Guía Práctica Laravel

Creamos un proyecto laravel con el siguiente comando.
composer create-project laravel/laravel componentes

En el archivo .env cambiamos el puerto y parámetros de la base de datos.

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

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel
DB_USERNAME=usuario
DB_PASSWORD=secret0
```

En config/app.php cambiamos el idioma local y el idioma de faker.

```
/*
|--------------------------------------------------------------------------
| Application Locale Configuration
|--------------------------------------------------------------------------
|
| The application locale determines the default locale that will be used
| by the translation service provider. You are free to set this value
| to any of the locales which will be supported by the application.
|
*/

'locale' => 'es',
```

```
/*
|--------------------------------------------------------------------------
| Faker Locale
|--------------------------------------------------------------------------
|
| This locale will be used by the Faker PHP library when generating fake
| data for your database seeds. For example, this will be used to get
| localized telephone numbers, street address information and more.
|
*/

'faker_locale' => 'es_ES',
```

Crearemos las tablas correspondientes en nuestra base de datos.

Crearemos el Modelo, Factory, Migrations y Controladores de nuestra tabla.
php artisan make:model Libro -mfcr

Hacer composer update si no funciona la primera vez. Pues a veces puede dar fallos.

En database/migrations modificaremos el archivo de subida de entradas de nuestra tabla, añadiendo los campos que queramos que tengan sus entradas.

```php
public function up()
{
    Schema::create('libros', function (Blueprint $table) {
        $table->id();
        $table->string('titulo')->unique();
        $table->text('resumen');
        $table->decimal('pvp', 5, 2);
        $table->integer('stock');
        $table->foreignId('user_id')->constrained()->onDelete('cascade');
        $table->timestamps();
    });
}
```

En app/models/Libro indicamos qué campos queremos que puedan ser modificados por el usuario.

```php
class Libro extends Model
{
    use HasFactory;
    protected $fillable = ['titulo', 'resumen', 'pvp', 'stock', 'user_id'];
```

Iremos a database/factories/LibroFactory para crear una preset de faker para nuestra tabla.

```php
class LibroFactory extends Factory
{
    /**
     * Define the model's default state.
     *
     * @return array<string, mixed>
     */
    public function definition()
    {
        return [
            'titulo'=>$this->faker->unique()->words(random_int(2,5), true),
            'resumen'=>$this->faker->text(),
            'pvp'=>$this->faker->randomFloat(2, 1, 100),
            'stock'=>random_int(0, 999),
            'user_id'=>User::all()->random()->id
        ];
    }
}
```

Vamos a database/seeders/DatabaseSeeder.php para hacer una llamada de cien creaciones de entradas a nuestra base de datos basadas en nuestra preset.

```php
class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
        // \App\Models\User::factory(10)->create();

        // \App\Models\User::factory()->create([
        //     'name' => 'Test User',
        //     'email' => 'test@example.com',
        // ]);
        \App\Models\User::factory(10)->create();
        \App\Models\Libro::factory(100)->create();

    }
}
```

Si dos tablas están relacionadas, debemos escribir primero la línea del seeder de la tabla de la que dependa la otra.

Ejecutamos el siguiente comando para crear las tablas y rellenarlas
php artisan migrate:fresh --seed

Podemos ejecutar el comando sin el --seed para comprobar si las migraciones se han realizado correctamente, pero no olvidar usar el --seed para tener luego datos de ejemplo.

Para relaciones entre dos tablas
(En este ejemplo 1:N)

Indicamos en el modelo que un libro tiene un único usuario autor.

```php
class Libro extends Model
{
    use HasFactory;
    protected $fillable = ['titulo', 'resumen',

    // Este libro tiene un usuario autor.
    public function user(){
        return $this->belongsTo(User::class);
    }
}
```

E indicamos que un usuario autor puede tener varios libros.

```php
class User extends Authenticatable
{
    use HasApiTokens, HasFactory, Notifiable;

    // Este usuario puede tener varios libros.
    public function libros(){
        return $this->hasMany(Libro::class);
    }
```

En el controlador de libros pasaremos el campo de usuario autor con with().
Esto es para evitar realizar varias consultas por registro.

