

Laravel. Motor de plantillas Blade

A pesar de que Laravel se clasifica como un framework PHP para el desarrollo en el lado del servidor (backend), integra de manera integral una herramienta de diseño para el lado del cliente (frontend). Esto se debe a que todo framework destinado al desarrollo web debe ser completo (fullstack). El motor de plantillas Blade se presenta como una herramienta robusta para la construcción de vistas HTML, guiándose por buenas prácticas de programación que garantizan la modularidad, mantenibilidad y escalabilidad de los proyectos Laravel. Blade nos habilita para llevar a cabo las siguientes operaciones:

- Aprovechar al máximo todas las capacidades nativas de HTML 5.
- Recuperar los datos provenientes del controlador y recorrerlos, realizando su procesamiento o representación directa, especialmente cuando se trata de arreglos.
- Utilizar estructuras de control para elegir los datos que se exhibirán.
- Diseñar diseños genéricos que se pueden reutilizar en otras plantillas, evitando la redundancia de elementos compartidos como encabezados, pies de página, etc.
- Configurar secciones que son susceptibles de ser incorporadas en distintos diseños.

Creación de una vista

Se han desarrollado vistas de manera provisional en las que únicamente se ha integrado el código esencial para presentar los resultados, prescindiendo de cualquier estructura HTML. Nuestro próximo paso consistirá en la creación de una vista destinada a la página de inicio, en la cual ya se establecerá una estructura HTML. Dicha vista se denominará "inicio.blade.php":

```
resources > views > inicio.blade.php
1  <!DOCTYPE html>
2  <html lang="{{ str_replace('_', '-', app()->getLocale()) }}">
3  <head>
4      <meta charset="UTF-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <title>ProyectoClase</title>
8  </head>
9  <body>
10     {{ $titulo }}
11 </body>
12 </html>
```

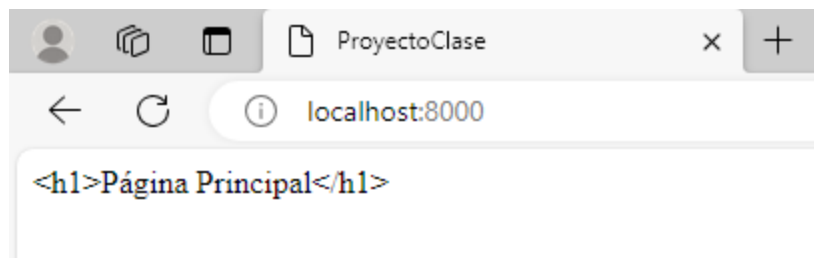
Se trata de una página HTML genérica en la que se han incorporado exclusivamente dos elementos:

- **Idioma (lang):** se capturará automáticamente el idioma de la aplicación.
- **Título:** se presentará como encabezado el texto que se reciba desde el controlador.

Posteriormente, procedemos a ajustar la función "raiz()" en nuestro controlador HolaControlador para que visualice la vista recién creada y transmita el valor del título:

```
public function raiz(){
    return view('inicio', ['titulo' => '<h1>Página Principal</h1>']);
}
```

Si ahora abrimos la página de inicio de nuestra aplicación obtenemos este resultado:



Como se observa, no se presenta un título (h1), sino que las etiquetas se visualizan como texto. Esta situación se origina debido a que Laravel reemplaza automáticamente los delimitadores de las etiquetas (<, >) por sus entidades correspondientes (<, >).

La solución más directa implica no incluir las etiquetas dentro de los parámetros y, en cambio, insertarlas directamente en la plantilla:

```
<body>
|   <h1>{{ $titulo }}</h1>
| </body>
```

Sin embargo, si es necesario enviar las etiquetas dentro de los parámetros, no hay inconveniente. Simplemente, se requiere cambiar el delimitador utilizado en la vista de {{ }} a {!! !!} para que Laravel no sustituya automáticamente los caracteres especiales por sus entidades:



Creación y utilización de layouts

La noción de utilizar layouts surge de la observación de que todas las páginas comparten elementos comunes, tales como:

- La declaración del tipo de documento (DOCTYPE).
- La etiqueta HTML con el atributo lang.
- La sección head que contiene elementos compartidos y otros que pueden variar entre páginas, como el título.
- La sección body que alberga los contenidos específicos de cada página.
- El cierre de la etiqueta HTML.

Cuando se trata de un sitio con pocas páginas, replicar esta estructura en cada una resulta sencillo. Sin embargo, a medida que el sitio crece, la tarea se vuelve más compleja, sobre todo porque cualquier cambio que afecte a todas las páginas requerirá repetir la modificación en cada una de ellas.

