Proyecto de Laravel

Crud-laravel

En este Git están los archivos necesarios para las plantila blade:

https://gist.github.com/YouDevs/8515e6c7a1859229e19ac01fac2466aa

Vamos a desarrollar una pequeña aplicación CRUD. Una aplicación CRUD es aquella que funciona a partir de la creación, lectura, actualización y borrado de datos de una base de datos.

Instalar un proyecto Laravel

Vamos a usar composer para instalar el proyecto. Así que abre tu consola, vete hasta el directorio en el que quieras crear tu proyecto y escribe

composer create-project laravel/laravel crud-project

A partir de aquí comienza la instalación del proyecto. Puede tardar más o menos, dependiendo del ancho de bandas que tengas en tu ordenador.

Una vez acabada la instalación del proyecto Laravel, nos movemos hasta la carpeta que se ha creado con el nombre del proyecto. Lo abrimos con Visual Studio Code o con el editor que uses.

Crear y conectar la base de datos

En nuestro phpmyadmin creamos la base de datos que va a servir de soporte a nuestra aplicación en Laravel. En este caso creamos una base de datos llamada crud_project.

Ahora necesitamos conectar la base de datos con nuestro proyecto. Para ello buscamos el archivo .env en el raiz de la carpeta del proyecto. En ese archivo, cambiamos los parámetros para que se pueda conectar sin problema. En mi caso:

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=8889
DB_DATABASE=crud-laravel
DB_USERNAME=root
DB_PASSWORD=root

Para comprobar que la conexión a la base de datos se realiza con normalidad, vamos a ejecutar las migraciones que vienen por defecto en la instalación de Laravel:

php artisan migrate

Si no hay ningún problema, veremos que se han creado unas tablas en tu base de datos. Si no es así, revisa los parámetros del archivo .env.

Crear migración

Nuestro proyecto va a ser una aplicación para gestionar tareas. Necesitamos una tabla donde almacenar esa información que vamos a crear, editar y borrar. A esa tabla la vamos a llamar tasks. Vamos a crear una migración, que usaremos para crear la tabla

php artisan make:migration create_tasks_table

Esto genera un archivo que se guarda dentro de la carpeta database/migrations. Delante del nombre del archivo lleva la fecha de creación de la migración. Abrimos el archivo y añadimos los campos de nuestra tabla tasks:

```
public function up(): void
  {
     Schema::create('tasks', function (Blueprint $table) {
        $table->id();
        $table->string("title");
        $table->text("description");
        $table->dateTime("fecha")->nullable();
        $table->enum("status", ["Pendiente","En progreso", "Completada"])->nullable();
     });
  }
```

Como se puede ver, definimos cada uno de los campos de nuestra tabla tasks, indicando el tipo y el nombre.

En fecha y en status hemos añadido nullable(), para indicar que esos campos pueden quedar vacios, mientras que el resto son obligatorios.

Una vez creada la migración, es hora de ejecutarla para que se cree nuestra tabla tasks:

php artisan migrate

Si no ha habido errores, podremos observar como se ha creado nuestra tabla tasks con los campos que le hemos indicado.

Si nos hemos equivocado al crear la migración, podemos hacer

php artisan migrate:rollback

De esta manera se desharán los cambios en la base de datos. Puedes cambiar tu archivo de migración y volver a ejecutarla para que tus tablas queden correctamente creadas.

Crear el controlador y el modelo

En primer lugar vamos a crear el controlador, usando:

php artisan make:controller TaskController

Podríamos usar esta instrucción, pero no queremos un controlador cualquiera, sino que queremos un controlador para un CRUD, además de que necesitamos un modelo. Todo esto podemos hacerlo en un solo comando:

php artisan make:controller TaskController —resource —model=Task

(Cuidado que son dos guiones)

Si todo ha ido bien, la consola nos va a preguntar si queremos también la creación del modelo, a lo que diremos que sí. Después de esto, veremos que se han creado dos archivos, el controlador TaskController en la carpeta app/HTTP/Controllers, y el modelo Task en app/Models.

Bueno, hemos creado un Resource Controller, pero ¿qué significa esto? El controlador ya viene equipado con todas los métodos clásicos necesarios para una aplicación CRUD.Puedes usarlos todos o sólo los que necesites.

Crear layout y la vista Index

Vamos a crear una vista index, que extenderá o heredará todo el código de una vista que llamaremos layout.

