest y CategoriaUpdateRequest**

Cuando se intente crear una categoría o modificar una ya creada, la única validación a realizar es la comprobación de que el título y el slug de la categoría sean únicos.

Por lo tanto en nuestras reglas observamos cómo está estructurada la función rules()

public function rules()  
{  
 return [  
 'titulo' => 'required|unique:lc\_categorias|max:255',  
 'slug' => 'required|unique:lc\_categorias|max:255'  
 ];  
}

**Validación de publicación**

En una publicación el único campo que tiene que ser único es el slug, por lo tanto al crear la publicación o al editarla comprobaremos que el slug no esté repetido.

El título, slug, body, categoría y fecha de publicación serán obligatorios, y como casos particulares podremos tratar el formato de fecha y que las imágenes subidas tengan un formato correcto.

$rules =  
 [  
 'titulo' => 'required',  
 'slug' => 'required|unique:lc\_posts',  
 'body' => 'required',  
 'categoria\_id' => 'required',  
 'published\_at' => 'required|date\_format:Y-m-d H:i:s',  
 'image' => 'mimes:jpg,jpeg,bmp,png'  
 ];

**Validación de usuarios**

Cuando validamos usuarios, tendremos que comprobar que tanto el slug como el email no estén ya repetidos, también tendremos que comprobar que el email tenga formato correcto, al igual que en las categorías, cuando hacemos un update a un usuario también tendremos que comprobar tanto email como slug.

Tenemos el caso excepcional de no poder eliminar al usuario principal, por lo tanto cuando iniciemos por primera vez nuestra aplicación y creemos el primer usuario, este tendrá la id de usuario número 1, por lo tanto bloquearemos la posibilidad de eliminar el primer usuario. De la misma manera que el usuario número 1 será administrador, no tendremos la opción de cambiarle el rol a dicho usuario.

**5.6 Laravel Authentication**

Laravel nos proporciona una implementación de autenticación simple, solo necesitamos instalar el siguiente componente:

php artisan make:auth

Al ejecutar el anterior comando, nos creará el controlador HomeContrller y En nuestras rutas automáticamente nos genera el código.

Auth::routes();  
  
Route::get('/admin', 'HomeController@index')->name('admin');

Por otra parte la autenticación de Laravel funciona con “providers” que definen como los usuarios son recuperados.

Simplemente accederemos a nuestro backend añadiendo al final de la url “/admin” o “/login”. El propio instalador ya nos genera las plantillas para visualizar el login, aunque los modificaremos levemente.

La aplicación no permitirá a los usuarios registrarse de una manera automática, ese trabajo solo lo podrá realizar el administrador des del apartado usuarios del backend.

**5.7 Api’s utilizadas**

**Carbon** [**[5]**](http://carbon.nesbot.com/docs/)**:**

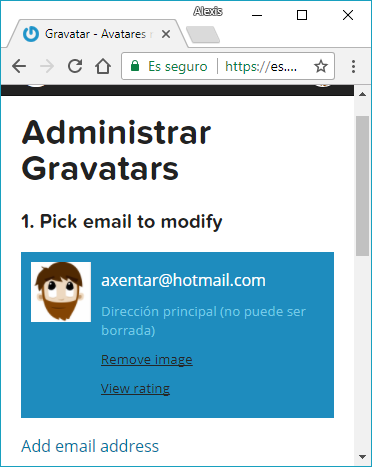
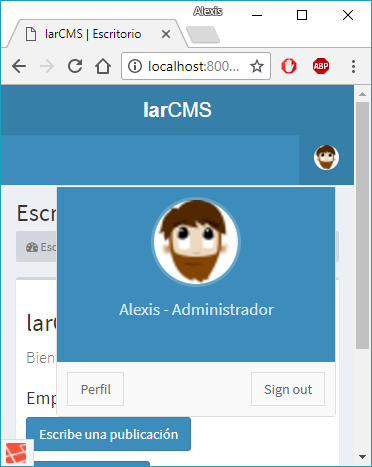
Para trabajar con fechas he utilizado la api Carbon que hereda de DateTime de php.

**Gravatar** [**[6]**](https://es.gravatar.com/)**:**

Gravatar es una herramienta que nos permite asociar una imagen de perfil a una cuenta de e-mail existente, esto es útil en blogs como wordpress, ya que solo poniendo tu e-mail te muestra la imagen de perfil.

