

# Sesión 2. Rutas y vistas

Una vez vistos los primeros pasos a dar con Laravel para crear proyectos y poderlos probar, en esta sesión nos centraremos en profundizar sobre el concepto de ruta, y cómo asociarlo a la definición de las distintas vistas o páginas que compondrán nuestra aplicación. Para este segundo paso, haremos uso del motor de plantillas incorporado con el framework Laravel, llamado Blade.

## 1. Rutas con Laravel

En la sesión anterior ya hemos tenido una primera toma de contacto con lo que suponen las rutas en Laravel. Podríamos decir que existen dos tipos principales de rutas:

- Las rutas **web** (almacenadas en el archivo `routes/web.php` de la aplicación), que nos permitirán cargar distintas vistas en función de la URL que indique el cliente.
- Las rutas **API** (almacenadas en el archivo `routes/api.php`), a través de las cuales definiremos distintos servicios REST, como veremos también más adelante.

Nos vamos a centrar durante esta sesión en el primer grupo, por lo que editaremos el contenido del archivo `routes/web.php`. Veamos qué tipos de rutas podemos definir, y qué características tienen.

### 1.1. Rutas simples

Las rutas simples tienen un nombre de ruta fijo, y una función que responde a dicho nombre emitiendo una respuesta. Un ejemplo es la ruta que definimos en la sesión anterior para nuestro proyecto de ejemplo, que mostraba la fecha y hora actuales:

```
Route::get('fecha', function() {  
    return date("d/m/y h:i:s");  
});
```

### 1.2. Rutas con parámetros

Es también posible pasar **parámetros en la URL** de la ruta. Para ello, incluimos el nombre del parámetro entre llaves, y lo pasamos también a la función del segundo parámetro. Por ejemplo, si definimos una ruta para saludar al nombre que nos llega como parámetro, el código quedaría así:

```
Route::get('saludo/{nombre}', function($nombre) {  
    return "Hola, " . $nombre;  
});
```

En este caso, si el parámetro es obligatorio y no lo indicamos en la URL, nos redirigirá a una página de error 404. Para indicar que un parámetro no es obligatorio, se termina su nombre con un interrogante, y también conviene darle un valor por defecto en la función PHP de respuesta. Así modificaríamos la ruta anterior para que el nombre del usuario sea opcional y, en caso de no ponerlo, se le asigne el nombre "Invitado".

```
Route::get('saludo/{nombre?}', function($nombre = "Invitado") {  
    return "Hola, " . $nombre;  
});
```

### 1.2.1. Validación de parámetros

Algunos parámetros será preciso que sigan un determinado patrón. Por ejemplo, un identificador numérico sólo contendrá dígitos. Para asegurarnos de eso, podemos emplear el método `where` al definir la ruta. A este método le pasamos dos parámetros: el nombre del parámetro a validar, y la expresión regular que tiene que cumplir. En el caso del nombre anterior, si queremos que sólo contenga letras (mayúsculas o minúsculas), podemos hacer algo así:

```
Route::get('saludo/{nombre?}', function($nombre = "Invitado") {  
    return "Hola, " . $nombre;  
})->where('nombre', "[A-Za-z]+");
```

En caso de que la ruta no cumpla el patrón, se obtendrá una página de error. Más adelante se explicará cómo podemos personalizar estas páginas de error.

### 1.3. Rutas con nombre o *named routes*

En ocasiones puede ser conveniente asociar un nombre a una ruta. Especialmente, cuando esa ruta va a formar parte de un enlace en alguna página de nuestro sitio, ya que en un futuro la ruta podría cambiar, y de este modo evitamos tener que actualizar los enlaces con el nuevo nombre.

Para ello, al definir la ruta, le asociamos con la función `name` el nombre que queramos. Por ejemplo:

```
Route::get('contacto', function() {  
    return "Página de contacto";  
})->name('ruta_contacto');
```

Ahora, si queremos definir un enlace a esta ruta en cualquier parte, basta con emplear la función `route` de Laravel, indicando el nombre que le hemos asignado a esta ruta. Por lo tanto, en lugar de poner esto:

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

Podemos hacer algo como esto otro, tal y como veremos a continuación cuando usemos el motor de plantillas Blade:

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

De este modo, ante futuros cambios en las rutas, sólo deberemos cambiar la URL en `Route::get`.

## 1.4. Combinación de elementos en rutas

Podemos combinar varias cláusulas `where` en una ruta para validar distintos parámetros que pueda tener, y también enlazar estas llamadas con una a la función `name` para nombrar la ruta. Por ejemplo, la siguiente ruta espera recibir un nombre con caracteres, y un *id* numérico, ambos con valores por defecto:

```
Route::get('saludo/{nombre?}/{id?}',  
function($nombre="Invitado", $id=0)  
{  
    return "Hola $nombre, tu código es el $id";  
})->where('nombre', "[A-Za-z]+")  
->where('id', "[0-9]+")  
->name('saludo');
```

Si accedemos a cada una de las siguientes URLs, obtendremos cada una de las respuestas indicadas:

URL	Respuesta
/saludo	Hola Invitado, tu código es el 0
/saludo/Nacho	Hola Nacho, tu código es el 0
/saludo/Nacho/3	Hola Nacho, tu código es el 3
/saludo/3	Error 404 (URL incorrecta)

Notar que el último caso es incorrecto. No podemos especificar un *id* sin haber especificado un nombre delante, porque incumple el patrón de la URL. Se puede dejar un parámetro omitido, siempre y cuando los posteriores también lo estén.