Para evitar este problema, podemos crear una plantilla (layout) que contenga todos esos elementos comunes y emplearla en cada vista. De esta manera, si necesitamos modificar alguno de esos elementos, solo será necesario hacerlo en la plantilla. En dicha plantilla, podemos utilizar valores obtenidos mediante funciones y helpers, delimitados por `{{ }}`, y también podemos definir zonas especiales cuyo contenido será establecido en cada vista que utilice la plantilla. Esto se logra mediante la directiva de Blade `@yield`, que incluirá como argumento un nombre único utilizado en la vista para definir el contenido que se desea incluir en esa zona. Ejemplo: `@yield('titulo')`

En primer lugar, vamos a crear la carpeta “layouts” dentro (resources/views), y en dicha carpeta creamos la plantilla “main-layout.blade.php”:

```
resources > views > layouts > main-layout.blade.php
1  <!DOCTYPE html>
2  <html lang="{{ str_replace('_', '-', app()->getLocale()) }}">
3  <head>
4      <meta charset="UTF-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <meta name="csrf-token" content="{{ csrf_token() }}" >
8      <title>@yield('page-title')</title>
9      <link rel="stylesheet"
10         href="https://fonts.googleapis.com/icon?family=Material+Icons">
11      <link rel="stylesheet" href="{{ asset('css/styles.css') }}">
12 </head>
13 <body>
14     <div class="container-fluid">
15         @yield('content-area')
16     </div>
17 </body>
18 </html>
```

En este caso se han definido mediante yield dos zonas cuyo contenido deberá ser establecido por las vistas que hagan uso de esta plantilla: page-title y content-area.

A continuación, modificamos la vista “inicio.blade.php” para que utilice el layout:

```
<body>
    @extends ('layouts.main-layout')
    @section ('page-title', 'Bienvenidos')
    @section ('content-area')
        {!! $titulo !!}
    @endsection
</body>
```

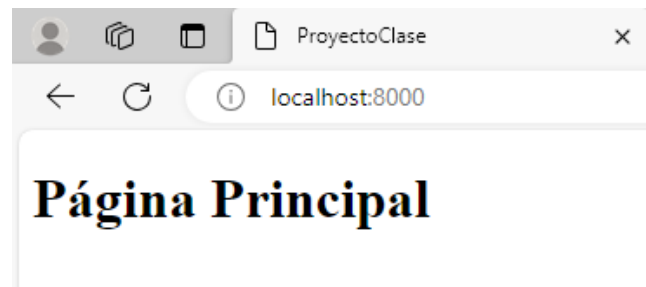
Todas las directivas de Blade inician con @, y en este contexto, estamos haciendo uso de dos de ellas:

- **@extends:** indica que la vista se extenderá a partir de la plantilla "main.layout" (sin incluir la extensión "blade.php"), la cual se encuentra en la carpeta "layouts" (considerando siempre como punto de partida "resources/views").

- **@section:** posibilita definir el contenido de una sección creada mediante la directiva @yield.

En el código, podemos apreciar las dos formas de establecer el contenido de una sección: la primera se realiza como segundo parámetro de @section, y la segunda se configura como el contenido del bloque comprendido entre @section y @endsection.

Si abrimos la página de inicio de nuestra aplicación el resultado no ha cambiado:



Pero lo interesante está en el código fuente de esa página:

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta http-equiv="X-UA-Compatible" content="IE=edge">
6   <meta name="viewport" content="width=device-width, initial-scale=1.0">
7   <title>ProyectoClase</title>
8 </head>
9 <body>
10
11
12
13 </body>
14 </html>
15 <!DOCTYPE html>
16 <html lang="en">
17 <head>
18   <meta charset="UTF-8">
19   <meta http-equiv="X-UA-Compatible" content="IE=edge">
20   <meta name="viewport" content="width=device-width, initial-scale=1.0">
21   <meta name="csrf-token" content="2fDdyTiV3IOuFh1lq7JB5C5nKpFkLvWw2sG2qh4S">
22   <title>Bienvenidos</title>
23   <link rel="stylesheet"
24     href="https://fonts.googleapis.com/icon?family=Material+Icons">
25   <link rel="stylesheet" href="http://localhost:8000/css/styles.css">
26 </head>
27 <body>
28   <div class="container-fluid">
29
30     <h1>Página Principal</h1>
31
32   </div>
33 </body>
34 </html>

```

En primer lugar, observamos que ambas secciones han sido completadas con la información establecida en la vista:

- **page-title:** la palabra "Bienvenidos".
- **content-area:** el contenido de la sección, que a su vez utiliza el título establecido en el controlador.

Además, se han empleado diversos helpers para configurar algunos valores de la página:

- **lang:** como ya habíamos visto, este valor se obtiene mediante `app()->getLocale()`.
- **csrf-token:** este valor es generado por Laravel como un mecanismo de seguridad para verificar que las solicitudes provengan del propio sitio: `csrf-token()`.
- **assets:** hace referencia a archivos externos que serán utilizados por nuestra aplicación, como hojas de estilo (css) y scripts (js), y se incluyen mediante `asset('ruta')`.

Es importante destacar que los recursos del usuario se almacenan en la carpeta "public", y todas las rutas del helper 'asset' se expresan desde esa carpeta. Por otro lado, los recursos de Laravel se guardan en la carpeta "resources" y deben ser procesados (a través de herramientas como Webpack, Vite).

Para comenzar, es necesario contar con la última versión de Node, la cual se puede obtener en <https://nodejs.org/es/>

Una vez instalado node ejecutamos:

npm install

npm run dev

Si además queremos que se actualicen automáticamente los cambios en dichos assets, tendremos que configurar nuestro archivo "package.json":

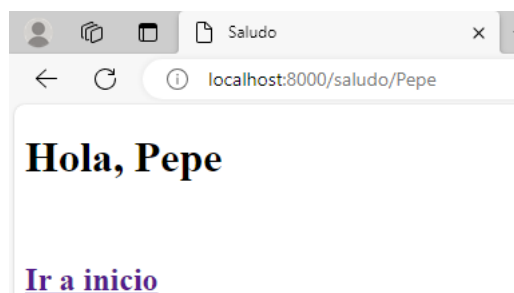
```
"scripts": {  
  "dev": "vite",  
  "build": "vite build",  
  "watch": "vite build --watch"  
},
```

Y a continuación, ejecutar el comando: **npm run watch // npm run watch-poll**

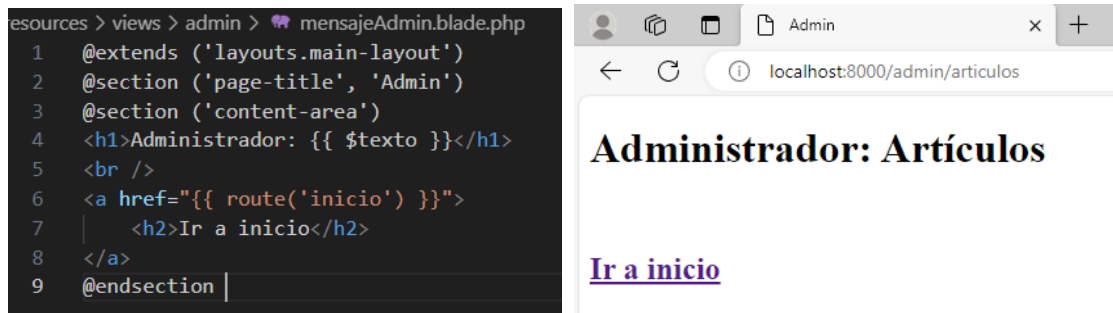
Reutilización de layouts

La ventaja de utilizar layouts es que podemos reutilizarlos en distintas páginas. Por ejemplo, vamos a hacerlo con nuestra página saludo que utiliza la vista: "mensaje.blade.php"

```
@extends ('layouts.main-layout')  
@section ('page-title', 'Saludo')  
@section ('content-area')  
<h1>{{ $texto }}</h1>  
<br />  
<a href="{{ route('inicio') }}">  
  <h2>Ir a inicio</h2>  
</a>  
@endsection |
```



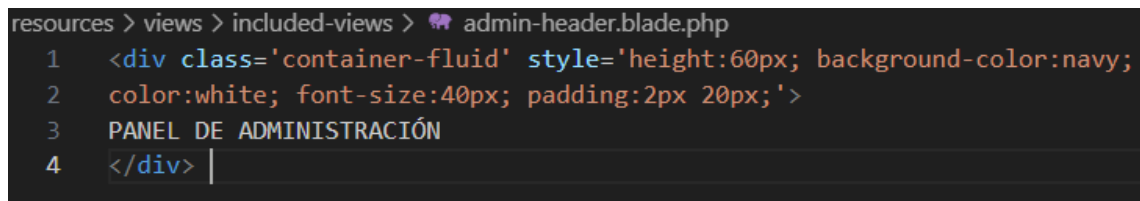
Y lo mismo podemos hacer con la vista de administrador “mensajeAdmin.blade.php”:



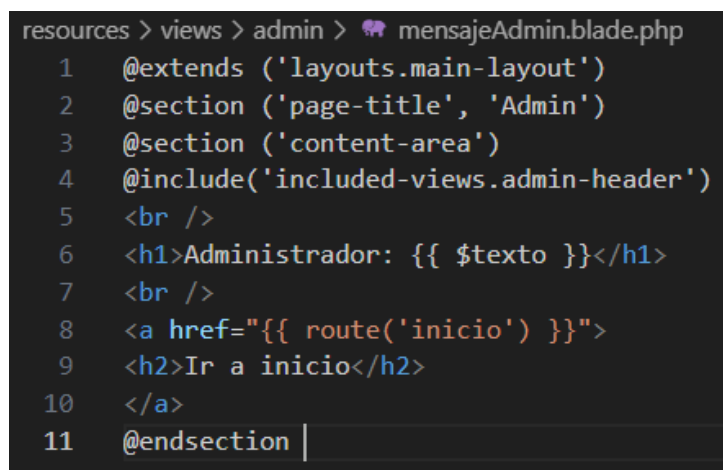
Vistas parciales

Hemos explorado la creación de layouts o plantillas que podemos emplear para establecer la estructura de nuestro sitio web. Sin embargo, en ocasiones, contamos con componentes como cabeceras, barras de navegación, paneles laterales, entre otros, que deben mostrarse solo en algunas páginas. En estos casos, no es apropiado incluirlos en una plantilla general, sino que es más conveniente crearlos de forma independiente para utilizarlos específicamente donde sean necesarios.

Para ilustrar cómo funcionan estas vistas parciales, vamos a desarrollar una cabecera para la página de administración llamada "admin-header.blade.php". Para mantener estos componentes bien organizados, los ubicaremos en la carpeta "included-views" dentro de "resources/views".



Una vez creada la vista parcial podemos incluirla en cualquier vista o layout donde se necesite utilizando la directiva @include. Por ejemplo, en la vista “mensajeAdmin.blade.php”:





Estructuras condicionales

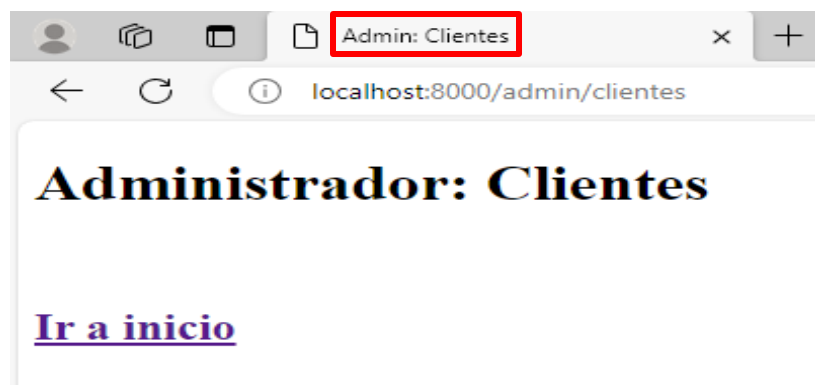
Blade permite el uso de condiciones para mostrar u ocultar contenidos en función del valor de alguna variable procedente del controlador.

@if() ... @endif

La directiva @if en Blade opera de manera similar a un condicional if en PHP. Es decir, evalúa la condición especificada y, en caso de ser verdadera, ejecuta el código que sigue antes de la directiva @endif. Además, Blade cuenta con las directivas @elseif() y @else, cuyo funcionamiento es análogo al que tienen en PHP. La estructura general es la siguiente:

```
@if () ... [@elseif() ...] [@else ...] @endif
```

Para verificar el funcionamiento de esta directiva, vamos a modificar la vista del administrador, "mensajeAdmin.blade.php", de modo que el título de la página dependa del texto recibido.



@switch() ... @endswitch

También funciona de forma idéntica al switch de PHP, e incluye las directivas @switch(), @case(), @break, @default y @endswitch. La vista "mensajeAdmin.blade.php" utilizando switch quedaría así:

```
resources > views > admin > mensajeAdmin.blade.php
1  @extends ('layouts.main-layout')
2
3  @switch ($texto)
4      @case ("Articulos")
5          @section ('page-title', 'Admin: Articulos')
6          @break
7      @case ("Clientes")
8          @section ('page-title', 'Admin: Clientes')
9          @break
10     @case ("Facturas")
11         @section ('page-title', 'Admin: Facturas')
12         @break
13     @default
14         @section ('page-title', 'Admin')
15 @endswitch
16
17 @section ('content-area')
18 <h1>Administrador: {{ $texto }}</h1>
19 <br />
20 <a href="{{ route('inicio') }}">
21     <h2>Ir a inicio</h2>
22 </a>
23 @endsection
```

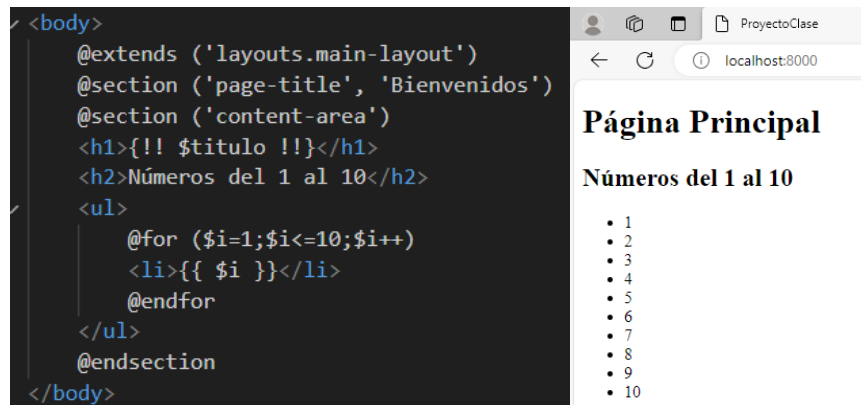


Bucles

Como sabemos, los bucles permiten repetir una operación, ya sea un número fijo de veces o dependiendo de una determinada condición, y su principal uso en una vista va a ser renderizar los datos de un array en una vista.

@for() ... @endfor

Se trata del bucle más sencillo y sirve para repetir una operación un número de veces predefinido. Vamos a probarlo en la vista "inicio.blade.php" creando una lista con valores de 1 a 10:



The screenshot shows a code editor on the left and a web browser on the right. The code editor displays the following Blade template code:

```
<body>
    @extends ('layouts.main-layout')
    @section ('page-title', 'Bienvenidos')
    @section ('content-area')
        <h1>{!! $titulo !!}</h1>
        <h2>Números del 1 al 10</h2>
        <ul>
            @for ($i=1;$i<=10;$i++)
                <li>{{ $i }}</li>
            @endfor
        </ul>
    @endsection
</body>
```

The web browser on the right shows the rendered output of the template. The title is "Página Principal" and the content is "Números del 1 al 10" followed by a bulleted list of numbers from 1 to 10.

Si queremos mostrar sólo los pares tenemos varias alternativas, la primera sería partir de 2 e incrementar el valor de la variable en 2 unidades cada iteración:

```
@for ($i=1;$i<=10;$i+=2)
    <li>{{ $i }}</li>
@endfor
```

También podemos utilizar las directivas @continue y @break que funcionan igual que en PHP.

```
@for ($i=2;$i<=10;$i+=2)
    @if ($i%2 != 0)
        @continue
    @endif
    <li>{{ $i }}</li>
@endfor
```

@while() ... @endwhile

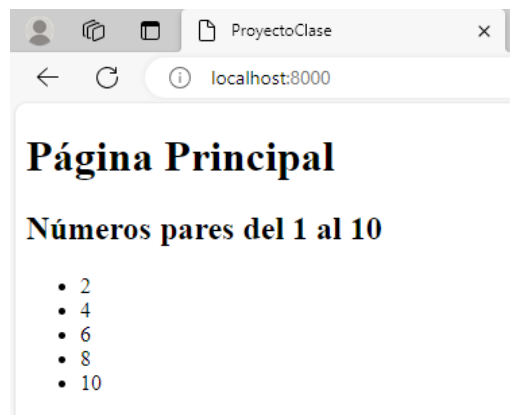
Este bucle se repite mientras una condición sea cierta, de manera que no es necesario conocer de antemano el número de repeticiones. Siguiendo con el ejemplo anterior de los números pares:

```
<?php $i = 2; ?>
<ul>
    @while ($i<=10)
        <li>{{ $i }}</li>
        <?php $i += 2; ?>
    @endwhile
```

Como vemos en el ejemplo, también podemos insertar bloques de código PHP, y se puede hacer con los delimitadores habituales, o utilizando las directivas de Blade `@php` y `@endphp`.

```
@php
$i = 2;
@endphp
<ul>
    @while ($i<=10) <li>{{ $i }}</li>
        @php
            $i += 2;
        @endphp
    @endwhile
</ul>
```

La salida para todos los casos sería la misma, la siguiente:



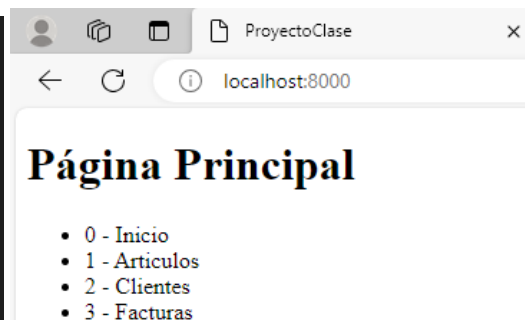
`@foreach() ... @endforeach`

Permite recorrer los elementos de un array y funciona de forma idéntica al `foreach` PHP. Para probar su funcionamiento vamos a crear un array en el método “`raiz()`” del controlador “`HolaControlador`” y se lo vamos a pasar a la vista.

```
public function raiz() {
    $titulo = "<h1>Página Principal</h1>";
    $paginas = ['Inicio', 'Articulos', 'Clientes', 'Facturas'];
    return view('inicio')->with([
        'titulo' => $titulo,
        'paginas' => $paginas,
    ]);
}
```

Se puede obtener la clave de cada elemento, como en el `foreach` de PHP:

```
<body>
    @extends ('layouts.main-layout')
    @section ('page-title', 'Bienvenidos')
    @section ('content-area')
        <h1>{{ $titulo }}</h1>
        <ul>
            @foreach ($paginas as $key=>$pagina)
                <li>{{ $key }} - {{ $pagina }}</li>
            @endforeach
        </ul>
    @endsection
```



Objeto Loop

Cuando utilizamos el bucle `@foreach` de Blade, tenemos acceso al objeto loop, el cual contiene propiedades relacionadas con las iteraciones:

- **\$loop->index:** indica el número de la iteración actual, comenzando desde 0.
- **\$loop->iteration:** indica el número de la iteración actual, comenzando desde 1.
- **\$loop->remaining:** indica la cantidad de iteraciones que aún faltan para finalizar.
- **\$loop->count:** indica el número total de elementos sobre los cuales se está iterando.
- **\$loop->first:** indica si el elemento actual es el primero en la iteración (true, false).
- **\$loop->last:** indica si el elemento actual es el último en la iteración (true, false).
- **\$loop->depth:** en el caso de bucles anidados, indica el nivel de profundidad del bucle actual.
- **\$loop->parent:** en el caso de bucles anidados, hace referencia al bucle inmediatamente superior y permite acceder a las propiedades de ese bucle. Por ejemplo: `$loop->parent->first`

Para probar el funcionamiento de `$loop`, vamos a dar un formato especial al primer y último elementos del array "`$paginas`":



The image shows a side-by-side comparison of a Blade template and its rendered output. On the left, the Blade code is displayed in a dark-themed editor. It includes an `@extends` directive for 'layouts.main-layout', an `@section` for 'page-title' with the value 'Bienvenidos', and another `@section` for 'content-area'. Inside the content section, there is an `<h1>` tag using `!! $titulo !!` and a `` tag. The `` contains an `@foreach` loop over `$paginas as $pagina`. Inside the loop, there is an `@if` condition `($loop->first || $loop->last)` that wraps the `` tag with `` and `` tags. An `@else` block follows. The loop ends with `@endif` and `@endforeach`. The section ends with `@endsection`. On the right, a browser window shows the rendered page. The title is 'Página Principal' and the list contains four items: 'Inicio' (bold), 'Articulos', 'Clientes', and 'Facturas' (bold). The browser's address bar shows 'localhost:8000'.

