Proyecto de Laravel

Chirp

Vamos a desarrollar una pequeña aplicación en Laravel, parecida a Twitter, que cada vez que introduzcamos un comentario, mandará un correo a todos los usuarios registrados, excepto al autor del mensaje. Empecemos creando el proyecto:

composer create-project laravel/laravel chipper

Lanzamos el servidor local con

php artisan serve

Podemos usar el comando

php artisan about

Con este comando, podemos ver una lista con el estado de nuestro proyecto. Ahora vamos a instalar el sistema de autencicación Breeze:

composer require laravel/breeze —dev

php artisan breeze:install blade

De esta manera, instalamos las vistas en blade del breeze, por si queremos retocarlas. Como ya sabemos, después de esta instalación podremos ver, de forma totalmente operativa, las opciones de lohin y de register en nuestra página. A continuación:

npm run dev

Esto hará que cualquier cambio que hagamos en el código, ya sea un css, un js o lo que sea, se va a actualizar automáticamente en el navegador. Es el momento de ejecutar las migraciones iniciales que vienen por defecto en Laravel (no se nos debe olvidar cambiar los parámetros en el archivo .env):

php artisan migrate

Recordad que las migraciones son un sistema que nos ofrece Laravel para que, mediante php, tengamos un control de versiones de nuestras tablas de las bases de datos, Es normal que, durante el desarrollo de una aplicación, añadamos, eliminemos o editemos alguno de los campos de una base de datos, por lo que este sistema de migraciones se va a encargar de hacer esos cambios.

Ahora puedes registrarte y crear un inicio de sesión.

Modelos, migraciones y controladores

Los modelos proporcionan una manera de interactuar con las tablas de nuestra base de datos. Por ejemplo, el modelo **Users** permite interactuar con la tabla **users** de nuestra base de datos. Las migraciones nos permiten llevar un control de las versiones de nuestras tablas. Y los controladores son el punto de comunicación de nuestra aplicación con el exterior; es decir, reciben una request y responden una response.

Ahora, vamos a crear el modelo, el controlador y la migración de Chirp:

php artisan make:model -mrc Chirps

Los parámetros mrc significan que se creará también la migración y un controlador resource (preparado con los métodos necesarios para un CRUD).

Routing

Las rutas son el punto de entrada de nuestra aplicación. Nosotros vamos a empezar con dos rutas:

- La ruta **index**, que nos muestra un formulario y el listade de chirps. Esta ruta va a recibir una petición GET.
- La ruta **store**, que usaremos para guardar los nuevos chirps. Va a recibir una petición POST.

Además de estas rutas, vamos a usar las del middleware. El concepto middleware es el de un código que se interpone entre la entrada del usuario y su autenticación. Usaremos dos middleware:

- el **auth** middleware, que se asegura de que sólo los usuarios logados accedan a la ruta

- El **verified** middleware, que decide si el correo se ha habilitado.

Nos vamos al archivo **routes/web.php,** donde añadimos lo siguiente:

```
use App\Http\Controllers\ChirpsController;
```

Y más abajo:

```
Route::resource('chirps', ChirpsController::class)
    ->only(['index', 'store'])
    ->middleware(['auth', 'verified']);
```

Con esto, ya tendríamos creadas las rutas para nuestros index y store. Vamos a testear nuestra ruta y controlador, mandando un mensaje desde el método index de nuestro controlador ChirpsController:

```
 public function index()
    {
        return response(('Hello, World'));
    }
```

Si ahora visitamos http://127.0.0.1:8000/chirps, veremos el mensaje que escribimos en el método del controlador.

