

Unidad 7

Framework Laravel 9 - Laravel Jetstream
Páginas web dinámicas con Laravel Livewire

Unidad 7

Framework Laravel 9 - Laravel Jetstream
Páginas web dinámicas con Laravel Livewire

INSTALACIÓN

Vamos a usar unas plantillas que ya vienen predefinidas. En este caso: **Laravel Jetstream**

Creamos un nuevo proyecto que se llame jetstream. Para ello:

En la consola de Windows o Gitbash: cd /c/xampp/htdocs

```
Laravel new jetstream -jet      ¡Son dos guiones!  
          0 (livewire)
```

Por otro lado, instalamos <https://nodejs.org/es/download/>

En Visual Studio, instalamos Laravel Jetstream: Composer require laravel/jetstream

```
Php artisan jetstream:install livewire
```

```
Npm install
```

```
Npm run build
```

```
Php artisan migrate
```

INSTALACIÓN

Vamos a instalar también 2 extensiones de Visual Studio Code:

Laravel goto view y Laravel goto controller

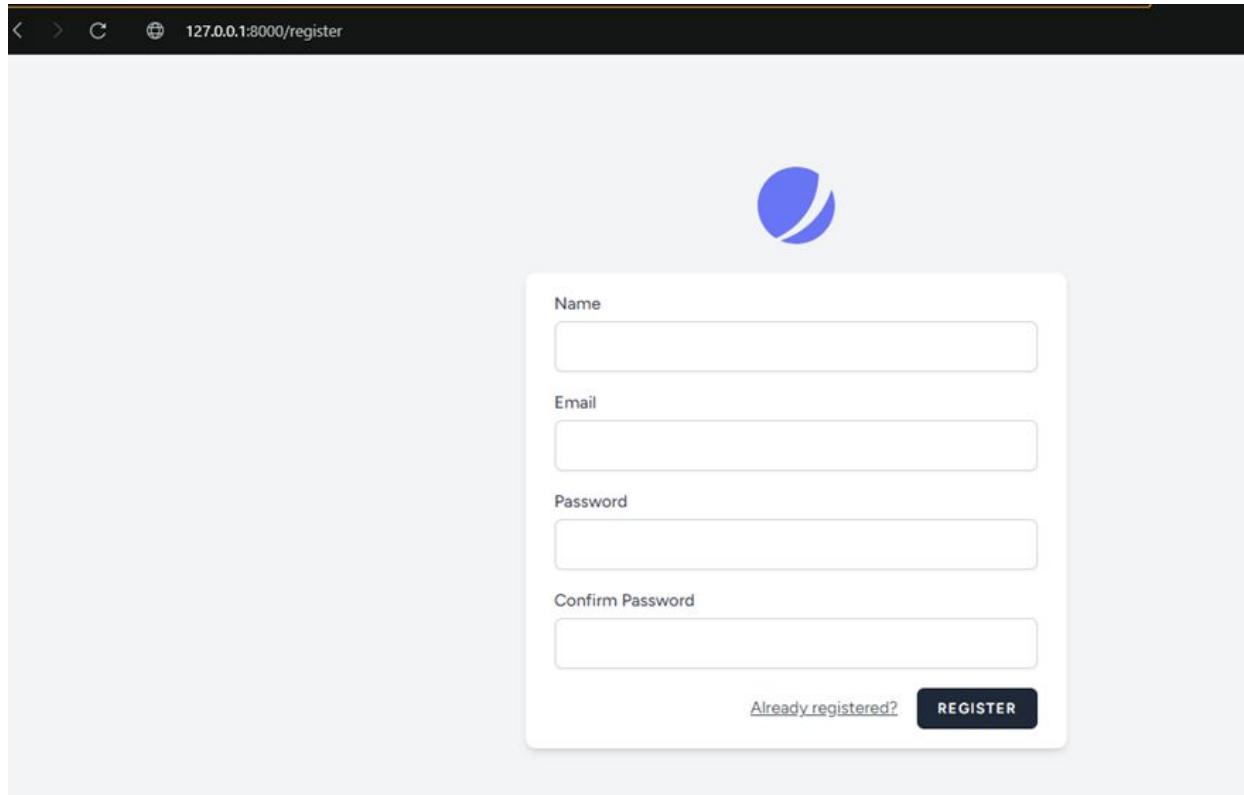
Una vez tengamos todo instalado:

Abrimos el proyecto en 127.0.0.1:8000 , escribiendo en consola:

`php artisan serve`

1. Plantilla JetStream

Si le damos a registrar



The screenshot shows a registration page from a local host at `127.0.0.1:8000/register`. The page features a large blue circular logo in the center. Below the logo is a form with four input fields: 'Name', 'Email', 'Password', and 'Confirm Password'. Each field has a corresponding placeholder text and a red border. At the bottom left of the form is a link 'Already registered?'. At the bottom right is a dark blue 'REGISTER' button.

127.0.0.1:8000/register

Name

Email

Password

Confirm Password

[Already registered?](#)