Nuestros usuarios de larCMS estarán registrados con un e-mail, si la cuenta de e-mail utilizada está registrada en gravatar y han modificado su imagen de perfil, automáticamente se verà reflejada en nuestra aplicación web, tanto en el apartado de backend como en las publicaciones del frontend.

Si asociamos la imagen en nuestro usuario de gravatar y en nuestro larCMS estamos registrados con la misma cuenta observaremos la misma imagen:

¿Cómo asociamos las imágenes?

El funcionamiento de gravatar consiste en una URL divida en partes.

La primera parte es estándar y siempre es igual:

http://www.gravatar.com/avatar/

La siguiente parte utiliza una versión hasheada del e-mail utilizando md5, es importante que el e-mail esté en minúscula y sin ningún espació. Por eso le aplicaremos la función to\_lower y trim.

/\*Devuelve la imagen utilizada en la web gravatar de un usuario\*/  
public function gravatar(){  
 $email = $this->email;  
 $default = asset("imagenpordefecto.jpg");  
 $size = 40;  
   
 return "https://www.gravatar.com/avatar/"

. md5( strtolower( trim( $email ) ) )

. "?d=" . urlencode( $default )

. "&s=" . $size;  
}

En el método anterior vemos como devuelve la url de gravatar aplicándole el md5 a nuestro email.

El parámetro ?d se encarga de poner una imagen por defecto si no tenemos cuenta en gravatar. Finalmente con el parámetro ?s escogemos la medida del avatar.

**Laravel Markdown** [**[7]**](https://github.com/GrahamCampbell/Laravel-Markdown)

El markdown es un lenguaje de marcado ligero con el que conseguiremos simplicidad para nuestras publicaciones.

Para el proyecto se ha utilizado Laravel Markdown, creado por Graham Cambell, su instalación es sencilla, para ello ayudándonos de composer, instalaremos el Markdown en nuestro proyecto de la siguiente manera:

Composer require Graham-campbell/markdown

Una vez instlado tenemos que registrar el provedor de servicio en nuestro fichero config/app.php dentro de la lista de providers:

GrahamCampbell\Markdown\MarkdownServiceProvider::*class*,

De la misma manera añadimos el alias a la lista de alias.

'Markdown' => GrahamCampbell\Markdown\Facades\Markdown::*class*,

El uso de markdowns es simple, la base de datos almacena el código en formato markdown, y el php se encarga de convertirlo a HTML, en la siguiente imagen podemos ver el efecto.



Para una mayor limpieza del código http, le encargamos la funcionalidad de convertir a html al modelo. No sin antes importar los datos en el modelo.

use GrahamCampbell\Markdown\Facades\Markdown;

Modelo “lc\_post”

public function getBodyHtmlAttribute($value) {  
 return Markdown::convertToHtml(e($this->body));  
}

Vista “post”

<div>  
 {!! $post->body\_html !!}  
</div>

**6. Frontend**

**6.1 SEO-Friendly**

Para hacer que las publicaciones sean encontradas fácilmente des de nuestros navegadores, tendremos que cumplir unas características que abarquen diferentes aspectos.

* Contenido optimizado: Las imágenes llevaran por etiqueta “ALT” el nombre del título, por otra parte, no repetiremos uso de las etiquetas “Heading” marcando “H1” una única vez como elemento principal y destacado.
* Metas:
* Diseño web: El diseño no solo debe ser atractivo de cara al usuario, si no que también debe favorecer la navegación y accesibilidad, adecuando el diseño responsive.
* Indexación: No utilizar contenidos Flash ya que no son indexables para los buscadores.
* Url-amigables: Nuestras urls deben ser simples y efectivas. Indexar una publicación por ID no es efectivo, así que cuando generemos una publicación podremos escoger el “SLUG” donde escogeremos el formato de nuestra URL.

El formato de la url de una publicación es el siguiente:

http://localhost:8000/blog/slug-de-la-noticia

Para modificar el valor de nuestra URL tenemos que modificar el proveedor que nos sirve rutas, se encuentra en la clase RouteServiceProvider.

