Contents

1 Blog con Laravel 9 sail

El código del proyecto se puede encontrar aquí.

1.1 Crear proyecto

```
curl -s https://laravel.build/blog-laravel9 | bash
cd curso-basico-laravel9
sail up
```

1.2 Modelo de datos

1.2.1 Modelos y sus migraciones

1. Creamos el modelo *Category* y su migración añadiendo `-m`

   ```
   sail artisan make:model Category -m
   ```

   El modelo tendrá una columna `id()` de tipo `BigInteger Unsigned Primary Key`, y una columna para el nombre de tipo `Varchar de 40`:

   ```
   <?php
   ...
   public function up()
   ```

```
{
    Schema::create('categories', function (Blueprint $table)
    {
        $table->id();
        $table->string("name", 40)->unique();
    });
}
...
```

Observa que en Laravel 9 (realmente desde la 7) la función `id()`, hace todo lo necesario sin necesidad de especificar el tipo o primary key...

Varchar viene expresado como string.

2. Ahora el modelo *Article*

```
sail artisan make:model Article -m
```

Un artículo tendrá una categoría (*Category*) y un usuario creador (*user*), por lo que enlazaremos con las tablas correspondientes con una «foreign key». El orden de creación de las migraciones es relevante. Además tendrá un título y el contenido. En este caso tendrá dos campos para indicar el momento de la creación y cuando ha sido modificado.

```
<?php
...
public function up()
{
    Schema::create('articles', function (Blueprint $table)
    {
        $table->id();
        $table->foreignIdFor(\App\Models\User::class)->constrained();
        $table->foreignIdFor(\App\Models\Category::class)->constrained();
        $table->string("title", 80)->unique();
        $table->text("content");
        $table->timestamps();
    });
}
...
```

1.2.2 Modelo Category

Vamos a tocar el modelo creado automáticamente para ajustarlo.

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\HasMany;

class Category extends Model
{
    use HasFactory;
```

```
    // por defecto Laravel crea timpestamps y esta variable es true
    // con false le indicamos al ORM que no existen
    public $timestamps = false;

    // evitar ataques de asignación masiva
    protected $fillable = [
        "name",
    ];

   //Definimos la relación entre Category y Articles
   //Una categoría puede estar en muchos articulos
   //En principio un artículo sólo pertenece a una categoría,
   //así que es una relación de uno a muchos
   //Accediendo a la función articles, des un objeto de tipo Category,
   //podremos saber todos los artículos que tienen esa categoría.
    public function articles(): HasMany
    {
        return $this->hasMany(Article::class);
    }
}
```

1. Modelo Article Aquí hay más cosas que cambiar:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Relations\BelongsTo;
use Illuminate\Support\Str;

class Article extends Model
{
    use HasFactory;
    //datos que permitimos rellenar
    protected $fillable = [
        "user_id", "category_id", "title", "content",
    ];
    //para la paginación, cuantos mostraremos por página
    //método paginate
    protected $perPage = 5;

    // método de Laravel que se ejecuta cuando se instancia un modelo
    protected static function boot()
    {
        parent::boot();
        //callback que recupera el id del autor y lo
        // relaciona con el user_id=> no es un campo rellenable
        // se rellena automáticamente con el id del usuario identificado
        self::creating(function (Article $article) {
            $article->user_id = auth()->id();
        });
    }

    //relación 1 a muchos, para saber a qué usaurio pertenece el artículo
    //
    public function user(): BelongsTo
    {
        return $this->belongsTo(User::class);
    }
    //relación 1 a muchos, a qué categoría pertence el artículo
    public function category(): BelongsTo
    {
        return $this->belongsTo(Category::class);
    }

    //poner la hora en formato legible para nosotros
    //carbon librería para trabajar con fechas
```

```
    public function getCreatedAtFormattedAttribute(): string
    {
        return \Carbon\Carbon::parse($this->created_at)->format('d-m-Y H:i');
    }
    //accesor para obtener un extracto del contenido del artículo
    public function getExcerptAttribute(): string
    {
        return Str::excerpt($this->content);
    }
}
```

Nota: excerpt Por si a alguien le pasa, o no. La función `excerpt` ha funcionado bien con los artículos creados desde los `seeders` y `factories`; sin embargo, con artículos creados desde el formulario falla, no sé la razón. La solución ha sido cambiar:

```
<?php
Str::excerpt(value: $this->content);
```

por

```
<?php
Str::words(value: $this->content, words: 90);
```

Donde 90, puede ser 100 o la cantidad de palabras que queráis que se muestren.

2. Modelo User Vamos a reutilizar el modelo que viene predefinido en Laravel, vamos a dejarlo casi tal cual viene, pero vamos a añadir la relación con los artículos:

```
<?php
//Un usuario va a tener muchos (hasMany) articles
public function articles(): HasMany
{
    return $this->hasMany(Article::class);
}
```

1.3 Seeders y Factories

Los «Seeders» (sembradores), junto con las Factories (factorías), son un medio para introducir datos de prueba, falsos, en la BD y poder probar la aplicación.

1. Seeder de Category Vamos a crear un *seeder* para Categoy:

```
sail artisan make:seed CategorySeeder
```

Abrimos el fichero (en database/seeders) y lo modificamos:

```
<?php

namespace Database\Seeders;

use App\Models\Category;
use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;

class CategorySeeder extends Seeder
{
```

```
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        // nuestros añadidos
        //Utilizamos nuestro modelo Category, y de este modelo
        //podemos usar varias funciones, insert para meter muchos datos
        //o create para un único dato. Vamos a utilizar insert, puesto
        // que vamos a introducir varios. Creamos un array por cada dato.
         Category::insert([
            ["name" => "Php",],
            ["name" => "Laravel",],
            ["name" => "Vue",],
            ["name" => "Docker",],
         ]);
    }
}
```

Cuando ejecutamos los *seeders*, el único que se ejecuta es el `DatabaseSeeder`, para ejecutar los demás tendremos que registrarlos, precisamente en `DatabaseSeeder`. Veremos dentro de un momento que hacer para registrarlos.