```php
/**
 * Display a listing of the resource.
 *
 * @return \Illuminate\Http\Response
 */
public function index()
{
    $libros = Libro::with('user')->paginate(10);
    return view('libros.index', compact('libros'));
}
```

En resouces/views se encuentran nuetras vistas, que son las páginas de nuestra aplicación; donde podremos crear nuestra estructura de archivos.

Se recomienda crear una plantilla aprovechando las funcionalidades de *blade*.

Las directrices @yield() añadirán a la plantilla los contenidos correspondientes a las @section() de todas nuestras páginas.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">

    <!-- CDNs -->
    <script src="https://cdn.tailwindcss.com"></script>
    <script src="//cdn.jsdelivr.net/npm/sweetalert2@11"></script>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.2.1/css/all.min.css">

    <title>@yield('titulo')</title>
</head>
<body style="background-color: #dadada">
    <h3 class="my-2 text-center text-lg">@yield('cabecera')</h3>
    <div class="container mx-auto">
        @yield('contenido')
    </div>
    @yield('js')
</body>
</html>
```

En el directorio de vistas crearemos una nueva carpeta para el las vistas del CRUD de nuestra tabla. La directriz @extends() indica qué plantilla usaremos, mientras que las @section indicarán los contenidos dirigidos a los correspondintes @yield()

```
@extends('plantillas.plantilla')
@section('titulo')
    libros
@endsection
@section('cabecera')
    Lista de libros
@endsection
@section('contenido')
```

Ahora tenemos una página de prueba creada con una tabla en su contenido y datos en los campos yield.

Para que nuestra aplicación rediriga la ruta interna de nuestra vista a la propia vista, debemos devolver su ruta a través método correspondinete del controlador de la página.

Pero primero debemos de crear dichas rutas.

Iremos a routes/web.php y escribiremos lo siguiente.

```php
/*
|--------------------------------------------------------------------------
| Web Routes
|--------------------------------------------------------------------------
|
| Here is where you can register web routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| contains the "web" middleware group. Now create something great!
|
*/

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

Route::resource('/libro', LibroController::class);
```

Importante usar resource para generar rutas para todos los métodos y no get como la ruta que viene por defecto, que solo generaría una ruta.

De este modo, habremos creado una ruta para el index de nuestras vistas 'Libro' que pordemos ver con el comando: php artisan r:l

```
G:\DWESE\prácticas\laravel\libreria>php artisan r:l

  GET|HEAD        / ..... 
  POST            _ignition/execute-solution ..... ignition.executeSolution › Spatie\LaravelIgnition › ExecuteSolutionController
  GET|HEAD        _ignition/health-check ..... ignition.healthCheck › Spatie\LaravelIgnition › HealthCheckController
  POST            _ignition/update-config ..... ignition.updateConfig › Spatie\LaravelIgnition › UpdateConfigController
  GET|HEAD        api/user ..... 
  GET|HEAD        libro ..... libro.index › LibroController@index
  POST            libro ..... libro.store › LibroController@store
  GET|HEAD        libro/create ..... libro.create › LibroController@create
  GET|HEAD        libro/{libro} ..... libro.show › LibroController@show
  PUT|PATCH       libro/{libro} ..... libro.update › LibroController@update
  DELETE          libro/{libro} ..... libro.destroy › LibroController@destroy
  GET|HEAD        libro/{libro}/edit ..... libro.edit › LibroController@edit
  GET|HEAD        sanctum/csrf-cookie ..... sanctum.csrf-cookie › Laravel\Sanctum › CsrfCookieController@show
```

Read

Iremos a app/Http/Controllers/Libro Controller.php y en su método index() devolveremos la ruta de nuestra vista index.

```php
public function index()
{
    $libros = Libro::with('user')->paginate(10);
    return view('libros.index', compact('libros'));
}
```

Como en nuestra vista de libros querremos usar datos guardados en nuestra base de datos, deberemos de realizar una consulta a la hora de cargar la vista y pasar los datos de la consulta a través de una variable a dicha vista.

Para ello usaremos el método compact()

Además usaremos el argumento paginate() para dividir la consulta en páginas.

Para activar la paginación en nuestra vista deberemos de escribir la siguiente directriz en nuestra tabla, preferiblemente al final.

```
    </tbody>
    <div>
        {{$coches->links()}}
    </div>
</table>
```

Create

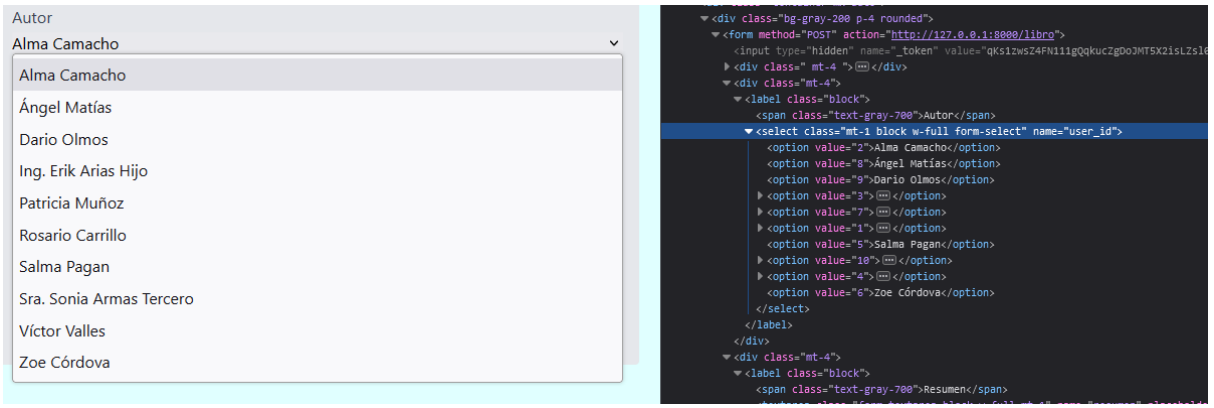
Para tablas relacionadas, querremos tener un campo select para seleccionar, por ejemplo, autores existentes para asignar a nuevos libros.

