1.- Instalación de Laravel 5.5

Antes de comenzar hay que aclarar que para que funcione correctamente Laravel 5.0 es necesario que tengamos instalado PHP v. 7.0.0 o superior.

Antes de nada vamos a ver cómo podemos descargar y configurar el denominado "Esqueleto (estructura) de Laravel 5.5" (basado en el FrameWork) en nuestro equipo.

1- Antes de nada deberíamos de instalar "Git".

   https://git-scm.com

2- El siguiente paso sería tener instalado "Composer[1]". Esto nos permitirá instalar todas las dependencias necesarias para configurar nuestro proyecto. Para descargarlo, vamos a la página de composer y buscamos la descarga para Windows (Windows Installer):

   https://getcomposer.org/download/

3- Para instalarlo, como indica en su documentación (https://laravel.com/docs/5.5), tenemos 2 opciones:
   a. A través del instalador de Laravel.

      ```
      composer global require "laravel/installer"
      laravel new nombre_proyecto
      ```

   b. A través del método "create-project" de composer:

      ```
      composer create-project --prefer-dist laravel/laravel nombre_proyecto
      ```

   La forma más sencilla y la que usaremos es mediante la opción "b". Vamos a abrir el "cmd" y a situarnos en la carpeta xampp/htdocs/, una vez ahí, vamos a escribir dicha línea de código[2], sustituyendo la parte nombre_proyecto por el nombre de la carpeta donde queramos crear el árbol de directorios. Para el desarrollo de este manual, crearemos una aplicación que nos permita gestionar un foro, con lo cual vamos a llamarle "forum".

4- La carpeta "forum", podríamos aprovecharla desde este momento como plantilla para todos los proyectos que vamos a realizar a partir de ahora, evitándonos así volver a realizar todos los pasos anteriores.

---

[1] En el caso de que Composer ya lo tuviésemos instalado, habría que actualizarlo (si acabamos de instalarlo en este momento no es necesario):

composer self-update (ó *php composer.phar self-update)*

[2] Para poder ejecutar estos comandos es necesario incluir el Path de PHP en las variables de entorno de Windows. Si has instalado XAMPP no es necesario que lo hagas.

2.- Configuración inicial de Laravel 5.0

Todos los ficheros de configuración de Laravel se encuentran dentro de la carpeta “config”. Antes de continuar hay varios aspectos que podemos configurar en Laravel:

1- Nombre de la aplicación (totalmente opcional)

Por defecto, el NameSpace de nuestra aplicación es “App”, pero es posible modificarlo mediante el comando “app:name” de Artisan. Para ejecutar el comando debes situarte dentro de la carpeta donde creaste el proyecto “forum”).

php artisan app:name forum

NOTA: Puedes observar, antes y después de ejecutar el comando, en el archivo siguiente como cambia la ruta del namespace:

Aplicación/app/Http/Controllers/Auth/Controller.php

2- Modo depuración

Es posible configurar el modo de depuración para nuestra aplicación como True o False. Esto se puede configurar mediante la variable “APP_DEBUG” que se encuentra en el archivo “.env”

En versiones anteriores de Laravel era posible ver directamente los mensajes de Error (observa en las siguientes figuras los mensajes que se muestran en caso de error en función del valor de dicha variable).

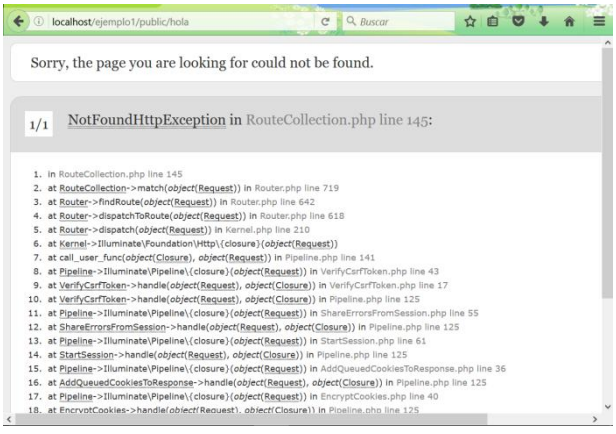


Imagen 1. APP_DEBUG=True

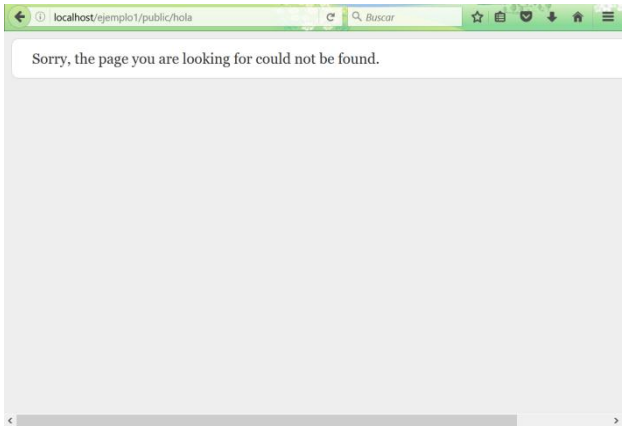


Imagen 2. APP_DEBUG=False

A partir de la versión 5.5, esto ya no es posible, debiendo recurrir a la instalación de paquetes para la visualización de dichos errores. Existen 2 opciones para poder hacer esto:

a) Instalar "filp/whoops" (disponible en el núcleo de Laravel hasta la versión 4.2): https://styde.net/como-instalar-filp-whoops-en-laravel-5/

Para ello tendremos que ejecutar el siguiente comando desde dentro de nuestro proyecto:

```
composer require filp/whoops
```

y modificar el código del método "render" del archivo "App\Exceptions\Handler.php" por este otro:

```
public function render($request, Exception $e)
{
    // Verificamos si el debug está activo
    if (config('app.debug')) {
        // Creamos una nueva instancia de la clase Run
        $whoops = new \Whoops\Run;
        // Registramos el manejador "pretty handler"
        $whoops->pushHandler(new \Whoops\Handler\PrettyPageHandler);

        // Devolvemos una nueva respuesta
        return response()->make(
            $whoops->handleException($e),
            method_exists($e, 'getStatusCode') ? $e->getStatusCode() : 500,
            method_exists($e, 'getHeaders') ? $e->getHeaders() : []
        );
    }
    // Si debug == false : devolvemos la respuesta para la excepción
    return parent::convertExceptionToResponse($e);

    //return parent::render($request, $exception);
}
```

b) Instalar la barra de depuración "Debugbar": https://styde.net/instalar-barra-de-debug-en-laravel/

(Esta opción la veremos en el Punto 14 del manual "Optimización de Consultas con Eloquent")

3- Archivo “config/app.php”

En este archivo podremos cambiar variables de configuración variadas:

- timezone: es posible establecer la zona horaria de nuestra aplicación modificando la que aparece por defecto “UTC”. Para ello, podemos visitar el siguiente enlace donde encontraremos las distintas zonas horarias del mundo (la nuestra en concreto es “Europe/Madrid”):

  http://php.net/manual/es/timezones.php

- locale: con este valor podemos modificar la localización (por defecto “en” – Inglés). Podemos establecerlo a “es”. Además de modificar esta variable es necesario realizar otros cambios. Observa la estructura de carpetas “resources/lang/en/”. Cada uno de los tres archivos contiene líneas de lenguaje que establecen descripciones de mensajes en el idioma especificado (inglés). Para poder cambiar la localización es necesario crear una nueva carpeta con el lenguaje seleccionado p.e. “resources/lang/es/” y crear los tres archivos con sus contenidos escritos en dicho lenguaje. Puede ayudarte para el contenido de los archivos el siguiente enlace (están dentro de la carpeta “src”):

  https://github.com/caouecs/Laravel-lang

  NOTA: También se puede cambiar el idioma en tiempo de ejecución utilizando el método setLocale de App. Este cambio no es permanente, en la siguiente ejecución se utilizará el valor de configuración por defecto:

  ```
  App::setLocale('es');
  ```

4- Eliminar “public” de la URL

Para eliminar la palabra “public[3]” de las URL sigue los siguientes pasos:

- Renombra el archivo “server.php” del directorio raíz de la aplicación a “index.php”
- Copia el archivo “.htaccess” desde el directorio “public” a la carpeta raíz de la aplicación.
- Cambia el contenido de la función “asset” del archivo “/vendor/laravel/framework/src/Illuminate/Foundation/helpers.php” como sigue:

```
function asset($path, $secure = null)
{
    return app('url')->asset("public/".$path, $secure);
}
```

[3] **NOTA**: Para dar más seguridad a la aplicación una vez que esté en producción se suele recomendar renombrar la carpeta “public” a otro nombre, modificando el valor tanto en la función anterior como en las dos apariciones que tiene dicha carpeta en el archivo “index.php” del directorio raíz.

3.- MVC: Modelo-Vista-Controlador

1- CONTOLADOR:

El Controlador se encarga de gestionar la lógica de la aplicación. También se encarga de controlar parámetros como la seguridad o funciones adicionales.
Hay que tener en cuenta que pueden existir varios controladores y realizan tareas como:

- Controlar las Sesiones
- Controlar la Seguridad
- Controlar la comunicación entre los módulos y el paso de parámetros

2- MODELO:

El Modelo se encarga de la interacción de la aplicación con los datos, de tal manera que recibe peticiones de datos, por parte del Controlador, y realiza las operaciones sobre la fuente de datos (BD) a la que esté conectada. Además ejecuta las Reglas de Negocio.

3- VISTA:

La Vista se encarga de interactuar con el usuario, de tal manera que recoge las acciones que el usuario quiere ejecutar y devuelve los resultados que ha devuelto la aplicación.

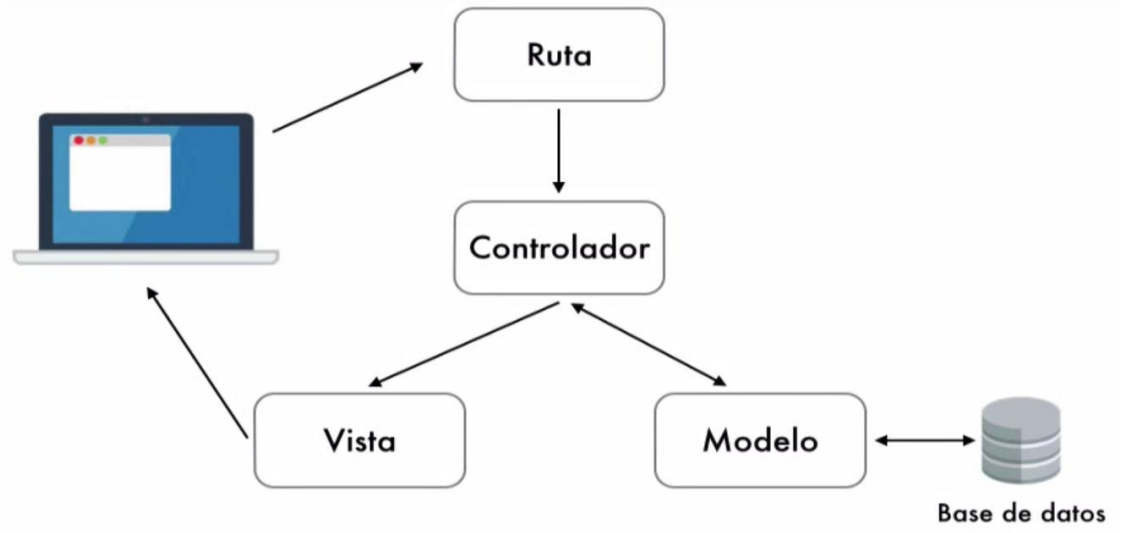


Fig 1. Funcionamiento del MVC (a)

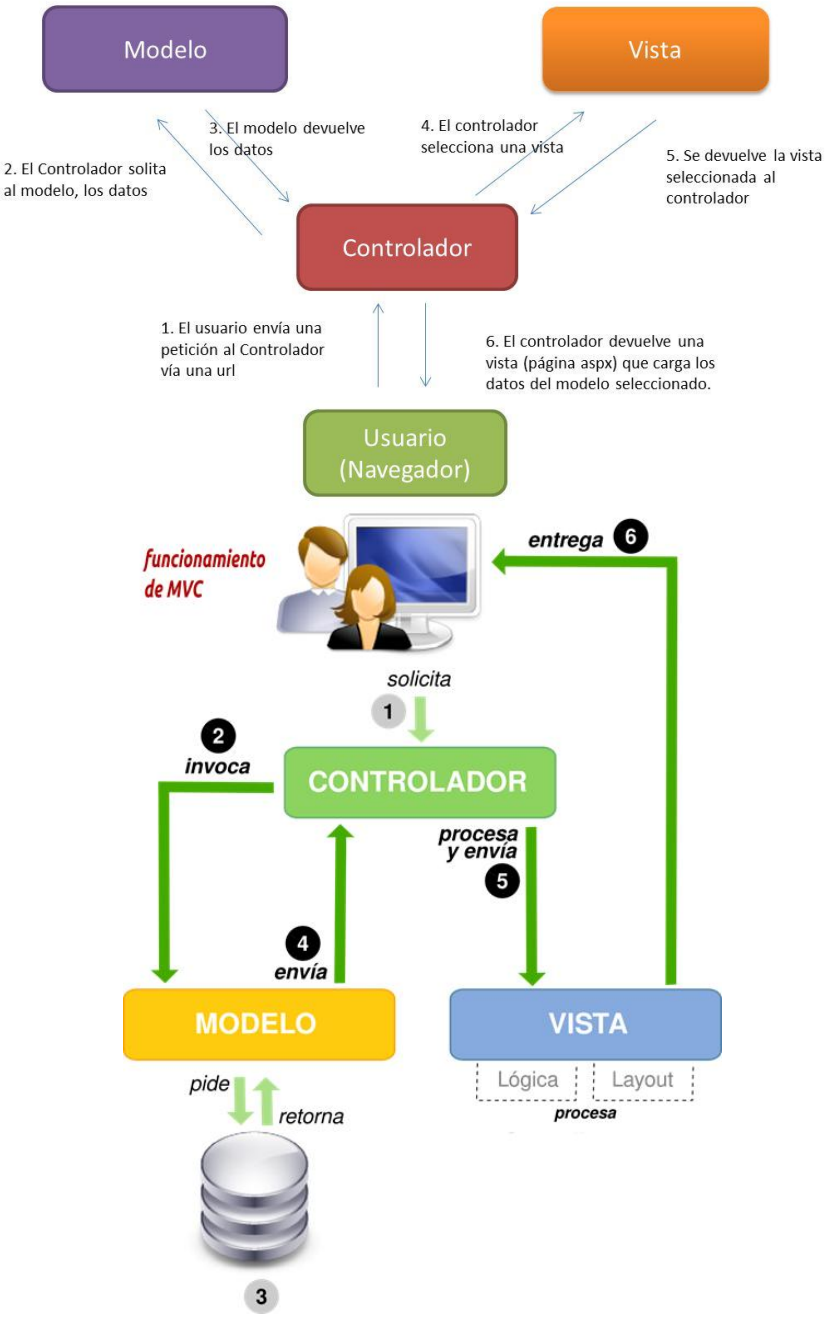


Fig 2. Funcionamiento del MVC

4.- Rutas

Cuando se trabaja con recursos web, es necesario especificar una ruta de acceso a dichos recursos, ya sea un archivo, una página, etc.

Como vimos en el apartado anterior una ruta es, básicamente, una petición por parte del usuario. Éste hace peticiones a través del navegador, las cuales son enrutadas indicando al controlador dicha petición y el Controlador procesa la petición decidiendo si es necesario llamar a un Modelo, porque se soliciten datos de una base de datos (el Modelo le retornará dichos datos al Controlador para generar la Vista), o generar directamente una Vista que le mostrará al usuario lo que ha solicitado.