Antes vamos ver cómo funcionan las factorías con una factoría para los artículos:

2. Factory para Article. Creamos la factoría como siempre, con artisan:

```
sail artisan make:factory ArticleFactory
```

Abrimos `ArticleFactory` y lo modificamos para que quede así:

```
<?php
namespace Database\Factories;

use App\Models\Category;
use App\Models\User;
use Illuminate\Database\Eloquent\Factories\Factory;

/**
 * @extends \Illuminate\Database\Eloquent\Factories\Factory<\App\Models\Article>
 */
class ArticleFactory extends Factory
{
    /**
     * Define the model's default state.
     *
     * @return array<string, mixed>
     */
    public function definition(): array
    {
```

```
        //introducimos los campos que queremos con su tip
        //los "faker" ya vienen dentro de los facotry
        //no hay que instanciarlos
        //
        return [
            //texto aleatorio de 30 caracteres
            "title" => $this->faker->text(30),
            //texto aleatorio para el "content"
            "content" => $this->faker->text,
            //obtenemos todos los usuarios que tenemos y asignamos
            //uno aleatoriamente
            "user_id" => User::all()->random(1)->first()->id,
            //idem con las categorías
            "category_id" => Category::all()->random(1)->first()->id,
            //hora de creación ahora con la función, de Carbon, now()
            "created_at" => now(),
        ];
    }
}
```

3. **Actualizar DatabaseSeeder** Para utilizar todo lo creado, debemos ir a DatabaseSeeder y registrar el seeder y la factory creados:

```
<?php

namespace Database\Seeders;

use App\Models\Article;
use App\Models\User;
use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
        //creamos un usuario con la User::factory
        //le pasamos datos, así que no usará lo del factory
        User::factory()->create([
            "name" => "CursosDesarrolloWeb",
            "email" => "laravel9@blogweb.es",
        ]);
        User::factory()->create([
            "name" => "Soporte",
            "email" => "soporte@blogweb.es",
```

```
        ]);
        //Para llamar al seeder sólo tenemos que llamarlo
        $this->call(CategorySeeder::class);
        //Creamos 20 artículos
        Article::factory(20)->create();
    }
}
```

Nota:

> No hemos mencionado la factoría de usuarios `UserFactory`, porque se crea automáticamente al crear el proyecto Laravel, y no hemos necesitado modificarla, sólo usarla. La puedes encontrar en `database/factories/UserFactory.php`

4. Ejecutar seeders Es el turno de ejecutar los seeders

```
sail artisan db:seed
```

Algunas veces pueden repetirse filas, así que habrá que volver a lanzar el seeder.

Además debemos modificar una función del modelo `Article` para poder lanzar los seeders:

```
<?php
protected static function boot()
{
        parent::boot();
        //callback que recupera el id del autor y lo
        // relaciona con el user_id=> no es un campo rellenable
        // se rellena automáticamente con el id del usuario identificado
        //Sólo se ejecutará si no estamos lanzando una operación desde consola,
        //porque no tenemos el usuario identificado
        if(!app()->runningInConsole())
        {
            self::creating(function (Article $article)
            {
                $article->user_id = auth()->id();
            });
        }
}
```

Si ya habíamos lanzado los seeder, aunque lo arreglemos fallará, porque se habrá ejecutado a medias, y ya habrá datos en BD. Así que habrá que hacer algo:

```
sail artisan migrate:fresh --seed
```

Con esto vaciamos las tablas, las eliminamos, recreamos y las rellenamos con los seeders.

1.4 Autenticación en Laravel

El siguiente paso es realizar el sistema de autenticación de nuestro blog. Para ello usaremos Breeze, una librería de Laravel:

1. **Laravel Breeze** Laravel Breeze es una implementación simple y mínima de todas las funciones de autenticación de Laravel, incluido el inicio de sesión, el registro, el restablecimiento de contraseña, la verificación de correo electrónico y la confirmación de contraseña. La capa de vista de Laravel Breeze se compone de plantillas Blade simples diseñadas con Tailwind CSS. Para comenzar, consulte la documentación sobre los kits de inicio de aplicaciones de Laravel.

Otras alternativas son:

2. **Laravel Fortify** es un backend de autenticación sin cabeza para Laravel que implementa muchas de las funciones que se encuentran en esta documentación, incluida la autenticación basada en cookies y otras funciones como la autenticación de dos factores y la verificación de correo electrónico. Fortify proporciona el backend de autenticación para Laravel Jetstream o se puede usar de forma independiente en combinación con Laravel Sanctum para proporcionar autenticación para un SPA que necesita autenticarse con Laravel.

3. **Laravel Jetstream** es un sólido kit de inicio de aplicaciones que consume y expone los servicios de autenticación de Laravel Fortify con una hermosa y moderna interfaz de usuario impulsada por Tailwind CSS, Livewire o Inertia. Laravel Jetstream incluye soporte opcional para autenticación de dos factores, soporte de equipo, administración de sesiones de navegador, administración de perfiles e integración incorporada con Laravel Sanctum para ofrecer autenticación de token API. Las ofertas de autenticación de API de Laravel se analizan a continuación.

   Aunque Laravel Breeze no nos ofrece tantas funcionalidades como Jetstream: 2FA, Inertia o Livewire, sí nos ofrece lo básico para cualquier proyecto, un completo proceso de autenticación, registro, login, confirmación de correo electrónico y recuperación de contraseña, y todo esto publicando todos los recursos en nuestro proyecto, tanto vistas con blade como controladores y requests. Instalar Laravel Breeze Para empezar a utilizar Laravel Breeze en tu proyecto Laravel simplemente sigue estos pasos (forma recomendada de instalar Breeze):

```
#Instala la dependencia Breeze
sail composer require laravel/breeze --dev
#Se monta el "andamio", en la aplicación, para usar
#autenticación con Breeze
sail artisan breeze:install
```

   Vamos a trabajar con Laravel Blade para las vistas, es la forma más sencilla. No es la más potente ni la única, Laravel se puede integrar fácilmente con Vue y con React, y con un poco más de trabajo con Mithril.js...

   Podemos comprobar que está instalado mirando que en `Controllers` tenemos el nuevo directorio `Auth` con todas las partes de la autenticación. Tenemos nuevos directorios y vistas en resources/views, entre otros `auth` (vistas de autenticación), `components` (componentes Blade que podemos utilizar en nuestra aplicación), `layouts` (diseño para usuarios autenticados, para invitados, navegación con acceso al dashboard -lo que se muestra al entrar en la página-).

   Además ejecutando el siguiente comando y viendo que se han añadido nuevas rutas:

```
sail artisan route:list
```

   Luego vamos a ejecutar `yarn` (alternativa a `npm`) para bajar las dependencias del lado del cliente y hacer seguimiento de los cambios que hagamos en el lado del cliente (para que se actualicen las vistas... sin necesidad de relanzar la aplicación).

   Laravel con webpack/laravel-mix versiones < 9.19.0:

```
sail yarn && sail yarn watch
```

   o bien

```
sail npm install && sail npm run watch
```

   Laravel con vite, versiones >= a la 9.19.0:

```
sail yarn  && sail yarn dev
```

   o bien

```
sail npm install && sail npm run dev
```

Webpack de laravel-mix y vite son herramientas para construir la parte «front-end» de una aplicación web, siendo vite más moderno y la herramienta de construcción que viene con la versión de Laravel que estamos usando.

Si alguien quiere, o necesita, volver a Laravel Mix, aquí hay una guía para hacerlo.

Una vez llegados a este punto ya podemos entrar en `localhost` y ver que nuestro sistema de "Login" está en marcha.

Versiones anteriores de Laravel usaban Bootstrap como framework de CSS, pero esta versión que estamos usando utiliza Tailwind.