Para pasar los nombres de los autores a la vista 'create' lo haremos de la siguiente forma.

De esta forma, se van a pasar los nombres como valores a mostrar en el select de x-form y sus ID como los valores a enviar.

```php
/**
 * Show the form for creating a new resource.
 *
 * @return \Illuminate\Http\Response
 */
public function create()
{
    $autores = User::orderBy('name')->pluck('name', 'id');
    return view('libros.create', compact('autores'));
}
```

```
<x-form-select name="user_id" label="Autor" :options="$autores"></x-form-select>
```



Instalaremos el siguiente paquete para facilitar la creación de formularios.
composer require protonemedia/laravel-form-components

En la vista create crearemos un formulario simple cuyo action dirija a la ruta .store de nuestra tabla.

```
<x-form :action="route('libro.store')">
    <x-form-input name="titulo" label="Titulo" placeholder="Titulo">
    <x-form-select name="user_id" label="Autor" :options="$autores">
    <x-form-textarea name="resumen" label="Resumen" placeholder="Res
    <x-form-input name="pvp" label="Precio" type="number" step="0.01
    <x-form-input name="stock" label="Stock" type="number" step="1"
    <div class="mt-2">
        <a href="{{route('libro.index')}}" class="mr-3 bg-blue-500 h
            <i class="fas fa-backward"> Volver</i>
        </a>
        <button type="submit" class="bg-green-500 hover:bg-green-700
            <i class="fas fa-upload"> Guardar</i>
        </button>
    </div>
</x-form>
```

En el método store() del controlador, realizaremos las validaciones necesarias simplemente con aplicando el método validate() del argumento del método.

```
public function store(Request $request)
{
    // Validaciones
    $request->validate([
        'titulo'=>['required', 'string', 'unique:libros,titulo'],
        'resumen'=>['required','string'],
        'pvp'=>['required', 'numeric', 'min:0', 'max:100'],
        'stock'=>['required', 'numeric', 'min:0', 'max:999'],
        'user_id'=>['required','exists:users,id']
    ]);

    // Guardar registro y redirigir a índice.
    Libro::create($request->all());
    return redirect()->route('libro.index')->with('mensaje', "Libro guardado");
}
```

Las reglas de validación que laravel posee por defecto son las siguientes:
https://laravel.com/docs/5.0/validation#available-validation-rules

A la hora de redirigirnos a nuestro índice, podemos tener el siguiente script en nuestra sección de JavaScript para mostrar mensajes de sweetalert2 con la variable de sisión flash que pasamos a la hora de redirigir.

```
@section('js')
@if (session('mensaje'))
<script>
    Swal.fire({
        icon: 'success',
        title: "{{ session('mensaje') }}",
        showConfirmButton: false,
        timer: 1500
    })
</script>
@endif
@endsection
```

(Esto va a ser igual para todos los métodos del CRUD)

Update

Pasamos el objeto a editar por la ruta del botón para ir a la vista de editar libro de nuestra vista index.

```
<a href="{{route('libro.edit', $libro)}}"
    <i class="fas fa-edit"></i>
```

Redirigimos la ruta a la vista pasando los parémetros que necesitemos con el controlador.

```
public function edit(Libro $libro)
{
    $autores = User::orderBy('name')->pluck('name', 'id');
    return view('libros.edit', compact('libro', 'autores'));
}
```