El código del layout y de otras vistas que vamos a usar posteriormente lo tienes en el enlace compartido al principio del documento. En primer lugar, vamos a copiar el contenido de base.blade.php. Dentro de resources/views, crearemos una carpeta llamada layouts. Y dentro de esta carpeta crearemos el archovo base.blade.php con el contenido copiado anteriormente.

Un layout es una plantilla base que define una estructura y un diseño común a todas las páginas del proyecto web. Permite reutilizar elementos de diseño y de código, como encabezados, menús o pies de página, en múltiples vistas sin tener que repetir el mismo código en cada una. De manera que si se necesita cambiar algo, se hace en el layout, sin tener que repetir la misma modificación en un montón de archivos.

En este layout estamos importando los cdns de bootstrap, tanto del css como del js. Si nos fijamos, dentro del body tenemos un container, y dentro de él tenemos una directiva **@yield(“content”)**. Esta directiva se utiliza para marcar un área en el layout donde se va a insertar contenido específico de cada vista. Actúa como un marcador de posición que permite a las vistas que utilizan ese layout inyectar contenido en esa posición designada.

Fuera de la carpeta layouts, creamos la vista index, index.blade.php. Su código lo copiamos del git hub y lo pegamos en el archivo correspondiente.

Lo que estamos haciendo en la index es que esta página hereda todo el código del layout base. **@extends(‘layouts.base')** quiere decir que extiende al archivo que está en layouts/base. Y todo lo que hay entre las directivas **@section(‘content')** y **@endsection** será inyectado en el layout, donde se encuentra la directiva **@yield.**

Para ver estos contenidos, nos vamos al archivo de rutas routes/web.php. Primero debemos importar nuestro controlador, puesto que Laravel lo necesita:

use App\Htttp\Controllers\TaskController;

Como es un controlador de tipo Resource, la sintaxis cambia para especificar las rutas, pues este tipo de controladores ya tiene todas las rutas asignadas. Lo único necesario es conectar las rutas con el controlador. En este caso:

Route::resource("tasks", TaskController::class);

tasks es la ruta y TasksController es nuestro controlador

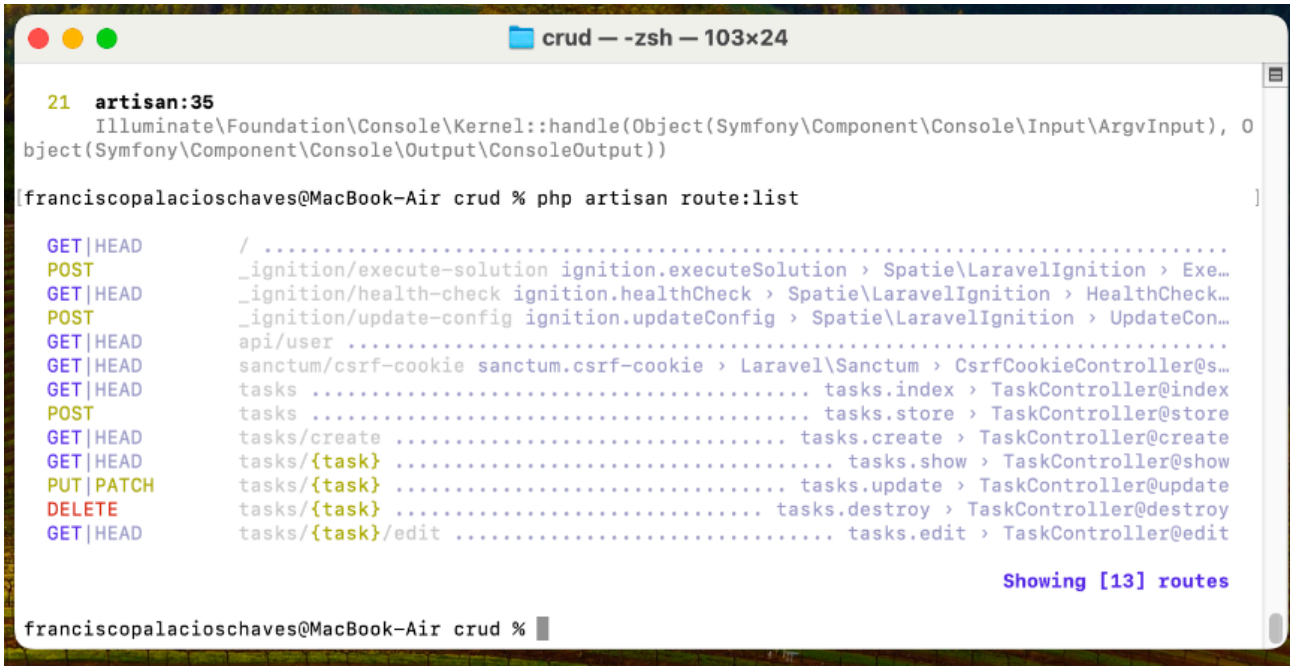
Para ver todas las rutas que tiene nuestro controlador, ejecutamos