1.5 Controladores, recursos y rutas. Tests parte 1.

1.5.1 Controlador de artículo: «ArticleController»

Vamos a empezar con el blog, para ello creamos nuestro primer controlador. El controlador es el puente entre la vista y el modelo y se ejecuta a través de las rutas definidas en el sistema de rutas.

El controlador va a ser del tipo *resource* (recurso), lo indicamos con el *flag* `-r`. Al ser del tipo recurso nos va a crear automáticamente los métodos:

- `index`: listar los artículos
- `create`: mostrar el formulario de creación
- `store`: guardar un artículo en BD
- `show`: mostrar un artículo en detalle
- `edit`: mostrar el formulario de edición de artículos
- `update`: actualizar el artículo que hayamos estado editando
- `destroy`: eliminar un recurso(artículo) de la base de datos

Básicamente todo lo que necesitamos para un CRUD.

Le vamos a pasar otra opción `--model=Article` para indicarle que el que vamos a gestionar objetos de tip `Article`. autoEjecutamos:

```
sail artisan make:controller ArticleController -r --model=Article
```

Si vamos al directorio de controladores vemos nuestro ArticleController.

1. **Creación de los tests** Más adelante veremos otra forma de hacer tests con el plugin Pest, pero por ahora vamos a verlos tal y como los incorpora Laravel. En los controladores, la idea es crear un test por cada función del controlador.

   ```
   sail artisan make:test Http/Controllers/ArticleController/IndexTest
        [--unit][--pest]
   ```

   Por defecto para Laravel todo son «feature tests» (carpeta `tests/Features`), si queremos crear test unitarios, en su correspondiente carpeta `/tests/Unit` hay que usar la opción `--unit`. La opción `--pest`, es para usar el framework, de tests, Pest, que veremos más adelante.

   Mirad el detalle de, que el nombre acaba en `Test`, para ayudar a Laravel a descubir los tests.

   En general crearemos tests de características (feature tests), puesto que probamos funcionalidades completas y no funciones individuales desconectadas del resto.

Podemos hacer un fichero de test para cada función del controlador, o hacer un fichero único con todos los tests del controlador, como en el ejemplo.

Una vez terminados los tes podemos hacer (cualquiera de ellas):

```
sail test
sail test --group orders
sail artisan test
```

Más adelante veremos otras alternativas que facilitan el testing.

1.5.2 Actualizar rutas

Si vamos a nuestro archivo de rutas ahora vemos que hay nuevas rutas que antes no teníamos. Entre otras se hace inclusión del fichero `routes/auth.php`, en el que están todas las rutas necesarias para todas las tareas de autenticación.

Pues bien, vamos a añadir una ruta para el controlador de artículos:

```php
<?php

Route::resource("articles", \App\Http\Controllers\ArticleController::class)
    ->middleware("auth");
```

Con esto estamos añadiendo todas las rutas para el CRUD de Article. En versiones anteriores habría que poner las rutas para cada una de las operaciones del CRUD. Además, por usar el middleware auth, no es posible ir a esta página sin estar autenticado.

1.5.3 Listado y paginación de artículos

Nos vamos a `ArticleController`, al principio, a la función `index()` que es la que servirá para hacer el listado de artículos. La opción por defecto de `index()` es devolver un objeto de tipo `Response`, pero nosotros vamos a devolver un `Renderable` de `Illuminate`. Con esto queremos decir que vamos a devolver una vista. Quedará así:

```php
<?php

/**
 * Display a listing of the resource.
 *
 * @return Renderable
 */
public function index(): Renderable
{
    $articles = Article::with("category")->latest()->paginate();
    //dd($articles)
    return view("articles.index", compact("articles"));
}
```

Con esto vamos a retornar todos los artículos, incluida su categoría. Con el método `with` le decimos que queremos cargar una relación, en este caso "category". Podríamos indicar las columnas que queremos así: `"category:id,name`, pero como sólo tiene dos columnas no ponemos nada y las recuperamos todas. También vamos a decir que queremos ordenar por la fecha de lata. Para ello usamos el método `latest()` para obtener desde el final. Finalmente llamamos a la función `paginate()`, que nos devuelve todos los resultados, pero paginados. Si queremos saber que está pasando, podemos usar la función, antes del return,

`dd($articles)`, para ver la información que pasa por ahí. Recordad, que la paginación muestra 5 artículos, porque así lo configuramos en el modelo.

Al final hacemos un `return view...`. Tenemos que crear la vista, para ello creamos el fichero `index.blade.php` en `resources/views/articles` (a veces PHPStorm te ayuda y te propone crear esa vista, supongo que con el plugin de pago, o el gratis que no está disponible para la última versión de PHPStorm; lo que sí se puede hacer «seguro», es crear los tests, pulsamos `Alt+Insert` y en el menú, una de las opciones, es Test...).

Vamos a modificar también el fichero `navigation.blade.php"`, vamos a `<!-- Navigation Links -->`, y duplicamos el `x-nav-link` modificando las rutas adecuadamente:

```
<!-- Navigation Links -->
<div class="hidden space-x-8 sm:-my-px sm:ml-10 sm:flex">
    <x-nav-link :href="route('dashboard')"
                :active="request()->routeIs('dashboard')">
        {{ __('Dashboard') }}
    </x-nav-link>
  <div class="hidden space-x-8 sm:-my-px sm:ml-10 sm:flex">
    <x-nav-link :href="route('articles.index')"
                :active="request()->routeIs('articles.*')">
      {{ __('Artículos') }}
    </x-nav-link>
</div>
```

Cuando creamos un controlador de tipo resource, como hicimos antes, se crean una serie de rutas para todas las funciones de ese resource, así ya tenemos creadas las rutas, `article.index, article.destroy`...

Con lo anterior hacer que, en la barra de navegación, aparezca el enlace a la ruta `article.index`, pero sólo si estamos en alguna ruta de "articles" `:active="request()->routeIs('articles.*')` `{{ __('Artículos') }}`.

Ahora vamos al Dashboard `dashboard.blade.php`. Observemos la etiqueta `<x-app-layout>`, nos está indicando que usa el layout *app*, en el fichero `app.blade.php`. Mirando en ese fichero vemos que es la disposición de base/layout (plantilla base) de nuestra aplicación. En él vemos que tenemos las partes que muestran la navegación, la cabecera, pie... Copiamos el contenido de `app.blade.php`, lo pegamos y lo iremos modificando. Podéis probar a quitar y poner cosas, y recargar.