Las rutas se definen en el archivo "routes/web.php"

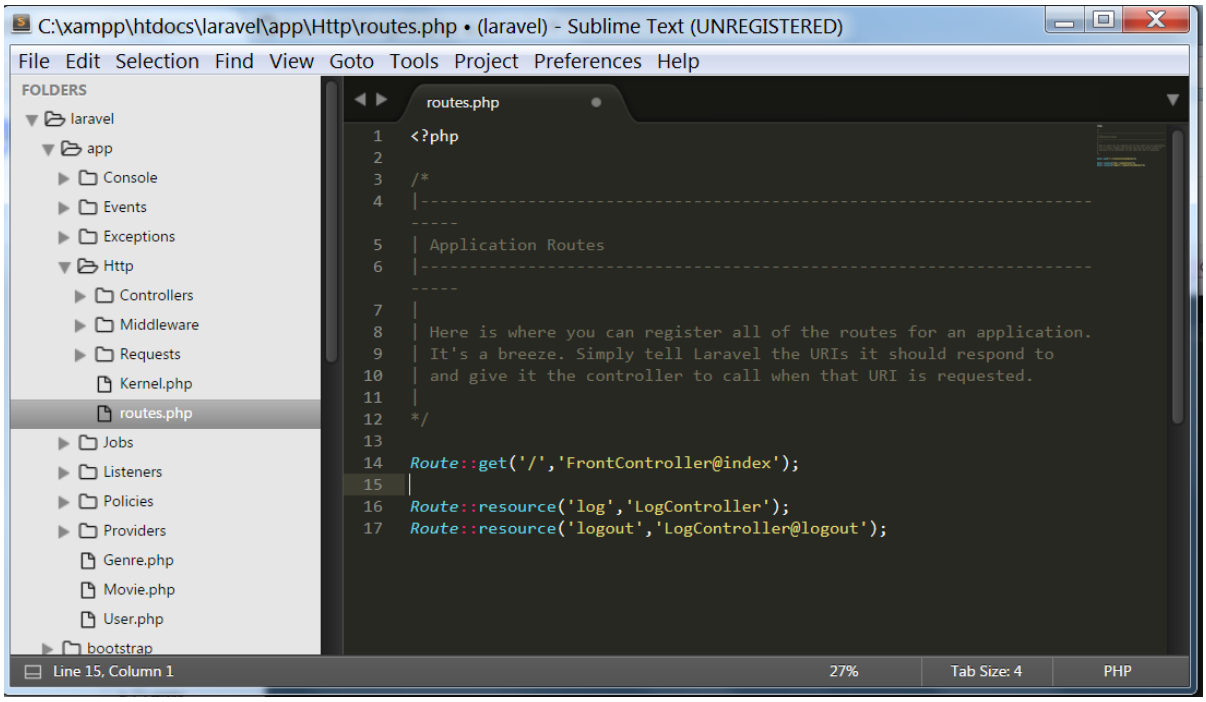


Imagen 3. Archivo "web.php"

Como podemos ver ya existen rutas establecidas y podremos crear nuevas rutas haciendo referencia a la clase "Route" indicándole el verbo HTTP que vamos a usar (get, post, delete, put, …) y como argumentos le vamos a indicar en primer lugar la "Ruta" que habrá que poner en el navegador y en segundo lugar, para este ejemplo, crearemos una función que retornará un mensaje al usuario.

```
Route::get('prueba', function(){
        return "Hola mundo";
});
```

Si ahora escribimos en el navegador "localhost/forum/public/prueba" veremos que nos muestra el mensaje que hemos indicado.

En Laravel existen otros tipos de rutas además de las anteriores (básicas), como son las rutas con parámetros (opcionales u opcionales con valores por defecto):

En este caso mediante llaves se indica el nombre del parámetro que vamos a recibir mediante la url

```
Route::get('prueba/{nombre}', function($nombre){
        return “Hola ” . $nombre;
});
```

En este caso la url será “localhost/forum/public/prueba/Raúl”

y el resultado mostrado será “**Hola Raúl**”.

Algo parecido sería para crear rutas básicas con valores por defecto. Para ello crearíamos la ruta poniendo la interrogación detrás del parámetro e igualando el parámetro de la función al valor por defecto:

```
Route::get('prueba/{edad?}', function($edad = 44){
        return "Mi edad es " . $edad;
});
```

Si omitimos el parámetro y escribimos la siguiente url “localhost/forum/public/prueba”

El resultado será **“Mi edad es 44”**

Sin embargo si añadimos el parámetro y escribimos “localhost/forum/public/prueba/26”

El resultado será **“Mi edad es 26”**

5.- Controladores

Los Controladores son los encargados de responder a las peticiones de los usuarios. En función de la petición el Controlador invocará a un Modelo (si necesita acceder a datos de un origen de datos) o directamente a la Vista para responder a la petición del usuario.

En Laravel los Controladores se encuentran en la carpeta “app/Http/Controllers” y estos pueden ser enrutados desde el fichero “web.php” como vimos anteriormente.

Como ejemplo vamos a crear un nuevo Controlador, para ello iremos a la carpeta de Controladores, crearemos un nuevo fichero que llamaremos “PruebaController.php” y escribiremos el siguiente código

```
<?php

namespace Forum\Http\Controllers;

class PruebaController extends Controller
{
        public function index(){
                return "Hola desde el nuevo controlador";
        }
}
```

A continuación nos vamos a nuestro archivo “web.php” y creamos nuestra ruta

```
Route::get('controlador','PruebaController@index');
```

Si vamos al navegador y escribimos “localhost/forum/public/controlador” veremos que nos mostrará el mensaje.

También podemos crear otra función en el controlador que reciba parámetros:

```
public function nombre($name){
        return "Hola " . $name;
}
```

Y creamos la ruta para acceder a ella:

```
Route::get('name/{name}', 'PruebaController@nombre');
```

Si vamos al navegador y escribimos “localhost/forum/public/name/raul” veremos que nos mostrará el mensaje.

Todo esto se volvería incontrolable si tuviésemos diez Controladores y cada uno de ellos con 8 métodos. Para ello existen los Controladores RESTful.

Para declarar un Controlador RESTful iríamos al archivo "web.php" y escribiríamos la siguiente línea de código:

```
Route::resource('forum','PruebaController');
```

Pero mejor lo vamos a hacer mediante "Artisan", que es la línea de comandos creada para trabajar con Laravel desde PHP. Para ello nos iremos a la línea de comandos (en la carpeta del proyecto) y escribiremos lo siguiente:

```
php artisan make:controller PruebaController --resource
```

Esta declaración va a generar múltiples rutas para cada método del Controlador (index, create, store, show, edit, update y destroy).

Si vamos ahora a nuestra carpeta de Controladores podemos ver como se ha creado nuestro Controlador y tiene todos sus métodos ya definidos. Ahora ya podemos cambiar algunos métodos para probarlo. Por ejemplo vamos a modificar el método index() y ponemos lo siguiente:

```
public function index(){
        return "Hola desde el index";
}
```

Y para probarlo pondremos en el navegador:

"localhost/forum/public/forum"

NOTA: Observa cómo, si no añadimos nada más en la ruta, se ejecuta el método "index".

6.- Modelos (Eloquent ORM)

Un modelo no es otra cosa que la representación de los datos con los que la aplicación va a trabajar. Es por tanto quien gestiona todos los accesos a dichos datos. Básicamente, el Modelo es la representación de una tabla de nuestra Base de Datos.

Para generar un Modelo podemos hacerlo mediante código, directamente sobre un archivo php, o podemos hacerlo mediante el siguiente comando Artisan:

php artisan make:model Forum

NOTA: Para hacerlo de una forma más completa, vamos a hacerlo de la siguiente forma:

php artisan make:model Forum -mcf

Con esto, además de crear el Modelo, estaremos creando una Migración, la Factoría y un Controlador (aunque todos estos objetos podemos crearlo, de forma independiente, mediante Artisan).

Podemos ver cómo ha creado nuestro Modelo "Forum" (en la carpeta "app"), la Migración (en la carpeta "database\migrations"), la Factoría (en "database\factories") y el Controlador (en "app\Http\Controllers").

NOTA: Si creamos antes el Controlador de forma unitaria no funcionarán después las factorías.

Eloquent se basa en lo que se denomina CodeFirst que consiste en crear las clases y sus relaciones y olvidarse de la creación de la Base de Datos, aunque existe la posibilidad de trabajar con Bases de Datos existentes.

Una vez creado el Modelo debemos ir al código que ha generado y crearemos la línea desde donde haremos referencia a nuestra Tabla. Para ello vamos a escribir:

```
<?php

namespace Forum;

use Illuminate\Database\Eloquent\Model;

class Forum extends Model {
    protected $table = 'forums';
}
```

Al usar Eloquent hay que tener en cuenta varias cosas:

- Se asume que el Modelo "User" almacena registros en la tabla "users"
- Se asume que cada tabla tiene una columna llamada "id" que es la clave primaria de dicha tabla (aunque esto es posible cambiarlo).
- Es posible establecer qué atributos van a poder ser rellenados en la tabla por un usuario. Esto se hace mediante la línea: ***protected $fillable = ['campo1','campo2','campo3'];***

7.- Migraciones

Una migración podríamos definirla como un tipo de control de versiones de nuestra Base de Datos, permitiendo a un equipo modificar el esquema de la misma y estar al día en dichas modificaciones.

Antes de avanzar observa que, en el apartado anterior, cuando creamos el Modelo denominado "Forum", automáticamente se creó la migración para el mismo dentro de la ruta "database/migrations".

Para crearlo mediante código tendremos que ver que columnas compondrán nuestra tabla y de qué tipo son. Para ello iremos a la migración que se ha creado de nuestro modelo y modificaremos la parte del código que sirve para crear el Esquema añadiendo aquellos campos que sean necesarios.

Vamos a crear primero el esquema de la tabla 'forums'

Los tipos de datos que podemos usar para crear los campos pueden verse en la documentación de Laravel:

https://laravel.com/docs/5.5/migrations#creating-columns

```
public function up() {
    Schema::create('forums', function (Blueprint $table) {
        $table->engine = "InnoDB";      // Esto permite escribir Relaciones y Claves Foráneas[4]
        $table->increments('id');
        $table->string('name');         // Nombre del Foro
        $table->text('description');    // Descripción del Foro
        $table->timestamps();
    });
}
```

Para continuar debemos configurar el fichero ".env" con los valores de acceso de nuestra base de datos (host, base_de_datos, usuario y contraseña) y debemos crear la base de datos (desde PHPMyAdmin).

Antes de realizar la migración, para evitar el error "SQLSTATE[42000]", debemos modificar el archivo "app/Providers/AppServiceProvider.php" de la siguiente forma:

```
use Illuminate\Support\ServiceProvider;
use Illuminate\Support\Facades\Schema;

public function boot()
{
    Schema::defaultStringLength(191);
}
```

[4] Para hacerlo de forma Global (en todas las tablas) debemos ir al archivo "database.php" y modificar esta línea:

```
'engine' => null,
```

por esta otra:

```
'engine' => 'InnoDB',
```

Una vez hecho esto debemos ejecutar el comando Artisan para migrar la BD:

php artisan migrate

Esto creará (o modificará) las tablas con los parámetros establecidos en el fichero anterior.

También es posible realizar "RollBack" de la base de datos si hemos cometido algún error, para ello escribimos:

php artisan migrate:rollback

Esto hará que se ejecute el método "down()" del archivo de migración.

NOTA: En el siguiente enlace se explica cómo hacer una migración de una base de datos existente:

https://styde.net/como-generar-migraciones-de-una-base-de-datos-existente/

8.- Factorías

Para poder poblar nuestras tablas vamos a utilizar los denominados “Factories” que se almacenan en la carpeta “database”. En el apartado 6, con el comando “php artisan make:model Forum –mcf” ya creamos nuestra factoría “ForumFactory.php”. Para poder utilizarla vamos a modificar el contenido de la misma como sigue:

```
<?php
use Faker\Generator as Faker;

$factory->define(forum\Forum::class, function (Faker $faker) {
    return [
        'name' => $faker->sentence,
        'description' => $faker->paragraph,
    ];
});
```

Para poder usar la Factoría podemos hacerlo de varias formas:

1- Desde la línea de comandos ejecutamos:

```
php artisan tinker
```

y una vez dentro de Tinker escribimos

```
factory(App\Forum::class, 50)->create();
```

creando así 50 registros en la tabla de Foros.

NOTA: Para salir de Tinker simplemente ejecutamos “exit”

2- Mediante el archivo “DatabaseSeeder.php” que se encuentra en la carpeta “database\seeds”, modificando el código del método “run” como sigue:

```
public function run()
{
   factory(\App\Forum::class, 100)->create();
}
```

Una vez hecho esto, hay que ejecutar el Seeder mediante Artisan:

php artisan migrate:refresh --seed

Como podemos ver ha eliminado (Rollback) y vuelto a crear las tablas y ha creado 100 registros con datos aleatorios

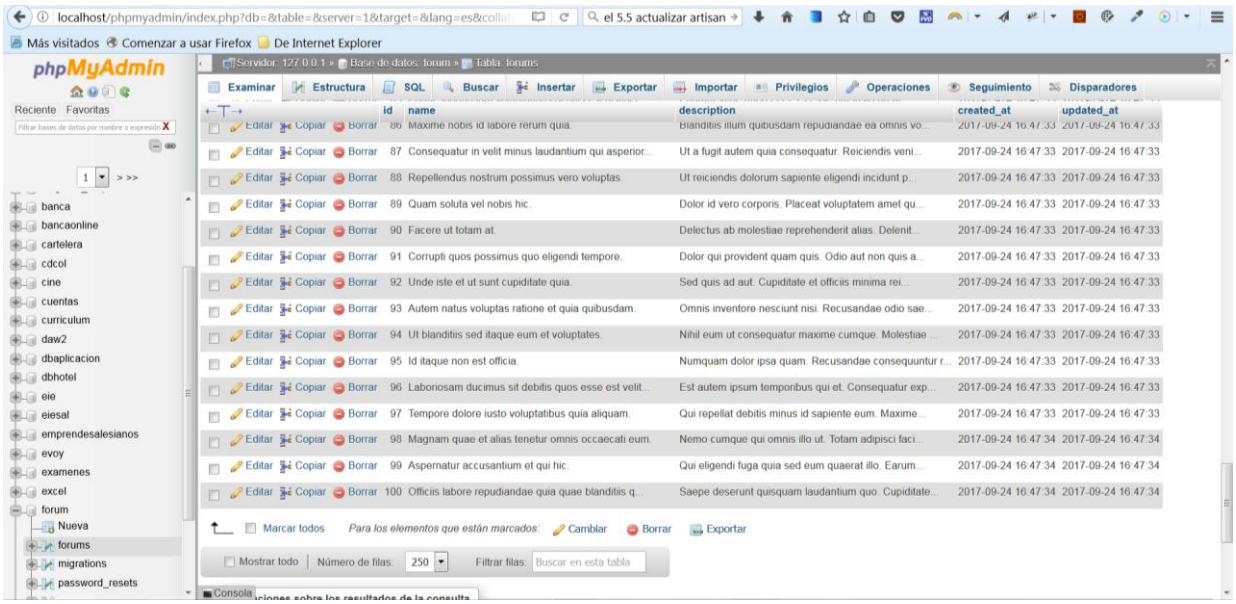


Imagen 4. Registros de la Tabla “Forum”

9.- Autenticación de usuarios

Laravel nos da la facilidad de crear, desde cero, un sistema de autenticación de usuarios sin tener que escribir código.

Para ello sólo tenemos que acceder a la línea de comandos y ejecutar la siguiente instrucción:

php artisan make:auth

Con esto dispondremos de un sistema de autenticación completo, sin tener que hacer nada más. Como podemos ver, en la parte superior derecha de la ventana nos aparecen ya los botones de "LOGIN" y "REGISTER", los cuales están ya totalmente validados.

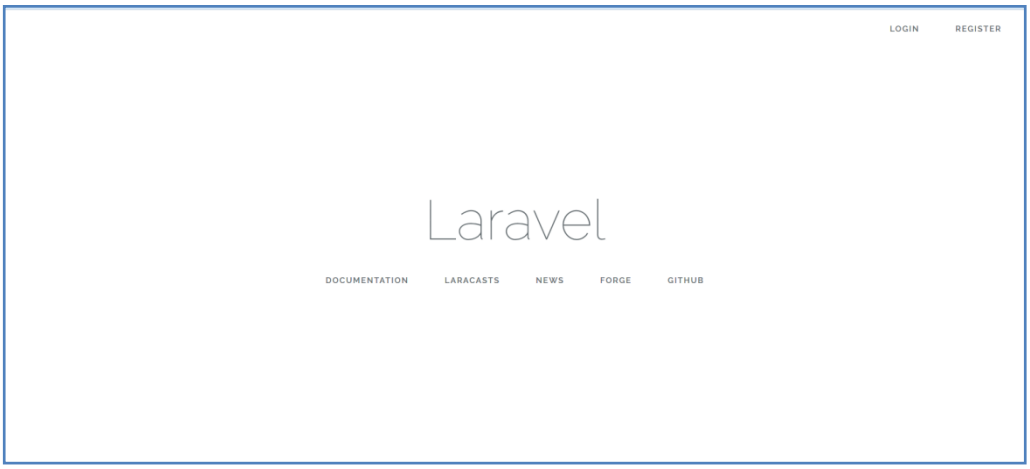


Imagen 5. Cabecera "Home"

Podemos ver también como en la carpeta de Vistas se ha creado una nueva carpeta llamada "auth" y dentro de ella varias Vistas. Además se ha creado otro Layout llamado "app.blade.php" que define la nueva barra de navegación y permite incluir en él un contenido "content" que será el que incorpore las distintas Vistas que hereden de este Layout.

10.- Vistas

Las “Vistas” son los archivos que contienen código Html que sirve nuestra aplicación y permiten separar los controladores y la lógica de la aplicación de la lógica de presentación. Estos van a mostrar la información a los usuarios finales. Además van a permitir que los usuarios interactúen con la aplicación mediante el uso de formularios.