REGISTER

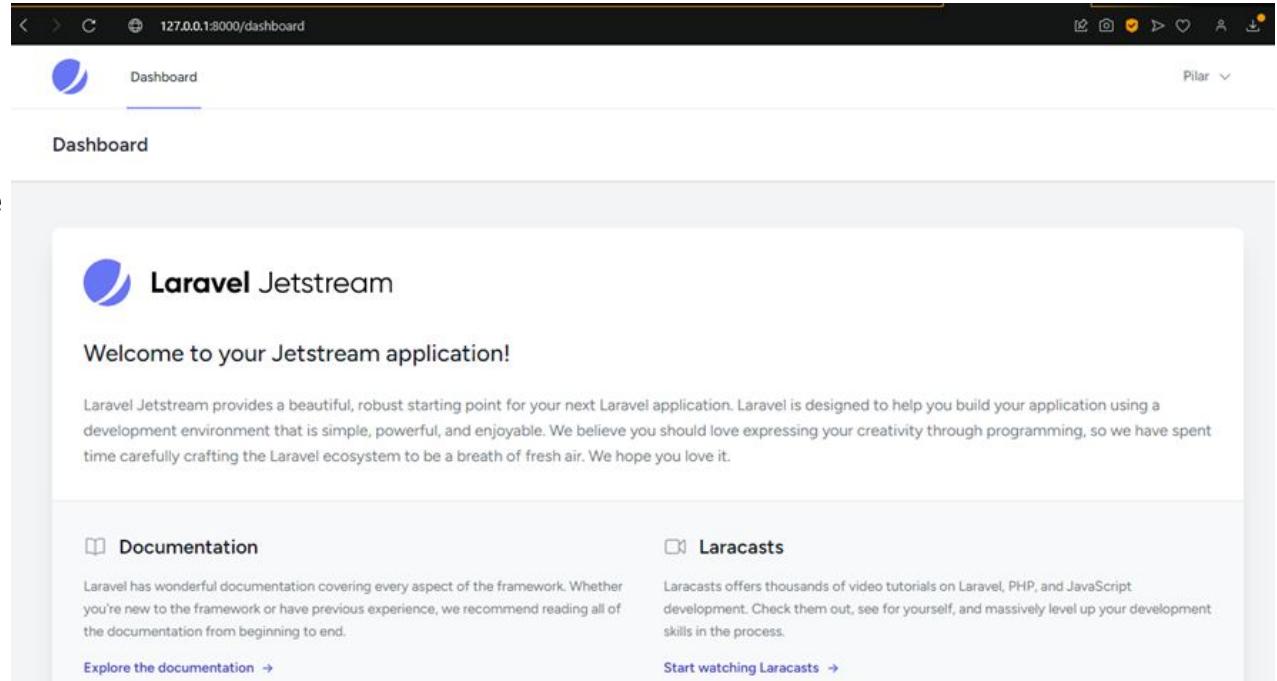
1. Plantilla JetStream

Y nos registramos, nos sale:

Se llama plantilla de Jetstream

Y nos proporciona un montón de opciones.

Desde opciones del usuario a información del mismo etc.



1.1 Personalización

Si queremos que el usuario elija la Foto de perfil

Como es un archivo de configuración, los archivos de configuración se encuentran en config.

En el archivo de configuración config\jetstream.php descomentamos la línea:

Features::profilePhotos(),

```
60     'features' => [
61         // Features::termsAndPrivacyPolicy(),
62         Features::profilePhotos(),
63         // Features::api(),
64         // Features::teams(['invitations' => true]),
65         Features::accountDeletion(),
66     ],
67 ]
```

1.1 Personalización

The screenshot shows a web application interface for user profile management. At the top, the browser's address bar displays the URL `localhost/jetstream/public/user/profile`. On the left, a vertical sidebar contains a blue circular icon with a white 'S' shape and a 'Dashboard' link. The main content area has a header 'Profile'. Below it, a section titled 'Profile Information' with the sub-instruction 'Update your account's profile information and email address.' To the right, there is a 'Photo' placeholder featuring a large letter 'P' inside a light blue circle. A button labeled 'SELECT A NEW PHOTO' is located below the placeholder.

< > C : localhost/jetstream/public/user/profile

 Dashboard

Profile

Profile Information

Update your account's profile information and email address.

Photo

P

SELECT A NEW PHOTO

1.1 Personalización

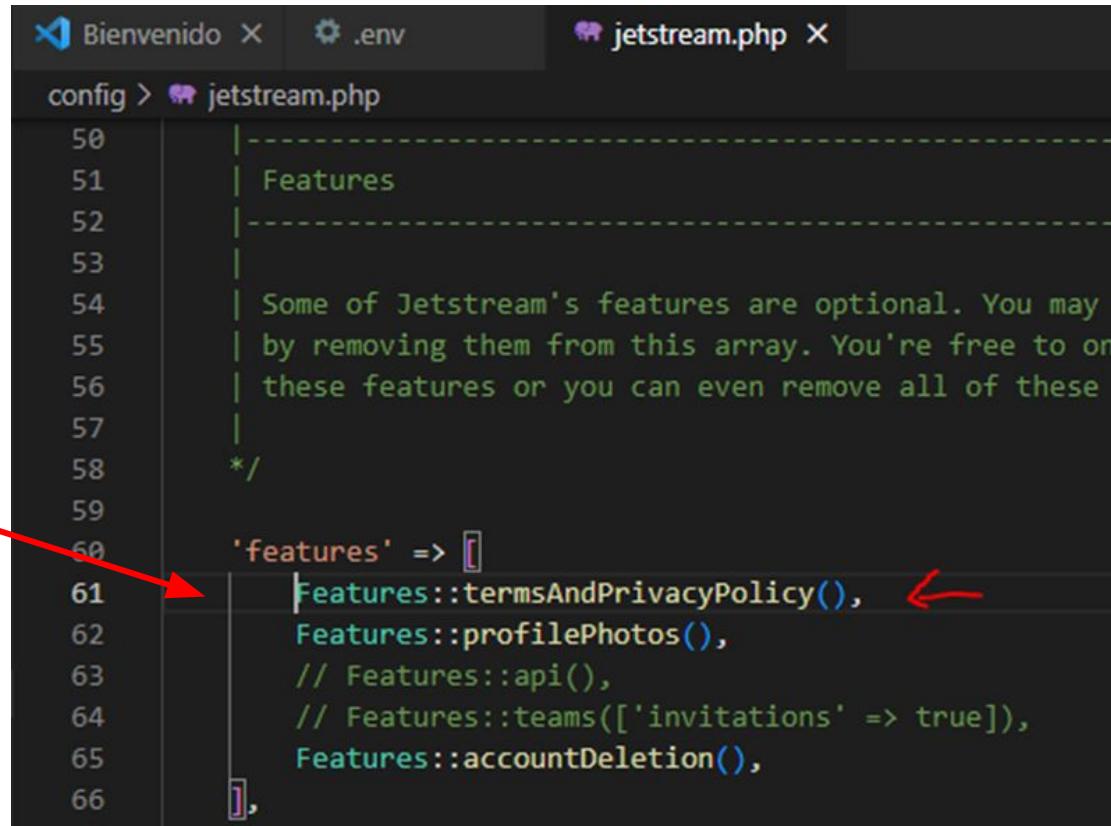
Si queremos que el usuario acepte términos y política de privacidad

Como es un archivo de configuración, los archivos de configuración se encuentran en config.

