Unidad 5. Vistas

En esta sesión nos centraremos en el concepto de vista, y en cómo asociar una Ruta a la definición de las diferentes vistas o páginas que conformarán nuestra aplicación. Para este segundo paso, haremos uso del motor de plantillas integrado con el framework Laravel, llamado Blade.

1. Vistas con Laravel

Hasta ahora las rutas que hemos definido devuelven un texto simple, a excepción de la que ya estaba creada por defecto en el proyecto, que apuntaba a la página de inicio. Si quisiéramos devolver contenido HTML, una opción (costosa) sería devolver este contenido generado desde el propio método path, a través de la sentencia 'return', pero en lugar de hacerlo desde dentro de la propia función de respuesta, lo más habitual (y recomendable) es generar una vista con el contenido HTML que se quiere enviar al cliente.

La forma general de mostrar vistas en Laravel es hacer que las rutas retornen ('return') una vista determinada. Para ello, se puede utilizar la función 'view' de Laravel, indicando el nombre de la vista que se va a generar o mostrar.

Por defecto, en la carpeta 'resources/views' tenemos disponible una vista de ejemplo llamada 'welcome.blade.php'. Es la que se utiliza como página de inicio en la ruta raíz en 'routes/web.php':

```
Ruta::get('/', función() { return
    view('bienvenido');
});
```

Nótese que no es necesario indicar la ruta o ruta al archivo de la vista, ni la extensión, ya que Laravel asume que por defecto las vistas están en la carpeta 'resources/views', con extensión '.blade.php' (que hace referencia al motor de plantillas Blade que veremos a continuación), o simplemente con extensión '.php' (en el caso de vistas simples que no utilicen Blade).

Podemos, por ejemplo, crear una vista simple dentro de esta carpeta de vistas (llamémosla 'start.blade.php'), con un Contenido HTML:

```
<html>
    <cabeza>
        <title>Iniciar</title>
    </head>
    <body>
        <h1>Página de inicio </
    h1> </
body> </html>
```

Y podemos usar esta vista como página de inicio:

```
Ruta::get('/', función() { return view('start'); });
```

1.1. Pasar valores a las vistas

Es muy habitual pasar cierta información a determinadas vistas, como por ejemplo listas de datos a mostrar, o datos de un elemento concreto. Por ejemplo, si queremos dar un mensaje de bienvenida a un nombre (supuestamente variable), deberemos almacenar el nombre en una variable en la ruta, y pasarlo a la vista al cargarla. Esto se puede hacer, por ejemplo, con el método 'with' tras generar la vista, indicando el nombre con el que vamos a asociarla a la vista, y el valor (variable) asociado a ese nombre. En nuestro caso quedaría así
este:

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

Más adelante, en la vista, tendremos que mostrar el valor de esta variable en algún lugar del código HTML. Podemos utilizar PHP tradicional para recopilar esta variable:

```
<html>
    <cabeza>
        <title>Iniciar</title>
    </head>
    <body>
        <h1> Página de inicio </
        h1> <p>Bienvenidos <?php echo $name; ?></p> </
    body> </
html>
```

Pero es más común y limpio utilizar una sintaxis Blade específica, como veremos a continuación.

Como alternativa al uso de 'with' discutido anteriormente, también podemos utilizar una matriz asociativa (asignando así varios nombres a varios valores):

```
devolver vista('inicio')->con(['nombre' => $nombre,...]);
```

También podemos utilizar esta misma matriz como segundo parámetro de la función 'view', y así prescindir de 'with':

```
retorno vista('inicio',['nombre' => $nombre,...]);
```

También podemos utilizar una función llamada 'compact' como segundo parámetro de 'view'. A esta función le pasamos únicamente el nombre de la variable y, siempre que la variable asociada se llame igual, nos establece la asociación:

```
retorno vista('inicio', compacta('nombre'));
```

La función 'compacto' admite tantos parámetros como queramos enviar a la vista por separado, cada uno con su nombre asociado.