Las vistas se almacenan en la carpeta “resources/views”

Antes de crear las vistas, lo primero que debemos hacer es crear una ruta (que representa una petición de un usuario), que en nuestro caso hará referencia a nuestro Controlador “ForumController”. Para crear la ruta iremos al archivo “routes/web.php” y escribiremos algo como esto:

```
Route::get('/', function () {
    return view('welcome');
});
```

NOTA: A partir de Laravel 5.5 es posible especificar una ruta de forma algo más abreviada, por ejemplo:

```
Route::view('/', 'welcome');
```

En nuestro ejemplo, para poder mostrar todos los foros en la ventana de inicio, lo que haremos será modificar la ruta de más arriba (la que usa la función anónima) con el siguiente código:

```
Route::get('/', 'ForumController@index');
```

Con esta ruta le estamos diciendo que debe llamar al método “index” del Controlador “ForumController”. Por tanto, lo siguiente que debemos hacer es ir al Controlador y crear dicho método. El contenido del Controlador debe ser el siguiente:

```
<?php

namespace App\Http\Controllers;

use App\Forum;

class ForumController extends Controller
{
    public function index()
        {
            $forums = Forum::all();
        }
}
```

Con esto le estamos diciendo a Laravel que recupere todos los datos de los foros mediante el Modelo "Forum". Para poder ver dichos datos de una forma rápida podríamos usar la función "dd()" de la siguiente forma:

```
$forums = Forum::all();
dd($forums);
```

Si vamos al navegador y mostramos la página de inicio podemos ver cómo nos dice que tenemos una colección de 100 registros devueltos.

Imagen 6. Mostrar Foros

Estos datos podríamos hacer que estuvieran ordenados y paginados de una forma muy sencilla, simplemente tendríamos que modificar la línea donde accedemos al Modelo y escribir lo siguiente:

```
$forums = Forum::latest()->paginate(5);
```

Esto devolvería los registros ordenados desde el último que fue creado y paginados de 5 en 5. Veremos a continuación que, además de esto, Laravel nos permite crear, de una forma muy sencilla los enlaces para poder movernos por las distintas páginas (para ello sólo tendremos que llamar al método "links()").

Para pasar dicha información a una Vista, en lugar de verlo con la función "dd()", lo que haremos será llamar al Helper "view" para devolver los datos a la vista compactándolos mediante la función "compact()":

```
return view('forums.index', compact('forums'));
```

Con esto le estamos diciendo a Laravel que mande los datos compactados a la Vista "index.blade.php" que crearemos dentro de una carpeta que llamaremos "forums" para tener las Vistas más ordenadas.

11.- Plantillas (Blade)

Blade es un motor de plantillas que ayuda a reducir las líneas de código que necesitamos para nuestra aplicación. La documentación oficial podemos encontrarla en:

https://laravel.com/docs/5.5/blade

Lo primero que debemos tener en cuenta es que todas las plantillas deben tener la extensión "blade.php". Para definir una plantilla Blade se apoya en "secciones" y "herencia" (extends).

En la documentación podemos ver que primero hay que definir la estructura de una plantilla y luego la usaremos desde otros archivos.

También podemos hacer una especie de "echo" de los datos si lo incluimos entre llaves: **{{ $dato }}**

En la carpeta "views" tenemos una carpeta denominada "layouts" y dentro de esta tenemos una plantilla que se llama "app.blade.php" desde la que heredarán el resto de vistas (aunque podemos crear nuevas plantillas y que hereden de estas otras).

El código de nuestra nueva Vista "index" que crearemos dentro de la nueva carpeta llamada "forums" será el siguiente:

```
@extends('layouts.app')

@section('content')
  <div class="row">
    <div class="col-md-8 col-md-offset-2">
        <h1 class="text-center text-mute"> {{ __("Foros") }} </h1>

        @forelse($forums as $forum)
            <div class="panel panel-default">
              <div class="panel-heading">
                <a href="forums/{{ $forum->id }}"> {{ $forum->name }} </a>
              </div>

              <div class="panel-body">
                {{ $forum->description }}
              </div>
            </div>
        @empty
          <div class="alert alert-danger">
            {{ __("No hay ningún foro en este momento") }}
          </div>
        @endforelse
    </div>
  </div>
@endsection
```

NOTA: Las líneas del estilo "{{ __("Foros") }}" son traducciones JSON disponibles a partir de Laravel 5.4 y que nos van a permitir traducir nuestras aplicaciones de una forma muy sencilla (lo veremos más adelante).

Para que nos quede un poco mejor, vamos a modificar la línea de la Plantilla “layout/app.blade.php” donde creamos la herencia de la sección “content” de la siguiente forma:

```
<div class="container">
        @yield('content')
</div>
```

Como podemos ver en el resultado, sólo nos está mostrando los 5 primeros registros. Para solucionar esto, como dijimos anteriormente, llamaremos al método “links()” que nos mostrará los enlaces de paginación. Simplemente tendremos que escribir el siguiente código después de la línea “@endforelse” de nuestra vista:

```
        @endforelse
        @if($forums->count())
          {{ $forums->links() }}
        @endif
      </div>
    </div>
@endsection
```

12.- Creación de los Posts de los Foros

Como sabemos, un foro está definido por muchos posts y cada post será creado por un usuario. Para crear esta parte del programa vamos a desarrollar nuestra Migración, la Factoría, el Controlador y el Modelo para dichos Posts.

Para ello vamos a escribir en la línea de comandos lo siguiente:

php artisan make:model Post -mcf

Esto creará el Modelo, la Migración y el Controlador para los posts

Lo primero que haremos será modificar nuestro archivo de migración de la siguiente forma:

```
public function up()
{
    Schema::create('posts', function (Blueprint $table) {
        $table->engine = "InnoDB";     // Esto permite escribir Relaciones y Claves Foráneas
        $table->increments('id');
        $table->unsignedInteger('user_id');    // Creamos la columna que hará de clave foránea
        $table->foreign('user_id')->references('id')->on('users');   // Definición de la clave foránea
        $table->unsignedInteger('forum_id');    // Creamos la columna que hará de clave foránea
        $table->foreign('forum_id')->references('id')->on('forums');   // Definición de la clave foránea
        $table->string('title');                    // Título del Post
        $table->text('description');                // Descripción del Post
        $table->timestamps();
    });
}
```

Con esas dos columnas que hemos creado (user_id y forum_id), estamos especificando quien será el creador del “Post” y a que “Foro” pertenece.

Para poder poblar la nueva tabla vamos a modificar la nueva Factoría “PostFactory” como sigue:

```
$factory->define(App\Post::class, function (Faker $faker) {
   return [
      'user_id' => \App\User::all()->random()->id,
      'forum_id' => \App\Forum::all()->random()->id,
      'title' => $faker->sentence,
      'description' => $faker->paragraph,
   ];
});
```

Con la primera línea le estamos diciendo que coja un Usuario aleatoriamente, pero que exista en la tabla Users y con la primera, lo mismo, pero de la tabla Forums.

Para poder usar la Factoría modificaremos el archivo "DatabaseSeeder.php" como sigue:

```
public function run()
{
   factory(\App\User::class, 50)->create();
   factory(\App\Forum::class, 20)->create();
   factory(\App\Post::class, 100)->create();
}
```

Una vez hecho esto, hay que ejecutar el Seeder mediante Artisan:

php artisan migrate:refresh --seed

Para mostrar los datos creados de una forma sencilla, vamos a nuestro archivo "web.php" y vamos a crear una nueva ruta:

```
Route::get('/forums/{forum}', 'ForumController@show');
```

Ahora tendremos que ir a nuestro Controlador "ForumController" y crear el método "show":

```
public function show(Forum $forum)  // Con esto estamos inyectando el Foro completo
{
        dd($forum);
}
```

Antes de desarrollar el método "show", tendremos que:

1. Relacionar el Modelo "User" con el Modelo "Forum".
2. Y tenemos que decirle como queremos coger las relaciones utilizando la carga dinámica de Eloquent para que no se dupliquen las consultas y los resultados se muestren de forma correcta.

NOTA: Si quisiéramos crear un usuario concreto para que se creara cada vez que reiniciemos la base de datos, podríamos poner, en la función "run()" la siguiente línea de código:

```
factory(User::class)->create(['email' => 'raulreyes@gmail.com']);
factory(\App\User::class, 50)->create();
```

Puedes observar que en el archivo "factories/UserFactory.php", al crear un nuevo Usuario desde la Factoría, por defecto, lleva la password "secret":

```
'password' => $password ?: $password = bcrypt('secret'),
```

13.- Relaciones entre Modelos y carga dinámica

Para definir las relaciones entre el Modelo “User” y el Modelo “Forum” vamos a modificar el fichero “Forum.php” de la siguiente forma:

```
class Forum extends Model
{
  protected $table = 'forums';

   protected $fillable = [
      'name', 'description',
   ];

   public function posts(){
         return $this->hasMany(Post::class);
   }
}
```

Con la función “posts”, le estamos diciendo a Laravel que un Foro puede tener muchos Posts. Ahora tendremos que hacer la relación inversa (aunque aquí no la vamos a usar pero es posible que lo necesitemos en alguna ocasión), para ello iremos al Modelo “Post.php” y lo modificaremos de la siguiente forma:

```
class Post extends Model
{
   protected $table = 'posts';

   protected $fillable = [
      'forum_id', 'user_id', 'title', 'description',
   ];

   public function forum(){
         return $this->belongsTo(Forum::class, 'forum_id');
   }

   public function owner(){
         return $this->belongsTo(User::class, 'user_id');
   }
}
```

En este caso le estamos diciendo que el Post pertenece a un Foro y a un Usuario concreto (propietario).

NOTA: Cuando usamos “protected $hidden” en un Modelo, los campos incluidos dejarán de mostrarse sólo si los estamos enviando en forma de Array, por ejemplo “$user->toArray()”.

Ahora tendremos que crear en el Modelo “user.php” la relación que existe entre un usuario y los post (un usuario puede tener muchos posts):

Para ello escribimos el siguiente código dentro del modelo:

```
public function posts(){
   return $this->hasMany(Post::class);
}
```

Ahora tendremos que modificar la función “show” del Controlador “ForumController” para que muestre dichos posts. Para ello cambiaremos la línea que hay dentro de “show” por la siguiente:

```
$posts = $forum->posts()->with(['owner'])->paginate(2);
```

Lo que le estamos diciendo es que llame al método “posts()” del Modelo “Forum” y mediante “with” le decimos cuáles son las relaciones que queremos que use. Al ser un array podríamos indicar varias.

Si mostrásemos los Posts “dd($posts)” podríamos ver que ahora, además de mostrarnos los datos de los Posts, nos mostraría los datos del “propietario” de dichos posts (owner).

Lo que haremos en lugar de eso será devolver dichos datos a una vista que llamaremos “detail” de la siguiente forma:

```
return view('forums.detail', compact('forum','posts'));
```

Vamos a crear ahora la Vista

```
@extends('layouts.app')

@section('content')

<div class="row">
  <div class="col-md-8 col-md-offset-2">
    <h1 class="text-center text-mute"> {{ __("Posts") }} </h1>

    @forelse($posts as $post)

      <div class="panel panel-default">
        <div class="panel-heading">
          <a href="../posts/{{ $post->id }}"> {{ $post->title }} </a>
          <span class="pull-right">
              {{ __("Owner") }}: {{ $post->owner->name }}
          </span>
        </div>

        <div class="panel-body">
          {{ $post->description }}
        </div>
      </div>
    @empty
      <div class="alert alert-danger">
        {{ __("No hay ningún post en este momento") }}
      </div>
    @endforelse

    @if($posts->count())
      {{ $posts->links() }}
    @endif
  </div>
</div>

@endsection
```

Podríamos mejorar la vista “index.blade.php” para que muestre, de cada Foro, cuántos Posts tienen. Para ello añadimos el siguiente código a la vista:

```
<a href="forums/{{ $forum->id }}"> {{ $forum->name }} </a>
    <span class="pull-right">
        {{ __("Posts") }}: {{ $forum->posts->count() }}
    </span>
```

14.- Optimización de Consultas con Eloquent

Para poder optimizar las consultas vamos a utilizar una herramienta que nos permitirá hacer debug tanto de las consultas, como de las rutas, de las vistas, de las excepciones, de los emails, de las redirecciones y de los requests en Laravel. Esta herramienta es “Laravel Debugbar” y está disponible en la siguiente dirección:

https://github.com/barryvdh/laravel-debugbar

para instalarlo ejecutaremos, en la línea de comandos, el siguiente comando:

composer require barryvdh/laravel-debugbar --dev

NOTA: Para versiones anteriores a Laravel 5.5, tendríamos que añadir en el fichero “config/app.php” en el apartado de ‘providers’ la siguiente línea:

Barryvdh\Debugbar\ServiceProvider::class,

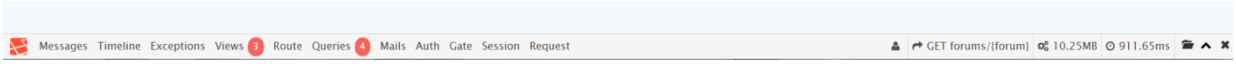
En el apartado ‘aliases’ esta otra:

'Debugbar' => Barryvdh\Debugbar\Facade::class,

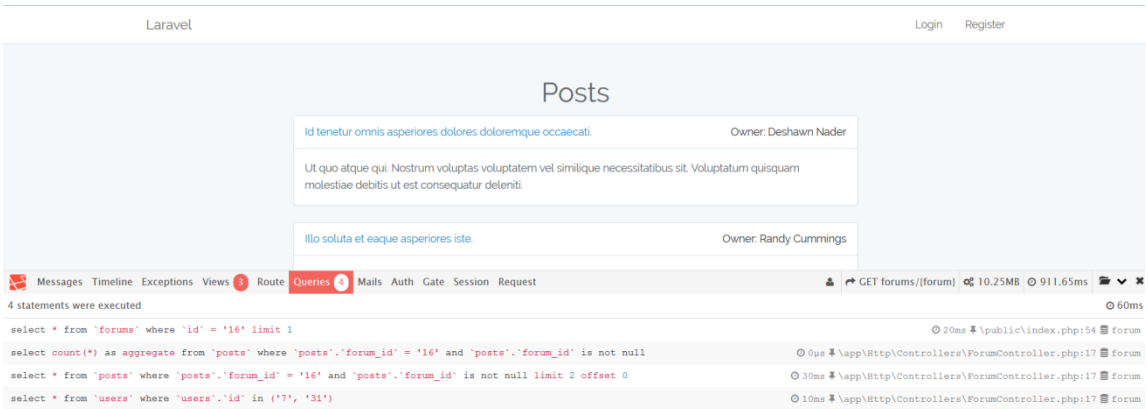
Y ejecutar en la línea de comandos lo siguiente:

php artisan vendor:publish --provider="Barryvdh\Debugbar\ServiceProvider"

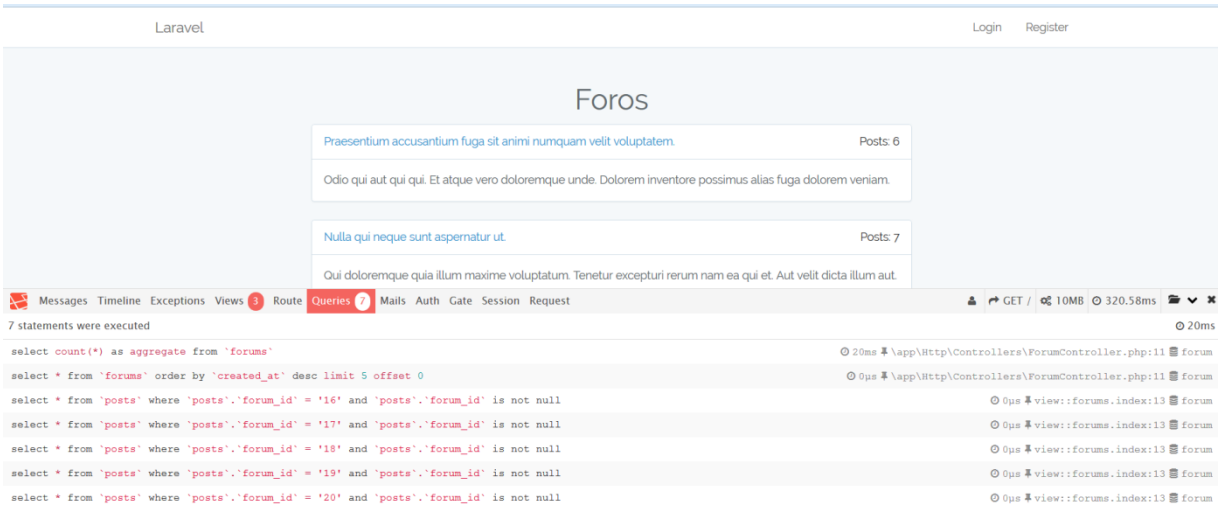
Con esto nos aparecería la barra del Debugbar en la parte inferior del navegador



Y podríamos ver, por ejemplo que se están ejecutando 4 consultas en la página actual, que podríamos verlas si pulsamos sobre el botón “Queries 4”



Si volvemos a la página de Foros y vemos las consultas que se están ejecutando, podemos ver que se está ejecutando una consulta por cada uno de los Foros que estamos paginando simultáneamente.