En el archivo de configuración config\jetstream.php descomentamos la línea:

Esta política se configura en:

Resource > markdown / policy.md y terms.md y aquí se modifica y se cambia la política y los términos que queramos poner.



```
Bienvenido X .env jetstream.php X
config > jetstream.php
50 | |
51 | Features
52 | -----
53 |
54 | Some of Jetstream's features are optional. You may
55 | by removing them from this array. You're free to on
56 | these features or you can even remove all of these
57 |
58 */
59
60 'features' => [
61     Features::termsAndPrivacyPolicy(), ←
62     Features::profilePhotos(),
63     // Features::api(),
64     // Features::teams(['invitations' => true]),
65     Features::accountDeletion(),
66 ],
67
```

1.1 Personalización

A screenshot of a web browser showing a registration form. The URL in the address bar is `localhost/jetstream/public/register`. The page features a large blue circular logo at the top center. Below it is a light gray card containing four input fields: Name, Email, Password, and Confirm Password. At the bottom left of the card, there is a checkbox labeled "I agree to the [Terms of Service](#) and [Privacy Policy](#)". A red arrow points from the text above this section to the checkbox. At the bottom right of the card are two buttons: "Already registered?" and a dark blue "REGISTER" button.

< > C : localhost/jetstream/public/register

Name

Email

Password

Confirm Password

I agree to the [Terms of Service](#) and [Privacy Policy](#).

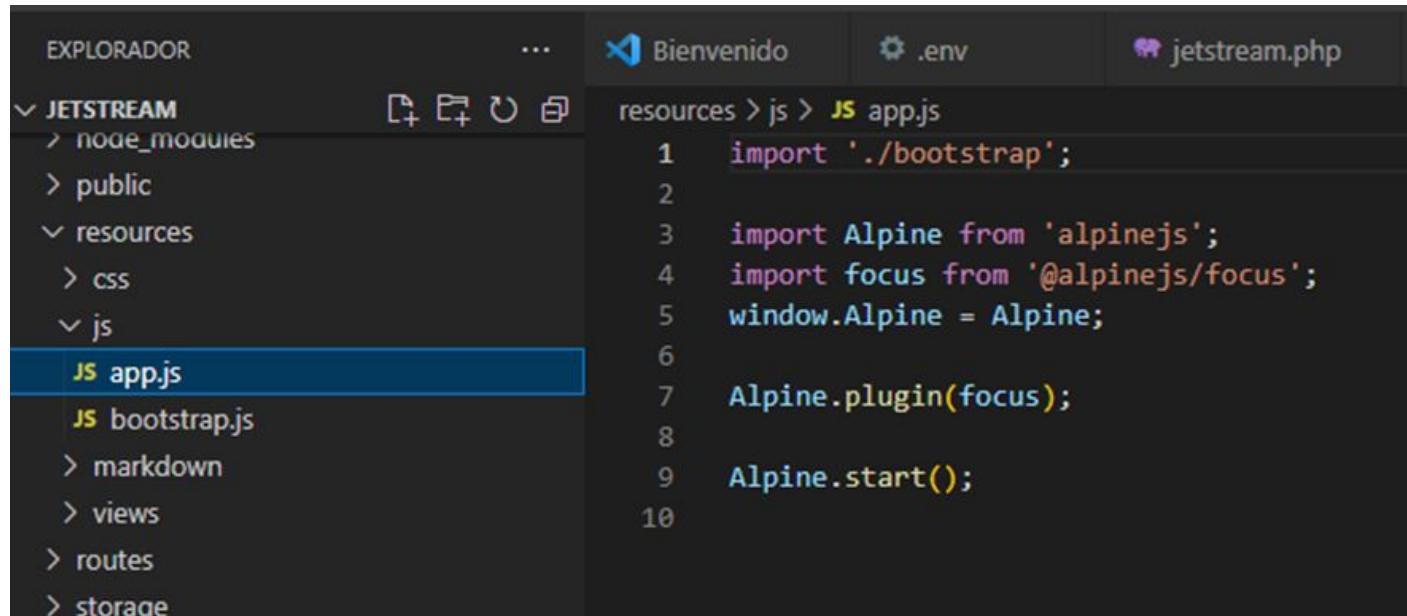
Already registered? **REGISTER**

1.2 Curiosidades

Al instalar Jetstream también se instala Alpine.

Alpine que es una librería JS que nos permite agregar interactividad a nuestra web.

Cuenta de esto lo da el archivo resources/js/app.js



The screenshot shows a code editor interface with a dark theme. On the left, the Explorer sidebar displays a project structure under 'JETSTREAM': node_modules, public, resources (with css and js subfolders), and views, routes, storage. The 'js' folder is expanded, and 'app.js' is selected, highlighted with a blue bar at the bottom of the list. The main editor area shows the following code:

```
resources > js > JS app.js
1 import './bootstrap';
2
3 import Alpine from 'alpinejs';
4 import focus from '@alpinejs/focus';
5 window.Alpine = Alpine;
6
7 Alpine.plugin(focus);
8
9 Alpine.start();
10
```

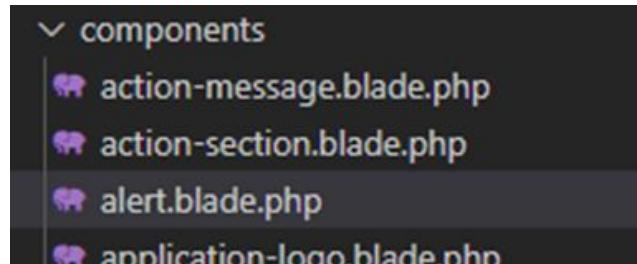
No vamos a entrar en esto.

2. Componentes blade

Crear de alertas

Para crear una alerta, creamos un nuevo fichero que se llame: alert.blade.php

Los componentes de Blade se encuentran en resources/views/ .



Buscamos una alerta en Google:

<https://v1.tailwindcss.com/components/alerts>

Y copiamos la que queramos (**la que os voy a pasar**) en alert.blade.php

2. Componentes blade

Crear de alertas

¿Cómo utilizo esta alerta?

