Unit 5. Views

In this session we will focus on the concept of view, and how to associate a Route with the definition of the different views or pages that will make up our application. For this second step, we will make use of the built-in template engine with the Laravel framework, called Blade.

1. Views with Laravel

Until now the paths that we have defined return a simple text, except for the one that was already created by default in the project, which pointed to the home page. If we wanted to return HTML content, a (expensive) option would be to return this content generated from the path method itself, through the 'return' statement, but instead of doing this from within the response function itself, the most common (and recommended) is to generate a **view** with the HTML content that you want to send to the client.

The general way to display views in Laravel is to make the paths return ('return') a certain view. To do this, you can use the 'view' function of Laravel, indicating the name of the view to be generated or displayed.

By default, in the folder 'resources/views' we have available an example view called 'welcome.blade.php'. It is the one that is used as a home page in the root route in 'routes/web.php':

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

Note that it is not necessary to indicate the *path* or path to the file of the view, nor the extension, since Laravel assumes that by default the views are in the folder 'resources/views',with the extension '.blade.php' (which refers to the Blade template engine that we will see below), or simply with extension '.php' (in the case of simple views that do not use Blade).

We can, for example, create a simple view within this folder of views (let's call it 'start.blade.php'),with a basic HTML content:

```
<html>
    <head>
        <title>Init</title>
    </head>
    <body>
        <h1>Home page</h1>
    </body>
</html>
```

And we can use this view as a home page:

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

1.1. Pass values to views

It is very common to pass certain information to certain views, such as lists of data to be displayed, or data of a specific element. For example, if we want to give a welcome message to a name (supposedly variable), we must store the name in a variable in the path, and pass it to the view when loading it. This can be done, for example, with the 'with' method after generating the view, indicating the name with which we are going to associate it with the view, and the value (variable) associated with that name. In our case it would look like this:

```
Route::get('/', function() {
$name =  "Nacho";
return view('start')->with('name',  $name);
});
```

Later, in the view, we will have to show the value of this variable somewhere in the HTML code. We can use traditional PHP to collect this variable:

```
<html>
    <head>
        <title>Init</title>
    </head>
    <body>
        <h1>Home page</h1>
        <p>Wellcome <?php echo $name; ?></p>
    </body>
</html>
```

But it is more common and clean to use a specific Blade syntax, as we will see below.

As alternatives to the use of 'with' discussed above, we can also use an associative array (thus assigning several names to several values):

```
return view('start')->with(['name' =>  $name,...]);
```

Also, we can use this same array as the second parameter of the 'view' function, and thus dispense with 'with':

```
return view('start',['name' =>  $name,...]);
```

And we can also use a function called 'compact' as the second parameter of 'view'. To this function we pass only the name of the variable and, whenever the associated variable is called the same, it establishes the association for us:

```
return view('start',  compact('name'));
```

The 'compact' function supports as many parameters as we want to send to the view separately, each with its associated name.

If we are simply going to return a view with little associated information, or little internal logic, we can also abbreviate the above code by calling directly to 'view',instead of 'route' first, in the file 'routes/web.php',and thus pass the information associated with the view:

```
Route::view('/',  'start',['name' =>  'Nacho']);
```

1.2. Getting Started with the Blade Template Engine

We have commented in the previous section that the use of Blade allows us to simplify the syntax and the way to process some things in our views. As long as we create the view file with the extension '.blade.php',it will automatically allow us to take advantage of the syntax and functionalities of Blade in our views.

For example, if we want to show the content of the variable 'name' that we have passed before to the home page, instead of making a rudimentary 'echo' in PHP, we can use a double key syntax, provided by Blade, to show the content of that variable. With this the line that showed the name goes from being like this...

```
Welcome <?php echo  $name  ?>
```

... to be so:

```
Welcome {{ $name  }}
```

> **NOTE** Each time a view is rendered in Laravel, the generated PHP content is stored in 'storage/framework/views',and is only re-generated after a change in the view, so calling back to an already rendered view does not affect the performance of the application. If we take a look at the view generated with only PHP or with blade, we will see a difference between both: With blade, instead of making a simple 'echo' to show the value of the variable, an intermediate function called 'e' is used. This function prevents XSS attacks(*Cross Site Scripting*),that is, to inject JavaScript *scripts* with the variable to display. In other words, the code is not interpreted. In some cases (especially when we generate HTML content from within the Blade expression) we may be interested in not protecting against these injections of code. In that case, the second key is replaced by a double exclamation:

```
Welcome {!! $name!!}
```

In addition to this basic syntax to display variable data in a certain place in the view, there are certain directives in Blade that allow us to perform checks or repetitions.