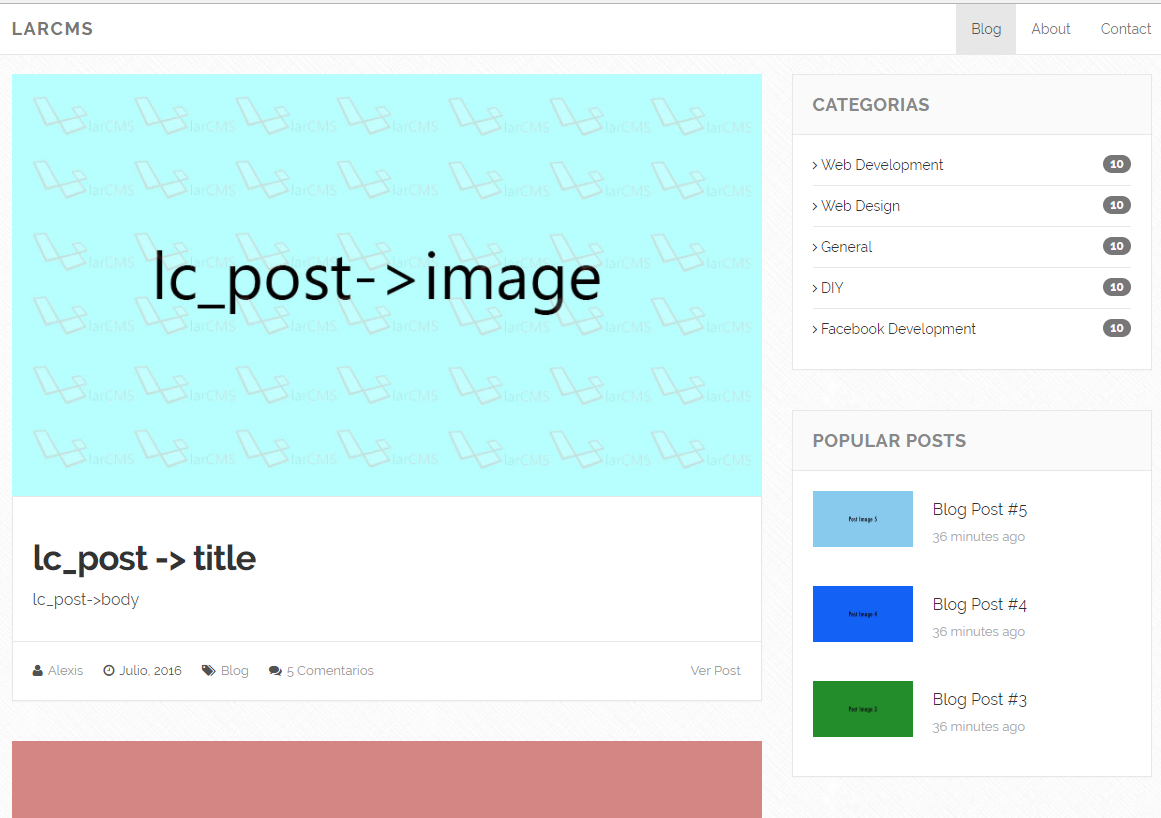
Para personalizar la lógica de resolución de rutas usamos el método Route::bind, el cual recibe el valor del segmento de la URL y devuelve la instancia de la clase que debe ser inyectada en la ruta.

Por lo tanto si queremos personalizar la URL de un post y que sea accedible utilizando el SLUG, necesitamos llamar al modelo del post.

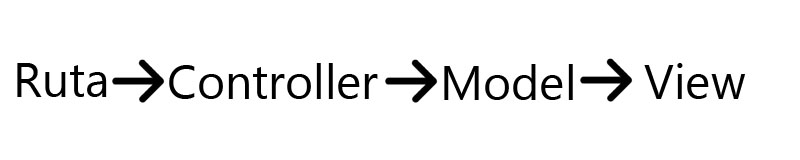
public function boot()  
{  
 parent::boot();  
  
 Route::bind('post',function($slug){  
 return lc\_post::publicado()->where('slug',$slug)->first();  
 });  
}

**6.2 Página principal**

En este paso observaremos como recuperar información de la base de datos, concretamente de la tabla lc\_posts para mostrarla dinámicamente en la lista principal de posts. Podemos ver el formato de un post en la lista de la web.



Uso de controllers



Para mostrar la lista de posts necesitaremos crear un controller. Los controller pueden organizar toda la lógica de gestión de solicitudes en una sola clase situada en app/http/controllers.

