Unit 6. Forms and data validation

1. Submitting forms with Laravel

Submitting forms in Laravel involves, on the one hand, defining a form, using simple HTML along with some options offered by Blade. On the other hand, we must collect the data sent by the form in some method from some controller, and process it appropriately.

1.1. Creating and submitting forms

If we define a form in a view, it is defined with the concepts we already know about HTML. As the only addition, in the 'action' field of the form we can use Blade and the 'route' function to indicate the route name to which we want to send the form.

Let's see, for example, how to define a form to register new books in our *library* project. We have to create a view called 'create.blade.php' in the 'resources/views/books' subfolder, with content like this:

```
@extends('template')

@section('title', 'New Book')

@section('content')

    <h1>New book</h1>

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

     @csrf

        <div class="form-group">

            <label for="title">Title:</label>

            <input type="text" class="form-control" name="title" id="title">

        </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="price">Price:</label>
                <input type="text" class="form-control" name="price" id="price">
            </div>

            <div class="form-group">
    <label  for="author">Author:</label>
                <select class="form-control" name="author" id="author">
                    @foreach ($authors the $author)
                        <option value="{{ $author->id }}">
                            {{ $author->name }}
                        </option>
                    @endforeach
                </select>
            </div>

            <input type="submit" name="send" value="Send"
                class="btn btn-dark btn-block">

        </form>
    @endsection
```

Note: We have to take into account that Laravel by default protects from XSS (*Cross Site Scripting*) attacks of identity theft, so we will get a 419 error if we send an unvalidated form. To solve this problem, simply use the '@csrf' directive in the form, which adds a hidden field with a user validation token:

In any case, this form will be sent to the indicated route. Since in our project we have defined a set of resources like this in 'routes/web.php', the route is already automatically defined as 'books.store': (If you have ->only(['index', 'show', 'create', 'edit', 'destroy']) delete it)

```
Route::resource('books', 'BookController');
```

Otherwise, we would have to add by hand the corresponding route to collect the form.

In addition, we must redefine the methods involved in BookController: on the one hand, the 'create' method will have to render the previous form. As we need to show the list of authors to associate one with the book, we will pass to the previous view the list of authors as a parameter (remember to include the 'use App\Models\Author; ' in the controller):

```
public function create()
{
    $authors = Author::get();
    return view('books.create', compact('authors'));
}
```

On the other hand, the 'store' method will be responsible for collecting the data of the request through the 'Request' parameter of this method. We have a 'get' method to access each field of the form from its name:

```
public function store(Request $request)
{
    $book = new Book();
    $book->title = $request->get('title');
    $book->editorial = $request->get('editorial');
    $book->price  = $request->get('price');
    $book->author()->associate(Author::findOrFail($request->get('author')));
    $book->save();
    return redirect()->route('books.index');
}
```

We can also use some auxiliary method of the request, such as ' has', which checks if there is a field with a certain name:

```
public function store(Request $request)

{

    if($request->has('title'))

    {

        ...

    }

}
```

In order to launch this insert operation, we need a link that shows the form. We can add a new option in the top navigation menu (file ' resources/views/partials/nav.blade.php'):

```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark">

<a class="navbar-brand" href="#">Library</a>
<button class="navbar-toggler" type="button" data-toggle="collapse" data-
target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-
label="Toggle navigation">
<span class="navbar-toggler-icon"></span>
</button>

<div class="collapse navbar-collapse" id="navbarNav">
<ul class="navbar-nav">
<li class="nav-item active">
<a class="nav-link" href="{{ route('start') }}">Start </a>
</li>
<li class="nav-item">
<a class="nav-link" href="{{ route('books.index') }}">List of books</a>
</li>
<li class="nav-item">
<a class="nav-link" href="{{ route('books.create') }}">New Book</a>
</li>
</ul>
</div>

</nav>
```

1.2. Updates and deletes

By default, the 'method' attribute of a form only supports GET or POST values. If we want to send an update or delete form, it must be associated with the PUT or DELETE methods, respectively. For this, we can use within the same form the directive '@method', indicating the name of the method we want to use:

Remember from the previous Unit, that we already did the delete with '@method('DELETE')'

```
<form action="{{ route('books.destroy', $book) }}" method="POST">

        @method('DELETE')
        @csrf
       <button>Delete</button>

</form>
```

Remember how we pass the book to be deleted to the path, so that it arrives as a parameter to the 'destroy' method of the controller. Within this method, we delete the book, and display the list of books again:

```
public function destroy(book $book)

{

    $book->delete();

    return redirect()->route('books.index');

}
```

Same and similar with the update. First, we add in index.blade.php the button to Update (under the Delete)

```
<form action="{{ route('books.edit', $book) }}">
@csrf
<button>Update</button>
</form>
```

Then, in BookController:

```
public function edit(book $book)
{
```

```
$authors = Author::get();
return view('books.edit', compact('book','authors'));
}
```