Vamos a dar formato a nuestro listado de artículos, para eso vamos a usar Tailwind, que ya viene en Laravel. Vamos a tailblocks, donde veremos bloques ya construidos con los que trabajar. Dentro de la cuarta opción, le damos a ver código y copiamos el siguiente trozo (y cerramos las etiqueta que queden abiertas):

```
<section class="text-gray-600 body-font overflow-hidden">
  <div class="container px-5 py-24 mx-auto">
    <div class="-my-8 divide-y-2 divide-gray-100">
      <div class="py-8 flex flex-wrap md:flex-nowrap">
        <div class="md:w-64 md:mb-0 mb-6 flex-shrink-0 flex flex-col">
          <span class="font-semibold title-font text-gray-700">
            CATEGORY
          </span>
          <span class="mt-1 text-gray-500 text-sm">12 Jun 2019</span>
        </div>
        <div class="md:flex-grow">
          <h2 class="text-2xl font-medium text-gray-900 title-font mb-2">
```

```
            Bitters hashtag waistcoat fashion axe chia unicorn
          </h2>
          <p class="leading-relaxed">
            Glossier echo park pug, church-key sartorial biodiesel
            vexillologist pop-up snackwave ramps cornhole.
            Marfa 3 wolf moon party messenger bag selfies,
            poke vaporware kombucha lumbersexual pork belly
            polaroid hoodie portland craft beer.
          </p>
          <a class="text-indigo-500 inline-flex items-center mt-4">Learn More
            <svg class="w-4 h-4 ml-2" viewBox="0 0 24 24"
                 stroke="currentColor" stroke-width="2"
                 fill="none" stroke-linecap="round"
                 stroke-linejoin="round">
              <path d="M5 12h14"></path>
              <path d="M12 5l7 7-7 7"></path>
            </svg>
          </a>
        </div>
      </div>
    </div>
  </div>
</section>
```

Vemos que el contenido realmente está en la etiqueta `<div class="-my-8 divide-y-2 divide-gray-100">`, vamos a poner directivas Blade para repetir una acción, listar, para cada artículo...:

```
<section class="text-gray-600 body-font overflow-hidden">
  <div class="container px-5 py-24 mx-auto">
    <div class="-my-8 divide-y-2 divide-gray-100">
       @foreach($articles as $article)
       <div class="py-8 flex flex-wrap md:flex-nowrap">
        ...
      </div>
      @endforeach
    </div>
  </div>
</section>
```

Vamos a modificar para mostrar lo que queremos.

- Cambiamos CATEGORY por la categoría del artículo, que viene en la variable de cada objeto de tipo Article.

```
<span class="font-semibold title-font text-gray-700">
  {{ $article->category->name }}</span>
```

- Cambiamos la fecha puesta a dedo por la fecha del artículo real, con el accesor que creamos al principio:

- Cambiamos el título:

```
<h2 class="text-2xl font-medium text-gray-900 title-font mb-2">
  {{ $article->title }}
</h2>
```

- Cambiamos el contenido por el extracto:

```
<p class="leading-relaxed">
  {{ $article->excerpt }}
</p>
```

Ya nos queda añadir los enlaces de paginación, que es tan fácil como, tras el `@endforeach`, añadir:

```
@endforeach
{{ $article->links() }}
```

y Laravel ya hace el resto por nosotros.

1.5.4 Creación de artículos

Antes de ir a hacer el formulario de creación de artículos y la función del controlador que lo llama, vamos a crear un botón en la vista *index* para dar acceso a ese formulario de creación. Es simplemente un enlace con la función de ayuda (o helper) `route()` a la función create del controlador de artículos:

```
<div class="mb-16 -my-8">
   <a href="{{ route("articles.create") }}"
      class="flex w-64 text-white bg-indigo-500
             border-0 py-2 px-8 focus:outline-none
             hover:bg-indigo-600 rounded text-lg">
      {{ __("Crear un nuevo artículo") }}
   </a>
</div>
```

Ahora mismo la función del controlador está vacía y no nos devuelve nada.

Creamos la función *create*:

```
<?php
 /**
 * Show the form for creating a new resource.
 *
 * @return Renderable
 */
public function create(): Renderable
{
  $article = new Article; //
  $title = __("Crear artículo");
  //ruta para el procesado del contenido
  //devuelto por el formulario
  $action = route("articles.store");
  return view("articles.form", compact("article", "title", "action"));
}
```

Dentro de la función hemos definido un nuevo artículo, para representar el formulario, que mostraremos después. Vamos a tener un título y una acción, donde vamos a procesar el formulario, que simplemente será la ruta de la función que recogerá los datos del formulario y los almacenará en BD. Finalmente retornamos una vista, `articles.form`, con todos esos parámetros. `articles.form` indica que accedemos a una vista `form` dentro de la carpeta `articles` (en `view`, claro). En Laravel muchas veces veremos, en el código, la notación con punto para rutas en el árbol de ficheros: `articles.form` equivale a `view/articles/form.blade.php`

El formulario nos servirá tanto para crear nuevos artículos, como para editarlos; economía del esfuerzo^_^.

Hay diversas manera de pasar parámetros a una vista, una de ellas es compact. Es la más sencilla; como inconveniente las variables que pasamos aquí, se tienen que llamar igual en el lugar de recepción.

Para el formulario vamos de nuevo a la página de bloques de TailWind y cogemos el sexto elemento, `formulario de contacto`, copiamos el código, lo pegamos en el formulario y lo revisamos para hacer algunos cambios. Queda así:

```
<x-app-layout>
  <x-slot name="header">
    <h2 class="font-semibold text-xl text-gray-800 leading-tight">
      {{ $title }}
    </h2>
  </x-slot>

  <div class="py-12">
    <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
      @if ($errors->any())
        <div class="bg-red-500 text-white p-4">
          <ul>
            @foreach ($errors->all() as $error)
              <li>{{ $error }}</li>
            @endforeach
          </ul>
        </div>
      @endif
      <form method="POST" action="{{ $action }}">
        @csrf
        @if($article->id)
          @method("PUT")
        @endif
        <div class="bg-white overflow-hidden shadow-sm sm:rounded-lg p-6">
          <h2 class="text-gray-900 text-lg mb-1 font-medium title-font">
              {{ __("Escribe tu artículo") }}
          </h2>
          <div class="relative mb-4">
            <label for="title"
                   class="leading-7 text-sm text-gray-600">
                   {{ __("Título") }}
            </label>
            <input type="text" id="title" name="title"
                   value="{{ old("title", $article->title) }}"
                   class="w-full bg-white rounded border border-gray-300
                          focus:border-indigo-500 focus:ring-2
                          focus:ring-indigo-200 text-base outline-none
                          text-gray-700 py-1 px-3 leading-8
                          transition-colors duration-200 ease-in-out">
          </div>
          <div class="relative mb-4">
            <label for="category_id"
                   class="leading-7 text-sm text-gray-600">
```

```
                        {{ __("Título") }}</label>
            <select id="category_id" name="category_id"
                    class="w-full bg-white rounded border border-gray-300
                          focus:border-indigo-500 focus:ring-2
                          focus:ring-indigo-200 text-base outline-none
                          text-gray-700 py-1 px-3 leading-8
                          transition-colors duration-200 ease-in-out">
              @foreach(\App\Models\Category::get() as $category)
                <option
                  {{ (int) old("category_id",
                   $article->category_id) === $category->id ? 'selected'
                    : '' }} value="{{ $category->id }}">
                    {{ $category->name }}
                  </option>
              @endforeach
            </select>
          </div>
          <div class="relative mb-4">
            <label for="content"
                   class="leading-7 text-sm text-gray-600">
            {{ __("Artículo") }}
            </label>
            <textarea id="content" name="content"
                      class="w-full bg-white rounded border
                      border-gray-300 focus:border-indigo-500
                      focus:ring-2 focus:ring-indigo-200 h-32
                      text-base outline-none text-gray-700 py-1 px-3
                      resize-none leading-6 transition-colors
                      duration-200 ease-in-out">
                    {{ old("content", $article->content) }}
            </textarea>
          </div>
          <button type="submit"
                  class="text-white bg-indigo-500 border-0
                  py-2 px-6 focus:outline-none hover:bg-indigo-600
                  rounded text-lg">
                 {{ $title }}
          </button>
        </div>
      </form>
    </div>
  </div>
</x-app-layout>
```