Si queremos mostrar la lista de posts a la que llamaremos “blog”, crearemos un controller para gestionar su lógica de gestión de solicitudes.

php artisan make:controller BlogController

Para que una ruta tome como acción un controller lo tenemos que indicar en el fichero de rutas.

/\*  
| Ruta del índice del frontend, muestra la lista de posts.  
\*/  
Route::get('/', [  
 'uses' => 'BlogController@index',  
]);

En la ruta indicamos que va a utilizar el controlador BlogController y la función index. En el controller recogeremos los datos del modelo para enviarlos a la vista. De la siguiente manera:

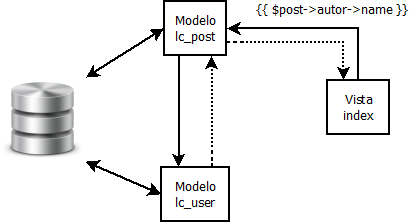
class BlogController extends Controller  
{  
 public function index()  
 {  
 $posts = lc\_post::all();  
  
 return view("frontend.index", compact('posts'));  
 }  
}

y a continuación ya podemos recoger los datos en la vista, almacenados en la variable @posts.

**6.3 Recuperar datos del usuario/categoria en una publicación**

Recuperar Datos del usuario de un post

Para recuperar los datos de un usuario, la vista que muestra la lista de post le pregunta al modelo que tiene el post actual: ¿cuál es el usuario que ha redactado el post? El modelo que guarda el post solo conoce la id del usuario, así que le envía la pregunta al modelo usuario.



**6.4 Filtrando por categoría**

**6.5 Filtrando por autor**

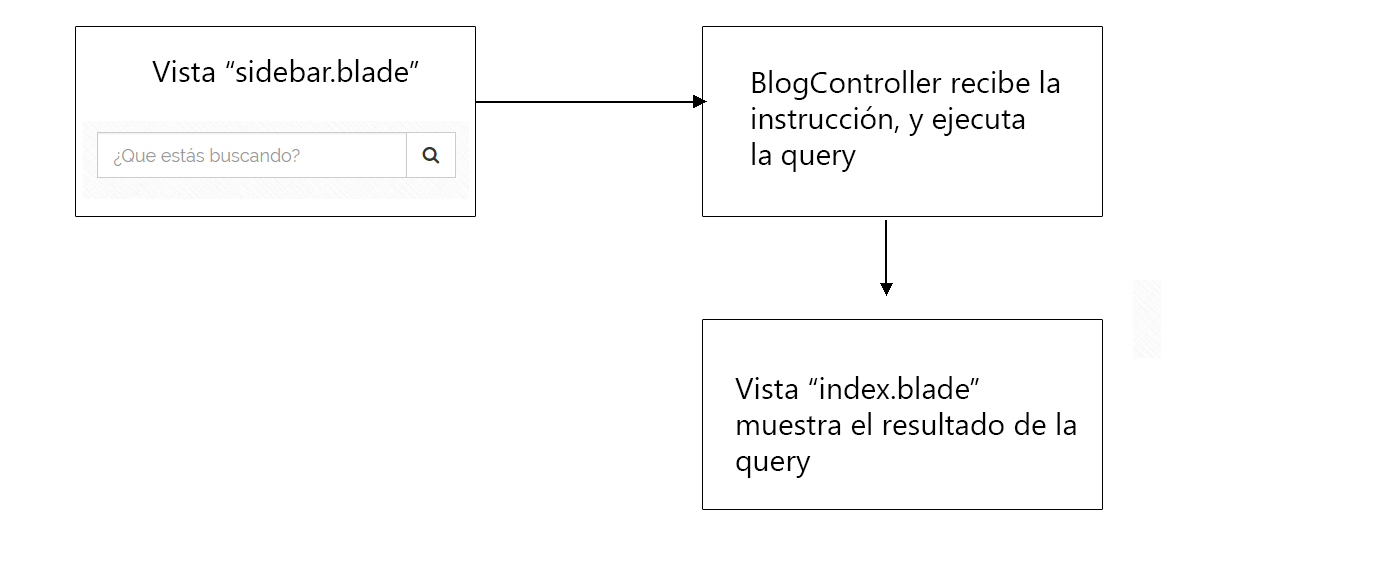
**6.7 Sidebar’s widgets**

En el lateral de nuestra página principal podremos interactuar con las publicaciones. El primer widget que observamos es un buscador de publicaciones.