Blade

Vamos a actualizar el método index del ChirpsController para usar plantillas Blade:

```
use Illuminate\View\View;

 public function index(): View
    {
        return view('chirps.index');
    }
```

En lugar de retornar una cadena, devolvemos una vista Blade. Esta vista la crearemos en **resources/views/chirps/index.blade.php.** Este archivo es donde vamos a mostrar todas las entradas. Vamos a crearlo:

```
<x-app-layout>
    <div class="max-w-2xl mx-auto p-4 sm:p-6 lg:p-8">
        <form method="POST" action="{{ route('chirps.store') }}">
            @csrf
            <textarea
                name="message"
                placeholder="{{ __('What\'s on your mind?') }}"
                class="block w-full border-gray-300 focus:border-indigo-300 focus:ring focus:ring-
indigo-200 focus:ring-opacity-50 rounded-md shadow-sm"
            >{{ old('message') }}</textarea>
            <x-input-error :messages="$errors->get('message')" class="mt-2" />
            <x-primary-button class="mt-4">{{ __('Chirp') }}</x-primary-button>
        </form>
    </div>
</x-app-layout>
```

Si refrescas la vista en tu navegador, verás el resultado del cambio.

Menú de navegación

Vamos a añadir un nuevo enlace al menú de navegación que nos proporciona Breeze. Para ellos, vamos a actualizar el archivo **resources/views/layouts/navigation.blade.php,** añadiendo estas lineas bajo el enlace al dashboard:

```
 <x-nav-link :href="route('chirps.index')" :active="request()->routeIs('chirps.index')">
            {{ __('Chirps') }}
</x-nav-link>
```

Si refrescamos, veremos como en la cabeceras aparece el nuevo enlace Chirps a la página en la que, de momento, se muestra un pequeño formulario. También debemos añadirlo en la parte del menú para móviles

```
<x-responsive-nav-link :href="route('chirps.index')" :active=“request()-
>routeIs('chirps.index')">
                        {{ __('Chirps') }}
</xx-responsive-nav-link>
```

En este caso, estamos usando el helper **route**. Además le indicamos que el enlace estará activo si estamos en la ruta que se indica en **:active**

Guardando el chirp

Nuestro formulario ha sido configurado para lanzar los mensajes a la ruta chirps.store que hemos creado anteriormente. Si nos fijamos en el formulario que tenemos en nuestra index, tiene un métido **POST** y el action es **chirps.store**. Actualizaremos este método en nuestro ChirpsController para validar los datos y crear un nuevo chirp:

```
use Illuminate\Http\RedirectResponse;
```

```
public function store(Request $request): RedirectResponse
    {
        //
        $validated = $request->validate([
                    'message' => ‘required|string|max:255',
            ]);

        $request->user()->chirps()->create($validated);

        return redirect(route('chirps.index'));
    }
```

Lo primero en los que nos debemos de fijar es en el Request, que es la petición que nos está llegando desde el formulario, guardada en la variable $request. Luego definimos el tipo de respuesta que vamos a dar, que es un RedirectResponse, lo que significa que va a ser una redirección.

Dentro de nuestro método hemos incluido la validación, en el que le decimos que el campo message es requerido y debe tener un máximo de 255 cararcteres de longitud. Si quisiésemos ver que se envían los datos de manera adecuada podríamos hacer

```
dd($validate)
```

Y comprobar que se está mandando de manera adecuada nuestro formulario. Ahora vamos a crear una relación, en la que vamos a decir que un usuario puede tener muchas entradas en este mocroblog. Para ello, nos vamos al modelo Users y añadimos:

```
use Illuminate\Database\Eloquent\Relations\HasMany;
```

Y dentro de la clase:

```
public function chirps(): HasMany
    {
        return $this->hasMany(Chirps::class);
    }
```

Esto lo que va a decir es que el usuario tiene muchos Chirps, o que la relación entre la tabla de usuarios y la de chirps es de uno a muchos.

Protección frente a la asignación masiva

Un problema de seguridad al que nos debemos enfrentar es el evitar que se pasen todos los requests de forma masiva, porque un usuario podria, de esta manera, editar su perfil y adjudicarse, por ejemplo, roles de administrador sin que nadie se los haya concedido.