- `@if($errors)`, nos muestra los errores en el formulario.
- El formulario usa el método **POST** con la acción `{{ $action }}`, que viene desde el controlador. Pero si el `id` existe, estamos modificando, no creando un artículo nuevo y necesitaríamos un **`PUT`** estándar para modificación (también se suele usar `PATCH`), lo hacemos con directivas Blade y así Laravel sabrá que es un `PUT`:

```
@if($article->id)
    @method("PUT")
@endif
```

- `@csrf` Directiva para proteger el formulario contra XSS.Crea un campo `hidden` en el formulario con un token. Cuando hagamos una petición a Laravel se usará ese token para comprobar que la petición se hace desde nuestro sitio y no desde otro (evitar XSS).
- Se ha cambiado el `value` del siguiente código:

```
<input type="text" id="title" name="title"
                value="{{ old("title", $article->title) }}"
```

  Lo que hace es, si hemos mandado el formulario, desde edición, y la validación del formulario ha fallado por lo que sea, se queda con el título válido que tenía antes, si no simplemente va a recoger el valor que habíamos enviado

- Recogemos todas las categorías, podríamos hacerlo con el controlador, o directamente como hemos hecho:

```
@foreach(\App\Models\Category::get() as $category)
  <option  {{ (int) old("category_id",
     $article->category_id) === $category->id ? 'selected'
      : '' }} value="{{ $category->id }}">
      {{ $category->name }}
  </option>
@endforeach
```

  Observad la conversión a entero `(int)`, ¿se puede quitar y usar == en lugar de ===, porque serían dos cadenas? Si el `id` de la categoría del artículo, nuevo o editado, se corresponde con lo que había, se maraca como `selected`.

1.5.5 Validación de formularios de forma segura con `Form Request` y match

Lo primero que vamos a hacer es crear un Form Request con artisan:

```
sail artisan make:request ArticleRequest
```

Los request en Laravel permiten interceptar la petición del formulario para poder realizar las validaciones correctamente

Una vez creado el fichero en `Http/Requests/ArticleRequest` lo modificamos:

```
<?php
namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class ArticleRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize(): bool
```

```
    {
        return true;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules(): array
    {
        return match ($this->method()) {
            "POST" => [
                "title" => "required|min:2|max:40|unique:articles",
                "content" => "required|min:10",
                "category_id" => "required|exists:categories,id",
            ],
            "PUT" => [
                "title" => "required|min:2|max:40|unique:articles,title," .
                           $this->route("article")->id,
                "content" => "required|min:10",
                "category_id" => "required|exists:categories,id",
            ],
        };
    }
}
```

La función `authorize` debe devolver `true` para que se active la siguiente función `rules()`, donde crearemos las reglas de validación del formulario. Las condiciones para que se ejecuten las reglas las controlamos nosotros con rutas, permisos o roles, como hemos hecho nosotros con el `middleware auth` en las rutas.

En lugar de hacer un return directamente, lo habitual, vamos a utilizar la función `match` de PHP 8 para comprobar si recibimos un `POST`, creación, o un `PUT`, modificación.

Dentro de cada uno las reglas correspondientes, por ejemplo para `title`, en creación, decimos que es obligatorio (`required`), longitud mínima 2 (`min:2`), máxima 40 (`max:40`), lo definido para el campo en la tabla), único (`unique`) en la tabla `articles` (hay que validarlo aquí para no recibir un error al intentar insertar en la tabla).

Lo mismo hacemos para `content`, mirad que hemos limitado el tamaño a 1000 para evitar errores (podemos ser más exactos mirando la longitud máxima admitida para el tipo de campo generado para la BD usada).

Para `category_id` tenemos que comprobar que, además de obligatorio, existe ese `id` en la tabla `categories`, pero allí no se llama `category_id`, sino `id`, por tanto debemos especificarlo (`exists:categories,id`), para evitar problemas al consultar la BD.

Cuando estamos editando las reglas van a ser muy parecidas, con algunos añadidos. En el caso del título tenemos que decirle que ese título puede existir en el caso de que el `id` recuperado sea el `id` *actual* con el que trabajamos (para poder dejar el mismo título y que al comprobar si existe un artículo con ese mismo título sepa que si es el mismo `id`, no está repetido). Lo hacemos añadiendo `.$this->route("article")->id`.

Para los demás campos no vamos a tocar nada.

Más sobre validación de formularios aquí.

Para ver todas las reglas que podemos usar podemos mirar en este apartado de la web anterior.

1.5.6 Persistir en BD los datos del formulario

Vamos a dar contenido a la función `store` de nuestro `ArticleController`, que será la función que persistirá en BD, lo que hemos introducido y validado en el formulario:

```php
<?php
/**
 * Store a newly created resource in storage.
 *
 * @param ArticleRequest $request
 * @return RedirectResponse
 */
public function store(ArticleRequest $request): RedirectResponse
{
  $validated = $request->safe()->only(['title', 'content', 'category_id']);
  Article::create($validated);
  //Para trabajar con traducciones la parte de __("...")
  session()->flash("success", __("El artículo ha sido creado correctamente"));
  return redirect(route("articles.index"));
}
```

Por defecto se devuelve un objeto de tipo `Response`, pero nosotros vamos a devolver uno del tipo `RedirectResponse`, una vez almacenado el artículo vamos a ir al listado de artículos para que se muestre que se ha creado.

Con `$request->safe()->only(['title', 'content', 'category_id']);` indicamos que vamos a hacer una petición segura, cadenas escapadas, verificadas... y además sólo permitimos los campos listado en el `only`.