La llamamos por ejemplo desde dashboard.blade.php

Borramos lo que hay y ponemos:

<x-alert/>

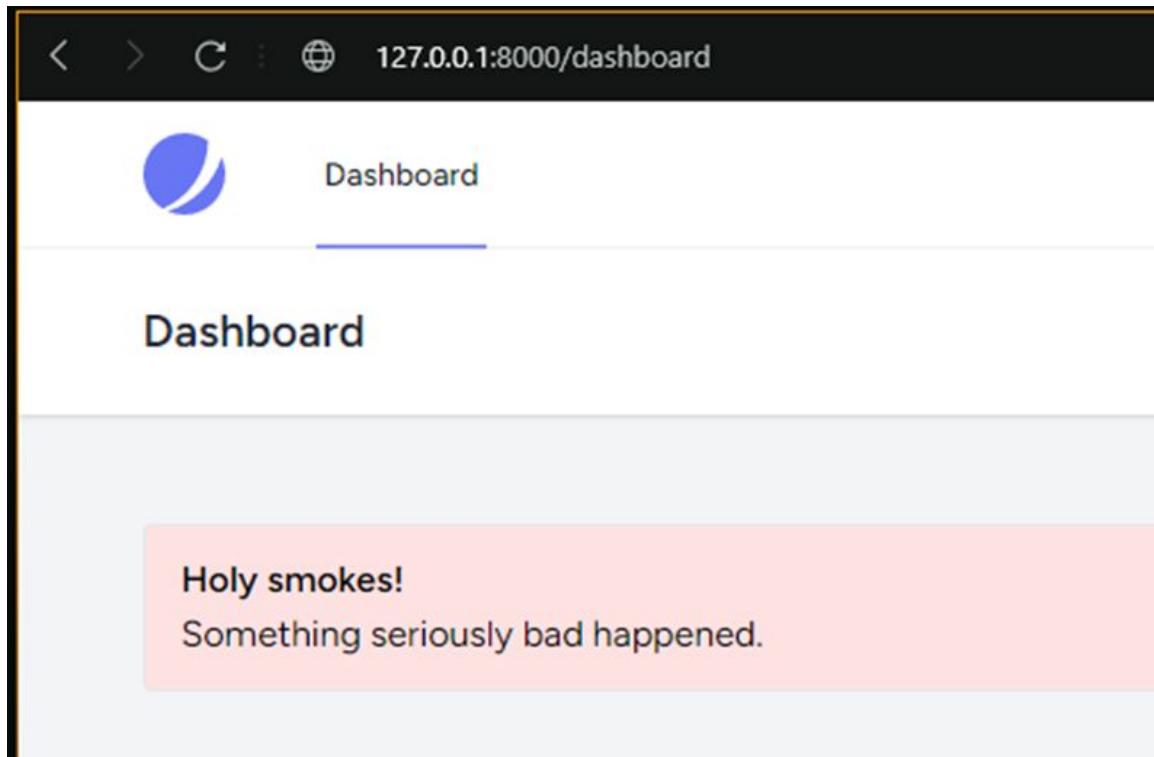
alert es el nombre de lo que acabamos de crear.

y x llama a los componentes, en este caso, alert.

```
resources > views > 📄 dashboard.blade.php > ⚒ x-app-layout > ⚒ div.py-12 > ⚒ d
1   <x-app-layout>
2     <x-slot name="header">
3       <h2 class="font-semibold text-xl text-gray-800 leading-tight">
4         {{ __('Dashboard') }}
5       </h2>
6     </x-slot>
7
8     <div class="py-12">
9       <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
10      |
11      |   <x-alert/>
12      |
13      </div>
14    </div>
15  </x-app-layout>
16
```

2. Componentes blade

Crear de alertas



2. Componentes blade

Crear de alertas

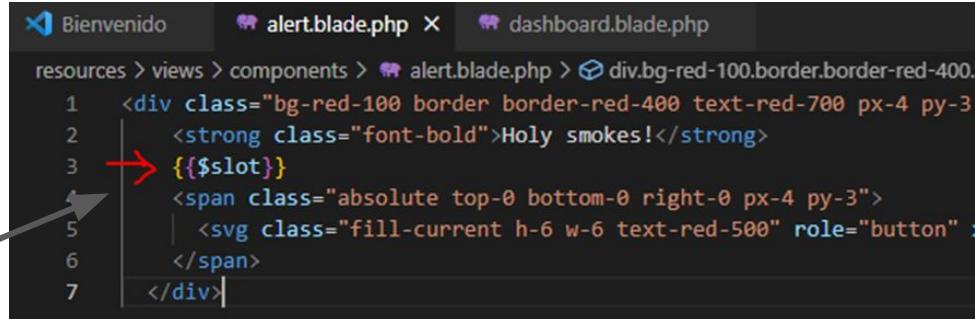
Configurar y modificar la alerta

Nos vamos a `alert.blade.php` y borramos la línea de:

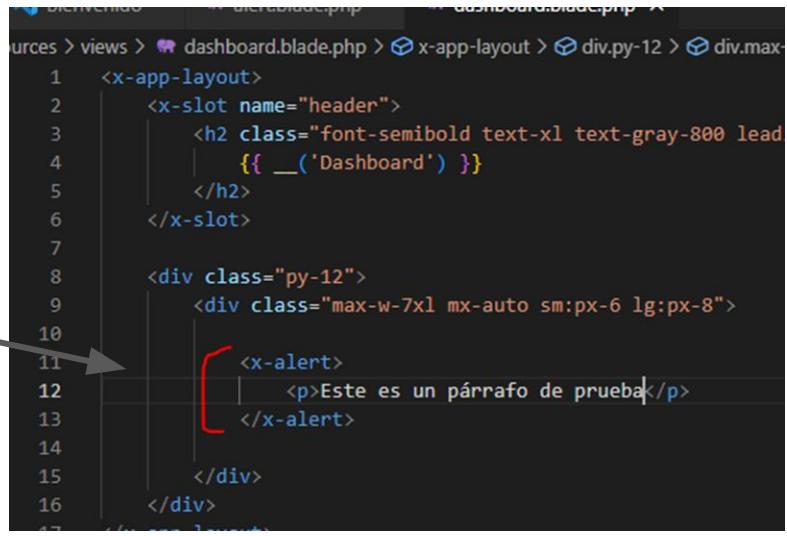
Something seriously bad happened y ponemos:

Si nos volvemos a ir a `dashboard.blade.php` y ponemos:

Cualquier cosa que pongamos entre `<x-alert>` y `</x-alert>` lo va a almacenar en la variable que acabamos de crear **slot**



```
Bienvenido alert.blade.php X dashboard.blade.php
resources > views > components > alert.blade.php >
1 <div class="bg-red-100 border border-red-400 text-red-700 px-4 py-3">
2   <strong class="font-bold">Holy smokes!</strong>
3   → {{ $slot }}
4   <span class="absolute top-0 bottom-0 right-0 px-4 py-3">
5     <svg class="fill-current h-6 w-6 text-red-500" role="button" >
6       </svg>
7     </span>
</div>
```

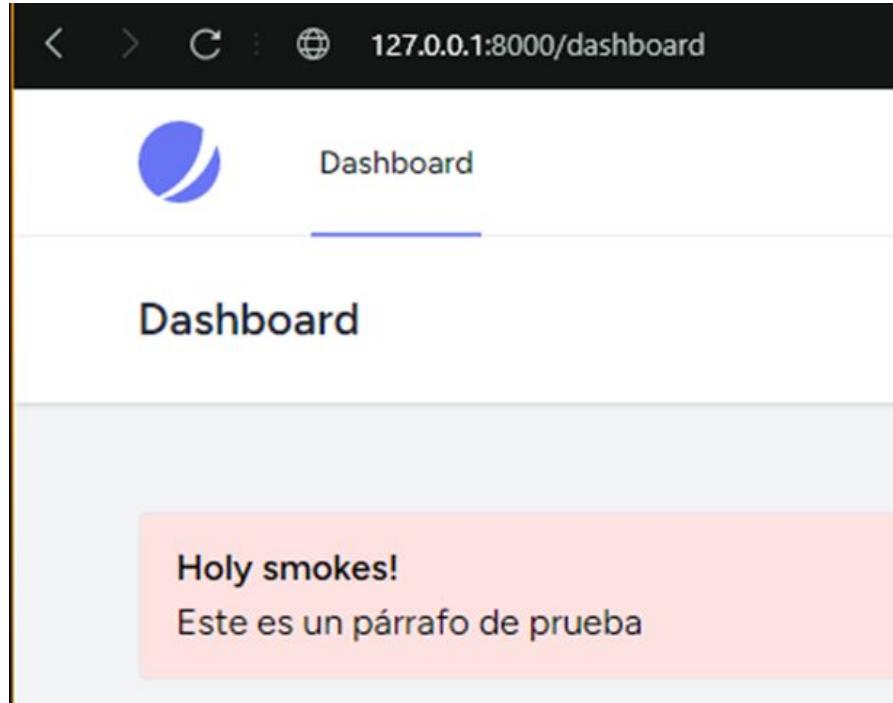


```
Bienvenido alert.blade.php X dashboard.blade.php
resources > views > dashboard.blade.php > x-app-layout > div.py-12 > div.max-w-7xl.mx-auto.sm:px-6.lg:px-8
1 <x-app-layout>
2   <x-slot name="header">
3     <h2 class="font-semibold text-xl text-gray-800 lead">
4       {{ __('Dashboard') }}
5     </h2>
6   </x-slot>
7
8   <div class="py-12">
9     <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
10    <x-alert>
11      <p>Este es un párrafo de prueba</p>
12    </x-alert>
13
14  </div>
15 </div>
16 </div>
17 </x-app-layout>
```

2. Componentes blade

Crear de alertas

Configurar y modificar la alerta



2. Componentes blade

Crear de alertas

Configurar y modificar la alerta

Si queremos modificar el título, podemos usar los atributos del componente.

Nos vamos a dashboard.blade

```
resources > views > 🏡 dashboard.blade.php > ⚒ x-app-layout > ⚒ div.py-12 > ⚒ div.ma
1   <x-app-layout>
2     <x-slot name="header">
3       <h2 class="font-semibold text-xl text-gray-800 leading-6">
4         {{ __('Dashboard') }}
5       </h2>
6     </x-slot>
7
8     <div class="py-12">
9       <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
10
11       <x-alert title="Título de prueba">
12         <p>Este es un párrafo de prueba</p>
13       </x-alert>
14
15     </div>
16   </div>
```

2. Componentes blade

Crear de alertas

Configurar y modificar la alerta

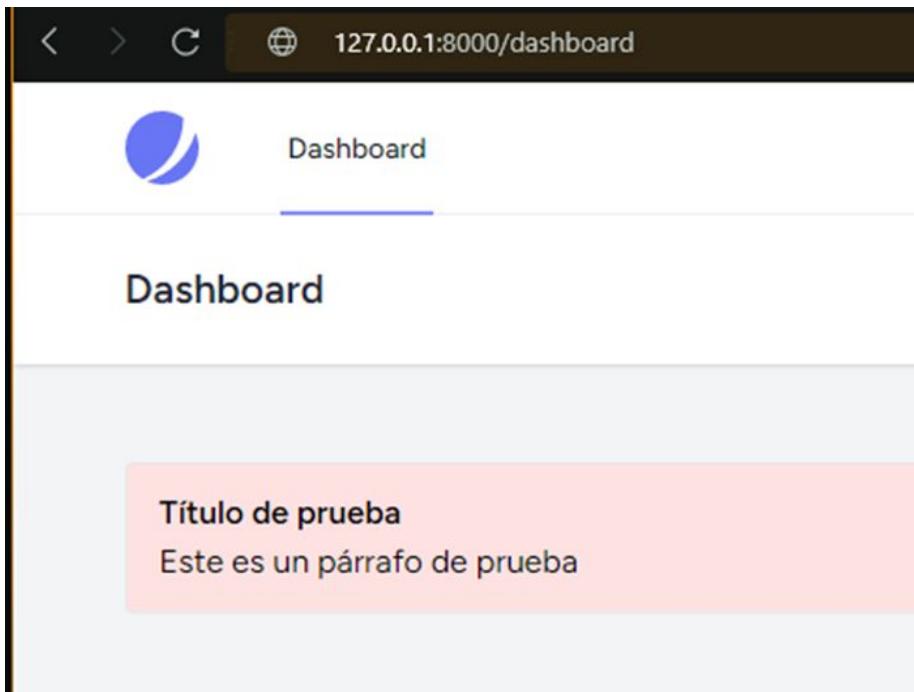
Y lo recibimos en el componente alert.blade.php haciendo uso de la directiva props:

```
resources > views > components > 🐾 alert.blade.php > ⚙️ div.bg-red-100.border.border-red-400.text-red-700.
1   @props(['title'])
2
3
4   <div class="bg-red-100 border border-red-400 text-red-700 px-4 py-3 rounded r
5   ↗   <strong class="font-bold">&{{ $title }}</strong>
6   {{ $slot }}
7   <span class="absolute top-0 bottom-0 right-0 px-4 py-3">
8     <svg class="fill-current h-6 w-6 text-red-500" role="button" xmlns="http://www.w3.org/2000/svg">
9       </span>
10      </div>
```

2. Componentes blade

Crear de alertas

Configurar y modificar la alerta



2. Componentes blade

Crear de alertas

Configurar y modificar la alerta - Otra forma

Esta forma de pasar valores a nuestras alertas nos vale cuando son valores pequeños. Pero si queremos pasar valores más grandes o estructuras.