El apartado de búsqueda será un formulario que enviará la información que queremos buscar al controller del blog.

Si la función principal del controller no recibe ningún parámetro de búsqueda, mostrará todas las publicaciones, en canvio si el formulario de búsqueda le envía la opción de buscar modificaremos la query para que el título,slug, excerpt, descripción contenga la palabra buscada.

De la misma manera, podremos buscar por autor o por categoría.

****

**6.7 Formato de una publicación**

**6.7 Página de error**

**7. Backend**

**7.1 Estructura del backend**

Nuevo controller Backend Controller.php

LoginController

Rules https://laravel.com/docs/5.0/validation

**6.2 Mostrando la lista de publicaciones**

**6.3 Añadiendo una publicación**

**laravelCollective/html (video 31)**

[**https://laravelcollective.com/docs/5.2/html**](https://laravelcollective.com/docs/5.2/html)

**Mirar requests**

**Creación de thumbnails** [**https://packagist.org/packages/intervention/image**](https://packagist.org/packages/intervention/image)

**Simplemde** [**https://simplemde.com/**](https://simplemde.com/)

**Bootstrap datetimepicker** [**https://eonasdan.github.io/bootstrap-datetimepicker/**](https://eonasdan.github.io/bootstrap-datetimepicker/)

**Jasny Bootstrap** [**http://www.jasny.net/bootstrap/**](http://www.jasny.net/bootstrap/)

**6.4 Editando una publicación**

**6.5 Eliminando una publicación**

SoftDeletes

¡

**6.6 Trabajando con las categorías**

En fichero CMS ponemos por defecto la categoría “sin\_categoria” con id 1¡

**6.7 Usuarios**

**7. Lista de control de acceso**

**7.1 Control de acceso y permisos de usuario.**

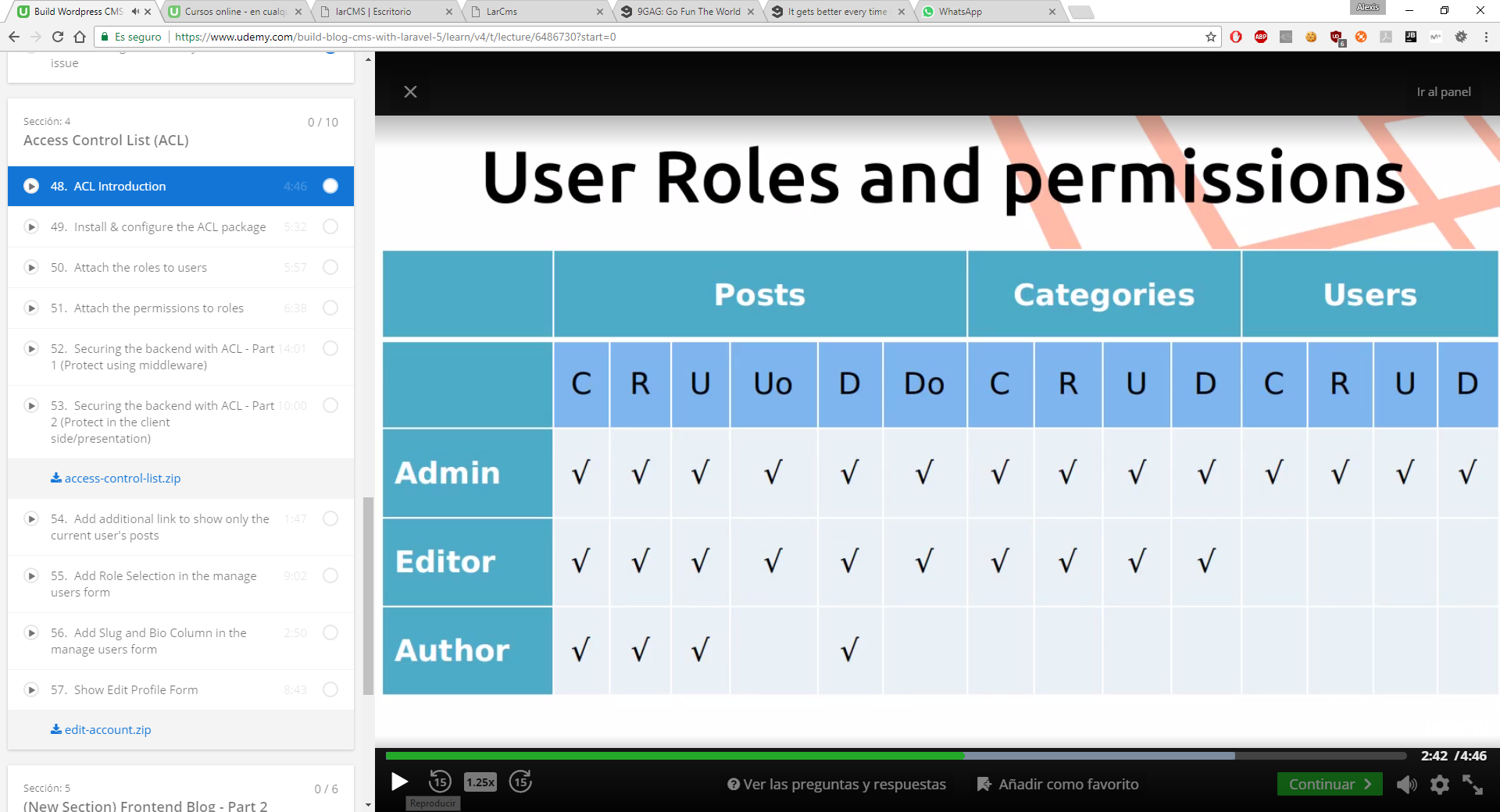
Usaremos una lista de control de acceso por roles (RBAC) (ACL)para saber que roles pertenecerá cada usuario.