Esto significa que si paginamos 10 foros por página nos haría 10 consultas para mostrarlos, lo cual no es eficiente (puedes comprobarlo modificando el archivo “ForumController”).

Esto ocurre porque estamos añadiendo a la información de cada Foro el número de Post que contienen:

{{ __("Posts") }}: {{ $forum->posts->count() }}

Puedes comprobar cómo, si quitamos dicha línea del archivo “index.blade.php”, no se ejecutarían dichas consultas.

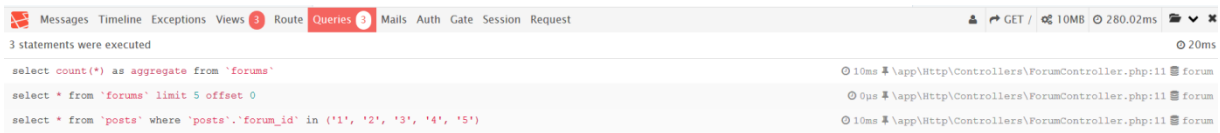
Para solucionar dicho problema, iremos al archivo “ForumController” y sustituiremos la línea siguiente:

$forums = Forum::latest()->paginate(5);

por esta otra:

$forums = Forum::with(['posts'])->paginate(5);

Si volvemos a actualizar el navegador, podemos ver como ahora sólo ejecuta 1 consulta para mostrar todos los foros:



Como puedes ver, lo que ha hecho es incluir todos los “forum_id” en una cláusula “in”. Esto lo conseguimos, automáticamente, con el uso del método “with”.

15.- Desarrollando las respuestas de los Foros

Después de los Foros y los Posts, tenemos que desarrollar las Respuestas de los Posts. Para crear esta parte del programa vamos a desarrollar nuestra Migración, la Factoría, el Controlador y el Modelo para dichas Respuestas.

Para ello vamos a escribir en la línea de comandos lo siguiente:

php artisan make:model Reply -mcf

Esto creará el Modelo, la Migración, el Controlador y la Factoría para los Posts

Lo primero que haremos será modificar nuestro archivo de migración de la siguiente forma:

```
public function up()
{
    Schema::create('replies', function (Blueprint $table) {
        $table->engine = "InnoDB";     // Esto permite escribir Relaciones y Claves Foráneas
        $table->increments('id');
        $table->unsignedInteger('post_id');
        $table->foreign('post_id')->references('id')->on('posts');
        $table->unsignedInteger('user_id');
        $table->foreign('user_id')->references('id')->on('users');
        $table->text('reply');   // Respuesta del Post
        $table->timestamps();
    });
}
```

Con esas dos columnas que hemos creado, estamos especificando quien será el creador de la "Respuesta" y a que "Post" pertenece.

Para poder poblar la nueva tabla vamos a modificar la nueva Factoría "ModelFactory" como sigue:

```
$factory->define(App\Reply::class, function (Faker $faker) {
   return [
      'user_id' => \App\User::all()->random()->id,
      'post_id' => \App\Post::all()->random()->id,
      'reply' => $faker->paragraph,
   ];
});
```

Para poder usar la Factoría modificaremos el archivo “DatabaseSeeder.php” como sigue:

```
public function run()
{
   factory(\App\User::class, 50)->create();
   factory(\App\Forum::class, 20)->create();
   factory(\App\Post::class, 50)->create();
   factory(\App\Reply::class, 100)->create();
}
```

Una vez hecho esto, hay que ejecutar el Seeder mediante Artisan:

php artisan migrate:refresh --seed

Lo primero que vamos a hacer es, en la ventana donde mostramos los Foros, además de mostrar cuántos Posts hay para cada Foro, vamos a indicar cuántas Respuestas hay. Para ello debemos acceder desde el Modelo “Forum” al Modelo “Reply” a través del Modelo “Post”.

Vamos a configurar primero el Modelo “Reply”:

```
class Reply extends Model
{
   protected $table = 'replies';

   protected $fillable = [
      'user_id', 'post_id', 'reply',
   ];

   // Para poder acceder al Foro desde esta tabla crearemos un atributo extra
   protected  $appends = ['forum'];

   public function post(){
         return $this->belongsTo(Post::class, 'post_id');
   }

   public function autor(){
         return $this->belongsTo(User::class, 'user_id');
   }

   // Y aquí definimos el Atributo extra
   // Para hacer eso, la función debe comenzar por "get"
   // y finalizar por "Attribute" (y lo de en medio CamelCase)
   public function getForumAttribute() {
         return $this->post->forum;
   }
}
```

Desde aquí parece fácil, porque podemos acceder al Post y desde el Modelo “Post” se puede acceder a su Foro (mediante la función “forum”), pero realmente, si nos fijamos en el Modelo “Forum”, ¿cómo lo hacemos?, ¿accedemos a todos los Post y accedemos a un Foro cualquiera? o ¿tenemos que recorrer todos los Posts para coger el Foro al que pertenece cada uno de ellos?

Para resolver esto, existe una solución muy sencilla que es definir en el Modelo “Forum” una función que llamaremos “replies” la cuál indicará que tenemos una relación “a distancia”, especificando cuál será la Clase que estará más lejos (“Reply”) y a través de qué clase vamos a enlazar con ella (“Post”):

```
public function posts(){
        return $this->hasMany(Post::class);
}

public function replies(){
        return $this->hasManyThrough(Reply::class, Post::class);
}
```

Para mostrar dichos datos, vamos a modificar la Vista “index.blade.php” añadiendo lo siguiente:

```
{{ __("Posts") }}: {{ $forum->posts->count() }},
{{ __("Respuestas") }}: {{ $forum->replies->count() }}
```

Puedes observar cómo, de nuevo, se nos han disparado el número de consultas que se han ejecutado. Para solucionar dicho problema, iremos al archivo “ForumController” y sustituiremos la línea siguiente:

```
$forums = Forum::with(['posts'])->paginate(5);
```

por esta otra:

```
$forums = Forum::with(['replies', 'posts'])->paginate(5);
```

16.- Mostrando las Respuestas

Lo siguiente que haremos es ir al Modelo “Post” y definir la relación que tiene un Post con las posibles Respuestas que puede tener. Para ello añadimos lo siguiente para indicar que un post puede tener muchas respuestas:

```
public function owner(){
        return $this->belongsTo(User::class, 'user_id');
}

public function replies(){
        return $this->hasMany(Reply::class);
}
```

Ahora vamos a abrir el fichero “web.php” para crear la ruta que nos mostrará las Respuestas para un Post concreto:

```
Route::get('/posts/{post}', 'PostController@show');
```

Lo siguiente que haremos es modificar el Controlador “PostController” para crear el método “show” al que estamos llamando:

```
use App\Post;

class PostController extends Controller
{
        public function show(Post $post)  // Con esto estamos inyectando el Post completo
        {
                $replies = $post->replies()->with('autor')->paginate(2);

                return view('posts.detail', compact('post','replies'));
        }
}
```

Ahora tendremos que crear la Vista “posts/detail.blade.php” con el siguiente código:

```
@extends('layouts.app')

@section('content')

<div class="row">
  <div class="col-md-8 col-md-offset-2">
      <h1 class="text-center text-mute"> {{ __("Respuestas al debate :name", ['name' => $post->title]) }} </h1>

      @forelse($replies as $reply)

        <div class="panel panel-default">
          <div class="panel-heading">
            <p>{{ __("Respuesta de") }}: {{ $reply->autor->name }}</p>
          </div>

          <div class="panel-body">
            {{ $reply->reply }}
          </div>
        </div>

      @empty

            <div class="alert alert-danger">
                {{ __("No hay ninguna respuesta en este momento") }}
            </div>

      @endforelse

        @if($replies->count())
            {{ $replies->links() }}
        @endif

  </div>
</div>

@endsection
```

Para poder diferenciar un poco las distintas páginas (Foros, Posts y Respuestas) vamos a aplicar algunos estilos a los headers de las Vistas. Vamos a crear un nuevo archivo .css para dichos estilos y lo llamaremos "styles.css", el cual contendrá el siguiente código:

```
.panel-default>.panel-heading-forum {
   color: #fff;
   background: #F4645F;
   border-color: #ddd;
}

.panel-heading-forum a{
   color: #fff;
   text-decoration: underline;
}

.panel-default>.panel-heading-post {
   color: #fff;
   background: #1f648b;
   border-color: #ddd;
}

.panel-heading-post a{
   color: #fff;
   text-decoration: underline;
}

.panel-default>.panel-heading-reply {
   color: #fff;
   background: #2F3133;
   border-color: #111;
}

.panel-heading-reply a{
   color: #fff;
   text-decoration: underline;
}
```

Ahora tenemos que añadir, en el layout principal "layouts/app.blade.php", dicha hoja de estilos:

```
<link href="{{ asset('css/app.css') }}" rel="stylesheet">
<link href="{{ asset('css/styles.css') }}" rel="stylesheet">
```

Y ahora tendremos que ir modificando tolas las vistas. Empezamos por "forums/index.blade.php":

```
<div class="panel-heading panel-heading-forum">
```

Ahora modificamos la vista “forums/detail.blade.php”:

```
<div class="panel-heading panel-heading-post">
```

Y, por último, vamos a modificar la vista “posts/detail.blade.php”:

```
<div class="panel-heading panel-heading-reply">
```

Vamos ahora a mostrar información adicional en la Vista de “Respuestas”. Para ello mostraremos quién es el autor de la respuesta y añadiremos un enlace para volver a la Vista de Posts. Para ello vamos al archivo “posts/detail.blade.php” y añadimos:

```
<h1 class="text-center text-mute"> {{ __("Respuestas al debate :name", ['name' => $post->title]) }} </h1>

<h4>{{ __("Autor del debate") }}: {{ $post->owner->name }}</h4>

<a href="../forums/{{ $post->forum->id }}" class="btn btn-info pull-right">
        {{ __ ("Volver al foro :name",  ['name' => $post->forum->name]) }}
</a>

<div class="clearfix"></div>

<br/>
```

Y por último, vamos a hacer lo mismo en la Vista de “Posts” para indicar a qué Foro pertenecen. Para ello editamos el fichero “forums/detail.blade.php” y vamos a modificar la siguiente línea:

```
<h1 class="text-center text-mute"> {{ __("Posts") }} </h1>
```

por estas otras:

```
<h1 class="text-center text-muted">
        {{ __("Posts del foro :name", ['name' => $forum->name]) }}
</h1>

<a href="/forum/public" class="btn btn-info pull-right">
        {{ __("Volver al listado de los foros") }}
</a>

<div class="clearfix"></div>

<br/>
```

17.- Dar de Alta un Foro nuevo

Lo primero que vamos a hacer es modificar el número de foros que vamos a paginar cada vez (en lugar de 5 como está establecido). Para ello vamos al Controlador "ForumController.php" y modificamos la siguiente línea:

```
$forums = Forum::with(['replies', 'posts'])->paginate(2);
```

En la pantalla principal de los Foros, lo que vamos a hacer es, después de la lista de foros paginados, vamos a mostrar un formulario que nos permita añadir nuevos Foros. Para hacer esto vamos a modificar el archivo "forums/index.blade.php" añadiendo el siguiente código que muestra un formulario:

```
@if($forums->count())
        {{ $forums->links() }}
@endif

<form method="POST" action="forums">
        {{ csrf_field() }}
        <div class="form-group">
                <label for="name" class="col-md-12 control-label">
                        {{ __("Nombre") }}
                </label>
                <input id="name" class="form-control" name="name" value="{{ old('name') }}"/>
        </div>
        <div class="form-group">
                <label for="description" class="col-md-12 control-label">
                        {{ __("Descripción") }}
                </label>
                <input id="description" class="form-control" name="description" value="{{
old('description') }}"/>
        </div>
        <button type="submit" name="addForum" class="btn btn-default">
                {{ __("Añadir Foro") }}
        </button>
</form>

</div>
</div>
```

A continuación tendremos que crear la ruta en el archivo "web.php" que nos permita mandar los datos por POST al "/forums":

```
Route::post('/forums', 'ForumController@store');
```

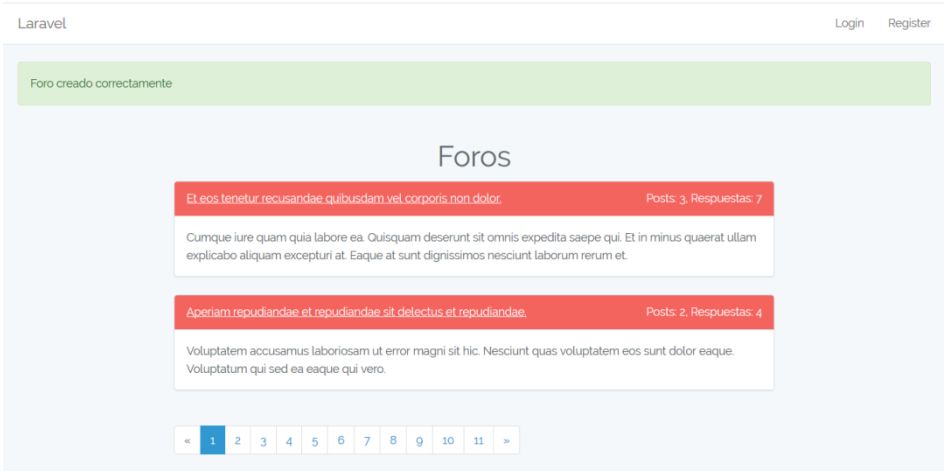
Y ahora tendremos que crear el método “store()” en el Controlador “ForumController”:

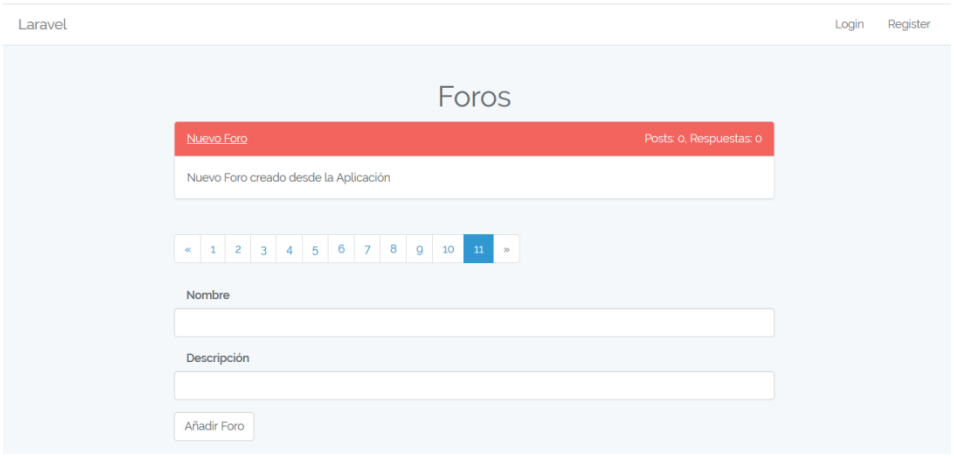
```
public function store()
{
        Forum::create(request()->all());
        // La siguiente línea nos devuelve a la url anterior (si es que existe), o a la raíz
        // y manda un mensaje, mediante una sesión flash, de éxito
        return back()->with('message', ['success', __("Foro creado correctamente")]);
}
```

Para mostrar la sesión flash que le estamos mandando, tendremos que ir al archivo “app.blade.php” y añadir lo siguiente:

```
<div class="container">

  @if(session('message'))
    <div class="alert alert-{{ session('message')[0] }}">
      {{ session('message')[1] }}
    </div>
  @endif

  @yield('content')
</div>
```


laravel
a clean and classy framework for PHP web development.