```
@extends ('layouts.main-layout')
@section ('page-title', 'Bienvenidos')
@section ('content-area')
<h1>{!! $titulo !!</h1>
<ul>
    @foreach ($paginas as $pagina)
        @if ($loop->first || $loop->last)
            <li><b>{!! $pagina !!</b></li>
        @else
            <li>{!! $pagina !!</li>
        @endif
    @endforeach
</ul>
@endsection
```

Página Principal

- Inicio
- Articulos
- Clientes
- Facturas

@forelse() ... @empty ... @endforelse

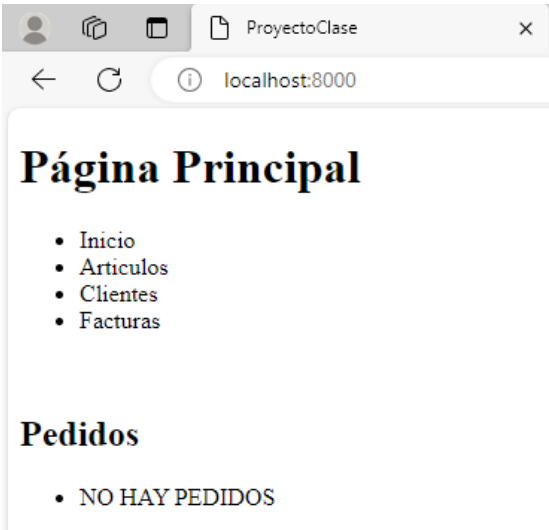
Cuando nos enfrentamos a la tarea de recorrer un array, es posible encontrarnos con la situación de que dicho array esté vacío. Si lo procesamos mediante un bucle `foreach`, esto no generará problemas, pero tampoco mostrará ningún resultado. El bucle `forelse` proporciona una solución, permitiendo especificar qué hacer o mostrar cuando el array está vacío, mediante la cláusula `empty`.

Para ilustrar este concepto, comenzamos creando un array vacío en el método "`raiz()`" del controlador "`HolaControlador`":

```
public function raiz() {
    $titulo = "<h1>Página Principal</h1>";
    $paginas = ['Inicio', 'Articulos', 'Clientes', 'Facturas'];
    $pedidos = [];
    return view('inicio')->with([
        'titulo' => $titulo,
        'paginas' => $paginas,
        'pedidos' => $pedidos,
    ]);
}
```

Y a continuación, lo procesamos en la vista “inicio.blade.php” utilizando forelse:

```
<body>
@extends ('layouts.main-layout')
@section ('page-title', 'Bienvenidos')
@section ('content-area')
<h1>{!! $titulo !!}</h1>
<ul>
    @foreach ($paginas as $pagina)
    <li>{!! $pagina !!}</li>
    @endforeach
</ul>
<br />
<h2>Pedidos</h2>
<ul>
    @forelse ($pedidos as $pedido)
    <li>{!! $pedido !!}</li>
    @empty
    <li>NO HAY PEDIDOS</li>
    @endforelse
</ul>
@endsection
```



Arrays bidimensionales

Cuando estamos involucrados en el procesamiento de datos, es común encontrarnos con la necesidad de trabajar con arrays de elementos que, a su vez, son arrays, lo que se conoce como arrays bidimensionales. Este escenario es especialmente frecuente cuando manejamos bases de datos relacionales, como tablas.

Por ejemplo, un array de pedidos podría contener varios pedidos, y cada uno de ellos podría incluir información como número de pedido, producto e importe.

```
public function raiz() {
    $titulo = "<h1>Página Principal</h1>";
    $paginas = ['Inicio', 'Articulos', 'Clientes', 'Facturas'];
    $pedidos = [
        ['num' => 101, 'producto' => 'Camisa', 'precio' => 49.95],
        ['num' => 102, 'producto' => 'Pantalón', 'precio' => 79.95],
        ['num' => 103, 'producto' => 'Sudadera', 'precio' => 29.95],
        ['num' => 104, 'producto' => 'Chandal', 'precio' => 69.95],
    ];

    return view('inicio')->with([
        'titulo' => $titulo,
        'paginas' => $paginas,
        'pedidos' => $pedidos,
    ]);
}
```

Y en la vista procesamos el array con @foreach, y accedemos a cada elemento con el nombre de la posición tal y como hacíamos en los arrays asociativos de PHP.