Roles:

Admin: Ver todos los posts, Añadir nuevo post, Editar y borrar cualquier post. Gestión de categorías y usuarios

Moderador: Gestion de posts, gestión de categorias

Autor: Acceso a sus publicaciones.



|  |  |  |
| --- | --- | --- |
| Admin | Moderador | Autor |
| Create-post  Read-post  Update-post  Update-others-post  Delete-post  Delete-others-post  Create-category  Read-category  Update-category  Delete-category  Créate-user  Read-user  Update-user  Delete-user | créate-post  read-post  update-post  update-others-post  delete-post  delete-others-post  créate-category  read-category  update-category  delete-category | Create-post  Read-post  Update-post  Delete-post |

Paquetes de laravel para ACL, uso Laratrust <http://laratrust.readthedocs.io/en/4.0/>

Funciona con CRUD (Create, Read, Update, Delete) y por cada rol le assignaremos esos valores.

Php artisan laratrust:migration

Php artisan migrate

<http://laratrust.readthedocs.io/en/4.0/configuration/models/role.html>

Creamos modelo php artisan make:model Role / php artisan make:model Permission

**<?**php

**namespace** App;

**use** Laratrust\LaratrustRole;

**class** **Role** **extends** LaratrustRole

{

}

Permiso

**<?**php

**namespace** App;

**use** Laratrust\LaratrustPermission;

**class** **Permission** **extends** LaratrustPermission

{

}

Modificando el modelo de usuario

**<?**php

**use** Laratrust\Traits\LaratrustUserTrait;

**class** **User** **extends** Model

{

**use** LaratrustUserTrait; *// add this trait to your user model*

**...**

}

Creando Roles y permisos con seed.

Acceso por código (@role([‘admin’] ) Filtrar Codigo

Acceso por middleware: Filtrar URL

https://laravel.com/docs/5.4/middleware

En el constructor del backend

public function \_\_construct()  
{  
 $this->middleware('auth');  
 $this->middleware('check-permissions');  
}

**8. Bibliografia y enlaces web**

Comparativa: <https://medium.com/@realvjy/why-laravel-is-best-php-framework-98a2784d76dc> [1]

WampServer: <http://www.wampserver.com/en/> [2]

Composer: <https://getcomposer.org/> [3]

Laravel: [https://laravel.com/docs/5.4 [4](https://laravel.com/docs/5.4%20%5b4)]

Carbon [http://carbon.nesbot.com/docs/[5](http://carbon.nesbot.com/docs/%5b5)]

Gravatar <https://es.gravatar.com/> [6]

Markdown <https://github.com/GrahamCampbell/Laravel-Markdown> [7]

Tutorial: <https://www.udemy.com/build-blog-cms-with-laravel-5/>