Laravel
Login
Register
Foro creado correctamente
Foros
Et eos tenetur recusandae quibusdam vel corporis non dolor.
Posts: 3, Respuestas: 7
Cumque iure quam quia labore ea. Quisquam deserunt sit omnis expedita saepe qui. Et in minus quaerat ullam explicabo aliquam excepturi at. Eaque at sunt dignissimos nesciunt laborum rerum et.
Aperiam repudiandae et repudiandae sit delectus et repudiandae.
Posts: 2, Respuestas: 4
Voluptatem accusamus laboriosam ut error magni sit hic. Nesciunt quas voluptatem eos sunt dolor eaque. Voluptatum qui sed ea eaque qui vero.
« 1 2 3 4 5 6 7 8 9 10 11 »


Laravel
Login
Register
Foros
Nuevo Foro
Posts: 0, Respuestas: 0
Nuevo Foro creado desde la Aplicación
« 1 2 3 4 5 6 7 8 9 10 11 »
Nombre
Descripción
Añadir Foro

18.- Validación de los Formularios

Para validar los datos de los formularios vamos a utilizar el método “valídate()” de los Controladores, el cual acepta como primer parámetro un “request” y como segundo un array de reglas de validación. Para ello, en el método “store()” del Controlador “ForumController” vamos a añadir el siguiente código:

```
$this->validate(request(), [
        'name' => 'required|max:100|unique:forums', // forums es la tabla dónde debe ser único
        'description' => 'required|max:500',
]);

Forum::create(request()->all());
```

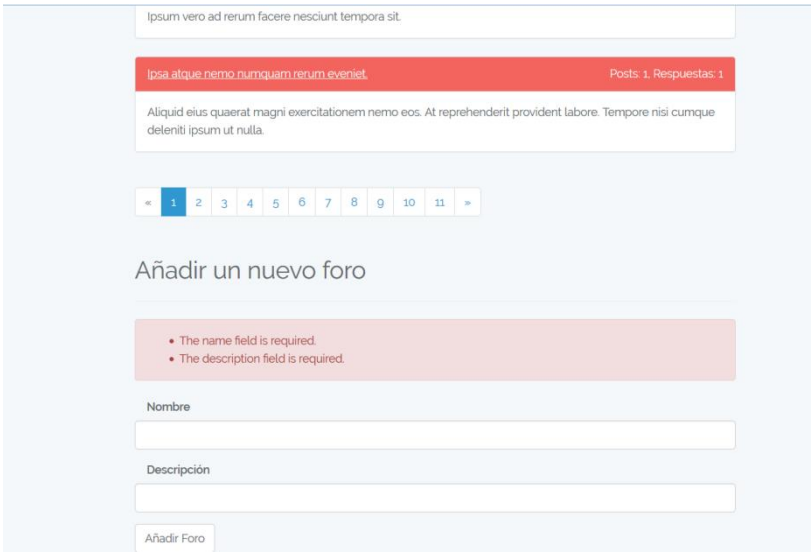
Para mostrar los posibles errores que puedan ocurrir en la validación vamos a crear una nueva Vista (en una nueva carpeta) llamada “partials/errors.blade.php” donde comprobaremos si existe algún error y crearemos una lista con los posibles errores. El código será el siguiente:

```
@if($errors->any())
  <div class="alert alert-danger">
    <ul>
      @foreach($errors->all() as $error)
        <li>{{ $error }}</li>
      @endforeach
    </ul>
  </div>
@endif
```

La variable “errors” que estamos comprobando vendrá del archivo donde está el formulario “forums/index.blade.php”, por eso será en este archivo donde tendremos que incluir el archivo creado anteriormente “partials/errors.blade.php”. El código necesario es el siguiente:

```
        @if($forums->count())
                {{ $forums->links() }}
        @endif

        <h2>{{ __("Añadir un nuevo foro") }}</h2>

        <hr />

        @include('partials.errors')
```

Como podemos ver, si ahora enviamos el formulario vacío, nos mostrará los mensajes de error que han ocurrido, pero si nos fijamos lo muestra en inglés:



Para poder traducir nuestra aplicación a Español, iremos al siguiente repositorio https://github.com/caouecs/Laravel-lang y en el apartado de "Install" podemos ver que debemos ejecutar la siguiente instrucción en la línea de comandos:

composer require caouecs/laravel-lang:~3.0

Una vez instalado, si vamos a la ruta "vendor/caouecs/laravel-lang/src" podemos ver como se han instalado las traducciones para 61 lenguajes. Simplemente tendremos que copiar la carpeta "es" en nuestra carpeta "resources/lang" y posteriormente, en el archivo "config/app.php" cambiaremos el valor de:

'locale' => 'en',

Por este otro:

'locale' => 'es',

Y como podemos ver ahora, el mensaje de error ha aparecido en español:

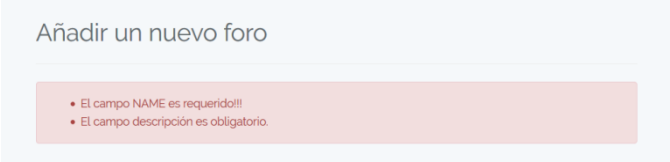
NOTA: En lugar de de instalar todos los lenguajes es posible descargar el archivo "ZIP" y copiar sólo la carpeta "es" en la ruta "resources/lang" de nuestro proyecto.

Observa que, tanto nombre como descripción aparecen directamente traducidos en el archivo "es/validation.php" porque son dos atributos de uso muy común. Si no fuera así, tendríamos que ir al final del archivo y añadir tanto el campo como la traducción al idioma.

Si quisiéramos poner un mensaje personalizado para el error, distinto a "El campo nombre es obligatorio" tendríamos que ir al Controlador y poner el siguiente código para el campo y error específicos:

```
$this->validate(request(), [
        'name' => 'required|max:100|unique:forums', // forums es la tabla dónde debe ser único
        'description' => 'required|max:500',
],
        [
                'name.required' => __("El campo NAME es requerido!!!")
        ]
);
```

Y como podemos ver el error aparece ahora personalizado:



19.- Usuarios autenticados

Aunque Laravel posee su propio módulo de Autenticación, para este ejercicio nosotros vamos a ver cómo manejar esto mediante los "Providers". En el archivo "app/Providers/AppServiceProvider.php" es dónde vamos a registrar las directivas nuevas para "Blade" (ejemplos de directivas en Blade son @if..@endif, @forelse…@endforelse, …).

Nosotros vamos a crear una directiva "if" específica para saber si un usuario está logueado. Dicha directiva se llamará "Logged" y la crearemos en la función "register" de dicho archivo añadiendo el siguiente código:

```
public function register() {
        \Blade::if('Logged', function() {
                // "auth" es el sistema de autenticación que estamos utilizando
                // y "check" nos dice si el usuario está o no autentificado
                return auth()->check();
        });
}
```

Para usar la nueva directiva iremos a la Vista donde mostramos los detalles de los foros "views/forums/detail.blade.php" y añadiremos el siguiente código:

```
@if($posts->count())
        {{ $posts-links() }}
@endif

@Logged()
        Está identificado
@else
        @include('partials.login_link', ['message' => __("Inicia sesión para crear un post")])
@endLogged
```

NOTA: La línea del "@include" sirve para, si no está logueado, crear un enlace a un nuevo archivo llamado "login_link.blade.php" que crearemos en la carpeta "partials" al cual pasaremos un mensaje.

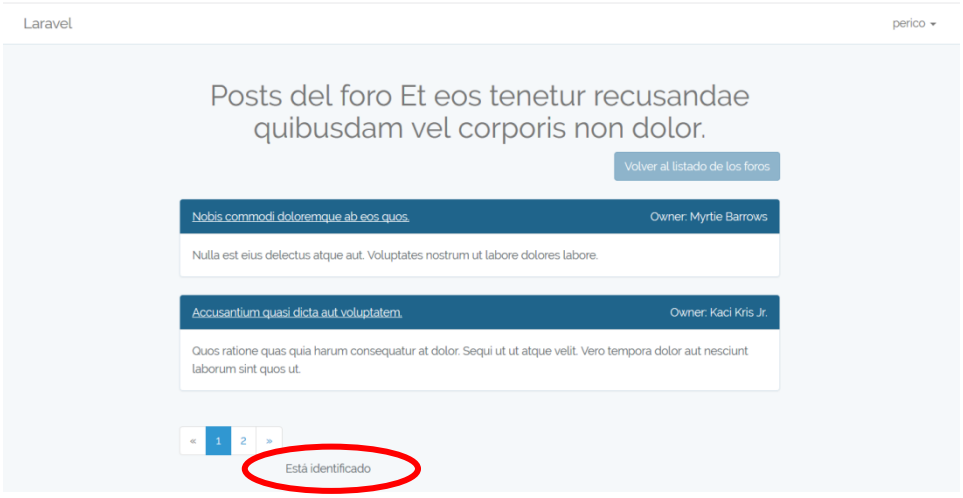
El archivo "login_link.blade.php" contendrá el siguiente código:

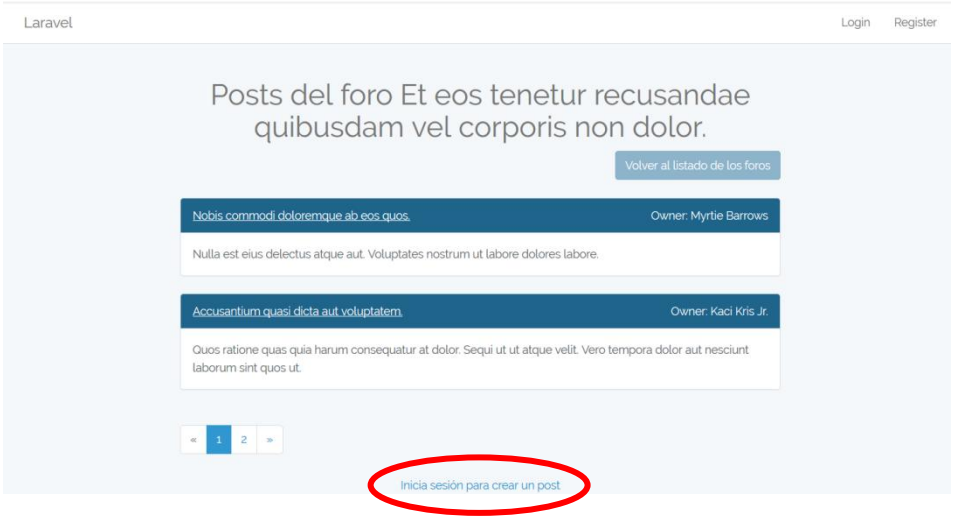
```
<div class="text-center">
        <a href="/login">{{ $message }}</a>
</div>
```

Para probarlo tendremos que registrar un usuario y entrar en los Posts que tiene un Foro cualquiera después de haber iniciado sesión con el usuario y hacerlo sin estar logueado.

NOTA: Para que no nos lleve a la Vista "home" cuando nos registramos o iniciamos sesión hay que:

- Eliminar del archivo "web.php" la ruta que redirecciona a "/home"
- Modificar, de los archivos "LoginController.php" y "RegisterController.php", la línea donde definimos la variable "$redirectTo" a "/home" por "/"





Lo siguiente que tenemos que hacer es crear el formulario que nos permitirá crear nuevos Posts para el Foro en el que nos encontremos.

20.- Formulario para crear nuevos Posts

Lo primero que vamos a hacer es cambiar el mensaje del archivo “views/forums/detail.blade.php” que dice “Está identificado” por otro más personalizado e incluir el archivo de errores por si ocurre alguno mostrar el mensaje:

```
@Logged()
        <h3 class="text-muted">{{ __("Añadir un nuevo post al foro :name", ['name' => $forum->name]) }}</h3>
        @include('partials.errors')
@else
```

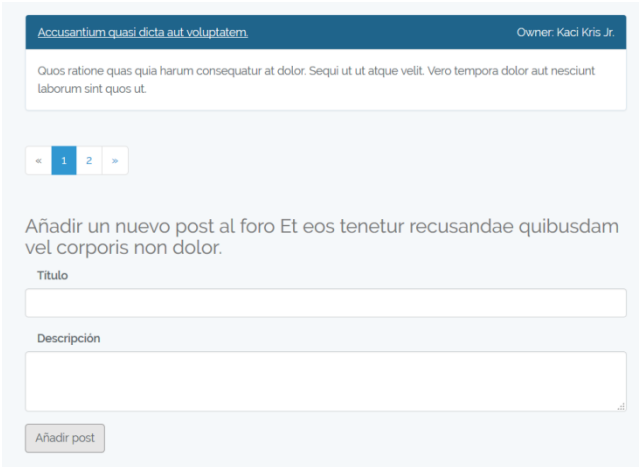
Si actualizamos el navegador podemos ver el nuevo mensaje:

Para darle un poco de espacio al mensaje con la parte inferior de la ventana, vamos a ir al archivo “app.blade.php” y vamos a modificar el siguiente código:

```
<div class="container" style="padding-bottom: 100px">
```

Vamos a seguir modificando el archivo “detail.blade.php” para incluir el nuevo formulario que nos permitirá crear los nuevos Posts:

```
@include('partials.errors')

<form method="POST" action="../posts">
  {{ csrf_field() }}
  <input type="hidden" name="forum_id" value="{{ $forum->id }}"/>

  <div class="form-group">
    <label for="title" class="col-md-12 control-label">{{ __("Título") }}</label>
    <input id="title" class="form-control" name="title" value="{{ old('title') }}"/>
  </div>

  <div class="form-group">
    <label for="description" class="col-md-12 control-label">{{ __("Descripción") }}</label>
    <textarea id="description" class="form-control"
        name="description">{{ old('description') }}</textarea>
  </div>

  <button type="submit" name="addPost" class="btn btn-default">{{ __("Añadir post") }}</button>
</form>

@else
```



Lo siguiente que haremos será crear la ruta en el archivo “web.php”:

```
Route::post('/posts', 'PostController@store');
```

Y a continuación lo que haremos será crear el método “store()” dentro del archivo “PostController.php” que se encargará de guardar el nuevo Post creado en la base de datos.

En este caso vamos a usar un nuevo método de validación de formularios utilizando lo que se llaman los FormRequests. De esta forma, el Controlador deja de tener la responsabilidad de validar los datos del formulario pasando esta al FormRequest.

Para crear un FormRequest iremos a la línea de comandos y ejecutaremos la siguiente función:

php artisan make:request PostRequest

Esto creará un archive llamado “PostRequest.php” dentro de la ruta “Http\Requests”. Lo primero que tendremos que cambiar del archivo es el método “authorize()” para que sólo lo puedan ejecutar los usuarios autenticados. Con “auth()->check()” lo que hacemos es que si el usuario está autenticado devolverá “true” y se ejecutará el Request, y si no lo está, devolverá “false” y no se ejecutará el Request:

```
public function authorize()
{
   return auth()->check();
}
```

Lo siguiente que haremos es modificar el método “rules()” para especificar las reglas de validación:

```
public function rules()
{
   return [
      'forum_id' => 'required|exists:forums,id', // con esto garantizamos que exista el “id” dentro de la tabla “forums”
      'title' => 'required|unique:posts|max:100',
      'description' => 'required',
   ];
}
```

NOTA: Es posible usar otro método llamado “messages()” que nos permite devolver un array de mensajes cuando existan errores de validación en el formulario (igual que hicimos cuando validamos en el método “store()” del Controlador “ForumController”).

Para usar el FormRequest iremos al archivo “PostController.php” y codificaremos el método “store()” del mismo de la siguiente forma:

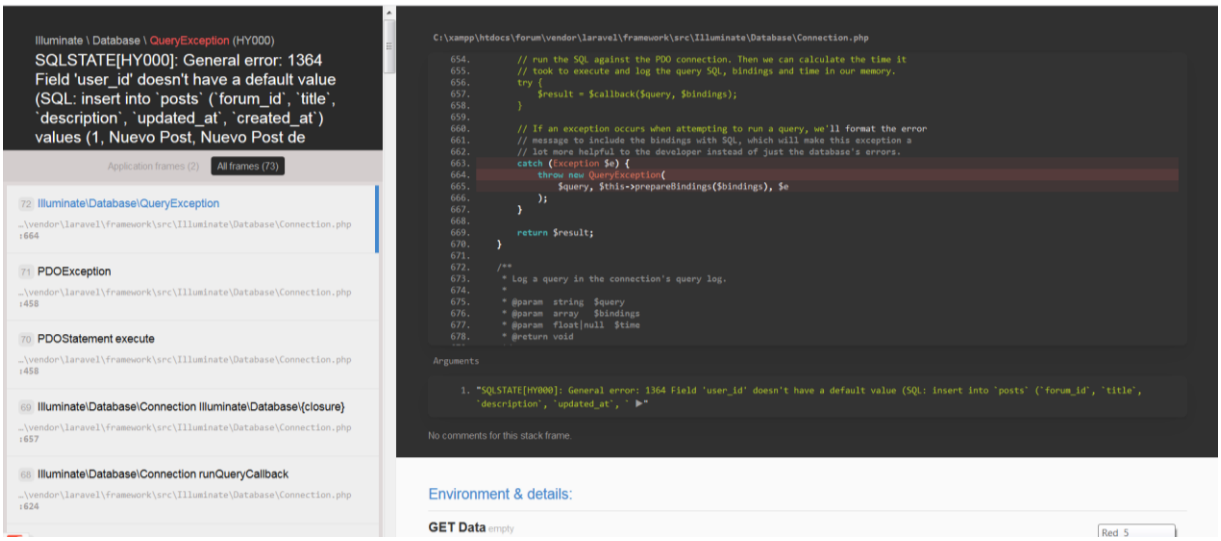
```
use App\Http\Requests\PostRequest;

public function store(PostRequest $post_request) {

}
```

Tan sólo con estas líneas podemos ver como si enviamos el formulario se validará y mostrará los errores que existan en el formulario.

