Sesión 6. Formularios y validación de datos

1. Envío de formularios con Laravel

El envío de formularios en Laravel implica, por un lado, definir un formulario, empleando HTML sencillo junto con algunas opciones ofrecidas por Blade. Por otra parte, debemos recoger los datos enviados por el formulario en algún método de algún controlador, y procesarlos adecuadamente.

1.1. Creación y envío de formularios

Si definimos un formulario en una vista, se define con los conceptos que ya sabemos de HTML. Como único añadido, en el campo `action` del formulario podemos utilizar Blade y la función `route` para indicar el nombre de ruta a la que queremos enviar el formulario.

Veamos, por ejemplo, cómo definir un formulario para dar de alta nuevos libros en nuestro proyecto de *biblioteca*. Creamos una vista llamada `create.blade.php` en la subcarpeta `resources/views/libros`, con un contenido como éste:

```
@extends('plantilla')

@section('titulo', 'Nuevo libro')

@section('contenido')
    <h1>Nuevo libro</h1>

    <form action="{{ route('libros.store') }}" method="POST">

        <div class="form-group">
            <label for="titulo">Título:</label>
            <input type="text" class="form-control" name="titulo"
                id="titulo">
        </div>

        <div class="form-group">
            <label for="editorial">Editorial:</label>
            <input type="text" class="form-control" name="editorial"
                id="editorial">
        </div>

        <div class="form-group">
            <label for="precio">Precio:</label>
            <input type="text" class="form-control" name="precio"
                id="precio">
        </div>

        <div class="form-group">
            <label for="autor">Autor:</label>
            <select class="form-control" name="autor" id="autor">
                @foreach ($autores as $autor)
                    <option value="{{ $autor->id }}">
                        {{ $autor->nombre }}
                    </option>
                @endforeach
            </select>
        </div>

        <input type="submit" name="enviar" value="Enviar"
            class="btn btn-dark btn-block">

    </form>
@endsection
```

Un segundo añadido más que tenemos que tener en cuenta es que Laravel por defecto protege de ataques XSS (*Cross Site Scripting*) de suplantación de identidad, por lo que obtendremos un error de tipo 419 si enviamos un formulario no validado. Para solucionar este problema, basta con utilizar la directiva `@csrf` en el formulario, que añade un campo oculto con un token de validación del usuario:

```
<form action="{{ route('libros.store') }}" method="POST">
    @csrf
    ...
</form>
```

En cualquier caso, este formulario se enviará a la ruta indicada. Dado que en nuestro proyecto hemos definido un conjunto de recursos como éste en `routes/web.php`, la ruta ya está automáticamente definida como `libros.store`:

```
Route::resource('libros', 'LibroController');
```

De lo contrario, tendríamos que añadir a mano la ruta correspondiente para recoger el formulario.

Además, debemos redefinir los métodos involucrados en el controlador: por un lado, el método `create` deberá renderizar el formulario anterior. Como necesitamos mostrar el listado de autores para asociar uno al libro, le pasaremos a la vista anterior el listado de autores como parámetro:

```
public function create()
{
    $autores = Autor::get();
    return view('libros.create', compact('autores'));
}
```

Por otra parte, el método `store` se encargará de recoger los datos de la petición a través del parámetro `Request` de dicho método. Disponemos de un método `get` para acceder a cada campo del formulario a partir de su nombre:

```
public function store(Request $request)
{
    $libro = new Libro();
    $libro->titulo = $request->get('titulo');
    $libro->editorial = $request->get('editorial');
    $libro->precio = $request->get('precio');
    $libro->autor()->associate(Autor::findOrFail($request->get('autor')));
    $libro->save();

    return redirect()->route('libros.index');
}
```

Podemos emplear también algún método auxiliar de la petición, como `has`, que comprueba si existe un campo con un nombre determinado:

```
public function store(Request $request)
{
    if($request->has('titulo'))
    {
        ...
    }
}
```

Para poder lanzar esta operación de inserción, necesitamos algún enlace que muestre el formulario. Podemos añadir una nueva opción en el menú superior de navegación (archivo `resources/views/partials/nav.blade.php`):

```
<nav class="navbar navbar-expand-lg navbar-dark bg-secondary">
    ...
    <div class="collapse navbar-collapse" id="navbarNav">
        <ul class="navbar-nav">
            ...
            <li class="{{ setActivo('libros.create') }} nav-item">
                <a class="nav-link" href="{{ route('libros.create') }}">
                Nuevo libro</a>
            </li>
        </ul>
    </div>
</nav>
```