1.2.1. Flow control structures in Blade

To iterate on a data set (array), we can use the directive '@foreach',with a syntax similar to the *foreach* of PHP, but without the keys. Simply end the loop with the '@endforeach' directive, like this:

```
<ul>
@foreach($elements  as  $element)
< li>{{  $element  }}</li>
    @endforeach
</ul>
```

In case you want to check something (for example, if the previous array is empty, to show a relevant message), we use the directive '@if',closed by its corresponding pair '@endif'. Optionally, you can use a '@else' directive for the alternate condition, or also '@elseif' to indicate another condition. The above example might look like this:

```
<ul>
    @if($elements)
@foreach($elements  as  $element)
< li>{{  $element  }}</li>
        @endforeach
    @else
< li>No  items to display</li>
    @endif
</ul>
```

We can also check if a variable is defined. In this case, we replace the '@if' directive with '@isset',with its corresponding '@endisset' closure.

```
<ul>
    @isset($elements)
@foreach($elements  as  $element)
< li>{{  $element  }}</li>
        @endforeach
    @else
< li>No  items to display</li>
    @endisset
</ul>
```

However, with any of these options we have a problem: in the first case, if the variable '$elements' is not defined, it will show a PHP error. In the second case, if the variable is defined but does not contain elements, nothing will be displayed per screen. A third alternative structure that groups these two cases together (controlling while the variable is defined and has elements) is to use the '@forelse' directive instead of '@foreach'. This policy allows an additional '@empty' clause to indicate what to do if the collection has no items or is undefined. The above example would now be abbreviated as follows:

```
<ul>
@forelse($elements  as  $element)
< li>{{  $element  }}</li>
    @empty
< li>No  items to display</li>
    @endforelse
</ul>
```

In this type of iterators('@foreach' or '@forelse'),we have available an object called '$loop',with a series of properties about the loop we are iterating, such as 'index' (position within the array through which we are going), or 'count' (total of elements), or 'first' and 'last' (Booleans who determine whether it is the first or last element, respectively), among others. We can see all the properties available in this object by calling 'var_dump':

```
<ul>
@forelse($elements  as  $element)
< li>{{  $element  }} {{  var_dump($loop)}}  </li>
    @empty
< li>No  items to display</li>
    @endforelse
</ul>
```

If, for example, we want to determine if it is the last item in the list, and display a special message or style, we can do something like this:

```
<ul>
@forelse($elements  as  $element)
<li>{{  $element  }}
{{ $loop->last  ? "Last item"  :  ""  }}
        </li>
    @empty
< li>No  items to display</li>
    @endforelse
</ul>
```

There are other types of iterative and selective structures in Blade, such as '@while', '@for' or '@switch',and a few others. You can consult about its use in the official documentation ofBlade.

Let's apply this in our *library* project example. We will define a route to get a list of books. By the moment we will create this list manually with an array (not a database). In the routing method, we will pass the array to a view called 'list.blade.php'. So, the new route for the list looks like:

```
Route::get('list', function() {
$books =  array(
array("title"  =>  "Ender's game",
              "author" => "Orson Scott Card"),
array("title"  =>  "The table of Flanders",
"author" =>  "Arturo Pérez Reverte"),
array("title"  =>  "The Endless Story",
"author" =>  "Michael Ende"),
array("title"  =>  "The Lord of the Rings",
"author" =>  "J.R.R. Tolkien")
    );


return view('list',  compact('books'));
});
```

For its part, the view 'list.blade.php' can look like this:

```
<html>
    <head>
        <title>List of books</title>
    </head>
    <body>
<h1>Listof books</h1>
        <ul>
        @forelse ($books as $book)
<li>{{ $book["title"] }} ({{ $book["author"] }})</li>
```

```
        @empty
<li>No books found</li>
        @endforelse
        </ul>
    </body>
</html>
```

1.2.2. About links to other routes

We have briefly commented in previous points that, thanks to Blade and the names on the routes, we can link one view with another in two ways: in a traditional way...

```
echo '<a href="/contact">contact</a>';
```

... or using the 'route' function followed by the name we have given to the route:

```
<a href="{{ route('path_contact') }}">contact</a>
```

In addition, through Blade there is a third way to link, using the 'url' function, which generates a complete URL to the path that we indicate:

```
<a href="{{ url('/contact') }}">contact</a>
```

1.3. Define common templates

If you want homogeneity in a website, it is usual that the header, the navigation menu or the footer are part of a template that is repeated in all the pages of the site, so that we avoid having to update all the pages before any possible change in these elements.

