Vistas con Blade

Hasta ahora las rutas que hemos definido devuelven un texto simple, salvo 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 dicho contenido generado desde el propio método de la ruta, a través de la instrucción `return`, pero en lugar de hacer esto 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 devuelvan (`return`) una determinada vista. Para ello, se puede emplear la función `view` de Laravel, indicando el nombre de la vista 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`:

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

Notar que no es necesario indicar el *path* o ruta hacia el archivo de la vista, ni tampoco la extensión, puesto que Laravel asume que por defecto las vistas se encuentran en la carpeta `resources/views`, con la 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 sencilla dentro de esta carpeta de vistas (llamémosla `inicio.blade.php`), con un contenido HTML básico:

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

Y podemos utilizar esta vista como página de inicio:

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

1. Pasar valores a las vistas

Es muy habitual pasar cierta información a ciertas vistas, como por ejemplo, listados de datos a mostrar, o datos de un elemento en concreto. Por ejemplo, si queremos dar un mensaje de bienvenida a un nombre (supuestamente variable), debemos almacenar el nombre en una variable en la ruta, y pasárselo a la vista al cargarla. Esto puede hacerse, por ejemplo, con el método `with` tras generar la vista, indicando el nombre con que lo vamos a asociar a la vista, y el valor (variable) asociado a dicho nombre. En nuestro caso quedaría así:

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

Posteriormente, en la vista, deberemos mostrar el valor de esta variable en algún lugar del código HTML. Podemos emplear PHP tradicional para recoger esta variable:

```
<html>
    <head>
        <title>Inicio</title>
    </head>
    <body>
        <h1>Página de inicio</h1>
        <p>Bienvenido/a <?php echo $nombre; ?></p>
    </body>
</html>
```

Pero es más habitual y limpio emplear una sintaxis específica de Blade, como veremos a continuación.

Como alternativas al uso de `with` comentado antes, también podemos utilizar un array asociativo (asignando así varios nombres a varios valores):

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

Asimismo, podemos utilizar este mismo array como segundo parámetro de la función `view`, y prescindir así de `with`:

```
return view('inicio', ['nombre' => $nombre, ...]);
```

Y también podemos utilizar una función llamada `compact` como segundo parámetro de `view`. A esta función le pasamos únicamente el nombre de la variable que usaremos en la vista y, siempre que la variable asociada se llame igual, establece la asociación por nosotros:

```
return view('inicio', compact('nombre'));
```

La función `compact` admite tantos parámetros como datos 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 `view`, en lugar de a `get` primero, en el archivo `routes/web.php`, y le pasamos así la información asociada a la vista:

```
Route::view('/', 'inicio', ['nombre' => 'Nacho']);
```

2. Primeros pasos con el motor de plantillas Blade

Hemos comentado en el apartado anterior que el uso de Blade permite simplificar la sintaxis y la forma de procesar algunas cosas en nuestras vistas. Siempre que creemos el archivo de la vista con la extensión `.blade.php`, nos permitirá automáticamente 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 rudimentario `echo` en PHP, podemos emplear una sintaxis de dobles llaves, facilitada por Blade, para mostrar el contenido de esa variable. Con esto la línea que mostraba el nombre pasa de ser así...

```
Bienvenido/a <?php echo $nombre ?>
```

... a ser así:

```
Bienvenido/a {{ $nombre }}
```

> **NOTA** Cada vez que se renderiza una vista en Laravel, se almacena el contenido PHP generado en `storage/framework/views`, y sólo se vuelve a re-generar ante un cambio en la vista, con lo que volver a llamar a una vista ya renderizada no afecta al rendimiento de la aplicación. Si echamos un vistazo a la vista generada con PHP plano y con Blade, veremos que hay una sutil diferencia entre ambas, y es que con Blade, en lugar de hacer un simple `echo` para mostrar el valor de la variable, se utiliza una función intermedia llamada `e`, que evita ataques XSS (*Cross Site Scripting*), es decir, que se inyecten *scripts* de JavaScript con la variable a mostrar. En otras palabras, el código no se interpreta, y se muestra tal cual. En algunos casos (especialmente cuando generamos contenido HTML desde dentro de la expresión Blade) nos puede interesar que no proteja contra estas inyecciones de código. En ese caso, se sustituye la segunda llave por una doble exclamación:

```
Bienvenido/a {!! $nombre !!}
```