php artisan route:list

De esta manera, obtenemos la lista de todas las rutas predefinidas por nuestro controlador, al ser un controlador Resource.

Todas las rutas de tasks están generadas automáticamente por el controlador. Ahora vamos a encender el servidor, para poder ver a nuestra aplicación en acción:

php artisan serve



Esa linea de comando hace que nuestra web sea visible en **http://127.0.0.1:8000/tasks.** De entrada nos va a llevar a una página en blanco. Está conectando con el método index, pero no le estamos diciendo que haga nada. Así que nos vamos al controlador TaskController y añadimos código en el método index:

return view(“index”);

Si hacemos esto y refrescamos nuestra ventana del navegador, ya veremos la aplicación andando. Lo que vemos es el código que tenemos en el archivo index, que de momento no son datos leídos de nuestra base de datos sino datos escritos hardcoded en el código.

Ahora ya podemos empezar a desarrollar las funcionalidades de nuestro CRUD, empezando por la operación de crear contenidos.

Create

Esta funcionalidad nos va a permitir agregar nuevas tareas a nuestra base de datos. Lo primero que hacemos es indicar en el metodo create que va a devolver una nueva vista. No nos debemos confundir: en el método **create()** vamos a incluir el formulario de creación de la tarea. Sin embargo, la lógica de creación de esta tarea en la base de datos se almacenará en el método **store().**

return view(“create”);

Copiamos el código del archivo create.blade.php y lo pegamos en nuestro archivo. Esta vista, al igual que la index, extiende o hereda el layout base y añade contenido dentro del **@yield(“content”)**. En este archivo tenemos el formulario. Como buena práctica, debemos considerar que cada uno de los campos del formulario debe tener el mismo name que los campos de la base de datos.

En el action no tenemos la ruta especificada. La agregamos:

action=“{{route(“tasks.store”)}}”

Es un buen momento para fijarnos en cómo deben cuadrar las rutas. Si nos fijamos en la lista de rutas, tenemos la ruta tasks.store (la que hemos puesto en el action). Esta ruta requiere que sea un método POST, como tenemos puesto en el method del formulario. Mientras sigamos las especificaciones de las rutas que nos proporciona el controlador, no vamos a tener ningún problema.

En el método store, vamos a usar una función, dd, que es como el var_dump tradicional de php

dd($request->all());

$request->all() contiene todos los datos que mandamos via formulario. De esta manera podemos comprobar si el formulario está mandando los datos de manera adecuada. Ahora, volvemos al formulario, rellenamos los campos y veamos si funciona.

Vemos que nos da error, porque si enviamos un formuiario usando POST, Laravel nos pide que usemos la directiva **@csrf,** que es una medida de seguridad utilizada para proteger a las aplicaciones web de ataques maliciosos. El token es un valor único y secreto que se genera para cada formulario en la aplicación. Al enviar un formulario, el token se envía junto con todos los datos, y si Laravel verifica que bo coincide el token almacenado con el enviado, Laravel rechaza la solicitud, ya que podría ser un intento de falsificación de petición de sitio. Esto ayuda a que sólo sean procesadas las peticiones legítimas y autenticadas.

Así que añadimos @csrf justo debajo del inicio del formulario. Recargamos la página, metemos datos de prueba y vemos lo que obtenemos en store.

Ahora, vamos a crear la lógica para almacenar los datos del formulario en la base de datos, usando el método store. Primero, necesitamos importar el modelo:

use App\Models\Task;

Dentro del método store escribimos:

Task::create($request->all());

Después de crear la tarea, redireccionamos a la página index:

return redirect()->route(“tasks.index”);

Y veremos qiue al refrescar nos sale un error. Esto es por un tema de seguridad, que es la asignación masiva, que se refiere a la capacidad de asignar múltiples atributos de una sola vez al modelo, utilizando un único comando. Cuando hagamos esto, debemos decirle al Laravel que vamos a hacerlo y en qué campos vamos a hacerlo. Lo vamos a hacer en el modelo:

protected $fillable = ["title", "description", "fecha", “status”];

Después de rellenar el formulario, podemos ver que está funcionando bien, viendo que se crean nuevos datos en la tabla tasks.