Si simplemente vamos a devolver una vista con poca información asociada, o poca lógica interna, también podemos abreviar el código anterior llamando directamente a 'vista', en lugar de 'ruta' primero, en el archivo 'rutas/web.php', y así pasar la información asociada a la vista:

```
Ruta::vista('/', 'inicio',['nombre' => 'Nacho']);
```

1.2. Introducción al motor de plantillas Blade

Hemos comentado en el apartado anterior que el uso de Blade nos permite simplificar la sintaxis y la forma de procesar algunas cosas en nuestras vistas. Siempre que creemos el archivo de vista con la extensión '.blade.php', automáticamente nos permitirá aprovechar la sintaxis y funcionalidades de Blade en nuestras vistas.

Por ejemplo, si queremos mostrar el contenido de la variable 'nombre' que hemos pasado antes a la página de inicio, en lugar de hacer un 'echo' rudimentario en PHP, podemos utilizar una sintaxis de doble clave, proporcionada por Blade, para mostrar el contenido de esa variable. Con esto la línea que mostraba el nombre pasa de estar así...

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

...ser así:

```
Bienvenido {{ $name }}
```

> NOTA Cada vez que se renderiza una vista en Laravel, el contenido PHP generado se almacena en 'storage/framework/views', y solo se vuelve a generar después de un cambio en la vista, por lo que volver a llamar a una vista ya renderizada no afecta el rendimiento de la aplicación. Si echamos un vistazo a la vista generada solo con PHP o con blade, veremos una diferencia entre ambas: Con blade, en lugar de hacer un simple 'echo' para mostrar el valor de la variable, se utiliza una función intermedia llamada 'e'.
> Esta función evita ataques XSS (Cross Site Scripting), es decir, inyectar scripts JavaScript con la variable a mostrar.
> Es decir, no se interpreta el código. En algunos casos (sobre todo cuando generamos contenido HTML desde dentro de la expresión Blade) puede interesarnos no protegernos de estas inyecciones de código. En ese caso, la segunda clave se sustituye por una doble exclamación:

```
Bienvenido {!! $name!!}
```

Además de esta sintaxis básica para mostrar datos variables en un lugar determinado de la vista, existen ciertas directivas en Blade que nos permiten realizar comprobaciones o repeticiones.

1.2.1. Estructuras de control de flujo en Blade

Para iterar sobre un conjunto de datos (array), podemos utilizar la directiva '@foreach', con una sintaxis similar a la del foreach de PHP, pero sin las claves. Simplemente finalizamos el bucle con la directiva '@endforeach', de esta forma:

```
<ul>
@foreach($elementos como $elemento) <
li>{{ $elemento }}</li> @endforeach </
        ul>
```

En caso de que desee comprobar algo (por ejemplo, si la matriz anterior está vacía, para mostrar un mensaje relevante), utilizamos la directiva '@if', cerrada por su par correspondiente '@endif'. Opcionalmente, puede utilizar una directiva '@else' para la condición alternativa, o también '@elseif' para indicar otra condición. El ejemplo anterior podría verse así:

```
<ul>
        @if($elementos)
@foreach($elementos como $elemento) <
li>{{ $elemento }}</li> @endforeach
                @else < li>No
        hay
elementos para mostrar</li> @endif </ul>
```

También podemos comprobar si una variable está definida. En este caso, reemplazamos la directiva '@if' por '@isset', con su correspondiente clausura '@endisset'.

```
<ul>
    @isset($elementos)
@foreach($elementos como $elemento) < li>{{ $elemento }}
</li> @endforeach @else < li>No hay
                elementos para
        mostrar</
li> @endisset </ul>
```