1.2. Actualizaciones y borrados

Por defecto, el atributo `method` de un formulario sólo admite los valores GET o POST. Si queremos enviar un formulario de actualización o borrado, éste debe ir asociado a los métodos PUT o DELETE, respectivamente. Para esto, podemos emplear dentro del mismo formulario la directiva `@method`, indicando el nombre del método que queremos usar:

```
<form ...>
    @csrf
    @method('PUT')
    ...
</form>
```

Por ejemplo, para borrar libros en nuestra aplicación de biblioteca, podríamos añadir un formulario como este en la ficha del libro (vista `resources/views/libros/show.blade.php`):

```
<form action="{{ route('libros.destroy', $libro->id) }}" method="POST">
    @csrf
    @method('DELETE')
    <input type="submit" class="btn btn-danger" value="Borrar libro" />
</form>
```

Observa cómo le pasamos a la ruta el *id* del libro a borrar, para que le llegue como parámetro al método `destroy` del controlador. Dentro de este método, buscamos el libro afectado, lo borramos, y mostramos el listado de libros nuevamente:

```
public function destroy($id)
{
    $libro = Libro::findOrFail($id);
    $libro->delete();
    return redirect()->route('libros.index');
}
```

En este punto, puedes realizar el Ejercicio 1 de los propuestos al final de la sesión, y el Ejercicio 2, que se deja como opcional.

2. Validación de formularios

Además de aplicar una validación en el cliente a través de HTML5, que también es recomendable, se deben validar los datos en el servidor. Para hacer esto, el propio objeto `request` proporciona un método llamado `validate`, al que le pasamos un array con las reglas de validación.

Por ejemplo, así comprobaríamos que el título y la editorial se han enviado, y que el título tiene un tamaño mínimo de 3 caracteres. Además, comprobamos que el precio es un valor numérico real positivo.

```
public function store()
{
    request()->validate(
        [
            'titulo' => 'required|min:3',
            'editorial' => 'required',
            'precio' => 'required|numeric|min:0'
        ]
    );

    // ... Código para procesar el formulario
}
```

> **NOTA**: notar que en varios campos se han añadido dos o más reglas de validación enlazadas por una barra vertical. Para el precio, por ejemplo, se comprueba que se ha enviado, que es numérico y que es mayor o igual que 0. Podéis consultar en la documentación de Laravel sobre otras reglas de validación disponibles, especialmente en el apartado de *Available Validation Rules*.

2.1. Utilizar *form requests* para validaciones más complejas

Si tenemos que validar unos pocos campos, puede ser adecuado llamar al método `validate` desde el propio método del controlador, pero para formularios más grandes el código puede crecer demasiado.

Una alternativa que ofrece laravel es crear un *form request*, una clase adicional que contiene la lógica de validación de una petición. Se crean con el comando `php artisan`, y la opción `make:request`, seguida del nombre de la clase a crear:

```
php artisan make:request LibroPost
```

Esta clase se almacena por defecto en `app/Http/Requests`, y contiene un par de métodos predefinidos:

- `authorize`: devuelve un booleano dependiendo de si el usuario actual está autorizado a enviar la petición o no. Para muchos formularios que no requieran autorización previa podemos simplemente devolver `true`. Será lo que haremos de momento en este formulario.
- `rules`: este es el método que más nos interesa. Devuelve un array de reglas de validación como las que teníamos en el controller, así que movemos ese código aquí:

```
public function rules()
{
    return [
        'titulo' => 'required|min:3',
        'editorial' => 'required',
        'precio' => 'required|numeric|min:0'
    ];
}
```

Ahora, en el método del controlador simplemente tenemos que inyectar este *form request* como parámetro (si observamos la clase que se ha creado, es un subtipo de `Request`), y usarlo para validar. La validación es automática, es decir, no tenemos que añadir más código al controlador que el objeto inyectado como parámetro, que se encargará de validar la propia petición que contiene a través del método `rules`.

```
public function store(LibroPost $request)
{
    // Si entramos aquí, el formulario es válido
}
```