Estamos devolviendo en algunas vistas una redirección, y a Laravel le gusta que declaremos lo que estamos retornando. Así que añadimos en la definición del método store:

public function store(Request $request): RedirectResponse

Nos dará un error a no ser que importemos esa clase en el controlador:

use Illuminate\Http\RedirectResponse;

Vamos a añadirle un poco de validación a ésto de crear nuevas tareas. En el método store añadimos:

```
$request->validate([
        "title" => "required",
        "description" => "required",
    ]);
```

Estamos diciendo que los campos title y description son requeridos, por lo que si se quedan vacíos en el formulario, no se va a procesar. Si lo probamos, veremos que se recarga la página, pero no sew muestra ningún error. Para esto copiamos el código de Git y lo copiamos en la vista create, entre el col y el formulario.

Lo que estamos diciendo, es que si hay algún error, mostrará por pantalla el contenido de ese div, y el foreach va a recorrer todo el array de errores y los va a mostrar por pantalla. Podemos comprobar que funciona, intentando crear una tarea con los campos vacíos.

Podemos mostrar un mensaje de éxito en la index, avisando que la tarea se ha creado de la forma adecuada. Para ello, en el retorno, a la hora de retornar, le añadimos un with:

return redirect()->route("tasks.index")->with("success", "La tarea se ha creado exitosamente");

Este mensaje se guarda en una variable de sesión de tipo flash, es decir, una variable que se usa una vez pero en el momento en que se recargue la página o se vaya a otra, ese valor desaparece.

Tenemos que incluir en la vista de index que si existe esa sesión, pinte el mensaje que queramos:

```
@if (Session::get("success"))

   <div class="alert alert-success mt-2">
     {{Session::get("success")}}
   </div>

@endif
```

Esto lo vamos a poner debajo del enlace de Crear Tarea. Podemos comprobar como al generar una tarea, vemos el mensaje de success en la vista de la index.

Read

Vamos a almecenar en una variable todas las tareas que estén guardadas en nuestra base de datos.

Podríamos hacer un **$tasks = Tasks::all();**, pero en lugar de eso vamos a tomar todas las tareas en orden descendente, y las vamos a pasar a la vista. En el método index:

```
$tasks = Task::latest()->get();
return view("index", ['tasks' => $tasks]);
```

Ahora nos vamos a la vista index, y en lugar de tener los datos de prueba, vamos a tener un foreach donde vamos a leer los datos de la tabla de tareas. Dentro de ese foreach vamos a meter el <tr> con la fila de cada tarea, pero rellenando con los datos leídos de la base de datos:

```
@foreach($tasks as $task)
        <tr>
           <td class="fw-bold">{{$task->title}}</td>
           <td>{{$task->description}}</td>
           <td>
              {{$task->fecha}}
           </td>
           <td>
              <span class="badge bg-warning fs-6">{{$task->status}}</span>
           </td>
```

```
        <td>
            <a href="" class="btn btn-warning">Editar</a>

            <form action="" method="post" class="d-inline">
                <button type="submit" class="btn btn-danger">Eliminar</button>
            </form>
        </td>
    </tr>
    @endforeach
```

El botón de editar y de eliminar los añadiremos más tarde, porque necesitamos crear esos métodos.

Todo va tomando forma, pero podemos añadirle algo más, como por ejemplo un paginador, para que en el caso de que tengamos un montón de tareas, no tengamos un listado interminable. Lo que ocurre es que Laravel viene con unos estilos predeterminados, por lo que, si queremos usar bootstrap para la paginación, debemos retocar algunas cosas. Tenemos que avisar a Laravel de que queremos otro estilo para el paginador. Para ello, tenemos que ir a **app/providers,** al archivo **AppServiceProvider**. Este es un archivo que nos permite registrar servicios de terceros. En este archivo vamos a conectar el paginador de bootstrap con nuestra aplicación:

use Illuminate\Pagination\Paginator;

Y dentro del método boot:

Paginator::useBootstrapFive();

De esa manera, le estamos diciendo que para el paginador debe usar Bootstrao 5. Volvemos al método index para llamar al paginador. En lugar de get(), vamos a usar paginate(numero_elementos):

$tasks = Task::latest()->paginate(3);

Y en la vista index, fuera de la tabla:

{{$tasks->links()}}

Refrescamos y vemos el resultado. Vamos a añadir un poco de código para que avisemos a Laravel de los métodos en los que vamos a retornar una vista:

use Illuminate\View\View;

Y en cada método que retorna una vista, como en la index, create y edit, añadimos:

public function index() : View

Update

De la misma manera que a la hora de crear necesitamos dos métodos, create y store, para actualizar necesitamos dos métodos también: edit y update. En el método edit vamos a enseñar el formulario, y en el método update tendremos toda la lógica encargada de actualizar la base de datos.

El método edit ya está preparado para recoger la tarea que vas a actualizar. En las rutas podemos ver que para editar tenemos tasks/{task}/edit. Es decir, en el lugar de las llaves va a ir el id de la tarea que vamos a modificar.

Vamos a añadir **dd($task)** en el método edit y vamos a introducir la ruta a mano en el navegador, para que veamos como recoge los valores de la tarea a editar.

Una vez que hemos comprobado que todo funciona bien, vamos a decirle al método que tiene que retornar una vista:

return view(“edit");

Creamos el archivo edit.blade.php en la carpeta de vistas, y lo que vamos a hacer es copiar lo que tenemos en create y pegarlo en edit, porque necesitamos el mismo formulario. Sin embargo, vamos a cambiar algunas cosas, como el título que aparece en la línea 7 para que ponga Editar Tarea, y el botón para que diga Actualizar en lugar de Crear.

Con esto, si recargamos, nos va a pintar la vista, pero es obvio que en el formulario faltan los datos de la tarea que vamos a editar. Primero, tenemos que decirle a la vista que pase los datos de la tarea que vamos a editar:

return view("edit", ["task" => $task]);

Y en la vista vamos añadiendo los atributos value junto con el valor de cada campo de la tarea:

value=“{{$task->title}}"

A la fecha debemos quitarles las comillas para que capte el valor. Para el select vamos a usar:

{{$task->status == "Pendiente" ? "selected": ""}}

O

@selected($task->status == “Pendiente")

Que es una directiva de Blade que pinta “selected” si lo que va dentro del paréntesis es true. Sólo nos falta el action del formulario, que debe ser:

<form action=‘{{route(“task.update”, “task")}}' method=“POST”>

El problema que tenemos ahora, si nos fijamos en la ruta del método update, es que es de tipo PUT, que no es un método aceptado por los navegadores y que suele usarse sobre todo en APIs. Para arreglarlo, usaremos una directiva en el formulario, debajo de la directiva **@csrf**:

@method(“PUT")

De esta manera, falsearemos el envio por POST y haremos creer a Laravel que lo estamos haciendo por el método PUT, que es el adecuado para el update. Si queremos, podemos añadir en el método update **dd($request->all())** para comprobar que llegan los datos de forma adecuada.

Ahora en el método update:

$task->update($request->all());

return redirect()->route("tasks.index")->with("success", "La tarea se ha editado exitosamente");

También podemos añadir la validación, de la misma manera que hicimos en el store:

```
$request->validate([
        "title" => "required",
        "description" => "required",
    ]);
```

También añadimos el RedirectResponse, al igual que hicimos en store:

public function update(Request $request, Task $task) : RedirectResponse

Delete

Éste es el último paso que nos queda pendiente por realizar. Si nos fijamos en la vista index, el botón de eliminar es un formulario. Así que nos vamos a la index, y debajo del form del botón añadimos:

```
@csrf
@method(“DELETE")
```

Y lo conectamos con la ruta adecuada:

<form action="{{route('tasks.destroy', $task)}}" method="POST" class=“d-inline">

Ahora, en el método destroy, vamos a escribir la funcionalidad de eliminar esa entrada, además de redirigir a la index con un mensaje adecuado:

```
public function destroy(Task $task) : RedirectResponse
   {
      $task->delete();
      return redirect()->route("tasks.index")->with("success", "La tarea se ha eliminado
exitosamente");
   }
```

Y eso es todo. Ya tenemos nuestro CRUD andando con toda sus funcionalidades.