Sin embargo, con cualquiera de estas opciones tenemos un problema: en el primer caso, si la variable '$elements' no está definida, se mostrará un error de PHP. En el segundo caso, si la variable está definida pero no contiene elementos, no se mostrará nada por pantalla. Una tercera estructura alternativa que agrupa estos dos casos (controlando mientras la variable esté definida y tenga elementos) es utilizar la directiva '@forelse' en lugar de '@foreach'. Esta política permite una cláusula adicional '@empty' para indicar qué hacer si la colección no tiene elementos o no está definida. El ejemplo anterior quedaría ahora abreviado de la siguiente manera:

```
<ul>
@forelse($elements as $element) < li>{{ $element }}</li>
@empty < li>No hay elementos para mostrar</
        li>
@endforelse </ul>
```

En este tipo de iteradores ('@foreach' o '@forelse'), disponemos de un objeto llamado '$loop', con una serie de propiedades sobre el bucle que estamos iterando, como 'index' (posición dentro del array por el que estamos recorriendo), o 'count' (total de elementos), o 'first' y 'last' (booleanos que determinan si es el primer o el último elemento, respectivamente), entre otras. Podemos ver todas las propiedades disponibles en este objeto llamando a 'var_dump':

```
<ul>
@forelse($elementos como $elemento) <
li>{{ $elemento }} {{ var_dump($bucle)}} </li>
        @empty
< li>No hay elementos para mostrar</li>
        @endforelse </
ul>
```

Si, por ejemplo, queremos determinar si es el último elemento de la lista y mostrar un mensaje o estilo especial, podemos hacer algo como esto:

```
<ul>
@forelse($elements as $element) <li>{{ $element }}
{{ $loop->last ? "Último elemento" :
</li> @empty < li>No hay elementos para mostrar</li>       "" }}


@endforelse </ul>
```

Existen otros tipos de estructuras iterativas y selectivas en Blade, como '@while', '@for' o '@switch', entre algunas otras. Puedes consultar su uso en la documentación oficial de Blade.

Apliquemos esto en nuestro ejemplo de proyecto de biblioteca . Definiremos una ruta para obtener una lista de libros. Por el momento, crearemos esta lista manualmente con una matriz (no una base de datos). En el método de enrutamiento, pasaremos la matriz a una vista llamada 'list.blade.php'. Entonces, la nueva ruta para la lista se verá así:

```
Ruta::get('list', function() { $libros =
array( array("title" =>
"El juego de Ender",
                        "author" => "Orson Scott Card"), array("title"
=> "La mesa de Flandes", "author" => "Arturo Pérez Reverte"),
array("title" => "La historia interminable", "author" =>
"Michael Ende"), array("title" => "El Señor de los Anillos",
"author" => "J. R. R. Tolkien")


        );


devolver vista('lista', compacta('libros')); });
```

Por su parte, la vista 'list.blade.php' puede verse así:

```
<html>
        <cabeza>
                <title>Lista de libros</title>
        </head>
        <body>
<h1>Lista de libros</h1>
                <ul>
                @forelse ($libros como $libro)
<li>{{ $libro["título"] }} ({{ $libro["autor"] }})</li>
```

```
                @empty
<li>No se encontraron libros </li>
                @endforelse </
                ul></
        cuerpo> </
html>
```

1.2.2. Sobre los enlaces a otras rutas

Hemos comentado brevemente en puntos anteriores que, gracias a Blade y a los nombres de las rutas, podemos enlazar una vista con otra de dos maneras: de forma tradicional...

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

... o utilizando la función 'ruta' seguida del nombre que le hayamos dado a la ruta:

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

Además, a través de Blade existe una tercera forma de enlazar, utilizando la función 'url', que genera una URL completa a la ruta que indiquemos:

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

1.3. Definir plantillas comunes

Si queremos homogeneidad en un sitio web, lo habitual es que la cabecera, el menú de navegación o el pie de página formen parte de una plantilla que se repita en todas las páginas del sitio, de forma que evitemos tener que actualizar todas las páginas ante cualquier posible cambio en estos elementos.