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

2.1. Estructuras de control de flujo en Blade

Para iterar sobre un conjunto de datos (array), podemos emplear la directiva `@foreach`, con una sintaxis similar al *foreach* de PHP, pero sin necesidad de llaves. Basta con finalizar el bucle con la directiva `@endforeach`, de este modo:

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

En el caso de querer realizar alguna comprobación (por ejemplo, si el array anterior está vacío, para mostrar un mensaje pertinente), usamos la directiva `@if`, cerrada por su correspondiente pareja `@endif`. Opcionalmente, se puede intercalar una directiva `@else` para el camino alternativo, o también `@elseif` para indicar otra condición. El ejemplo anterior podría quedar así:

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

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

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

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

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

En este tipo de iteradores (`@foreach` o `@forelse`), tenemos disponible un objeto llamado `$loop`, con una serie de propiedades sobre el bucle que estamos iterando, como por ejemplo:

- `index`: posición dentro del array por la que vamos

- `count`: total de elementos
- `first` y `last`: booleanos que determinan si es el primer o último elemento, respectivamente
- ...

Podemos ver todas las propiedades disponibles en este objeto llamando a `var_dump`:

```
<ul>
    @forelse($elementos as $elemento)
        <li>{{ $elemento }} {{ var_dump($loop) }} </li>
    @empty
        <li>No hay elementos que 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($elementos as $elemento)
        <li>{{ $elemento }}
            {{ $loop->last ? "Ultimo elemento" : "" }}
        </li>
    @empty
        <li>No hay elementos que mostrar</li>
    @endforelse
</ul>
```

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

Vamos a aplicar esto en nuestro ejemplo del proyecto *biblioteca*. Definiremos una ruta para sacar un listado de libros y, de momento, vamos a crear a mano dicho listado en el propio método de enrutamiento, y se lo pasaremos a una vista llamada `listado.blade.php`. Por un lado, la nueva ruta para el listado puede quedar así:

```
Route::get('listado', function() {
    $libros = array(
        array("titulo" => "El juego de Ender",
              "autor" => "Orson Scott Card"),
        array("titulo" => "La tabla de Flandes",
              "autor" => "Arturo Pérez Reverte"),
        array("titulo" => "La historia interminable",
              "autor" => "Michael Ende"),
        array("titulo" => "El Señor de los Anillos",
              "autor" => "J.R.R. Tolkien")
    );

    return view('listado', compact('libros'));
})->name('listado_libros');
```

Por su parte, la vista `listado.blade.php` puede quedar así:

```
<html>
    <head>
        <title>Listado de libros</title>
    </head>
    <body>
        <h1>Listado de libros</h1>
        <ul>
        @forelse ($libros as $libro)
            <li>{{ $libro["titulo"] }} ({{ $libro["autor"] }})</li>
        @empty
            <li>No se encontraron libros</li>
        @endforelse
        </ul>
    </body>
</html>
```

2.2. Sobre los enlaces a otras rutas

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

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

... o bien empleando la función `route` seguida del nombre que le hemos dado a la ruta:

```
<a href="{{ route('ruta_contacto') }}">Contacto</a>
```

Por ejemplo, podemos poner un enlace a la vista del listado de libros que hemos creado antes (y que hemos nombrado como `listado_libros` de este modo en nuestra vista de `inicio.blade.php`:

```
<p>Bienvenido/a {{ $nombre }}</p>
<p><a href="{{ url('listado_libros') }}">Listado de libros</a></p>
```

2.3. Definir plantillas comunes

A la hora de dar homogeneidad a una web, es habitual que la cabecera, el menú de navegación o el pie de página formen parte de una plantilla que se repite en todas las páginas del sitio, de modo que evitamos tener que actualizar todas las páginas ante cualquier posible cambio en estos elementos.