En este punto, puedes realizar el [Ejercicio 1](#) de los propuestos al final de la sesión.

## 2. Vistas con Laravel

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');  
});
```

## 2.1. 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 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 `route` primero, en el archivo `routes/web.php`, y le pasamos así la información asociada a la vista:

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

## 2.2. 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 creamos 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.

### 2.2.1. 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), o `count` (total de elementos), o `first` y `last` (booleanos que determinan si es el primer o último elemento, respectivamente), entre otros. 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`, y algunos otros. 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'));
});
```

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>
```

### 2.2.2. 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>
```

Además, mediante Blade existe una tercera forma de enlazar, empleando la función `url`, que genera una URL completa hasta la ruta que indiquemos:

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

## 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>
  Bienvenido/a {{ $nombre }}
@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
```

### 2.3.1. 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>  
    &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~  
    <a href="{ route('libros_listado') }">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 menú `<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.

### 2.3.2. 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'));
});
```

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.

### 2.4. 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>
    Documento no encontrado
@endsection
```

En este punto, puedes realizar el [Ejercicio 2](#) de los propuestos al final de la sesión.

## 3. Enlazando con CSS y JavaScript en el cliente

Ahora que ya tenemos una visión bastante completa de lo que el motor de plantillas Blade puede ofrecernos, llega el momento de terminar de perfilar nuestras vistas. Hasta ahora no hemos hablado nada de estilos CSS, y eso es algo que toda vista que se precie debe incluir. 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.

### 3.1. Infraestructura para archivos CSS y JavaScript

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

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

```
npm install
```

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 esta última contiene dependencias PHP (no JavaScript). Ninguna de estas carpetas debe subirse a un repositorio *git*, ya que ambas pueden reconstruirse con el correspondiente comando de instalación de *npm* o de *composer*, según el caso, y además, pueden ocupar mucho espacio.

Además, por un lado, tenemos el archivo `resources/css/app.css`, o bien `resources/sass/app.scss` (dependiendo de la versión de Laravel que usemos), donde podemos definir estilos CSS propios, o incorporar librerías externas como veremos después, utilizando o bien CSS

plano o bien Sass. Por ejemplo, podemos editar este archivo para añadir algún estilo propio para el cuerpo del documento:

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

Por otro lado, tenemos el archivo `resources/js/app.js` para incluir nuestras propias funciones en JavaScript, o incluso funcionalidades externas (a través de jQuery, por ejemplo).

### 3.2. Generación automática de CSS y JavaScript

Estos dos archivos necesitan ser procesados para generar el código resultante (CSS y JavaScript) que formará parte de la aplicación, aunando todas las librerías y funciones que hayamos especificado. Para esto, se tiene el archivo `webpack.mix.js` en la raíz del proyecto, que emplea la herramienta WebPack para compilar, empaquetar y minificar estos archivos resultado CSS y JavaScript.

```
mix.js('resources/js/app.js', 'public/js')
    .css('resources/sass/app.scss', 'public/css');
```

Como podemos intuir, desde este archivo `webpack.mix.js` se tomará todo lo que hay en el archivo `resources/js/app.js` y se generará un archivo optimizado ubicado en `public/js/app.js`. De forma similar, se tomarán los estilos definidos en `resources/sass/app.scss` o en `resources/css/app.css` (dependiendo de la versión de Laravel) y se generará un archivo CSS optimizado en `public/css/app.css`. Para desencadenar este proceso, Laravel y WebPack se valen de la librería `laravel-mix`, incluida en el archivo `package.json`. Por eso es importante esta librería, y por eso debemos dejarla instalada previamente con el comando `npm install` que hemos explicado antes. Una vez instalada, para generar los CSS y JavaScript debemos ejecutar este comando desde la raíz del proyecto:

```
npm run dev
```

Esto generará los archivos `public/css/app.css` y `public/js/app.js`, y después ya podremos añadir estos archivos en nuestras vistas, con algo como esto, respectivamente:

```
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="/css/app.css">
    <script type="text/javascript" src="/js/app.js">
    </script>
  ...
```

### 3.3. Incluir estilos Bootstrap

Uno de los frameworks de diseño web más utilizados a la hora de elaborar una web es [Bootstrap](#). En este curso no vamos a dar demasiadas nociones sobre él, pero sí utilizaremos algunas pinceladas para que nuestras vistas tengan un aspecto más profesional.

Para incluir este framework en Laravel, debemos incluir una librería en el servidor llamada *ui*, que se encarga de incorporar distintas herramientas para diseño de interfaces de usuario (*UI*, *User Interface*).