Para crear una plantilla en Blade, creamos un archivo normal (por ejemplo, 'template.blade.php'), en la carpeta views, con el contenido general de la plantilla. En aquellas zonas del documento donde vamos a permitir contenido variable en función de la propia vista, añadimos una sección llamada '@yield', con un nombre asociado. Nuestra plantilla podría ser ésta (nótese que se permiten varios '@yield' con nombres diferentes):

```
<html>
        <cabeza>
                <título>
                @yield('título') </título>

        </cabeza>
        <cuerpo>
                <navegación>
                        <!-- ... Menú de navegación --> </nav>
```

```
            @yield('contenido') </
        cuerpo> </
html>
```

Luego, en cada vista en la que queramos utilizar esta plantilla, añadimos la directiva blade '@extends', indicando el nombre de la plantilla que vamos a utilizar. Con la directiva '@section', seguida del nombre de la sección, definimos el contenido para cada uno de los '@yield' que se hayan indicado en la plantilla. Finalizaremos cada sección con la directiva '@endsection'. De esta forma, para nuestra página de inicio ('start.blade.php'), el contenido ahora puede ser este:

```
@extends('plantilla')


@section('título', 'Inicio')


@section('content')
        <h1>Página de inicio</h1>
Bienvenido {{ $name }}
@endsection
```

Nótese, además, que a la directiva '@section' se le puede pasar un segundo parámetro con el contenido de esa sección, y en este caso no es necesario cerrarla con '@endsection'. Esta opción es útil para contenidos en los que no hay caracteres en blanco ni saltos de línea innecesarios al principio o al final, como en el ejemplo anterior con el título (title) de la página.

De manera similar, nuestra vista de la lista de libros se vería así:

```
@extends('plantilla')


@section('title', 'Lista de libros')


@section('content') <h1>Lista
de libros</h1> <ul>


  @forelse ($libros como $libro)
<li>{{ $libro["título"] }} ({{ $libro["autor"] }})</li>
        @empty
<li>No se encontraron libros</li>
        @endforelse </
        ul>
@endsection
```

1.3.1. Incluir vistas dentro de otras

También es habitual definir contenidos parciales (normalmente se definen en una subcarpeta 'partials' dentro de 'resources/views') e incluirlos en las vistas. Para ello, utilizaremos la directiva '@include' de Blade.

Por ejemplo, definamos un menú de navegación. Supongamos que el menú está en el archivo 'resources/views/partials/nav.blade.php'.

```
<navegación>
<a href="{{ route('start') }}"> Inicio</a>    <a

href="{{ route('book_list') }}"> Lista de libros</a> </nav>
```

Tenga en cuenta que, en este ejemplo, se supone que a cada una de las dos rutas involucradas se le han asignado los nombres "start" y "book_list", respectivamente, utilizando el método 'name' al definir la ruta. De lo contrario, las propiedades 'href' de los dos enlaces deben apuntar a '/' y '/list', respectivamente.

Para incluir el menú en la plantilla anterior, podemos hacer esto (y eliminaríamos la etiqueta "nav" de la plantilla):

```
<html>
    <cabeza>
        <título>
        @yield('título') </título
        > </cabeza>
    <cuerpo>

    @include('partials.nav') @yield('contenido')
        </cuerpo> </html>
```

Como puedes ver, podemos usar tanto el punto como el / para indicar el separador de carpeta en la vista.

1.3.2. Estructura de vistas en carpetas

Cuando la aplicación se va haciendo compleja, pueden ser necesarias varias vistas, y tenerlas todas en una misma carpeta puede resultar complicado de gestionar. Lo habitual es, como veremos en sesiones posteriores, estructurar las vistas de la carpeta 'resources/views' en subcarpetas, de forma que, por ejemplo, cada carpeta haga referencia a las vistas de una entidad o modelo de la aplicación, o a un controlador concreto. De momento, en nuestro ejemplo de la librería, estamos

vamos a ubicar la vista 'list.blade.php' en una subcarpeta 'libros', por lo que en la ruta que renderiza esta vista, ahora debemos indicar también el nombre de la subcarpeta:

```
Ruta::get('lista', función() {
        ...
        devolver vista('libros.lista', compacta('libros'));
});
```

Ahora mismo, en nuestra carpeta 'resources/views' del proyecto de la librería tendremos únicamente la plantilla base y la página de inicio (y la vista 'welcome.blade.php', que de hecho ya podemos eliminar si queremos). El resto de vistas las estructuraremos en subcarpetas.