1.5.7 Añadir flash en app.blade.php

Para que se muestre el mensaje, vamos a `app.blade.php`, que es a donde estamos redireccionando, y buscamos la sección `Page Content`, donde vemos la etiqueta `<main>`. Aquí vamos a «recibir» la sesión creada en la función `store` («success»), y poner en la página el mensaje enviado con la sesión:

```php
<?php
<!-- Page Content -->
<main>
  @if (session()->has("success"))
    <div class="bg-green-500 text-white p-4">
      <ul>
        <li>{{ session("success") }}</li>
      </ul>
    </div>
  @endif
      {{ $slot }}
</main>
```

Así obtendremos el mensaje de creado correctamente cuando se cree un artículo nuevo y se haya podido insertar en BD.

Tareilla: En este punto, podemos probar la validación del formulario, enviando el formulario vacío o con

datos fuera del rango permitido. Si cambiamos el **true** por **false** en la función `authorize()` de `ArticleRequest`, veremos que nos da un operación no autorizada. Lo dejamos como estaba y seguimos con la prueba del formulario. Si lo ponemos todo correctamente el `dd()` podemos ver lo que llega desde el formulario.

1.5.8 Editar artículos. Formulario en modo edición.

Vamos a añadir lo necesario para poder editar un artículo. Como ya avanzamos vamos a utilizar el mismo formulario que cuando creamos un artículo.

Primero vamos a `index.blade.php` para añadir un enlace que nos lleve a la edición del artículo. Vemos en nuestro código que hay un enlace que pone `{{ __("Ver detalle") }}`, que en el código original era «Learn more» y lo hemos cambiado. Al final de ese bloque hemos añadido un carácter de pipeline `|`. En este mismo bloque, antes de añadir, la parte de editar, hemos añadido lo siguiente:

```
<?php
href="{{ route("articles.show", ["article" => $article]) }}"
```

Esto nos permite ir a mostrar el artículo al detalle, pero Laravel necesita el `id` del artículo, para ello podemos pasarle en `id` o el artículo completo. En este caso pasamos todo el artículo y Laravel ya sabrá que hacer con él.

Lo siguiente es añadir un bloque similar al del detalle para la edición del artículo:

```
<?php
<a href="{{ route("articles.edit", ["article" => $article]) }}"
  class="text-indigo-500 inline-flex items-center mt-4">{{ __("Editar") }}
 <svg class="w-4 h-4 ml-2" viewBox="0 0 24 24"
       stroke="currentColor" stroke-width="2"
       fill="none" stroke-linecap="round"
    stroke-linejoin="round">
    <path d="M5 12h14"></path>
    <path d="M12 5l7 7-7 7"></path>
  </svg>
</a> |
```

Observad a qué `url` vamos, `articles.edit`. Es la función del controlador que vamos a implementar a continuación. También el carácter de pipeline al final del bloque. Vamos a nuestro fichero `ArticleController.php` y editamos el método `edit`

```
<?php
/**
 * Show the form for editing the specified resource.
 *
 * @param Article $article
 * @return Renderable
 */
public function edit(Article $article): Renderable
{
    //dd($article);
    $title = __("Actualizar artículo");
    //***al pasar el $article, al ir al formulario ya estará todo relleno***
    $action = route("articles.update", ["article" => $article]);
    return view("articles.form", compact("article", "title", "action"));
}
```

Observad que es muy parecido a lo que hicimos en la creación de artículos. Devolvemos un `Renderable`, tenemos título, una acción (llama al método update, situado debajo en la clase), crea la acción que es una llamada a la ruta `"articles.update"` y devuelve la vista `"articles.form"`, con los datos del artículo, el título y la acción. El orden no importa. Como bonus mágico adicional, ya puesto como comentario en el código, pasar a la ruta el artículo hace que, al editar, el formulario esté relleno con los datos que tenía el artículo.

1.5.9 Procesar edición de artículos. Actualizar registro en la BD.

Este trozo de código que ya hemos mencionado anteriormente

```
@if($article->id)
    @method("PUT")
@endif
```

es el que permite que el método `update` sepa el tipo de petición, POST o PUT, y entremos en modo edición y no creación de nuevo artículo.

E Vamos a nuestro ArticleController y vamos a modificar el método `update()`, que es donde se va a procesar el formulario cuando estemos en modo edición. Cambiamos el tipo de respuesta a `RedirectResponse`, y añadimos `$article->update($validated);` y ya está. Queda crear una sesión con un mensaje que pasar al `index.blade.php`, y hacer la redirección. Y eso, es todo amigos.

```
<?php
/**
 * Update the specified resource in storage.
 *
 * @param  ArticleRequest $request
 * @param Article $article
 * @return RedirectResponse
 */
public function update(ArticleRequest $request, Article $article)
{
    $validated = $request->safe()->only(['title', 'content', 'category_id']);
    $article->update($validated);
    //es una línea, pero para que salga bien...
    session()->flash("success", __("El artículo ha sido" .
                                   "actualizado correctamente"));
    return redirect(route("articles.index"));
}
```

1.5.10 Mostrar el detalle de un artículo, cargando sus relaciones.

Para esto vamos a usar el método `show()`; vamos al ArticleController y cambiamos el tipo de dato devuelto a `Renderable`. Ya tenemos el artículo en sí, porque está llegando desde la ruta y ya sólo nos queda cargar cierta información para este artículo, la que proviene de las relaciones. Para ello vamos a usad `load`. A esta función le pasamos qué queremos cargar, en este caso usuario con su id y nombre y la categoría con id y nombre, también ("user:id,name", "category:id,name").

Si no usamos load para cargar las relaciones, y usamos `dd($article)`, podemos ver que los datos provenientes de las relaciones están vacíos.

Esta es la forma de cargar información dinámicamente con Eloquent, el ORM de Laravel, ya que con una sola consulta se carga todo. Si no lo hiciéramos así, y recorriéramos, por ejemplo, una lista de etiquetas, tendríamos

un problema de rendimiento, puesto que haríamos muchas más consultas de las necesarias. Podemos usar "Laravel Debugbar" para verlo, así como para depurar el código.

Lo que nos queda es devolver una vista `articles.show`, que no existe y vamos a crear, pasándole el artículo modificado..

```php
<?php
/**
 * Display the specified resource.
 *
 * @param Article $article
 * @return Renderable
 */
public function show(Article $article): Renderable
{
    $article->load("user:id,name", "category:id,name");
    //dd($article);
    return view("articles.show", compact("article"));
}
```