```
<table width='60%' border='1'>
  <thead style='background:lavender;'>
    <tr>
      <th colspan='3'>PEDIDOS</th>
    </tr>
    <tr>
      <th>NUM</th>
      <th>PRODUCTO</th>
      <th>PRECIO</th>
    </tr>
  </thead>
  <tbody style='text-align:center;'>
    @foreach ($pedidos as $pedido)
      <tr>
        <td>{{ $pedido['num'] }}</td>
        <td>{{ $pedido['producto'] }}</td>
        <td>{{ $pedido['precio'] }}</td>
      </tr>
    @endforeach
  </tbody>
</table>
```

Comentarios

Igual que cualquier lenguaje de programación, Blade permite agregar comentarios en el código, y esto se puede realizar de dos maneras:

- **Estilo HTML:** los comentarios se insertan de la forma tradicional en HTML y se muestran en el código fuente.
- **Estilo Blade:** se utilizan los delimitadores {{-- y --}}, y no aparecen en el código fuente de la página.

```
<!-- Tabla de productos -->
{{-- Tabla de productos --}}
```

Crear directivas

Blade ofrece una variedad de directivas, pero además nos brinda la posibilidad de crear nuestras propias directivas para utilizarlas en nuestras vistas según sea necesario. Vamos a explorar esto con un ejemplo simple en el que crearemos una directiva llamada @mayus, la cual tomará una cadena y la convertirá a mayúsculas.

En primer lugar, abrimos el archivo "app/Providers/AppServiceProvider.php" y observamos que contiene dos métodos vacíos:

- **boot():** Este método se ejecuta al iniciar la aplicación y es el lugar adecuado para definir directivas, rutas, listeners y cualquier otro componente que deba estar disponible desde el lanzamiento de la aplicación.
- **register():** Se utiliza para vincular componentes al servicio de contenedor de Laravel, facilitando la gestión automática de dependencias e inyección de dependencias.

Vamos a continuar con el desarrollo de la directiva @mayus en el método boot(). Por tanto, el lugar para crear la directiva será el método boot(), que quedará así:

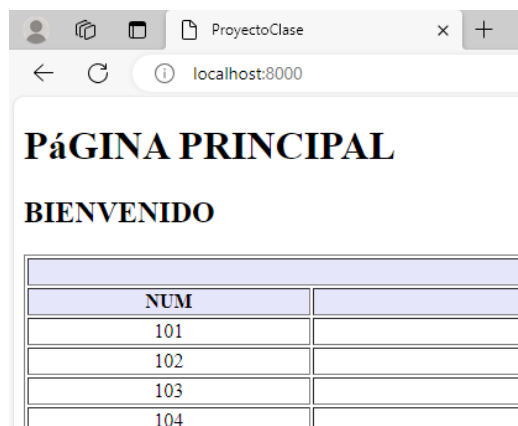
```
public function boot() {  
    Blade::directive('mayus', function ($string) {  
        return "<?php echo strtoupper($string) ?>";  
    });  
}
```

No debemos pasar por alto la importancia de incluir la clase "Blade" en nuestro archivo. Para ello, agregamos la siguiente línea al principio del archivo, o también podemos hacer clic con el botón derecho sobre la palabra "Blade" y seleccionar "import class" si disponemos de la extensión "PHP Namespace resolver".

```
use Illuminate\Support\Facades\Blade;
```

Una vez creada la directiva, vamos a utilizarla en la vista "inicio.blade.php":