2.4. Vistas de páginas de error

A lo largo de estas sesiones, algunas acciones que realicemos provocarán páginas de error con determinados códigos, como el 404 para páginas no encontradas. Si queremos definir la apariencia y estructura de estas páginas, basta con crear la vista correspondiente en la carpeta 'resources/views/errors', por ejemplo, 'resources/views/errors/404.blade.php' para el error 404 (ponemos el código de error antes del sufijo de la vista).

```
@extends('plantilla')


@sección('título', 'Error 404')


@section('contenido')
        <h1>Error</h1>
Documento no encontrado
@sección final
```

2. Vinculación a CSS y JavaScript en el cliente

Ahora que ya tenemos una visión bastante completa de lo que nos puede ofrecer el motor de plantillas Blade, es hora de mejorar nuestras vistas. Para ello vamos a incluir estilos CSS. Además, también puede ser necesario en algunos casos incluir alguna librería JavaScript en el lado del cliente para ciertos procesamientos. Veremos cómo gestiona Laravel estos recursos.

2.1. Infraestructura para archivos CSS y JavaScript

Para poder agregar estilos CSS o archivos JavaScript a nuestro proyecto Laravel, el framework ya proporciona algunos archivos donde puedes centralizar estas opciones.

En primer lugar, debemos tener en cuenta que todas las dependencias de las librerías de la parte cliente están centralizadas en el archivo 'package.json', disponible en la raíz del proyecto. Inicialmente ya tiene una serie de dependencias pre-añadidas. Algunas de ellas son importantes, como 'laravel-mix', y otras puede que no las necesitemos y las podamos eliminar. Es recomendable instalar las dependencias cuando creamos el proyecto, para tenerlas disponibles, con este comando desde el contenedor de laravel:

Desde la terminal, ingrese al contenedor en modo interactivo:

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

Luego, ejecute el comando:

```
Instalación de npm
```

Y por último:

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

Se creará una carpeta 'node_modules' en la raíz del proyecto con las dependencias instaladas. Esta carpeta es similar a la carpeta 'vendor', también en la raíz del proyecto, pero la segunda contiene dependencias de PHP (no JavaScript). Ninguna de estas carpetas debe cargarse en un repositorio git , ya que ambas pueden reconstruirse con el comando de instalación correspondiente de npm o composer . Ambas carpetas ocupan mucho espacio.

Además, disponemos del archivo 'resources/css/app.css', donde podemos definir nuestros propios estilos CSS, o incorporar librerías externas como veremos más adelante, ya sea usando CSS plano o Sass. Por ejemplo, podemos editar este archivo para añadir algún estilo propio para el cuerpo del documento:

```
cuerpo
{
        color de fondo: #CCC; familia de
        fuentes: Arial; alineación
        del texto: justificar;
}
```

y en plantilla.blade

```
<html>
<cabeza>
@vite(['recursos/css/app.css', 'recursos/js/app.js']) <título> @yield('título') </título> </
cabeza>
<cuerpo>



@include('partials.nav')
@yield('contenido') </
cuerpo> </
html>
```

Además, tenemos el archivo 'resources/js/app.js' para incluir funciones propias en JavaScript, o incluso funcionalidades externas (a través de jQuery, por ejemplo).