Laravel nos protege de estos problemas, bloqueando la asignación masiva por defecto. Para solucionar esto, tenemos la propiedad **$fillable** en nuestro modelo **Chirps**:

```
protected $fillable = [
        'message',
    ];
```

De esta manera protegemos a la columna message de la asignación masiva de datos.

Actualizando la migración

Nos falta actualizar la migración, para crear la tabla donde se van a guardar los chirps, y su relación con la tabla **users** de usuarios. En la migración de chirps, añadimos:

```
public function up(): void
  {
     Schema::create('chirps', function (Blueprint $table) {
        $table->id();
        $table->foreignId('user_id')->constrained()->cascadeOnDelete();
        $table->string('message', 255);
        $table->timestamps();
     });
  }
```

Le hemos añadido dos colunmnas, una user_id que es clave foránea de la tabla users, y una columna message, que es donde vamos a guardar los chirps. Los timestamps nos permiten guardar el momento en el que se crean los registros y cuándo se actualizan. Ha llegado el momento de migrar la base de datos

php artisan migrate

Después de correr la migración, veremos que se ha creado nuestra tabla y que ya podemos escribir en el microblog, guardándose el resultado.

Cada migración sólo se ejecuta una sola vez. Para hacer cambios adicionales a una tabla, necesitamos crear una nueva migración. Durante el desarrollo, puede ser que necesites actualizar una migración y reconstruir tu base de datos. Esto puede hacerse con

php artisan migrate:fresh

Artisan Tinker

Tinker nos permite ejecutar código php desde el propio terminal. Para acceder a Tinker usamos

php artisan tinker

Se nos va a abrir un nuevo terminal en el que podemos hacer cosas como esta:

User::all();

Donde podremos ver una lista de todos los usuarios registrados. Para salir de Tinker puedes usar el comando **exit** o pulsar **Control + C.**

Mostrando los Chirps

Vamos a actualizar nuestro método index del ChirpController para pasarle los chirps de cada usuario en nuestra página de inicio.

```
public function index(): View
  {
     return view('chirps.index', [
        'chirps' => Chirps::with('user')->latest()->get(),
     ]);
  }
```

Como podemos ver, no solo estamos rertornando una vista, sino que también le añadimos un array de datos. Pero para que esto funcione, aún necesitamos conectar los Chirps con los usuarios que los han introducido. Para eso, nos vamos al modelo Chirps:

```
use Illuminate\Database\Eloquent\Relations\BelongsTo;
```

Y en el método index:

```
public function user(): BelongsTo
   {
      return $this->belongsTo(User::class);
   }
```

Aquí le estamos diciendo al modelo que la relación que tiene el **Chirp** con el **User** es **BelongsTo**, es decir, que cada chirp pertenece a un usuario en concreto.

Actualizando la vista

Vamos a actualizar nuestro archivo chirp.index para poder mostrar la lista de los chirps. Justo cuando acaba nuestro formulario añadimos

```
<div class="mt-6 bg-white shadow-sm rounded-lg divide-y">
        @foreach ($chirps as $chirp)
                <div class="p-6 flex space-x-2">
                        <svg xmlns="http://www.w3.org/2000/svg" class="h-6 w-6 text-
gray-600 -scale-x-100" fill="none" viewBox="0 0 24 24" stroke="currentColor" stroke-
width="2">
                        <path stroke-linecap="round" stroke-linejoin="round" d="M8
12h.01M12 12h.01M16 12h.01M21 12c0 4.418-4.03 8-9 8a9.863 9.863 0 01-4.255-.949L3
20l1.395-3.72C3.512 15.042 3 13.574 3 12c0-4.418 4.03-8 9-8s9 3.582 9 8z" />
                        </svg>
                        <div class="flex-1">
                                <div class="flex justify-between items-center">
                                        <div>
                                                <span class="text-gray-800">{{ $chirp->user-
>name }}</span>
                                                small class="ml-2 text-sm text-
gray-600">{{ $chirp->created_at->format('j M Y, g:i a') }}</small>
                                        </div>
                                </div>
                                <p class="mt-4 text-lg text-gray-900">{{ $chirp->message }}</
p>
                        </div>
                </div>
         @endforeach
</div>
```

Lo que hemos añadido es un div donde dentro tenemos un bucle foreach, en el que vamos leyendo el usuario propietario del chirp (**$chirp->user->name**), la fecha de creación (**$chirp->created_at->format('j M Y, g:i a')**) y el mensaje (**$chirp->message**). De esta manera, iterando dentro del foreach, vamos leyendo cada uno de los chirps y pintamos toda la información correspondiente.