1. Vista `articles.show` Esta es la vista que creamos para el detalle de los artículos. No hay mucho que destacar. Tenemos una cabecera, luego vamos metiendo, según nos interese, los distintos campos de un artículo, contenido, usuario..., así como un enlace para volver al index. Con esto ya está la vista de detalle.

```blade
<x-app-layout>
  <x-slot name="header">
    <h2 class="font-semibold text-xl text-gray-800 leading-tight">
      {{ __('Detalle artículo') }}
    </h2>
  </x-slot>

  <div class="container px-5 py-24 mx-auto flex flex-col">
    <div class="lg:w-4/6 mx-auto">
      <div class="rounded-lg overflow-hidden">
        <h1 class="text-3xl">{{ $article->title }}</h1>
      </div>
      <div class="flex flex-col sm:flex-row mt-10">
        <div class="sm:w-1/3 text-center sm:pr-8 sm:py-8">
          <div class="w-20 h-20 rounded-full inline-flex
            items-center justify-center bg-gray-200 text-gray-400">
            <svg fill="none" stroke="currentColor"
                 stroke-linecap="round" stroke-linejoin="round"
                 stroke-width="2" class="w-10 h-10" viewBox="0 0 24 24">
              <path d="M20 21v-2a4 4 0 00-4-4H8a4 4 0 00-4 4v2"></path>
              <circle cx="12" cy="7" r="4"></circle>
            </svg>
          </div>
          <div class="flex flex-col items-center text-center justify-center">
            <h2 class="font-medium title-font mt-4 text-gray-900 text-lg">
            {{ $article->user->name }}</h2>
            <div class="w-12 h-1 bg-indigo-500 rounded mt-2 mb-4"></div>
```

```
            </div>
          </div>
          <div class="sm:w-2/3 sm:pl-8 sm:py-8 sm:border-l border-gray-200
               sm:border-t-0 border-t mt-4 pt-4 sm:mt-0 text-center sm:text-left">
            <span class="font-semibold title-font text-gray-400 underline">
            {{ $article->category->name }}</span>
            <p class="leading-relaxed text-lg mb-4">{{ $article->content }}</p>
            <a href="{{ route("articles.index") }}"
              class="text-indigo-500 inline-flex items-center">{{ __("Volver") }}
              <svg fill="none" stroke="currentColor" stroke-linecap="round"
                   stroke-linejoin="round" stroke-width="2"
                   class="w-4 h-4 ml-2" viewBox="0 0 24 24">
                <path d="M5 12h14M12 5l7 7-7 7"></path>
              </svg>
            </a>
          </div>
        </div>
      </div>
</x-app-layout>
```

1.5.11 Eliminar registros de la base de datos de forma correcta

Vamos a ver cómo eliminar artículos desde el listado de artículos.

Primero nos vamos a `index.blade.php` y añadir una opción para poder hacerlo. Va a ser un formulario que añadimos después de la opción de `Editar`, tras el carácter de pipeline:

```
</a> | <--
<form class="inline" method="POST"
      action="{{ route("articles.destroy", ["article" => $article]) }}">
 @csrf
 @method("DELETE")
 <button type="submit"
         class="text-red-500 inline-flex items-center mt-4">{{ __("Eliminar") }}
  <svg class="w-4 h-4 ml-2" viewBox="0 0 24 24" stroke="currentColor" stroke-width="2"
       fill="none" stroke-linecap="round" stroke-linejoin="round">
    <path d="M5 12h14"></path>
    <path d="M12 5l7 7-7 7"></path>
  </svg>
 </button>
</form>
```

Observad que el borrado lo hacemos con un formulario, lo ponemos `inline` con el resto de elementos, y es de tipo `POST`, pero lo vamos a mandar como `DELETE`, sin comprobar nada porque sólo va a tener la función de borrar. Lo de poner un formulario y no como un enlace, es para poder protegerlo con `@csrf` y evitar que cualquiera desde otro sitio pueda acceder a nuestros recursos y borrarlos, con una simple llamada `GET`, sin `csrf` ni *token*, a ese enlace.

La acción del formulario, va a ser llamar directamente a la ruta `articles.destroy` con el artículo actual. Se podría pasar sólo el `id`, pero no es necesario, Laravel sabe qué hacer.

También definimos un botón, de tipo submit, con el texto *Eliminar* para realizar la acción.

Si borramos `@method('DELETE')` no funcionaría, probadlo.

Mejora, pedir confirmación del borrado. Se podría lanzar un alert, u otro elemento para pedir la confirmación. Si se confirma el borrado ya se llamaría a `destroy`; si no se confirma, se vuelve al listado.

Ahora vamos a crear la función llamada desde el enlace de eliminar. En nuestro `ArticleController` vamos a dar contenido a la función `destroy`, quedará así:

```
<?php
/**
 * Remove the specified resource from storage.
 *
 * @param Article $article
 * @return RedirectResponse
 */
public function destroy(Article $article)
{
    $article->delete();
    session()->flash("success", __("El artículo ha sido eliminado correctamente"));
    return redirect(route("articles.index"));
}
```

La primera línea es todo lo que necesitamos. La siguientes es para mostrar un mensaje, como siempre, y finalmente volvemos a la lista de artículos.

CRUD terminado.

1.6 Por hacer del CRUD

Quedan algunas cosas por hacer como subir un fichero, imagen... y gestionar roles. Lo primero lo vamos a ver, lo segundo...

1.7 Test funcionales con Pest Framework

Podríamos hacer todos los test de las diferentes funciones del controlador tal y como hicimos unas páginas atrás y luego, con los test rellenos, usar `sail test`, para ejecutar los test.

Vamos a usar una framework que se llama Pest. Trabaja sin clases, lo que eliminar mucho código y además si estás a costumbrado a Jest en javascript, te resultará familiar la forma de trabajar. El elemento para estos test son las funciones `it()` encadenadas, tantas como queramos, con tantos test como creamos necesario.

Primero hay que instalar Pest, si no lo tenemos instalado.

```
sail composer require pestphp/pest --dev --with-all-dependencies
```

Luego el plugin para Laravel

```
sail composer require pestphp/pest-plugin-laravel --dev
```

Necesitamos que se cree `Pest.php` para artisan (configuración), para ello ejecutamos:

```
sail artisan pest:install
```

También disponemos del comando `pest:dataset` para generar datos para las pruebas conforme se van realizando los test (usaremos factorías en su lugar). Y para crear test usaremos `pest:test` , aunque también se podría crear con el comando artisan general para los tests y añadir al final `--pest`. Si queremos test unitarios habría que añadir `--unit`.

1.7.1 Nuestro primer test

Vamos a crear test funcionales para el CRUD de artículos

```
sail artisan pest:test ArticleTest
```

Nos crea un fichero con un `it()` que vamos a borrar y empezar de cero.

Antes de continuar vamos a abrir el fichero `phpunit.xml` y añadir, en la etiqueta `<php>`:

```
<env name="DB_CONNECTION" value="sqlite"/>
<env name="DB_DATABASE" value=":memory:"/>
```

Si tenemos alguno de esos atributos ya creados, con otros valores, los comentamos.

Con esto vamos a ejecutar los tests en una BD en memoria con sqlite, sin interferir con nuestra BD de la aplicación.

Si no nos importa, podemos dejar activa:

```
<env name="DB_DATABASE" value="testing"/>
```

Empezamos a crear los tests. Lo primero es importar lo que vayamos a necesitar de Pest/laravel. Con Pest vamos a usar funciones y no clases. De Pest:

```php
<?php
use function Pest\Laravel\{actingAs, get};
```

Importamos dos funciones del *namespace* `Pest\Laravel` como son `actingAs` y `get`.