2.2 Generación automática de CSS y JavaScript

Estos dos archivos deben procesarse para generar el código resultante (CSS y JavaScript) que formará parte de la aplicación, combinando todas las librerías y funciones que hemos especificado. Para ello, se dispone del archivo 'manifest.json' en public/build, que compila, empaqueta y minimiza estos archivos de resultados CSS y JavaScript.

```
{
    "recursos/css/app.css": { "archivo":
        "activos/app-BQeTYtUt.css", "origen": "recursos/
        css/app.css", "isEntry": verdadero }, "recursos/
        js/app.js": { "archivo":

    "activos/app-z-Rg4TxU.js",
        "nombre": "app", "origen": "recursos/js/app.js",
        "isEntry": verdadero


    }
}
```

Como podemos intuir, de este archivo 'manifest.json' se tomará todo lo que esté en el archivo 'resources/js/app.js' y se generará un archivo optimizado ubicado en 'assets/app-z-Rg4TxU.js'. De igual forma, se tomarán los estilos definidos en 'resources/css/app.css' y se generará un archivo CSS optimizado en 'assets/app-BQeTYtUt.css'.
Una vez instalado, para generar el CSS y JavaScript debemos ejecutar este comando desde la raíz del proyecto:

```
npm ejecuta compilación
```

Esto generará los archivos 'assets/app-BQeTYtUt.css' y 'assets/app-z-Rg4TxU.js'.

2.3. Incluir estilos Bootstrap

Uno de los frameworks de diseño web más utilizados a la hora de desarrollar un sitio web es Bootstrap. En este curso no daremos demasiadas nociones al respecto, pero sí utilizaremos un poco, para que nuestras vistas tengan un aspecto más profesional.

Para incluir este framework en Laravel, debemos incluir una librería en el servidor llamada ui, la cual se encarga de incorporar diferentes herramientas para el diseño de interfaz de usuario (UI, User Interface). Desde el docker nuevamente, ejecutamos (docker exec -it laravel-myapp-1 bash)

```
El compositor requiere laravel/ui
```

Una vez agregada la herramienta, podemos utilizarla a través del comando 'artisan' para incorporar Bootstrap al proyecto:

```
interfaz de usuario de php artisan bootstrap
```

```
npm instala sass --save-dev
```

Tenemos una nueva carpeta recursos/sass con un archivo css 'app.scss'

... agregará un enlace a la biblioteca bootstrap en el archivo 'resources/sass/app.scss', para que podamos generar un archivo CSS optimizado con Bootstrap incluido:

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

Tenemos que mover nuestro estilo CSS de 'resources/css/app.css' a 'resources/sass/app.scss':

```
@import 'bootstrap/scss/bootstrap'; cuerpo
{ color de fondo: rojo; familia de fuentes: Arial; alineación del texto: justificar; }
```

Y cambia en template.blade:

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

Agregue una barra de navegación Bootstrap a su proyecto (en nav.blade.php):

```
<nav class="navbar navbar-expand-lg navbar-dark bg-dark"> <a class="navbar-
    brand" href="#">Biblioteca</a> <button class="navbar-toggler"
    type="button" data-toggle="collapse" data-
target="#navbarNav" aria-controls="navbarNav" aria-expanded="false" aria-label="Alternar
navegación">
    <span class="navbar-toggler-icon"></button> <div class="colapsar
    navbar-
    collapse" id="navbarNav"> <ul class="navbar-nav"> <li class="nav-item active"> <a
    class="nav-link"
    href="{{ route('start') }}">Inicio </a> </li>
```

```
<li class="elemento de navegación">
<a class="nav-link" href="{{ route('book_list') }}">Lista de libros</a> </li> </ul> </div> </nav>
```

Para poder utilizar finalmente Bootstrap, debemos ejecutar nuevamente (y cada vez que hagamos algún cambio en css o bootstrap):

```
npm ejecuta compilación
```

La primera instrucción descargará e instalará Bootstrap en el proyecto (en la subcarpeta node_modules), y la segunda generará los archivos CSS y JavaScript incluyendo la librería Bootstrap. Con esto ya tendremos disponibles las clases y estilos de Bootstrap para nuestras vistas.