Editando los Chirps

Vamos a añadir la característica de poder editar los chirps que hayan sido enviados anteriormente. Para ello, nos vamos a ir a nuestras rutas y vamos a añadir las rutas edit y update, que son las que necesitamos para editar los chirps enviados.

```
Route::resource('chirps', ChirpsController::class)
    ->only(['index', 'store', 'edit', 'update'])
    ->middleware(['auth', 'verified']);
```

Como podemos ver, el cambio ha sido añadir las dos nuevas rutas de edit y de update. Lo siguiente es enlazar cada uno de nuestros chirps de la página de inicio con la edición. Para ello, vamos a cambiar nuestra plantilla para añadir esos enlaces:

```
<x-app-layout>
    <div class="max-w-2xl mx-auto p-4 sm:p-6 lg:p-8">
        <form method="POST" action="{{ route('chirps.store') }}">
            @csrf
            <textarea
                name="message"
                placeholder="{{ __('What\'s on your mind?') }}"
                class="block w-full border-gray-300 focus:border-indigo-300 focus:ring
focus:ring-indigo-200 focus:ring-opacity-50 rounded-md shadow-sm"
            >{{ old('message') }}</textarea>
            <x-input-error :messages="$errors->get('message')" class="mt-2" />
            <x-primary-button class="mt-4">{{ __('Chirp') }}</x-primary-button>
        </form>

        <div class="mt-6 bg-white shadow-sm rounded-lg divide-y">
            @foreach ($chirps as $chirp)
                <div class="p-6 flex space-x-2">
                    <svg xmlns="http://www.w3.org/2000/svg" class="h-6 w-6 text-gray-600
-scale-x-100" fill="none" viewBox="0 0 24 24" stroke="currentColor" stroke-width="2">
                        <path stroke-linecap="round" stroke-linejoin="round" d="M8 12h.01M12
12h.01M16 12h.01M21 12c0 4.418-4.03 8-9 8a9.863 9.863 0 01-4.255-.949L3
20l1.395-3.72C3.512 15.042 3 13.574 3 12c0-4.418 4.03-8 9-8s9 3.582 9 8z" />
                    </svg>
                    <div class="flex-1">
                        <div class="flex justify-between items-center">
                            <div>
                                <span class="text-gray-800">{{ $chirp->user->name }}</span>
                                <small class="ml-2 text-sm text-gray-600">{{ $chirp->created_at-
>format('j M Y, g:i a') }}</small>
+                               @unless ($chirp->created_at->eq($chirp->updated_at))
+                                   <small class="text-sm text-gray-600"> &middot; {{ __('edited') }}</
small>
+                               @endunless
                            </div>
+                           @if ($chirp->user->is(auth()->user()))
+                               <x-dropdown>
+                                   <x-slot name="trigger">
+                                       <button>
```

```
+                                    <svg xmlns="http://www.w3.org/2000/svg" class="h-4 w-4 text-
gray-400" viewBox="0 0 20 20" fill="currentColor">
+                                        <path d="M6 10a2 2 0 11-4 0 2 2 0 014 0zM12 10a2 2 0 11-4 0 2
2 0 014 0zM16 12a2 2 0 100-4 2 2 0 000 4z" />
+                                    </svg>
+                                </button>
+                            </x-slot>
+                            <x-slot name="content">
+                                <x-dropdown-link :href="route('chirps.edit', $chirp)">
+                                    {{ __('Edit') }}
+                                </x-dropdown-link>
+                            </x-slot>
+                        </x-dropdown>
+                    @endif
                </div>
                <p class="mt-4 text-lg text-gray-900">{{ $chirp->message }}</p>
            </div>
        </div>
    @endforeach
  </div>
 </div>
</x-app-layout>
```

Este va a ser el nuevo contenido de nuestra página index. Como puede verse, se ha incluido dentro de cada chirp un dropdown con un enlace que apunta a la edición del contenido.