Para crear una plantilla en Blade, creamos un archivo normal y corriente (por ejemplo, `plantilla.blade.php`), en la carpeta de vistas, con el contenido general de la plantilla. En aquellas zonas del documento donde vamos a permitir contenido variable dependiendo de la vista en sí, añadimos una sección llamada `@yield`, con un nombre asociado. Nuestra plantilla podría ser esta (notar que se permiten varios `@yield` con diferentes nombres):

```
<html>
    <head>
        <title>
        @yield('titulo')
        </title>
    </head>
    <body>
        <nav>
            <!-- ... Menú de navegación -->
        </nav>
        @yield('contenido')
    </body>
</html>
```

Después, en cada vista en que queramos utilizar esta plantilla, añadimos la directiva `@extends` de Blade, indicando el nombre de 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`. Así, para nuestra página inicial (`inicio.blade.php`), el contenido puede ser ahora éste:

```
@extends('plantilla')

@section('titulo', 'Inicio')

@section('contenido')
    <h1>Página de inicio</h1>
    <p>Bienvenido/a {{ $nombre }}</p>
@endsection
```

Notar, 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 donde no interesen caracteres en blanco o saltos de línea innecesarios al principio o al final, como ocurre en el ejemplo anterior con el título (*title*) de la página.

Del mismo modo, nuestra vista para el listado de libros quedaría de esta forma:

```
@extends('plantilla')

@section('titulo', 'Listado de libros')

@section('contenido')
    <h1>Listado de libros</h1>
    <ul>
    @forelse ($libros as $libro)
        <li>{{ $libro["titulo"] }} ({{ $libro["autor"] }})</li>
    @empty
        <li>No se encontraron libros</li>
    @endforelse
    </ul>
@endsection
```

2.4. Incluir vistas dentro de otras

También suele ser habitual definir contenidos parciales (se suelen definir en una subcarpeta `partials` dentro de `resources/views`), e incluirlos en las vistas. Para esto, utilizaremos la directiva `@include` de Blade.

Por ejemplo, vamos a definir un menú de navegación. Supongamos que dicho menú está en el archivo `resources/views/partials/nav.blade.php`.

```
<nav>
    <a href="{{ route('inicio') }}">Inicio</a>
      
    <a href="{{ route('listado_libros') }}">Listado de libros</a>
</nav>
```

Notar que, en este ejemplo, se supone que a cada una de las dos rutas implicadas les hemos asignado los nombres "inicio" y "libros_listado", respectivamente, empleando el método `name` al definir la ruta. De lo contrario, las propiedades `href` de los dos enlaces deberían apuntar a `/` y `/listado`, respectivamente.

Para incluir el menú en la plantilla anterior, podemos hacer esto (y eliminaríamos el elemento `<nav>` de la plantilla):

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

Como puede verse, podemos utilizar tanto el punto como la barra para indicar el separador de carpeta en la vista.

2.5. Estructurar vistas en carpetas

Cuando la aplicación es algo compleja, pueden ser necesarias varias vistas, y tenerlas todas en una misma carpeta puede ser algo difícil de gestionar. Es habitual, como iremos viendo en sesiones posteriores, estructurar las vistas de la carpeta `resources/views` en subcarpetas, de modo que, por ejemplo, cada carpeta se refiera a las vistas de una entidad o modelo de la aplicación, o a un controlador específico. De momento, en nuestro ejemplo de la biblioteca, vamos a ubicar la vista `listado.blade.php` en una subcarpeta `libros`, de modo que en la ruta que renderiza esta vista, ahora deberemos indicar también el nombre de la subcarpeta:

```
Route::get('listado', function() {
    ...
    return view('libros.listado', compact('libros'));
})->name('listado_libros');
```

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

2.6. Vistas para páginas de error

A lo largo de estas sesiones, algunas acciones que hagamos provocarán páginas de error con determinados códigos, como por ejemplo 404 para páginas no encontradas. Si queremos definir el aspecto 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 (anteponemos el código de error al sufijo de la vista).

```
@extends('plantilla')

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

@section('contenido')
    <h1>Error</h1>
    <p>Documento no encontrado</p>
@endsection
```