Ahora sólo nos queda modificar el contenido del método “store()” para que inserte los datos del nuevo Post en la base de datos y nos muestre un mensaje informando que se ha realizado correctamente:

```
public function store(PostRequest $post_request) {
    Post::create($post_request->input()); // Esto coge todos los datos que vienen vía Post y los inserta
    return back()->with('message', ['success', __('Post creado correctamente')]);
}
```

Si enviamos los datos del nuevo Post, nos dará un error diciendo que el campo “user_id” no tiene ningún valor:



Para solucionar este problema podemos hacerlo de varias formas:

1. Insertando en el PostRequest un nuevo par de clave-valor con el id del usuario que ha iniciado sesión

```
$post_request->merge(['user_id' => auth()->id()]);
Post::create($post_request->input());
```

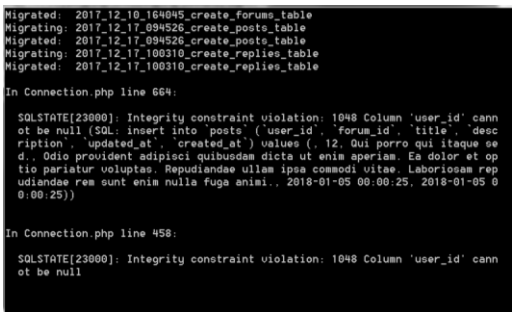
2. Otra forma más elegante de hacerlo es usando el método “boot()” dentro del Modelo “Post”. Este método nos sirve para codificar los Eventos del Modelo, es decir, que haremos cuando se está creando, cuando se ha creado y cuando se está borrando

```
protected $fillable = [
    'forum_id', 'user_id', 'title', 'description',
];

protected static function boot() {
    parent::boot();

    static::creating(function($post) {
        $post->user_id = auth()->id();
    });
}
```

Todo esto sólo tiene un problema, que si vamos a la línea de comandos y ejecutamos lo siguiente:

php artisan migrate:fresh --seed

se ejecutará también el método “boot()” y nos mostrará un error diciéndonos que la variable “user_id” no puede ser Null, ya que no hemos iniciado sesión desde la línea de comandos:



Para solucionar este error, simplemente tendremos que indicar en el método “boot()” que haga eso si no se está ejecutando desde la línea de comandos:

```
use Illuminate\Support\Facades\App;

    static::creating(function($post) {
        if( ! App::runningInConsole() ) {
            $post->user_id = auth()->id();
        }
    });
```

21.- Formulario para crear nuevas Respuestas (Mejorando las Respuestas)

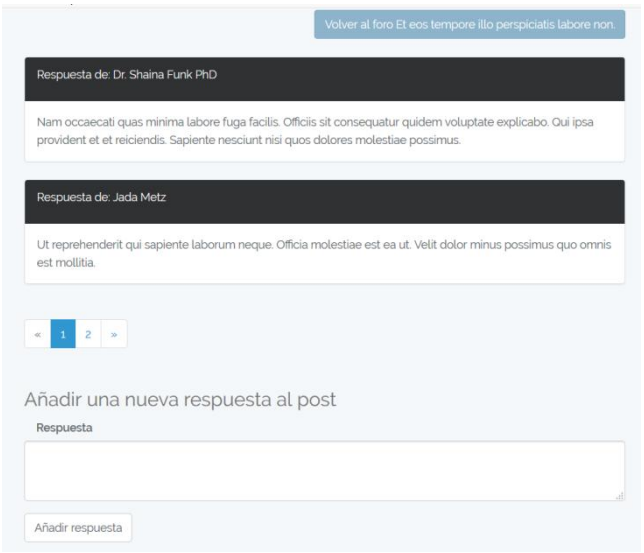
Lo siguiente que vamos a hacer es crear, en la vista de respuestas de los Posts, un formulario para que, los usuarios logueados, puedan crear nuevas Respuestas.

Lo primero que haremos es, en el archivo "posts/detail.blade.php" usaremos la nueva directiva "Logged" que creamos anteriormente:

```
    @Logged()
      <h3 class="text-muted">{{ __("Añadir una nueva respuesta al post :name", ['name' => $post->name]) }}</h3>
      @include('partials.errors')

      <form method="POST" action="../replies">
        {{ csrf_field() }}
        <input type="hidden" name="post_id" value="{{ $post->id }}" />

        <div class="form-group">
          <label for="reply" class="col-md-12 control-label">{{ __("Respuesta") }}</label>
          <textarea id="reply" class="form-control" name="reply">{{ old('reply') }}</textarea>
        </div>

        <button type="submit" name="addReply" class="btn btn-default">{{ __("Añadir respuesta") }}</button>
      </form>
    @else
      @include('partials.login_link', ['message' => __("Inicia sesión para responder")])
    @endLogged()
```

Podemos ver cómo, en caso de no estar logueado, nos muestra el mensaje de iniciar la sesión:

Pero si hemos iniciado sesión, nos mostraría el formulario para añadir una respuesta:



Igual que anteriormente, lo siguiente será definir la ruta a cual nos lleva el formulario:

```
Route::post('/replies', 'ReplyController@store');
```

Antes de crear el método “store()” en el Controlador, vamos a ver cómo usar un nuevo método de Validación (disponible desde Laravel 5.5) que son las “Rules”. Para crear una nueva “Rule” tendremos que ir a la línea de comandos y escribir:

```
php artisan make:rule ValidReply
```

Esto creará una nueva carpeta y dentro una Clase llamada “app/Rules/ValidReply.php”. Una vez creada, modificaremos el método “passes( )” con el siguiente código:

```
public function passes($attribute, $value)
{
   return strlen($value) > 10 && strlen($value) < 500;
}
```

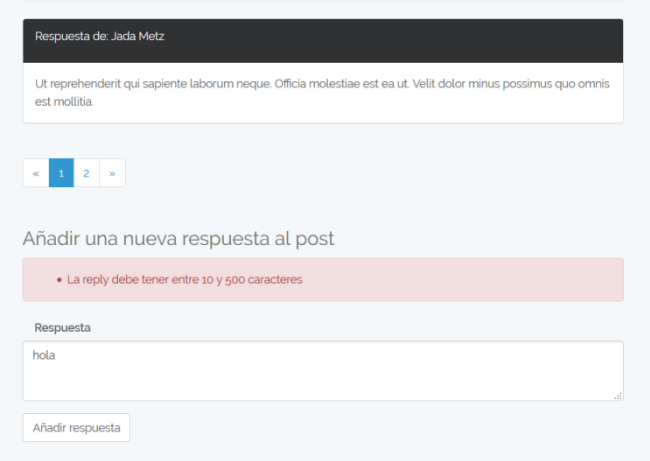
Y el método “message()” con el siguiente:

```
public function message()
{
   return __("La :attribute debe tener entre 10 y 500 caracteres");
}
```

Y posteriormente, tendremos que ir al Controlador “RepliesController” y escribir el código para el método “store()” que utilice la validación que acabamos de crear:

```
use App\Rules\ValidReply;

class ReplyController extends Controller
{
    public function store() {
        $this->validate(request(), [
            'reply' => ['required', new ValidReply]
        ]);
    }
```

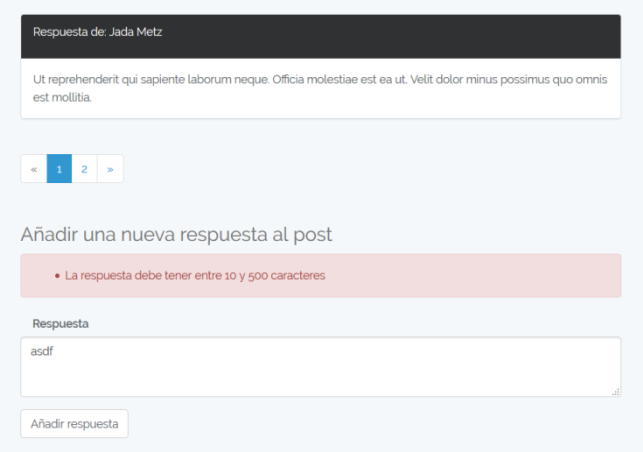
Podemos ver que si la Respuesta tiene menos de 10 caracteres muestra el error:



Como podemos ver, el mensaje es un poco extraño (La reply debe tener…). Para poder modificar dicho mensaje iremos al archivo “resources/lang/es/validation.php” y, en el array “atributes”, añadiremos un nuevo par “clave-valor” que será:

```
        'message'           => 'mensaje',
        'reply'             => 'respuesta',
    ],
```

Si enviamos de nuevo el formulario podemos ver que, ahora sí, el mensaje es correcto:



Lo siguiente sería crear la Respuesta en el Controlador "ReplyController" y mandar el mensaje de que la respuesta se ha creado correctamente:

```
use App\Rules\ValidReply;
use App\Reply;

class ReplyController extends Controller
{
    public function store() {
        $this->validate(request(), [
                'reply' => ['required', new ValidReply]
        ]);

        Reply::create(request()->input());

        return back()->with('message', ['success', __('Respuesta añadida correctamente')]);
    }
```

Si lo ejecutamos veremos que nos dará un mensaje de error, indicando que no existe el “user_id”. Para solucionarlo podemos copiar el código del método “boot()” desde el Modelo “Post.php” y pegarlo modificando “$post” por “$reply”:

```
use Illuminate\Database\Eloquent\Model;
use Illuminate\Support\Facades\App;
...
protected  $appends = ['forum'];

protected static function boot() {
    parent::boot();

    static::creating(function($reply) {
        if( ! App::runningInConsole() ) {
            $reply->user_id = auth()->id();
        }
    });
}
```

Podemos ver como si ahora creamos una nueva respuesta nos mostrará el mensaje de que se ha creado la respuesta correctamente:

22.- Urls amigables

Lo siguiente que vamos a hacer es modificar el código, para que, al acceder a un foro, post o respuesta concreta, en lugar de hacerlo mediante su código (p.e. "/forums/1") lo hagamos mediante "slugs", haciendo las URLs más amigables. Para ello tendremos que modificar, en primer lugar, las Migraciones para añadir a las tablas un nuevo campo que llamaremos "slug". Vamos a modificar primero el archivo de las migraciones de los Posts añadiendo el siguiente código:

```
Schema::create('posts', function (Blueprint $table) {
    …
});

Schema::table('posts', function (Blueprint $table) {
    $table->string('slug');
});
```

Observa que lo que hacemos es usar el método "table" del "Schema" para crear nuevos campos. Esto se hace así cuando queremos mantener los datos anteriores que puedan existir en la Base de Datos creando un nuevo archivo de migraciones con el contenido actual y el nuevo método "table". En nuestro caso, como los datos no son reales vamos a incluir el nuevo campo dentro del método "create" y volver a realizar la migración. Como vamos a hacer búsquedas por este campo, también le diremos que se trata de un índice de la tabla:

```
public function up()
{
    Schema::create('posts', function (Blueprint $table) {
        $table->engine = "InnoDB";     // Esto permite escribir Relaciones y Claves Foráneas
        $table->increments('id');
        $table->unsignedInteger('user_id');    // Creamos la columna que hará de clave foránea
        $table->foreign('user_id')->references('id')->on('users');    // Definición de la clave foránea
        $table->unsignedInteger('forum_id');    // Creamos la columna que hará de clave foránea
        $table->foreign('forum_id')->references('id')->on('forums');    // Definición de la clave foránea
        $table->string('title');          // Título del Post
        $table->string('slug');
        $table->index('slug');
        $table->text('description');       // Descripción del Post
        $table->timestamps();
    });
}
```

Ahora tendríamos que hacer lo mismo en la migración de los Foros:

```
public function up()
{
    Schema::create('forums', function (Blueprint $table) {
        $table->engine = "InnoDB";     // Esto permite escribir Relaciones y Claves Foráneas
        $table->increments('id');
        $table->string('name');    // Nombre del Foro
        $table->string('slug');
        $table->index('slug');
        $table->text('description');    // Descripción del Foro
        $table->timestamps();
    });
}
```

Lo siguiente será modificar las Factorías, tanto de los Foros como de los Posts. Empezaremos por la de los Foros:

```
$factory->define(App\Forum::class, function (Faker $faker) {
    $name = $faker->sentence;
    return [
        "name" => $name,
        'slug' => str_slug($name, '-'),      // str_slug hace algo parecido a la función “Split” (el 2º parámetro es el separador)
        "description" => $faker->paragraph
    ];
});
```

Hacemos lo mismo ahora con la Factoría de Posts:

```
$factory->define(App\Post::class, function (Faker $faker) {
    $title = $faker->sentence;
    return [
        'user_id' => \App\User::all()->random()->id,
        'forum_id' => \App\Forum::all()->random()->id,
        'title' => $title,
        'slug' => str_slug($title, '-'),
        "description" => $faker->paragraph,
    ];
});
```

En los modelos de Posts y de Foros tendremos que decirles que hay un nuevo campo que se llama "slug".

Primero en los Posts "Post.php":

```
protected $fillable = [
    'forum_id', 'user_id', 'title', 'description', 'slug',
];
```

Y luego en los Foros "Forum.php":

```
protected $fillable = [
    'name', 'description', 'slug',
];
```

Una vez hecho esto, vamos a ir a la terminal y ejecutaremos el comando:

php artisan migrate:fresh --seed

Podemos ver en la Base de Datos como en las tablas de Posts y de Foros ha creado un nuevo campo llamado "slug". Por último tendremos que decirle a Laravel que la clave para las rutas de los Posts y de los Foros no será el "id", sino el "slug". Para hacer esto, en cada uno de los modelos definiremos la función "getRouteKeyName()".

Primero en el Modelo Forum:

```
public function getRouteKeyName() {
    return 'slug';
}

public function posts() {
    return $this->hasMany(Post::class);
}
```

Y luego en el modelo Post:

```
public function getRouteKeyName() {
    return 'slug';
}

public function forum(){
    return $this->belongsTo(Forum::class, 'forum_id');
}
```

Si ahora volvemos a la aplicación e intentamos entrar en un foro, nos dará el error porque seguirá intentando entrar mediante el "id" del Foro. Para solucionar este problema tendremos que cambiar, en todas las vistas donde hagamos referencia al campo "id" y referenciar al campo "slug":

1- Vista "forums/index.blade.php" (línea 11):

```
<a href="forums/{{ $forum->slug }}"> {{ $forum->name }} </a>
```

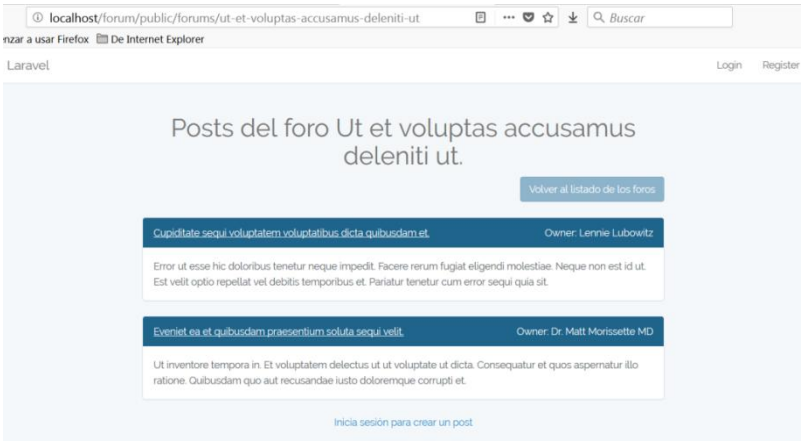
2- Vista "forums/detail.blade.php" (línea 23):

```
<a href="../posts/{{ $post->slug }}"> {{ $post->title }} </a>
```

3- Vista "posts/detail.blade.php" (línea 11):

```
<a href="../forums/{{ $post->forum->slug }}" class="btn btn-info pull-right">
```

Ahora sí, si volvemos a probar la aplicación entrando en los Foros y volviendo hacia atrás veremos que funcionarán nuestras nuevas rutas "amigables":



Hay que modificar el Modelo "Forum" para que no de error al crear un nuevo Foro

```
use Illuminate\Support\Facades\App;
class Forum extends Model
{
   ...
   protected static function boot() {
      parent::boot();

      static::creating(function($forum) {
         if( ! App::runningInConsole() ) {
            $forum->slug = str_slug($forum->name , "-");
         }
      });
   }
```

23.- Añadir Imágenes a los Posts

Lo que vamos a ver en este capítulo es ver cómo podemos añadir una imagen a nuestros Posts. En un principio lo que haremos será rellenar la tabla, mediante la "Factory", con imágenes reales obtenidas de una Web. Para ello vamos a usar lo que se denomina en Laravel "Storage". Es posible hacerlo de una forma más sencilla guardándolo en el directorio "public", pero "Storage" nos va a permitir guardarlo en cualquier sitio y después mostrarlo.