To create a template in Blade, we create a regular file (for example, 'template.blade.php'),in the views folder, with the general contents of the template. In those areas of the document where we are going to allow variable content depending on the view itself, we add a section called '@yield',with an associated name. Our template could be this (note that several '@yield' with different names are allowed):

```
<html>
    <head>
        <title>
        @yield('title')
        </title>
    </head>
    <body>
        <nav>
            <!-- ... Navigation menu -->
        </nav>
```

```
        @yield('content')
    </body>
</html>
```

Then, in each view in which we want to use this template, we add the '@extends' blade directive, indicating the name of the template that we are going to use. With the directive '@section', followed by the name of the section, we define the content for each of the '@yield' that have been indicated in the template. We will end each section with the '@endsection' directive. Thus, for our start page ('start.blade.php'),the content can now be this:

```
@extends('template')


@section('title',  'Home')


@section('content')
    <h1>Home page</h1>
Welcome {{ $name  }}
@endsection
```

Note, in addition, that the directive '@section' can be passed a second parameter with the content of that section, and in this case it is not necessary to close it with '@endsection'. This option is useful for content where there are no blank characters or unnecessary line breaks at the beginning or end, as in the previous example with the title (*title*) of the page.

Similarly, our view for the list of books would look like this:

```
@extends('template')


@section('title',  'List of books')


@section('content')
<h1>List of books</h1>
    <ul>

 @forelse ($books as $book)
<li>{{ $book["title"] }} ({{ $book["author"] }})</li>
    @empty
<li>No  books found</li>
    @endforelse
    </ul>
@endsection
```

1.3.1. Include views within others

It is also common to define partial contents (they are usually defined in a subfolder 'partials' within 'resources/views'),and include them in the views. For this, we will use the Blade '@include' directive.

For example, let's define a navigation menu. Suppose that menu is in the file 'resources/views/partials/nav.blade.php'.

```
<nav>
<a href="{{ route('start') }}">Start</a>
      
<a href="{{ route('book_list') }}">List of books</a>
</nav>
```

Note that, in this example, it is assumed that each of the two routes involved has been assigned the names "start" and "book_list", respectively, using the 'name' method when defining the route. Otherwise, the 'href' properties of the two links should point to '/' and '/list',respectively.

To include the menu in the above template, we can do this (and we would remove the "nav" label from the template):

```
<html>
    <head>
        <title>
        @yield('title')
        </title>
    </head>
    <body>
        @include('partials.nav')
        @yield('content')
    </body>
</html>
```

As you can see, we can use both the point and the / to indicate the folder separator in the view.

1.3.2. Structure views in folders

When the application is getting complex, several views may be necessary, and having them all in the same folder can be difficult to manage. It is usual, as we will see in later sessions, to structure the views of the 'resources/views' folder in subfolders, so that, for example, each folder refers to the views of an entity or model of the application, or to a specific controller. At the moment, in our example of the library, we are

going to locate the view 'list.blade.php' in a subfolder 'books',so that in the path that renders this view, now we must also indicate the name of the subfolder:

```
Route::get('list', function() {
    ...
    return view('books.list', compact('books'));
});
```

Right now, in our 'resources/views' folder of the library project we will have only the base template and the home page (and the 'welcome.blade.php' view, which in fact we can already delete if we want). The rest of the views will be structured in subfolders.

2.4. Views for error pages

Throughout these sessions, some actions we do will cause error pages with certain codes, such as 404 for pages not found. If we want to define the appearance and structure of these pages, it is enough to create the corresponding view in the folder 'resources/views/errors',for example, 'resources/views/errors/404.blade.php' for error 404 (we put the error code before the suffix of the view).

```
@extends('template')


@section('titulo', 'Error 404')


@section('content')
    <h1>Error</h1>
Document not  found
@endsection
```

2. Binding to CSS and JavaScript in the client

Now that we have a fairly complete view of what the Blade template engine can offer us, it's time to improve our views. For this we are going to include CSS styles. In addition, it may also be necessary in some cases to include some JavaScript library on the client side for certain processing. We will see how Laravel manages these resources.