Se hace de otra manera, a través de slots, slots secundarios.

En dashboard.blade.php:



```
resources > views > 🐾 dashboard.blade.php > ⚡ x-app-layout > ⚡ div.py-12 > ⚡ d
4           {{ __('Dashboard') }}
5           </h2>
6           </x-slot>
7
8           <div class="py-12">
9               <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
10
11               <x-alert>
12                   <x-slot name="title">
13                       Este es el título desde el slot
14                   </x-slot>
15                   <p>Este es un texto de prueba</p>
16               </x-alert>
17
18           </div>
```

A red bracket highlights the code block starting at line 11, specifically the opening and closing tags for the `<x-alert>` component and its nested `<x-slot name="title">` and `<p>` tags. A red callout box points to the text "Este es el título desde el slot".

2. Componentes blade

Crear de alertas

Configurar y modificar la alerta - Otra forma

Y lo recibimos en alert.blade.php sin necesidad de la directiva props porque ya lo hemos pasado por name del slot.

```
resources > views > components > 🐾 alert.blade.php > ...
1  |
2  <div class="bg-red-100 border border-red-400 text-red-700 px-4 py-3 rounded">
3    <strong class="font-bold">{$title}</strong>
4    {$slot}
5    <span class="absolute top-0 bottom-0 right-0 px-4 py-3">
6      <svg class="fill-current h-6 w-6 text-red-500" role="button" xmlns="http://www.w3.org/2000/svg" viewBox="0 0 16 16"></svg>
7    </span>
8  </div>
```

2. Componentes blade

Crear de alertas

Varios tipos de alertas

Imaginemos que queremos tener varios tipos de alerta. Por ejemplo, una alerta de **info**.

En dashboard.blade.php



```
resources > views > 🐾 dashboard.blade.php > 📁 x-app-layout > 📁 div.py-12 > 📁 div
4           |   {{ __('Dashboard') }}
5           |   </h2>
6           |   </x-slot>
7
8   <div class="py-12">
9       <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
10
11      <x-alert type="info">          ↗
12          <x-slot name="title">        |
13              |   Este es el título desde el slot
14          </x-slot>
15          <p>Este es un texto de prueba</p>
16      </x-alert>
17
18  </div>
```

2. Componentes blade

Crear de alertas

Varios tipos de alertas

Como le pasamos un atributo, tenemos que recibirlo desde el componente.

Luego, nos vamos a alert.blade.php

Y ponemos:

```
resources > views > components > 📄 alert.blade.php > ⚙️ div.bg-red-100.b
1 → @props(['type'])
2
3
4 <div class="bg-red-100 border border-red-400 text-r
5   <strong class="font-bold">$title</strong>
6   $slot
7   <span class="absolute top-0 bottom-0 right-0 px
8     <svg class="fill-current h-6 w-6 text-red-500
9       </span>
10      </div>
```

2. Componentes blade

Crear de alertas

Varios tipos de alertas

Para modificar el color de cada alerta:

Agregamos la directa @php que nos permite crear código php
Y escribimos:



```
resources > views > components > 🗃 alert.blade.php > ...
1  @props(['type'])
2
3  @php
4      switch($type) {
5          case 'info':
6              $clases = "bg-blue-100 border-blue-500 text-blue-700";
7              break;
8          case 'danger':
9              $clases = "bg-red-100 border-red-500 text-red-700";
10             break;
11         default:
12             $clases = "bg-orange-100 border-orange-500 text-orange-700";
13             break;
14     }
15     @endphp
```

2. Componentes blade

Crear de alertas

Varios tipos de alertas

Y abajo:



```
resources > views > components > alert.blade.php > ...
5      switch($type) {
6          case 'info':
7              $clases = "bg-blue-100 border-blue-500 text-blue-700";
8              break;
9          case 'danger':
10             $clases = "bg-red-100 border-red-500 text-red-700";
11             break;
12         default:
13             $clases = "bg-orange-100 border-orange-500 text-orange-700";
14             break;
15     }
16     @endphp
17     <div class="border px-4 py-3 rounded relative {{$clases}}" role="alert">
18         <strong class="font-bold">$title</strong>
19         {{$slot}}
20         <span class="absolute top-0 bottom-0 right-0 px-4 py-3">
```

2. Componentes blade

Crear de alertas

Varios tipos de alertas

Si lo probamos

Y si cambiamos el tipo en nuestro dashboard en lugar de info ponemos danger:

```
resources > views > dashboard.blade.php > app.layout > div.py-12 > div
 4     |     {{ __('Dashboard') }}
 5     |     </h2>
 6     |     </x-slot>
 7
 8     <div class="py-12">
 9         <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
10
11         <x-alert type="danger"> ←
12             <x-slot name="title">
13                 Este es el título desde el slot
14             </x-slot>
15             <p>Este es un texto de prueba</p>
16         </x-alert>
17
18     </div>
19 </div>
```