Si abrimos el fichero "config/filesystems.php" podemos ver un array denominado "disks" (discos) donde podemos ver los que se incluyen por defecto: "local" (para guardar en 'app'), "public" que es de ámbito público (para guardar en 'app/public') y "s3" (que podría ser un almacenamiento de 'Amazon').

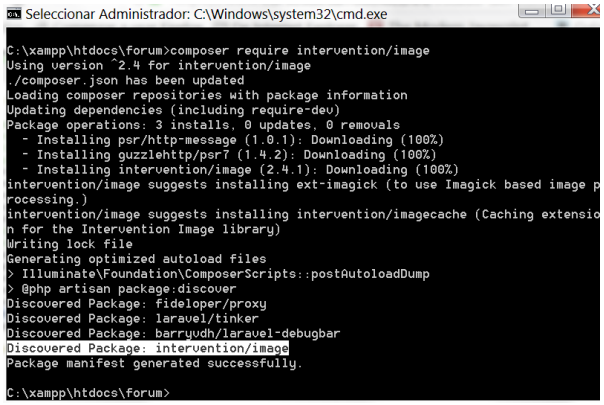
Cuando guardemos mediante Storage, simplemente pondremos una línea como la siguiente:

```
\Illuminate\Support\Facades\Storage::disk('s3')->storage();
```

Con esa línea guardaríamos en "s3" y simplemente cambiando ese valor podríamos guardar en cualquiera de los almacenamientos que tengamos configurados.

Para poder usar esto, lo primero que hay que hacer es añadir a nuestra aplicación el paquete "Image Intervention" (image.intervention.io) . Para ello iremos a la consola de comandos y ejecutaremos lo siguiente:

```
composer require intervention/image
```



Como podemos ver, se ha descubierto el Paquete "intervention/image", con lo cual no hay que realizar los pasos que vienen en la web donde dicen que hay que añadir las líneas en "config/app.php" en los arrays "$providers" y en "$aliases" (esto ocurre con la instalación de muchos paquetes a partir de la versión 5.5 de Laravel).

Una vez hecho esto, como vamos a guardar las imágenes en los Posts, tendremos que modificar la Migración de "Posts" y añadir el nuevo campo de texto que podrá ser nulo (ya que todos los Post no tendrán que tener imagen necesariamente) y que llamaremos "attachment":

```
$table->text('description');      // Descripción del Post
$table->string('attachment')->nullable();
$table->timestamps();
```

Tendremos también que modificar el Modelo Post para decirle que hay un nuevo campo que se llama "attachment".

```
protected $fillable = [
        'forum_id', 'user_id', 'title', 'description', 'slug', 'attachment',
];
```

Lo siguiente que haremos es ir a la Factoría de Posts para añadirle la línea que rellenará, de forma aleatoria, las imágenes reales de los Posts obteniéndolas de la web www.lorempixel.com:

```
"description" => $faker->paragraph,
// Los argumentos son:
// 1: Ruta -> Concatenamos la ruta de Storage con la ruta interna que crearemos
// 2: Ancho  // 3: Alto //  4: Categoría de la imagen
// 5: Si queremos guardar la Ruta Completa o sólo el nombre y la extensión del archivo
'attachment'  =>  \Faker\Provider\Image::image(storage_path()  .  '\app\posts',  200,  200,
'technics', False),
```

Como la ruta no existe tendremos que crearla sobre la ruta de "Storage" (storage/app/posts).

Lo siguiente será crear nuestro nuevo disco en "config/filesystems.php". Para ello copiaremos la configuración del disco "public" y la pegaremos debajo modificando el código de la siguiente forma:

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

Por último iremos a la terminal y ejecutaremos la siguiente instrucción:

```
php artisan migrate:fresh --seed
```

Podemos ver tanto en el directorio "storage/app/posts" como en la Base de Datos como se han creado para los post las imágenes correspondientes, creando los archivos en la carpeta y escribiendo su nombre en la BD.

Lo siguiente que habría que hacer es mostrar las imágenes cuando mostremos los Posts.

24.- Mostrar las Imágenes de los Posts

Vamos a mostrar las imágenes en los Posts y lo haremos justo debajo del texto de la descripción del Post. Para ello tendremos que seguir los siguientes pasos:

1- Crearemos una ruta en el archivo “web.php” de la siguiente forma:

```
use Illuminate\Support\Facades\Storage;
use Intervention\Image\Facades\Image;

Route::get('/images/{path}/{attachment}', function ($path, $attachment){
        // Lo siguiente devuelve el Path absoluto de "Storage"
        $storagePath = Storage::disk($path)->getDriver()->getAdapter()->getPathPrefix();
        $imageFilePath = $storagePath . $attachment;

        if(File::exists($imageFilePath)) {
                return Image::make($imageFilePath)->response();
        }
});
```

2- En el Modelo “Post” vamos a crear un nuevo método llamado “pathAttachment” con el siguiente código:

```
public function pathAttachment(){
        return "/images/posts/" . $this->attachment;
}
```

Este método nos devolverá la ruta de la imagen en el formato que hemos especificado en “web.php”.

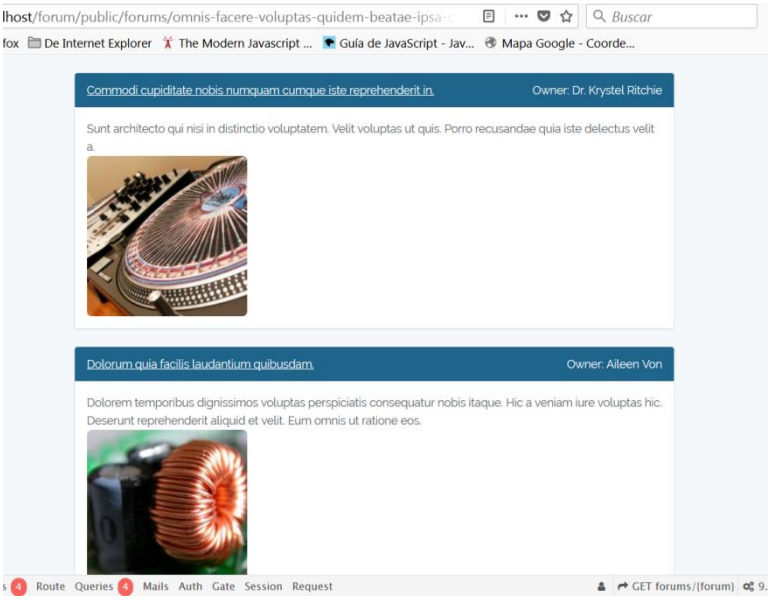
3- Ahora tenemos que ir al archivo de detalles de los Foros “forums/detail.blade.php” y modificar el código de la siguiente forma:

```
{{ $post->description }}

@if($post->attachment)
        <img src="{{ $post->pathAttachment() }}" class="img-responsive img-rounded" />
@endif
```

4- Antes de nada, saber que para que Laravel pueda mostrar las imágenes, debido a que en la Ruta que hicimos debe hacer una transformación para coger las imágenes, es necesario levantar el servidor que trae Laravel para que funcione y muestre dichas imágenes. Para eso debemos escribir en la línea de comandos:

php artisan serve

5- Como podemos ver, nos muestra en la consola la dirección a la que debemos acceder en el navegador para mostrar nuestra web (127.0.0.1:8000). Si la abrimos en el navegador ya aparecen las imágenes en cada uno de los Posts.



25.- Subir Imágenes validadas

Lo primero que vamos a hacer es repetir los pasos vistos en los puntos 23 y 24 para añadir y mostrar imágenes en las "Respuestas". Los pasos son los siguientes:

Para preparar la BD y llenarla de imágenes

1. Modificamos la Migración de "Replies" añadiendo el nuevo campo

```
$table->text('reply');   // Respuesta del Post
$table->string('attachment')->nullable();
$table->timestamps();
```

2. Modificamos la Factoría de "Replies"

```
'reply' => $faker->paragraph,
'attachment' => \Faker\Provider\Image::image(storage_path() . '\app\replies', 200, 200, 'animals', false),
```

3. Añadimos un nuevo disco en "config/filesystem.php" para las respuestas y creamos el directorio dentro de "storage/app"

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

4. Ejecutamos en la línea de comandos

php artisan migrate:fresh --seed

Para mostrar las imágenes en la vista de Respuestas

5. La ruta del archivo "Web.php" no es necesaria crearla ya que usaremos la misma

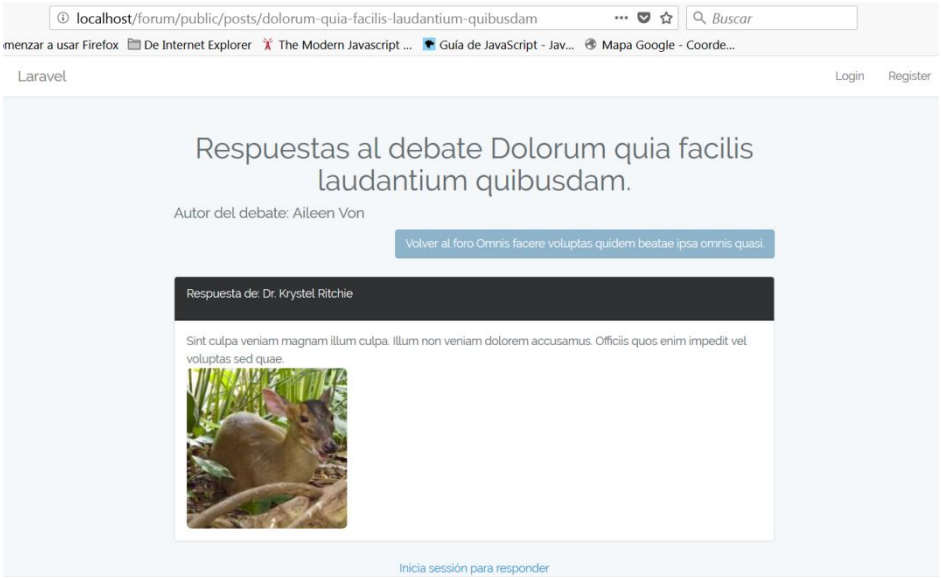
6. En el Modelo "Reply" creamos el método "pathAttachment()"

```
public function pathAttachment() {
        return "/images/replies/" . $this->attachment;
}
```

7. Modificamos el código de la Vista de detalles de los Posts “forums/detail.blade.php”

```
{{ $reply->reply }}

@if($post->attachment)
        <img src="{{ $reply->pathAttachment() }}" class="img-responsive img-rounded" />
@endif
```

Como podemos ver, ya se ven las imágenes también en las respuestas



Lo siguiente que haremos es añadir un nuevo botón, al formulario que nos permite crear un Post, que nos permita añadir (opcionalmente) una imagen a ese nuevo Post.

Para ello vamos a modificar la vista que muestra los Posts (“forums/detail.blade.php”) y vamos a modificar el código añadiendo, en el formulario, un campo de tipo “file”:

```
<label class="btn btn-warning" for="file">
        <input id="file" name="file" type="file" style="display:none;">
        {{ __("Subir archivo") }}
</label>

<button type="submit" name="addPost" class="btn btn-default">{{ __("Añadir post") }}</button>
```

Para poder subir archivos desde el formulario, hay que modificar la línea principal del formulario añadiéndole la propiedad “enctype”

```
<form method="POST" action="../posts" enctype="multipart/form-data">
```

A continuación iremos al FormRequest “PostRequest.php” y añadiremos una regla de validación para que sólo acepte ficheros de tipo “imagen”:

```
'description' => 'required',
'file' => 'image',
```

Como vamos a querer subir archivos de imagen desde aquí y desde las respuestas, vamos a crear un Helper que nos ayude a reutilizar la funcionalidad de las subidas de imágenes. Crearemos un archivo llamado “helpers.php” en la ruta “app/http” y dentro definiremos las función “uploadFile()” que tendrá dos argumentos: “key”: que será la clave que tendrá el archivo que vamos a mandar por Post (en nuestro caso “file”) y “path”: que hará referencia a la ruta dentro de “storage” dónde lo vamos a guardar:

```
<?php

function uploadFile($key, $path) {
        request()->file($key)->store($path);
        // Devolveremos el Hash del Name que es lo que se guarda
        return request()->file($key)->hashName();
}
```

Para utilizar esta función iremos al método “store()” del Controlador “PostController” y añadiremos las siguientes líneas de código que comprueban si el archivo existe y si es un archivo de imagen válido:

```
public function store(PostRequest $post_request) {
        if($post_request->hasFile('file') && $post_request->file('file')->isValid()) {
                $filename = uploadFile('file', 'posts');
                $post_request->merge(['attachment' => $filename]);
        }

        Post::create($post_request->input());
```

Además debemos decirle en el Modelo “Post” que cuando cree un nuevo Post cree un valor para el campo “slug”:

```
$post->user_id = auth()->id();
$post->slug = str_slug($post->title, "-");
```

Para que todo esto funcione, es necesario añadir el path de nuestro nuevo archivo “helpers.php” al archivo “composer.json”:

```
"autoload-dev": {
        "psr-4": {
                "Tests\\": "tests/"
        },
        "files": ["app/Http/helpers.php"]
},
```

Y autocargarlo desde la línea de comandos:

composer dump-autoload

Esto último hará que se carguen de nuevo todos los archivos, va a revisar Composer que hemos añadido un nuevo “Path”, va a registrar el archivo “helpers.php” y si lo probamos en el navegador nos debe de subir la imagen en el nuevo Post sin problemas.

26.- Subir Imágenes en las Respuestas

Igual que hicimos con los Posts, para poder adjuntar imágenes a las respuestas tenemos que seguir los siguientes pasos:

Añadimos un botón, al formulario que nos permite crear una Respuesta, que nos permita añadir (opcionalmente) una imagen a esa nueva Respuesta.

Para ello vamos a modificar la vista que muestra las Respuestas ("posts/detail.blade.php") y vamos a modificar el código añadiendo, en el formulario, un campo de tipo "file":

```
<label class="btn btn-warning" for="file">
        <input id="file" name="file" type="file" style="display:none;">
        {{ __("Subir archivo") }}
</label>

<button type="submit" name="addReply" class="btn btn-default">{{ __("Añadir Respuesta") }}</button>
```

Recuerda que para poder subir archivos desde el formulario, hay que modificar la línea principal del formulario añadiéndole la propiedad "enctype"

```
<form method="POST" action="../replies" enctype="multipart/form-data">
```

Para volver a utilizar el Helper que creamos anteriormente, iremos al método "store()" del Controlador "ReplyController" y añadiremos las siguientes líneas de código que comprueban si el archivo existe y si es un archivo de imagen válido:

```
public function store() {
        $this->validate(request(), [
                'reply' => ['required', new ValidReply],
                'file' => 'image'
        ]);

        if(request()->hasFile('file') && request()->file('file')->isValid()) {
                $filename = uploadFile('file', 'replies');
                request()->merge(['attachment' => $filename]);
        }

        Reply::create(request->input());
```

Vamos a modificar el Modelo Reply para decirle que hay un nuevo campo que se llama "attachment".

```
 protected $fillable = [
        'user_id', 'post_id', 'reply', 'attachment',
];
```

27.- Eliminar Posts, Respuestas y archivos adjuntos

Vamos a crear ahora la funcionalidad que nos va a permitir eliminar Posts, Respuestas y los archivos (imágenes) adjuntos que pueda tener, pero controlando que un usuario sólo pueda eliminar los que sean de su propiedad.

Lo primero que tendremos que hacer entonces, antes de mostrar el botón de eliminar, es comprobar si el Post pertenece al usuario que ha iniciado sesión. Para ello vamos a crear un nuevo método en el Modelo "Post" que llamaremos "isOwner()" con el siguiente código:

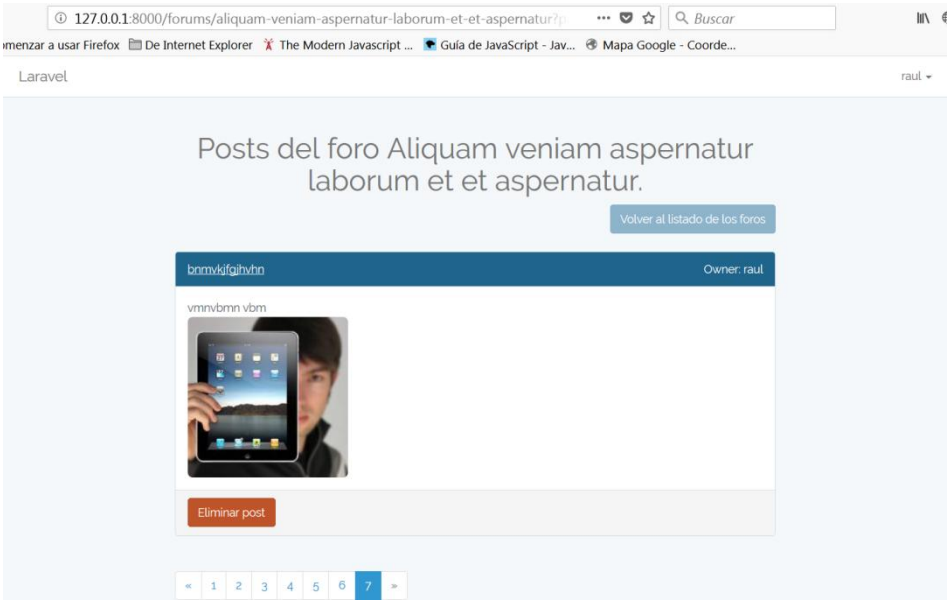
```
public function isOwner() {
        return $this->owner->id === auth()->id();
        // También es posible ponerlo de la siguiente forma
        // return $this->owner == auth()->user();
}
```

Esto nos devolverá "True" o "False" indicando si el propietario del Post es el mismo que el usuario que ha iniciado sesión.