El @unless comprueba si la fecha de edición coincide con la actual, por lo que, en ese caso, escribirá **Editado,** mientras que no lo hará en caso contrario. La otra parte añadida lo que hace es que, si el usuario que está logado es el propietario del chirp, aparecerá un dropdown para editarlo.

Ahora necesitamos el formulario de edición. Vamos a crear un archivo chirps.edit:

```
<x-app-layout>
  <div class="max-w-2xl mx-auto p-4 sm:p-6 lg:p-8">
    <form method="POST" action="{{ route('chirps.update', $chirp) }}">
      @csrf
      @method('patch')
      <textarea
        name="message"
        class="block w-full border-gray-300 focus:border-indigo-300 focus:ring focus:ring-
indigo-200 focus:ring-opacity-50 rounded-md shadow-sm"
      >{{ old('message', $chirp->message) }}</textarea>
      <x-input-error :messages="$errors->get('message')" class="mt-2" />
      <div class="mt-4 space-x-2">
        <x-primary-button>{{ __('Save') }}</x-primary-button>
        <a href="{{ route('chirps.index') }}">{{ __('Cancel') }}</a>
      </div>
    </form>
  </div>
</x-app-layout>
```

Podemos ver que es un formulario con método POST con un action a chirps.update. Usando **@method**, falseamos el action del formulario para que sea de tipo PATCH, que es lo necesario para editar ese post.

Ahora, debemos actualizar nuestro controlador para que reciba esa petición que hacemos desde el action del formulario, en el que le mandamos $chirp, que contiene los datos del formulario:

```
public function edit(Chirps $chirp): View
  {
     //
    // $this->authorize('update', $chirp);

     return view('chirps.edit', [
        'chirp' => $chirp,
     ]);
  }

public function update(Request $request, Chirps $chirp): RedirectResponse
  {
     //
    // $this->authorize('update', $chirp);

     $validated = $request->validate([
        'message' => 'required|string|max:255',
     ]);

     $chirp->update($validated);

     return redirect(route('chirps.index'));
  }
```

Hemos añadido el código necesario a los métodos edit y update. En edit, retornamos la vista con los datos del chirp a editar, mientras que en update, validamos el valor del campo message, actualizamos el registro y retornamos a la index.

Borrando Chirps

A veces no es suficiente con poder editar un chirp, sino que podemos necesitar el borrado de alguno de nuestros contenidos.

Empezaremos actualizando nuestras rutas para permitir la ruta **chirps.destroy:**

```
Route::resource('chirps', ChirpsController::class)
   ->only(['index', 'store', 'edit', 'update', 'destroy'])
    ->middleware(['auth', 'verified']);
```

Ahora necesitamos añadir el códifo necesario en el método destroy del controlador, para decirle qué debe hacer en ese caso:

```
public function destroy(Chirps $chirp): RedirectResponse
  {
     //
    // $this->authorize('delete', $chirp);

     $chirp->delete();

     return redirect(route('chirps.index'));
  }
```

Finalmente, debemos actualizar la vista para añadir el botón de borrado del chirp Bajo el enlace de edición, donde se cierra </x-dropdown-link>, añadimos:

```
<form method="POST" action="{{ route('chirps.destroy', $chirp) }}">
        @csrf
        @method('delete')
        <x-dropdown-link :href="route('chirps.destroy', $chirp)"
onclick="event.preventDefault(); this.closest('form').submit();">
                                {{ __('Delete') }}
        </x-dropdown-link>
</form>
```