```
composer require laravel/ui
```

Una vez añadida la herramienta, la podemos emplear a través del comando `artisan` para incorporar Bootstrap al proyecto:

```
php artisan ui bootstrap
```

Esto incorporará Bootstrap al archivo `package.json`, en la sección de dependencias...

```
"devDependencies": {
  ...
  "bootstrap": "^4.0.0",
  ...
}
```

... y también añadirá un enlace a dicha librería en el archivo `resources/sass/app.scss`, para que podamos generar un archivo CSS optimizado con Bootstrap incluido:

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

Para poder utilizar finalmente Bootstrap, debemos ejecutar nuevamente las instrucciones:



```
npm install
npm run dev
```

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 en ellos la librería Bootstrap. Con esto ya tendremos disponibles las clases y estilos de Bootstrap para nuestras vistas.

En este punto, ya puedes realizar el [Ejercicio 3](#) de los propuestos al final de la sesión.

## 4. Ejercicios propuestos

### Ejercicio 1

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

- Añade una nueva ruta parametrizada a `posts/{id}`, de manera que el parámetro `id` sea numérico (es decir, sólo contenga dígitos del 0 al 9) y obligatorio. Haz que la ruta devuelva el mensaje "Ficha del post XXXX", siendo XXXX el id que haya recibido como parámetro.
- Pon un nombre a las tres rutas que hay definidas hasta ahora: a la página de inicio ponle el nombre "inicio", a la del listado la llamaremos "posts\_listado" y a la de ficha que acabas de crear, la llamaremos "posts\_ficha".

### Ejercicio 2

Continuamos con el proyecto **blog**. En este caso vamos a definir una plantilla y una serie de vistas que la utilicen.

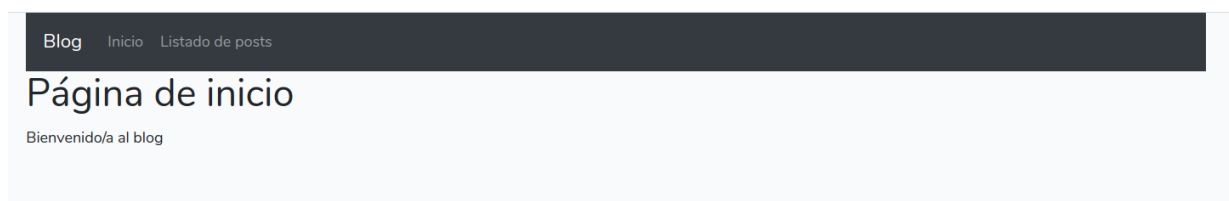
- Comenzaremos definiendo una plantilla llamada `plantilla.blade.php` en la carpeta de vistas del proyecto ( `resources/views` ). Define una cabecera con una sección `yield` para el título, y otra para el contenido de la página, como la del ejemplo que hemos visto anteriormente.
- Define en un archivo aparte en la subcarpeta `partials`, llamado `nav.blade.php`, una barra de navegación que nos permita acceder a estas direcciones de momento:
  - Página de inicio
  - Listado de posts
- Incluye la barra de navegación en la plantilla base que has definido antes
- A partir de la plantilla base, define otras dos vistas en una subcarpeta `posts`, llamadas `posts/listado.blade.php` y `posts/ficha.blade.php`. Como título de cada página pon un breve texto de lo que son (por ejemplo, "Listado posts" y "Ficha post"), y como contenido de momento deja un encabezado `h1` que indique la página en la que estamos: "Listado de posts" o "Ficha del post XXXX", donde XXXX será el identificador del post que habremos pasado por la URL (y que deberás

pasar a la vista). Haz que las rutas correspondientes de `routes/web.php` que ya has definido rendericen estas vistas en lugar de devolver texto plano.

## Ejercicio 3

Sobre el mismo proyecto **blog** que venimos desarrollando, incorpora ahora los estilos de Bootstrap siguiendo los pasos vistos en estos apuntes:

- Instala con *composer* la librería `laravel/ui`, y utilízala para incorporar Bootstrap al proyecto
- Descarga Bootstrap con `npm install`, y actualiza los archivos CSS y JavaScript con `npm run dev`
- Incorpora los estilos `/css/app.css` a la plantilla base del proyecto, para que los utilicen todas las vistas que heredan de ella.
- Edita el archivo `partials/nav.blade.php` para modificar la barra de navegación y dejarla con un estilo particular de Bootstrap. Puedes consultar [esta página](#) para tomar ideas de algunos diseños que puedes aplicar en la barra de navegación.
- Renombra el archivo `welcome.blade.php` a `inicio.blade.php` y cámbialo para que también herede de la plantilla base. Añade algún texto introductorio como contenido. Puede quedarte más o menos así (la barra de navegación superior puede variar en función del estilo que hayas querido darle)



## ¿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 la sesión anterior. Renombra el archivo comprimido a `blog_02.zip`.