2.2. Mostrar mensajes de error

Si la validación es correcta, se retornará el dato del final de la función, pero si falla algún campo, se volverá a la página del formulario, con la información del error que se haya producido. Podemos acceder desde cualquier lugar de Laravel a la variable `$errors` con los errores que se hayan producido en una operación determinada. Esta variable tiene un método booleano llamado `any` que comprueba si hay algún error, y otro método llamado `all` que devuelve el array de errores producidos. Combinando estos dos métodos con Blade, podemos mostrar el listado de errores de validación antes del formulario, de esta forma:

```
@if ($errors->any())
    <ul>
    @foreach($errors->all() as $error)
        <li>{{ $error }}</li>
    @endforeach
    </ul>
@endif
<form ...>
    @csrf
    ...
</form>
```

También podemos emplear el método `first` del array de errores para obtener el primer error asociado a un campo, y mostrarlo bajo o sobre el campo en cuestión. Por ejemplo:

```
<form action="{{ route('libros.store') }}" method="POST">

    @csrf

    <div class="form-group">
        <label for="titulo">Título:</label>
        <input type="text" class="form-control" name="titulo"
            id="titulo">
        @if ($errors->has('titulo'))
            <div class="text-danger">
                {{ $errors->first('titulo') }}
            </div>
        @endif
    </div>

    ...
```

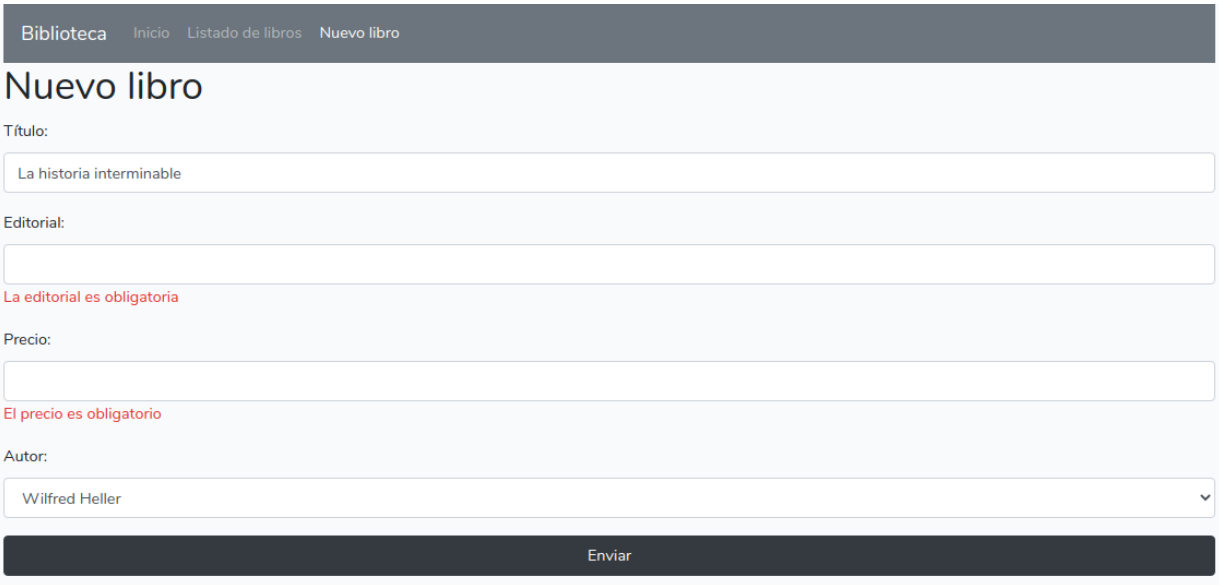
Además, podemos **personalizar el mensaje de error** a mostrar, redefiniendo en la clase del *form request* el método `messages`. En este método devolvemos un array con el mensaje a mostrar para cada posible error de validación. Por ejemplo:

```
public function messages()
{
    return [
        'titulo.required' => 'El título es obligatorio',
        ...
    ];
}
```