Podemos mirar el fichero `Pest.php` donde podríamos añadir más configuración, entre otras podríamos añadir `Dusk` para ejecutar test en el navegador.

Para instalar y configurar Dusk sigue este enlace y este otro

ATENCIÓN nunca instalar Dusk en un servidor en producción. Podría pasar, que personas ajenas se puedan autenticar con la aplicación sin ser usuarios.

El fichero `index.blade.php` terminado:

```
<x-app-layout>
  <x-slot name="header">
    <h2 class="font-semibold text-xl text-gray-800 leading-tight">
      {{ __('Artículos') }}
    </h2>
  </x-slot>

  <section class="text-gray-600 body-font overflow-hidden">
    <div class="container px-5 py-24 mx-auto">
      <div class="mb-16 -my-8">
        <a href="{{ route("articles.create") }}"
           class="flex w-64 text-white bg-indigo-500
                  border-0 py-2 px-8 focus:outline-none
                  hover:bg-indigo-600 rounded text-lg">
          {{ __("Crear un nuevo artículo") }}
        </a>
      </div>

      <div class="-my-8 divide-y-2 divide-gray-100">
        @foreach($articles as $article)
          <div class="py-8 flex flex-wrap md:flex-nowrap">
            <div class="md:w-64 md:mb-0 mb-6 flex-shrink-0 flex flex-col">
              <span class="font-semibold title-font text-gray-700">
                 {{ $article->category->name }}
              </span>
              <span class="mt-1 text-gray-500 text-sm">
                {{ $article->created_at_formatted }}
              </span>
            </div>
            <div class="md:flex-grow">
              <h2 class="text-2xl font-medium text-gray-900 title-font mb-2">
                 {{ $article->title }}
              </h2>
              <p class="leading-relaxed">{{ $article->excerpt }}</p>
              <a href="{{ route("articles.show", ["article" => $article]) }}"
                 class="text-indigo-500 inline-flex items-center mt-4">
                    {{ __("Ver detalle") }}
```

```
                <svg class="w-4 h-4 ml-2" viewBox="0 0 24 24"
                     stroke="currentColor" stroke-width="2"
                     fill="none" stroke-linecap="round"
                     stroke-linejoin="round">
                  <path d="M5 12h14"></path>
                  <path d="M12 5l7 7-7 7"></path>
                </svg>
              </a> |
              <a href="{{ route("articles.edit", ["article" => $article]) }}"
                 class="text-indigo-500 inline-flex items-center mt-4">
                      {{ __("Editar") }}
                <svg class="w-4 h-4 ml-2" viewBox="0 0 24 24"
                     stroke="currentColor" stroke-width="2"
                     fill="none" stroke-linecap="round"
                     stroke-linejoin="round">
                  <path d="M5 12h14"></path>
                  <path d="M12 5l7 7-7 7"></path>
                </svg>
              </a> |
              <form class="inline" method="POST"
                    action="{{ route("articles.destroy", ["article" => $article]) }}">
                @csrf
                @method("DELETE")
                <button type="submit"
                        class="text-red-500 inline-flex items-center mt-4">
                        {{ __("Eliminar") }}
                  <svg class="w-4 h-4 ml-2" viewBox="0 0 24 24"
                       stroke="currentColor" stroke-width="2"
                       fill="none" stroke-linecap="round" stroke-linejoin="round">
                    <path d="M5 12h14"></path>
                    <path d="M12 5l7 7-7 7"></path>
                  </svg>
                </button>
              </form>
            </div>
          </div>
        @endforeach

        {{ $articles->links() }}
      </div>
    </div>
  </section>
</x-app-layout>
```

2 Anexo

2.1 Pasos tras clonar un repositorio de un proyecto Laravel sail

Cuando subimos un proyecto hecho con Laravel sail, hay ciertas partes que no se suben al repositorio, como binarios, librerías, dependencias y ficheros de configuración con información sensible como el `.env`. Por esto, una vez clonado el repositorio tendremos que hacer algunas cosas, a saber:

1. Rehacer la información del `.env`

   (a) Información de los servidores, contraseñas etc., deberemos copiar el fichero `.env.example` a `.env`, ver qué servicios tenemos activos en nuestro `docker-compose.yml` y cambiar la información correspondiente en el nuevo `.env`, por ejemplo para el servicio de BD con MySQL, los datos a rellenar son:

   ```
   DB_CONNECTION=mysql
   DB_HOST=/ mysql /
   DB_PORT=3306
   DB_DATABASE=/ blog_laravel9 /
   DB_USERNAME=/ sail /
   DB_PASSWORD=/ password /
   ```

   Entre // los datos que hay que cambiar, he puesto sus valores correctos, pero recuerda que no tendrá datos adecuados cuando empecemos a editarlo.

   (b) Ejecutar `composer` para instalar todo el aparataje de Laravel, binarios, librerías, scripts..., incluido `sail` para poder trabajar en el proyecto. Si el proyecto estuviera en su totalidad en el contenedor, o tuviéramos `composer` y php instalado en la máquina un simple `composer install` sería suficiente. Pero nuestro proyecto está fuera de los contenedores, que es donde está PHP, así que tenemos que hacerlo de otra manera, con un contenedor donde está PHP y composer. La versión de PHP debe ser la misma con la que se creó el proyecto.

   ```
   docker run --rm \
   -u "$(id -u):$(id -g)" \
   -v "$(pwd):/var/www/html" \
   -w /var/www/html \
   laravelsail/php81-composer:latest \
   composer install --ignore-platform-reqs
   ```

   (c) Ya tenemos lo necesario para ejecutar y levantar el proyecto con

   ```
   sail up
   # o
   sail up -d
   ```

   (d) Una de las partes que no se preserva es la variable `APP_KEY=`, que deberemos regenerar:

   ```
   sail artisan key:generate
   ```

   (e) Ahora rehacemos la BD con las migraciones y las semillas, si tenemos *seeders*:

   ```
   sail artisan migrate:fresh --seed
   ```

   Normalmente *fresh* no sería necesario en este caso, pero por si las moscas.

   (f) Si hemos instalado Breeze u otro componente que trabaje con el *frontend*, deberemos instalar las dependencias de `node` y ejecutar el servicio de desarrollo.

```
sail npm install
sail npm run dev
```

Si tenemos `yarn` en lugar de `npm`:

```
yarn #o yarn install
yarn dev #o yarn run dev
```

Nota 1: Yarn es más seguro, eso dicen, y puede bajar las dependencias y hacer el build en paralelo, con lo que es más rápido.

Nota 2: Cuando el proyecto pase producción se ejecutará `node`, o `yarn`, con la opción `prod` o `produdction`, eso generará las partes del front definitivas, que deberemos copiar en el sitio correspondiente del proyecto, directorios para JavaScript y CSS.

Si falta algo, se irá añadiendo en este documento.