```
<h1>@mayus($titulo)</h1>  
<h2>@mayus("bienvenido")</h2>
```



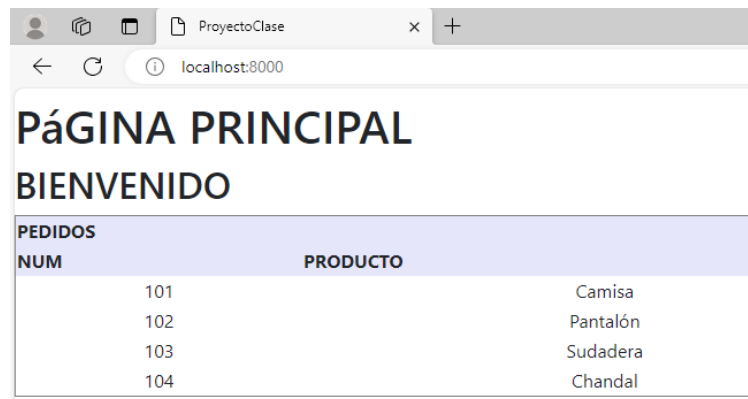
| NUM | |
|-----|--|
| 101 | |
| 102 | |
| 103 | |
| 104 | |

Utilizar Bootstrap en Laravel con CDN

Podemos incorporar Bootstrap a nuestro proyecto Laravel de la misma manera que en cualquier proyecto PHP, simplemente añadiendo los enlaces CDN según se explica en la página oficial de Bootstrap.

[Get started with Bootstrap · Bootstrap v5.3 \(getbootstrap.com\)](https://getbootstrap.com/docs/5.3/getting-started/introduction/)

Si tras añadir el CDN accedemos a nuestra aplicación, veremos que el aspecto ha cambiado ya que se están utilizando los estilos de Bootstrap.



The screenshot shows a web browser window with the address bar at localhost:8000. The page content includes a heading 'PÁGINA PRINCIPAL BIENVENIDO' and a table with the following data:

| PEDIDOS | | |
|---------|----------|--|
| NUM | PRODUCTO | |
| 101 | Camisa | |
| 102 | Pantalón | |
| 103 | Sudadera | |
| 104 | Chandal | |

Integrar Bootstrap en Laravel

A pesar de que es posible emplear Bootstrap en nuestros proyectos de Laravel mediante el uso de CDN, es más apropiado proceder a su instalación e integración directa en el propio proyecto. Esto permite aprovechar las capacidades de optimización, empaquetado y otras funcionalidades avanzadas. Para llevar a cabo la instalación de Bootstrap, es necesario ejecutar los siguientes comandos dentro de la carpeta correspondiente de nuestro proyecto:

composer require laravel/ui

php artisan ui bootstrap

npm install

npm run dev

Una vez instalado, la forma de utilizar Bootstrap va a depender de la versión de Laravel

Laravel 8

Hasta la versión 8, Laravel empleaba Webpack como herramienta de empaquetado a través de Laravel Mix. Tras completar el proceso de empaquetado mediante el comando "npm run dev", los archivos CSS y JS optimizados quedan almacenados en la carpeta "public", y podemos acceder a ellos utilizando el helper "asset()".

```
<!--Assets-->
<link href="{{ asset('css/app.css') }}" rel="stylesheet">
<script src="{{ asset('js/app.js') }}" defer></script>

<!--Fonts-->
<link rel="dns-prefetch" href="//fonts.gstatic.com">
<link href="https://fonts.googleapis.com/css?family=Nunito">
```

Laravel 9

A partir de la versión 9.2, Laravel ha reemplazado WebPack con Vite para llevar a cabo el empaquetado de manera más eficiente, lo que resulta en una drástica reducción en el tiempo necesario para realizar este proceso. Para hacer uso de los recursos CSS y JS generados por Vite, es necesario emplear la directiva Blade '@vite' dentro del bloque head de nuestro documento o plantilla web.

```
<head>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css" rel="styl
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>ProyectoClase</title>
  @vite(['resources/sass/app.scss', 'resources/js/app.js'])
</head>
```

Además, como estamos utilizando la extensión sass, podremos personalizar los estilos de Bootstrap muy fácilmente utilizando las variables sass. Bastará con definir el valor de estas variables en el archivo "sass/app.scss" antes de importar bootstrap:

```
// Bootstrap
$body-bg: #000;
$body-color: #fff;
@import 'bootstrap/scss/bootstrap';
```

Integración de otros frameworks CSS en Laravel

De manera similar a cómo hemos incorporado Bootstrap a nuestro proyecto, también podríamos haber optado por otro framework CSS como Tailwind, Materialize, Material Design, entre otros. La mayoría de estos frameworks proporcionan la opción de ser utilizados tanto a través de la versión en línea (CDN) como integrándolos directamente dentro de Laravel.

<https://tailwindcss.com>

<https://materializecss.com/getting-started.html>

<https://material.io/design>

Agregar nuestros estilos

Aparte de emplear los estilos predefinidos por un framework CSS como Bootstrap, Tailwind, Materialize, etc., tenemos la opción de agregar nuestros propios estilos. Ya lo realizamos al crear la cabecera para la página de administración, aunque no de la manera adecuada, ya que utilizamos la propiedad "style" directamente dentro del HTML, en lugar de mantenerlo en un archivo CSS independiente ubicado en la carpeta "public".

1. En primer lugar, creamos el archivo "styles.css" dentro de la carpeta "public/css", y definimos una clase "cabecera-admin" con los estilos de la cabecera:

```
.cabecera-admin {  
  height:60px;  
  background-color: navy;  
  color: white;  
  font-size:40px;  
  padding:2px 20px;  
}
```

2. A continuación, eliminamos los estilos de nuestro "admin-header.blade.php" y los sustituimos por la clase "cabecera-admin":

```
<div class='container-fluid cabecera-admin'>  
  PANEL DE ADMINISTRACIÓN  
</div>
```

3. Finalmente, importamos nuestro archivo "styles.css" mediante el helper "asset()". Es crucial realizar esta importación después de los archivos de Vite, para otorgar prioridad a nuestros propios estilos.

```
@vite(['resources/sass/app.scss', 'resources/js/app.js'])  
<link rel="stylesheet" href="{{ asset('css/styles.css') }}">
```

4. Tras realizar estos cambios podemos abrir que la vista de administrador y veremos que nuestra página de administración está utilizando tanto los estilos definidos en Bootstrap, como los que hemos definidos nosotros (styles.css).