2.1. Infrastructure for CSS and JavaScript files

In order to add CSS styles or JavaScript files to our Laravel project, the framework already provides some files where you can centralize these options.

First of all, we must bear in mind that all library dependencies on the client part are centralized in the 'package.json' file, available at the root of the project. Initially it already has a series of pre-added dependencies. Some of them are important, such as 'laravel-mix',and others we may not need and we can delete them. It is recommended to install the dependencies when we create the project, to have them available, with this command from the laravel container:

From the terminal, enter the container in interactive mode:

```
docker exec -it laravel-myapp-1 bash
```

Then, execute the command:

```
npm install
```

And Finally:

```
npm install -g npm@10.9.0
```

A 'node_modules' folder will be created at the root of the project with the dependencies installed. This folder is similar to the 'vendor' folder, also at the root of the project, but the second contains PHP dependencies (not JavaScript). Neither of these folders should be uploaded to a *git* repository, since both can be rebuilt with the corresponding *npm* or *composer* installation command. Both folders take up a lot of space.

In addition, we have the file 'resources/css/app.css', where we can define our own CSS styles, or incorporate external libraries as we will see later, using either plain CSS or Sass. For example, we can edit this file to add some own style for the body of the document:

```
body
{
    background-color: #CCC;
    font-family: Arial;
    text-align: justify;
}
```

and in template.blade

```
<html>
<head>
@vite(['resources/css/app.css', 'resources/js/app.js'])
<title>
@yield('title')
</title>
</head>
<body>
@include('partials.nav')
@yield('content')
</body>
</html>
```

Besides, we have the file 'resources/js/app.js' to include our own functions in JavaScript, or even external functionalities (through jQuery, for example).

2.2. Automatic generation of CSS and JavaScript

These two files need to be processed to generate the resulting code (CSS and JavaScript) that will be part of the application, combining all the libraries and functions that we have specified. For this, you have the file 'manifest.json' in public/build, which compile, package and minimize these CSS and JavaScript result files.

```
{
  "resources/css/app.css": {
    "file": "assets/app-BQeTYtUt.css",
    "src": "resources/css/app.css",
    "isEntry": true
  },
  "resources/js/app.js": {
    "file": "assets/app-z-Rg4TxU.js",
    "name": "app",
    "src": "resources/js/app.js",
    "isEntry": true
  }
}
```

As we can guess, from this file 'manifest.json' everything in the file 'resources/js/app.js' will be taken and an optimized file located in 'assets/app-z-Rg4TxU.js' will be generated. Similarly, the styles defined in 'resources/css/app.css' will be taken and an optimized CSS file will be generated in 'assets/app-BQeTYtUt.css'. Once installed, to generate the CSS and JavaScript we must execute this command from the root of the project:

```
npm run build
```

This will generate the files 'assets/app-BQeTYtUt.css' and 'assets/app-z-Rg4TxU.js'.

2.3. Include Bootstrap styles

One of the most used web design frameworks when developing a website is Bootstrap. In this course we will not give too many notions about it, but we will use a little bit, so that our views have a more professional look.

To include this framework in Laravel, we must include a library on the server called *ui*,which is responsible for incorporating different tools for user interface design (*UI*, *User Interface*). From the docker again, execute (docker exec -it laravel-myapp-1 bash)

```
composer require laravel/ui
```

Once the tool is added, we can use it through the 'artisan' command to incorporate Bootstrap into the project:

```
php artisan ui bootstrap
```

```
npm install sass --save-dev
```

We have a new folder resources/sass with a css file 'app.scss'

... it will add a link to the bootstrap library in the file 'resources/sass/app.scss',so that we can generate an optimized CSS file with Bootstrap included:

```
...
@import '~bootstrap/scss/bootstrap';
```

We have to move our css style from 'resources/css/app.css' to 'resources/sass/app.scss':

```
@import 'bootstrap/scss/bootstrap';
body { background-color: red; font-family: Arial; text-align: justify; }
```

And change in template.blade:

```
@vite(['resources/sass/app.scss', 'resources/js/app.js'])
```

Add a Bootstrap navbar to your project (in 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('book_list') }}">List of books</a>
      </li>
      </ul>
      </div>
      </nav>
```

In order to finally use Bootstrap, we must execute again (and every time we make any changes to css or bootstrap):

```
npm run build
```

The first instruction will download and install Bootstrap in the project (in the *node_modules* subfolder),and the second will generate the CSS and JavaScript files including the Bootstrap library. With this we will already have available the classes and styles of Bootstrap for our views.