< > C 127.0.0.1:8000/dashboard



Dashboard

Dashboard

Este es el título desde el slot

Este es un texto de prueba

Dashboard

Este es el título desde el slot

Este es un texto de prueba

2. Componentes blade

Crear de alertas

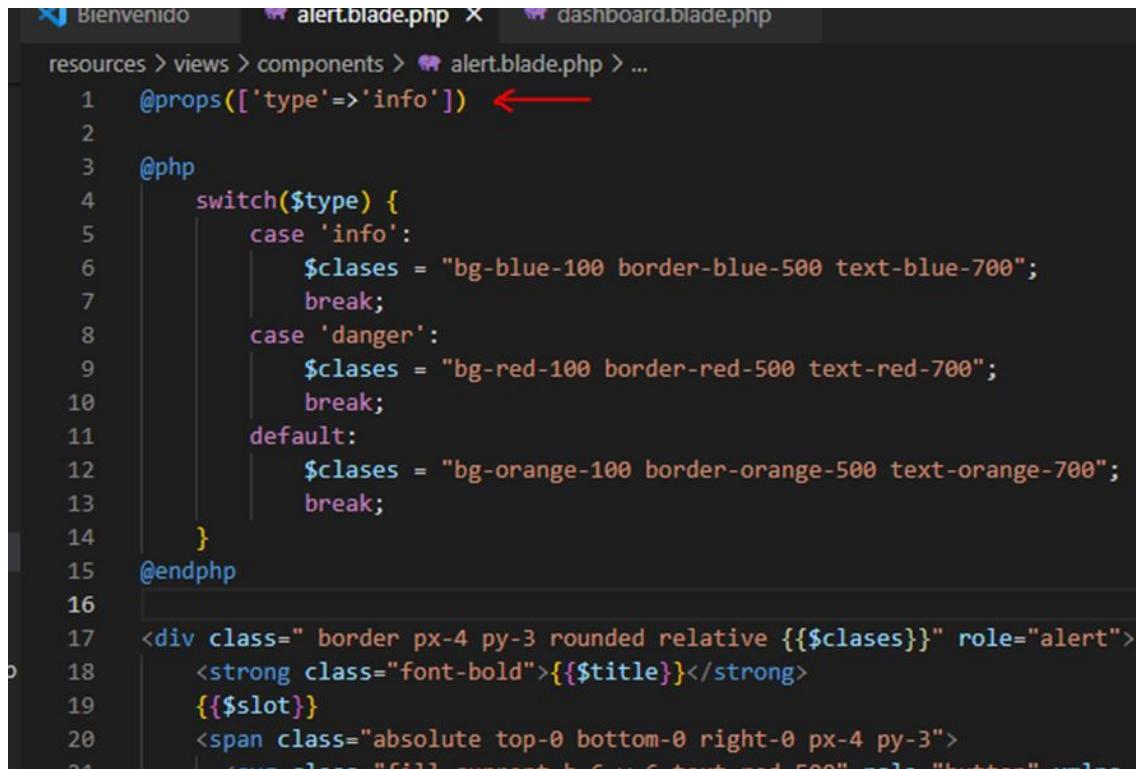
Varios tipos de alertas

Si no le pasamos nada, nos va a dar un error.

En este caso, en el caso en el que no definamos el tipo de alerta que queremos, exista un tipo por defecto.

El tipo por defecto va a ser info.

Para ello, en alert.blade.php



```
resources > views > components > alert.blade.php > ...
1  @props(['type'=>'info']) ←
2
3  @php
4      switch($type) {
5          case 'info':
6              $clases = "bg-blue-100 border-blue-500 text-blue-700";
7              break;
8          case 'danger':
9              $clases = "bg-red-100 border-red-500 text-red-700";
10             break;
11         default:
12             $clases = "bg-orange-100 border-orange-500 text-orange-700";
13             break;
14     }
15     @endphp
16
17 <div class=" border px-4 py-3 rounded relative {{$clases}}" role="alert">
18     <strong class="font-bold">$title</strong>
19     {{$slot}}
20     <span class="absolute top-0 bottom-0 right-0 px-4 py-3">
21         <button class="flex items-center justify-center w-full h-full bg-white border border-gray-300 rounded" type="button">
```

2. Componentes blade

Crear de alertas

Varios tipos de alertas

Podemos pasarle código a las alertas. Por ejemplo, si queremos usar variables en los atributos del componente.

Nos vamos a dashboard.blade.php

```
resources > views > dashboard.blade.php > x-app-layout
4           {{ __('Dashboard') }}
5           </h2>
6           </x-slot>
7
8           <div class="py-12">
9               <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
10
11             @php
12             $type = 'info';
13             @endphp
14
15             → <x-alert :type="$type">
16                 <x-slot name="title">
17                     Este es el título desde el slot
18                 </x-slot>
19                 <p>Este es un texto de prueba</p>
20             </x-alert>
21
22             </div>
23         </div>
24     </x-app-layout>
```

2. Componentes blade

Crear de alertas

Varios tipos de alertas

Pero si no queremos pasarle un valor, si no un atributo.

Por ejemplo id

Y lo recuperamos en nuestro componente. En alert.blade.php

```
resources > views > components > alert.blade.php > ...
1 @props(['type' => 'info', 'id'])
```

The diagram illustrates the flow of data from a component to a view. A black arrow points from the '@props' line in the alert.blade.php component code to the '\$type' variable in the dashboard.blade.php view code. A red arrow points from the 'id' prop in the component code to the 'id="alerta"' attribute in the 'x-alert' component in the view code.

```
resources > views > dashboard.blade.php > x-app-layout > div.py-12 > div
4     |           {{ __('Dashboard') }}
5     |           </h2>
6     |           </x-slot>
7
8     <div class="py-12">
9         <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
10            @php
11                $type = 'info';
12            @endphp
13
14
15            <x-alert :type="$type" id="alerta">
16                <x-slot name="title">
17                    Este es el título desde el slot
18                </x-slot>
19                <p>Este es un texto de prueba</p>
20            </x-alert>
21
22        </div>
23    </div>
```

Pero esta forma es **poco práctica** porque debemos definir todos los posibles atributos que esperamos recibir.

2. Componentes blade

Crear de alertas

Varios tipos de alertas

Los componentes tienen una variable llamada attributes.