De forma alternativa, si optamos por validar el formulario en el propio controlador, este array de mensajes se pasa como segundo parámetro en la llamada al método `validate`:

```
request()->validate(
    [
        'titulo' => 'required|min:3',
        'editorial' => 'required',
        'precio' => 'required|numeric|min:0'
    ], [
        'titulo.required' => 'El título es obligatorio',
        ...
    ]
);
```

En definitiva, conseguiremos mostrar mensajes de error para los campos que hayan dado errores al validar:



2.3. Recordar valores enviados

Un problema derivado de la validación de datos es que, al volver a la página del formulario tras un error, los campos que ya se han examinado hasta el error, aunque fueran correctos, han perdido el valor que tenían, y puede resultar engorroso tenerlos que rellenar otra vez. Para mantener su antiguo valor, podemos añadir el atributo `value` en cada campo del formulario, y utilizar con Blade una función llamada `old`, que permite acceder al anterior valor de un determinado campo, referenciado por su nombre:

```
<form action="{{ route('libros.store') }}" method="POST">

    @csrf

    <div class="form-group">
        <label for="titulo">Título:</label>
        <input type="text" class="form-control" name="titulo"
            id="titulo" value="{{ old('titulo') }}">
        @if ($errors->has('titulo'))
            <div class="text-danger">
                {{ $errors->first('titulo') }}
            </div>
        @endif
    </div>

    ...
```

En el caso de áreas de texto, usamos esta expresión dentro del área (es decir, entre la etiqueta de apertura y la de cierre del *textarea*):

```
<textarea name="mensaje" placeholder="Mensaje...">
{{ old('mensaje') }}
</textarea>
```

En este punto, puedes realizar el Ejercicio 3 de los propuestos al final de la sesión.

3. Ejercicios propuestos

Ejercicio 1

Sobre el proyecto **blog** de la sesión anterior, vamos a añadir estos cambios:

- Crea un formulario para dar de alta nuevos posts, en la vista `resources/views/posts/create.blade.php`. Añade un par de campos (un texto corto y un texto largo) para rellenar el título y el contenido, y como autor o usuario del post de momento deja uno predefinido; por ejemplo, el autor con id = 1, o el primer autor que encuentres en la base de datos (`Autor::get()->first()`). Más adelante ya lo haremos dependiente del usuario que se haya autenticado. Recuerda definir el método `store` en el controlador de posts para dar de alta el post, y redirigir después al listado principal de posts. Para cargar el formulario, añade una nueva opción en el menú principal de navegación.
- En la ficha de un post, añade un botón con un formulario para borrar el post. Deberás definir el código del método `destroy` para eliminar el post y redirigir de nuevo al listado. En el caso de que hayas hecho el ejercicio opcional de la sesión anterior para añadir comentarios a los posts, deberás previamente eliminar todos los comentarios asociados a ese post, y después borrar el post. Para filtrar los comentarios de un post y borrarlos, utiliza la cláusula `where` que se explicó en la sesión 4:

```
Comentario::where('post_id', $id)->delete();
// Aquí ya borramos el post
```

Ejercicio 2

Opcional

Continuamos con el proyecto **blog** anterior. Ahora añadiremos el formulario de edición de un post, también desde la vista de la ficha del post. El formulario deberá mostrar los datos ya rellenos del post. Dicho formulario se carga a partir del método `edit` (que deberá renderizar la vista con el formulario de edición, `resources/views/posts/edit.blade.php`), y el formulario se enviará al método `update` del controlador, pasándole como parámetro el *id* del post a modificar.

Ejercicio 3

Continuamos con el proyecto **blog** anterior. Crea un *form request* llamado `PostRequest`, que valide los datos del post. En concreto, deben cumplirse estos requisitos:

- El título del post debe ser obligatorio, y de al menos 5 caracteres de longitud
- El contenido del post debe ser obligatorio, y de al menos 50 caracteres de longitud

Define mensajes de error personalizados para cada posible error de validación, y muéstralos junto a cada campo afectado, como en el ejemplo de la biblioteca. Además, utiliza la función `old` para recordar el valor antiguo correcto, en el caso de que un campo pase la validación pero otro(s) no.

¿Qué entregar?

Como entrega de esta sesión deberás comprimir el proyecto **blog** con todos los cambios incorporados, y eliminando las carpetas `vendor` y `node_modules` como se explicó en las sesiones anteriores. Renombra el archivo comprimido a `blog_06.zip`.