Edit es muy similar a create, solo que usamos directrices @bind() para rellenar los campos ya existentes (por eso debemos de usar los mismos nombres en los campos que en la base de datos), además de usar el método PUT para el formulario, la directriz @csrf ya la introduce los formularios de protonomedia.

```
<x-form :action="route('libro.update', $libro)">
    @method('PUT')
    @bind($libro)
    <x-form-input name="titulo" label="Titulo" placeholder="Titulo"></x-form-input>
    <x-form-select name="user_id" label="Autor" :options="$autores"></x-form-select>
    <x-form-textarea name="resumen" label="Resumen" placeholder="Resumen"></x-form-textarea>
    <x-form-input name="pvp" label="Precio" type="number" step="0.01" min="0" max="100" placeholder="Precio (€)"></x-form-input>
    <x-form-input name="stock" label="Stock" type="number" step="1" min="0" max="999" placeholder="Unidades"></x-form-input>
    @endbind
    <div class="mt-2">
        <a href="{{route('libro.index')}}" class="mr-3 bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded">
            <i class="fas fa-backward"> Volver</i>
        </a>
        <button type="submit" class="bg-green-500 hover:bg-green-700 text-white font-bold py-2 px-4 rounded">
            <i class="fas fa-redo"> Actualizar</i>
        </button>
    </div>
</x-form>
```

En el método update volveremos a realizar las validaciones salvo que esta vez permitiremos que el campo título puedan estar repetidos, pues estamos modificando una entrada del registro ya existente.

```
public function update(Request $request, Libro $libro)
{
    // Validaciones
    $request->validate([
        'titulo'=>['required', 'string', 'unique:libros,titulo,'.$libro->titulo.',titulo'],
        'resumen'=>['required','string'],
        'pvp'=>['required', 'numeric', 'min:0', 'max:100'],
        'stock'=>['required', 'numeric', 'min:0', 'max:999'],
        'user_id'=>['required','exists:users,id']
    ]);

    // Guardar registro y redirigir a índice.
    $libro->update($request->all());
    return redirect()->route('libro.index')->with('mensaje', "Libro guardado");
}
```

Delete

A la hora de crear el delete, es muy simple, en el formulario de acciones de nuestra tabla, como siempre, nos redirigimos a la ruta **.destroy** hacinedo uso del método DELETE con la directiva @method() y usar POST para el method normal.

```
PUT|PATCH       libro/{libro} ..... libro.update
DELETE          libro/{libro} ..... libro.destroy
GET|HEAD        libro/{libro}/edit ..... libro.ed
```

```
<td class="text-sm text-gray-900 font-light px-6 py-4 whitespace-nowrap">
    <form action="{{route('libro.destroy', $libro)}}" method="POST">
        @csrf
        @method("DELETE")
        <a href="" class="bg-yellow-500 hover:bg-yellow-700 text-white fo
            <i class="fas fa-edit"></i>
        </a>
        <button type="submit" class="ml-2 bg-red-500 hover:bg-red-700 tex
            <i class="fas fa-trash"></i>
        </button>
```

Y en controlador tan solo debemos llamar a la eliminación del objeto y redirigirnos de vuelta a la página con un mensaje de notificación.

```
public function destroy(Libro $libro)
{
    $libro->delete();
    return redirect()->route('libro.index')->with('mensaje', "Libro borrado.");
}
```

Show

En este proyecto también tenemos un show. Es muy sencillo de hacer.

Como a nuestra vista index ya pasamos todos los libros, mostrando cada uno en una fila de la tabla con un bucle foreach, podemos asignar al botón de show el valor de la variable a pasar a la ruta del botón.

```
<tbody>
    @foreach ($libros as $libro)
    <tr class="bg-gray-100 border-b">
        <td class="px-6 py-4 whitespace-nowrap text-sm font-medium text-gray-900">
            <a href="{{route('libro.show', $libro)}}" class="bg-cyan-500 hover:bg-cyan-700 text-white font-bold py-2 px-4 rounded">
                <i class="fas fa-info"></i>
            </a>
        </td>
```

En el controlador, sólo tenemos que encargarnos de devolver la ruta.

```
public function show(Libro $libro)
{
    return view('libros.show', compact('libro'));
}
```

Y en la vista show, rellenaremos los campos de la siguiente forma, obteniendo los datos de los atributos del objeto libro.

```
@extends('plantillas.plantilla')
@section('titulo')
    detalle libro
@endsection
@section('cabecera')
    Información sobre el libro
@endsection
@section('contenido')
<div class="mx-auto max-w-sm rounded overflow-hidden shadow-lg bg-teal-200">
    <div class="px-6 py-4">
      <div class="font-bold text-xl mb-2">{{$libro->titulo}}</div>
      <u>Autor/a:</u><p class="text-gray-700 text-base">{{$libro->user->name}}</p>
      <u>Resumen:</u>
      <p class="text-gray-700 text-base">
        {{$libro->resumen}}
      </p>
      <u>Precio</u><p class="text-gray-700 text-base">{{$libro->pvp}}</p>
      <u>Stock</u><p class="text-gray-700 text-base">{{$libro->stock}}</p>
      <br><hr>
      <div class="flex">
            <a href="{{route('libro.index')}}" class="mt-3 bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded">
                <i class="fas fa-backward"> Volver</i>
            </a>
      </div>
    </div>
  </div>
@endsection
```