Esta función la llamaremos desde la Vista de Post ("forums/detail.blade.php") y escribiremos el siguiente código:

```
<div class="panel-body">
        {{ $post->description }}

        @if($post->attachment)
                <img src="{{ $post->pathAttachment() }}" class="img-responsive img-
        rounded" />
        @endif
</div>

@if($post->isOwner())
        <div class="panel-footer">
                <form method="POST" action="../posts/{{ $post->slug }}">
                        // Es necesario enmascarar el método Delete en Laravel
                        {{ method_field('DELETE') }}
                        {{ csrf_field() }}
                        <button type="submit" name="deletePost" class="btn btn-danger">
                                {{ __("Eliminar post") }}
                        </button>
                </form>
        </div>
@endif
```

Como podemos ver, si nosotros somos los autores del Post, nos aparece el botón para eliminar dicho Post:



Lo siguiente será crear la nueva ruta que nos permita llamar al método del Controlador:

```
Route::post('/posts', 'PostController@store');
Route::delete('/posts/{post}', 'PostController@destroy');
```

Ahora tendremos que crear el método “destroy()” dentro del Controlador “PostController”:

```
public function destroy(Post $post) {
        if( ! $post->isOwner())
                abort(401);

        $post->delete();
        return back()->with('message', ['success', __('Post y respuestas eliminados correctamente')]);
}
```

Pero este código sólo eliminaría el Post si no tiene Respuestas asociadas. Para que elimine el Post y las Respuestas asociadas, podríamos hacerlo desde el mismo Controlador, pero es mejor dejarlo más “limpio” y hacerlo desde el Modelo “Post”. Para ello vamos a modificar la función “boot()” añadiendo un método “static” con el siguiente código:

```
use Illuminate\Support\Facades\Storage;
…
static::deleting(function($post) {
        if( ! App::runningInConsole() ) {
                if($post->replies()->count()) {
                        foreach($post->replies as $reply) {
                                if($reply->attachment) {
                                        Storage::delete('replies/' . $reply->attachment);
                                }
                        }
                        $post->replies()->delete();
                }

                if($post->attachment) {
                        Storage::delete('posts/' . $post->attachment);
                }
        }
});
```

28.- Eliminar Respuestas y adjuntos si somos Autores

Vamos a hacer lo mismo que hemos hecho con los Posts pero ahora con las Respuestas. Para ello vamos a ir al fichero “posts/detail.blade.php” y vamos a añadir un botón para borrar pero comprobando antes si el usuario que está logueado es el autor de dicha Respuesta. Para saber si es el autor, primero iremos al Modelo “Reply.php” y crearemos un nuevo método:

```
public function isAuthor() {
    return $this->autor->id === auth()->id();
    // También es posible ponerlo de la siguiente forma
    // return $this->autor == auth()->user();
}
```

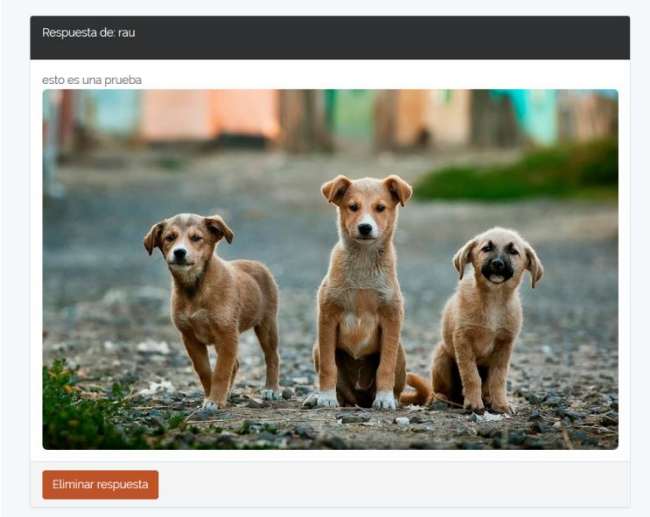
Una vez hecho esto, iremos a la vista “detail” y justo debajo del “panel-body” vamos a escribir el siguiente código:

```
<div class="panel-body">
  {{ $reply->reply }}

  @if($post->attachment)
    <img src="{{ $reply->pathAttachment() }}" class="img-responsive img-rounded" />
  @endif

</div>

@if($reply->isAuthor())
  <div class="panel-footer">
    <form method="POST" action="{{ route('replies.delete', [$reply->id]) }}">
      {{ method_field('DELETE') }}
      {{ csrf_field() }}
      <button type="submit" name="deleteReply" class="btn btn-danger">
        {{ __("Eliminar respuesta") }}
      </button>
    </form>
  </div>
@endif
```

Nota que en esta ocasión, al llamar al “action” del formulario estamos llamando a una ruta “replies.delete” enviándole un parámetro “$reply->id”. Para declarar dicha ruta iremos al archivo “web.php” y añadiremos el siguiente código:

```
Route::post('/replies', 'ReplyController@store');
Route::delete('/replies/{reply}', 'ReplyController@destroy')->name('replies.delete');
```

Observa como ahora si nos aparece el botón de “Eliminar Respuesta” en las respuestas que somos propietarios:



Nos queda aún crear el método “destroy()” dentro del Controlador “ReplyController”:

```
public function destroy(Reply $reply) {
        $reply->delete();
        return back()->with('message', ['success', __('Respuesta eliminada correctamente')]);
}
```

Y en el Modelo “Reply” tenemos que crear también el Evento “deleting()” justo debajo del Evento “creating()”:

```
use Illuminate\Support\Facades\Storage;
…
static::deleting(function($reply) {
        if( ! App::runningInConsole() ) {
                if($reply->attachment) {
                        Storage::delete('replies/' . $reply->attachment);
                }
        }
});
```

Si pulsamos el botón eliminar podemos ver como elimina la Respuesta mostrándonos el mensaje “Respuesta eliminada correctamente”.

29.- Evitar múltiples inicios de sesión con una cuenta

En esta sección veremos cómo podemos evitar que dos personas inicien sesión, al mismo tiempo, en nuestra aplicación usando una misma cuenta de usuario.

Para ello vamos a crear una nueva aplicación de Laravel y vamos a configurar nuestro archivo ".env" con los datos necesarios para acceder a la BD que usaremos (recuerda que la BD debes crearla también). Además iniciaremos la propiedad "APP_URL" de dicho archivo para poder enviar correos de recuperación de contraseña:

```
APP_URL = http://localhost:8000
```

Una vez hecho esto, iremos al archivo de migraciones de usuarios y crearemos un nuevo campo de tipo booleano que llamaremos "is_logged" y cuyo valor por defecto sea "false".

```
$table->string('password');
        $table->boolean('is_logged')->default(false);
```

Ahora iremos a la terminal y ejecutaremos las migraciones:

```
php artisan migrate
```

Una vez que tenemos nuestra base de datos preparada, crearemos el sistema de autenticación de Laravel:

```
php artisan make:auth
```

Lo siguiente será ir al layout "app.blade.php" y justo encima de "@yield('content')" vamos a escribir el siguiente código que mostrará un mensaje informativo si existe una variable de sesión llamada "message":

```
@if(session('message'))
   <div class="row justify-content-center">
      <div class="col-md-10">
         <div class="alert alert-{{ session('message')[0] }}">
            <h4 class="alert alert-heading">{{ __("Mensaje informativo") }}</h4>
            <p>{{ session('message')[1] }}</p>
         </div>
      </div>
   </div>
@endif
@yield('content')
```

Vamos a sobrescribir ahora el login, para que cuando un usuario inicie sesión, ponga el campo "is_logged" de la BD a "true". Para ello vamos al Controlador "LoginController.php" y como podemos ver hay un Trait denominado "AuthenticatesUsers":

```
use AuthenticatesUsers;
```

si vamos a la clase, podemos ver que hay varios métodos y entre ellos está el método "authenticated" que se ejecutará cuando el usuario se haya autenticado. Para utilizar este método no vamos a escribir código aquí, ya que estaremos escribiendo en el directorio "vendor" y esto no es una buena práctica, ya que si actualizamos las dependencias es posible que se sobrescriba la clase eliminando nuestro código. Por tanto lo que haremos será copiar el método completo y lo pegamos justo al final del "LoginController" (realizando el import del "Request"):

```
use Illuminate\Http\Request;
…

protected function authenticated(Request $request, $user)
{
   dd($user);     // Podemos ver que funciona cuando iniciamos sesión
}
```

Lo que vamos a hacer es, dentro del método, comprobar si el campo "is_logged" está a "false" (para ponerlo a "true" y permitir que se loguee) o si está a "true" (para no dejarlo que se loguee).

```
protected function authenticated(Request $request, $user)
{
   if($user->is_logged){
         $this->guard()->logout();         // Cerramos la sesión
         $request->session()->invalidate();  // Invalidamos la sesión
         session()->flash('message', ['danger', 'Ya hay un usuario logueado con esta cuenta']);
         return redirect('/login');
   }
   else{
      $user->is_logged = true;
      $user->save();
   }

   // Y enviamos al usuario a la página de "home" si ha iniciado sesión
   return redirect($this->redirectPath());
}
```

Para probarlo primero en la Base de Datos ponemos el campo "is_logged" a "0" y luego abriremos dos navegadores. Iniciaremos sesión en el primero (podemos comprobar cómo el campo "is_logged" estará a "1" en la Base de Datos) y si volvemos a iniciar sesión en el otro debería de mostrarnos el mensaje y llevarnos a la pantalla de login.

Para que funcione correctamente, tendremos que poner el valor “is_logged” al “0”, cuando el usuario haga logout en su cuenta. Para ello volveremos al Trait “AuthenticatesUsers” y buscaremos el método “logout”. Lo vamos a copiar completo y lo vamos a pegar al final de nuestro Controlador “LoginController”. Antes de cerrar la sesión, debemos recuperar las credenciales del usuario para poner el valor del campo a “0”:

```
use App\User;

public function logout(Request $request)
{
   $user = User::find(auth()->id());
   $user->is_logged = false;
   $user->save();

   $this->guard()->logout();

   $request->session()->invalidate();

   return $this->loggedOut($request) ?: redirect('/login');
}
```

NOTA: Recuerda que para probarlo hay que poner a “0” el campo en la Base de Datos manualmente.

Vamos a sobrescribir también el método “loggedOut” para mostrar un mensaje flash:

```
protected function loggedOut(Request $request)
{
   session()->flash('message', ['success', 'Has cerrado sesión correctamente']);
   return redirect('/login');
}
```

Aún no está terminada la aplicación. Si registramos un usuario podemos ver cómo, automáticamente, se inicia sesión con dicho usuario, pero en la Base de Datos el campo “is_logged” tiene el valor “0” en lugar de “1”. Vamos a escribir el código suficiente para que cuando el usuario se registre ponga dicho campo a “1”. Para ello, si vamos al Controlador “RegisterController”, vemos que se hace referencia al Trait “RegistersUsers”. Si abrimos la Clase podemos encontrar un método llamado “registered”. Igual que hicimos antes, vamos a copiar el método y a sobrescribirlo en “RegisterController” (recuerda importar la clase “Request”):

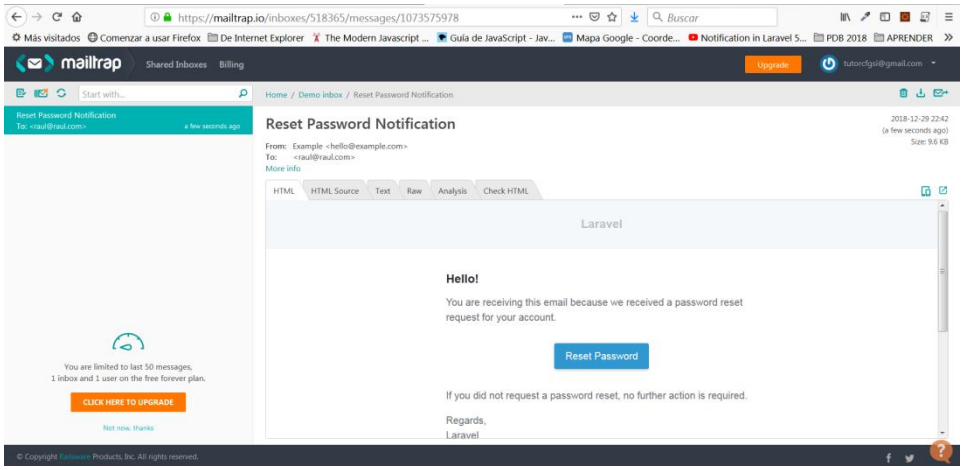
```
use Illuminate\Http\Request;

protected function registered(Request $request, $user)
{
   $user->is_logged = true;
   $user->save();

   return redirect($this->redirectPath());
}
```

Pasaría lo mismo cuando un usuario pulsa el enlace “Forgot your Password”, ya que se le enviará un email para restablecer la contraseña y cuando la modifique iniciará sesión de forma automática pero el campo no se pondrá a “1” en la Base de Datos.

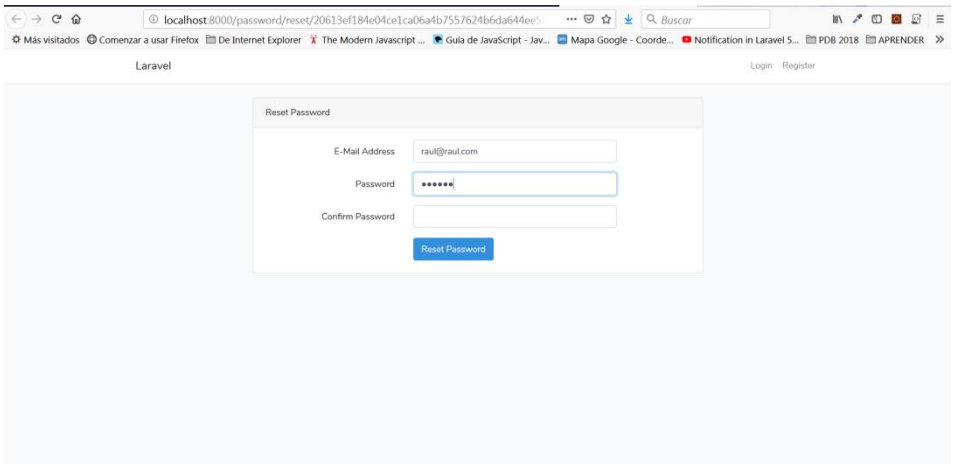
Para realizar esto vamos a usar el servidor de correos “mailtrap”. Para ello vamos a la web “mailtrap.io” y creamos una cuenta gratuita. Dentro del enlace “Demo inbox” podemos encontrar los valores de Username y Password que tendremos que utilizar en nuestro fichero “.env”.

NOTA: Recuerda que después de copiar las credenciales en el archivo “.env” hay que reiniciar el servidor.

Si pulsamos ahora en el enlace y ponemos el correo del usuario de la aplicación, podemos ver que nos llega un correo con un botón para resetear dicha contraseña



Y si pulsamos el botón “Reset Password”, nos llevará a la página “Reset Password” de nuestra aplicación



Podemos ver cómo al resetear la password, se inicia sesión automáticamente, pero el valor del campo "is_logged" no se pone a "1". Para solucionar esto iremos al Controlador "ResetPasswordController" y en el Trait "ResetsPasswords" buscamos el método "resetPassword". Lo copiamos y lo pegamos en nuestro Controlador para sobrescribirlo. En este caso tenemos que importar las clases "Hash", "Str" y "PasswordReset", y poner a true la propiedad "is_logged":

```
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Str;
use Illuminate\Auth\Events\PasswordReset;


$user->setRememberToken(Str::random(60));

$user->is_logged = true;

$user->save();
```

35.- Base de Datos de la Aplicación Forum