We create a new edit.blade.php view:

```
@extends('template')
@section('title', 'Edit Book')
@section('content')
<h1>Edit book</h1>

<form action="{{ route('books.update',$book) }}" method="POST">
@method('PUT')
@csrf

<div class="form-group">

<label for="title">Title:</label>

<input type="text" class="form-control" name="title" id="title" value="{{ $book-
>title }}">

</div>

<div class="form-group">

<label for="editorial">Editorial:</label>

<input type="text" class="form-control" name="editorial"

id="editorial" value="{{ $book->editorial }}">

</div>

<div class="form-group">

<label for="price">Price:</label>

<input type="text" class="form-control" name="price"

id="price" value="{{ $book->price }}">

</div>
<div class="form-group">

<label for="author_id">Author:</label>

<select class="form-control" name="author_id" id="author_id">

@foreach ($authors as $author)
@if ($author->id==$book->author_id)
```

```
<option value="{{ $author->id }}" selected>
{{ $author->name }}
</option>

@else
<option value="{{ $author->id }}">
{{ $author->name }}
</option>

@endif

@endforeach

</select>

</div>

<input type="submit" name="send" value="Send"

class="btn btn-dark btn-block">

</form>

@endsection
```

And in BookController, the update method:

```
public function update(Request $request, book $book)
{

$book->title = $request->get('title');
$book->editorial = $request->get('editorial');
$book->price = $request->get('price');
$book->author_id=$request->get('author_id');

$book->save();
return redirect()->route('books.index');

}
```

2. Form validation

In addition to applying client-side validation via HTML5, which is also recommended, the data must be validated on the server. To do this, the 'request' object itself provides a method called 'validate', to which we pass an array with the validation rules.

For example, this would check that the title and editorial have been sent, and that the title has a minimum size of 3 characters. In addition, we verify that the price is a positive real numerical value.

```
public function store()
{
    request()->validate(
        [
            'title' => 'required|min:3',
            'editorial' => 'required',
            'price' => 'required|numeric|min:0'
        ]
    );
    // ... Code to process the form
}
```

> **NOTE**: Note that two or more validation rules linked by a vertical bar have been added to several fields. For the price, for example, it is checked that it has been sent, that it is numeric and that it is greater than or equal to 0. You can consult the Laravel documentation about other available validation rules, especially in the *Available Validation Rules* section.

2.1. Display error messages

If the validation is successful, the data from the end of the function will be returned, but if any field fails, it will be returned to the form page, with the information of the error that occurred. We can access from anywhere in Laravel to the variable '$errors' with the errors that have occurred in a given operation. This variable has a Boolean method called 'any' that checks for any errors, and another method called 'all' that returns the array of errors produced. Combining these two methods with Blade, we can display the list of validation errors before the form, in this way (in create.blade.php view):

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

```
    @endforeach
    </ul>
@endif
<form ... >
    @csrf
    ...
</form>
```

We can also use the 'first' method of the error array to obtain the first error associated with a field, and show it under or on the field in question. For example:

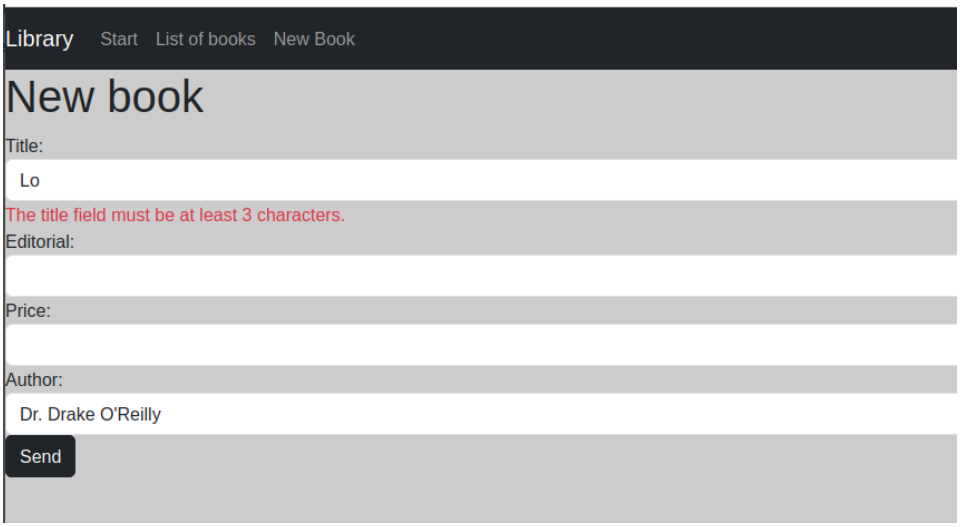
```
<form action="{{ route('books.store') }}" method="POST">

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

In addition, we can **customize the error message** to display, redefining the 'messages'.

The message array is passed as the second parameter in the call to the 'validate' method:

```
request()->validate(
    [
        'title' => 'required|min:3',
        'editorial' => 'required',
        'price' => 'required|numeric|min:0'
    ], [
'title.required' =>  'The title is compulsory',
        ...
    ]
);
```

In short, we will be able to show error messages for the fields that have given errors when validating:



2.3. Remember submitted values

One problem with data validation is that when you return to the form page after an error, the fields that have already been examined to the error, even if they were correct, have lost their value, and it can be a lot of work to fill them in again. To maintain its old value, we can add the attribute 'value' in each field of the form, and use with Blade a function called 'old', which allows access to the previous value of a certain field, referenced by its name:

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

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

In the case of text areas, we use this expression within the area (that is, between the opening and closing tags of the *textarea*):

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