Si no definimos los valores enviados en props se guardarán en attributes.

```
resources > views > components > alert.blade.php > div
  1 @props(['type' => 'info'])
  2
  3 @php
  4     switch($type) {
  5         case 'info':
  6             $clases = "bg-blue-100 border-blue-500 text-blue-900 p-4 rounded-lg";
  7             break;
  8         case 'danger':
  9             $clases = "bg-red-100 border-red-500 text-white p-4 rounded-lg";
 10             break;
 11         default:
 12             $clases = "bg-orange-100 border-orange-500 text-orange-900 p-4 rounded-lg";
 13             break;
 14     }
 15 @endphp
 16
 17 <div {{ $attributes }}>
 18     <strong class="font-bold">{{ $title }}</strong>
 19     {{ $slot }}
 20     <span class="absolute top-0 bottom-0 right-0 px-4 py-2 text-white">
 21         <svg class="fill-current h-6 w-6 text-red-500" role="img"></svg>
 22     </span>
 23 </div>
```

2. Componentes blade

Crear de alertas

Varios tipos de alertas

Si por ejemplo, nosotros ahora queremos enviar un atributo de clase, para que me cree un margen debajo de la alerta para que no ocurra:



Este es el título desde el slot
Este es un texto de prueba

Hola mundo

En `dashboard.blade.php`
tendríamos que colocar:

```
14 | 
15 |     <x-alert :type="$type" id="alerta" class="mb-4">
16 |         <x-slot name="title">
```

2. Componentes blade

Crear de alertas

Varios tipos de alertas

PERO lo que va a pasar es que como ya definimos en alert.blade.php una clase en \$attributes, no podemos definir otra clase por lo que se perderán los estilos



Para solucionarlo en alert.blade.php

```
15 @endphp
16
17 <div {{$attributes->merge(['class'=> "border px-4 py-3 rounded relative $clases"])}>
18   <strong class="font-bold">{{$title}}</strong>
19   {{$slot}}
```

Este es el título desde el slot
Este es un texto de prueba

Lo que hace merge es combinar los atributos que estamos pasando con los atributos base de clase.

2. Componentes blade

Crear de alertas

Varios tipos de alertas

Otros tipos de atributos no se combinan sino que se reemplazan. Por ejemplo, si ponemos:

En alert.blade.php

```
<div {{$attributes->merge(['class'=> "border px-4 py-3 rounded relative $clases", 'role'=>"alert"])}>
  <strong class="font-bold">$title</strong>
  {$slot}
```

Y en dashboard.blade.php

```
<x-alert :type="$type" id="alerta" class="mb-4" role="prueba">
  <x-slot name="title">
```

Lo que va a suceder es que el valor por defecto alert, lo va a sustituir por prueba.

En conclusión, si le pasamos una clase, combina estilos. Si no, reemplaza.

2. Componentes blade

Crear de alertas ¡Separar código!

Llegados a este punto podemos observar que estamos poniendo código php en la vista. En este caso, no es tan grande y ocupa poco.

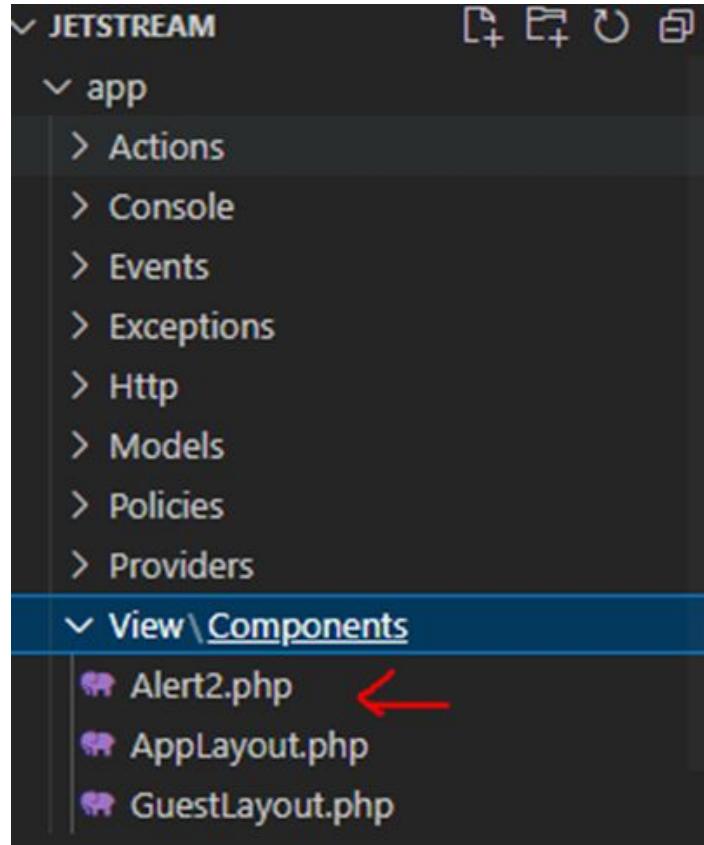
Pero habrá ocasiones en las que la lógica (el código php) sea bastante más complicado.

Y vamos a querer tener separado la lógica de la vista.

Para ello, vamos a crear un componente de clase.
Abrimos la terminal:

```
php artisan make:component Alert2
```

Esto me crea dos archivos:



2. Componentes blade

Crear de alertas

El primero: la lógica. Que se encuentra en
app/View/Components/Alert2.php



El segundo: la vista. Que se encuentra en
resources/views/components

resources > views > components > alert2.blade.php >  div

```
1  <div>
2    <!-- He who is contented is rich. - Laozi -->
3  </div>
```

```
app > View > Components > Alert2.php > ...
1  <?php
2
3  namespace App\View\Components;
4
5  use Closure;
6  use Illuminate\Contracts\View\View;
7  use Illuminate\View\Component;
8
9  class Alert2 extends Component
10 {
11
12     /**
13      * Create a new component instance.
14     */
15
16     public function __construct()
17     {
18         //
19     }
20
21     /**
22      * Get the view / contents that represent the component.
23     */
24     public function render(): View|Closure|string
25     {
26         return view('components.alert2');
27     }
}
```

2. Componentes blade

Crear de alertas

Ahora, en la vista alert2.blade.php ponemos lo que teníamos en la alerta.blade.php de antes, copiamos y pegamos:

```
resources > views > components > 🐾 alert2.blade.php > ⚙️ div
1  <div {{$attributes->merge(['class'=> "border px-4 py-3 rounded relative $clases", 'role'=>"alert"
2    | <strong class="font-bold">{{$title}}</strong>
3    | {{$slot}}
4    | <span class="absolute top-0 bottom-0 right-0 px-4 py-3">
5    |   <svg class="fill-current h-6 w-6 text-red-500" role="button" xmlns="http://www.w3.org/2000/svg">
6    |     <path d="M12.5 10.5l-5 5L12.5 17l5-5L17 12.5l-5 5L12.5 17l5 5L17 12.5z" />
7  </div>
```

2. Componentes blade

Crear de alertas

Y la lógica, es decir, el código php lo definimos de la siguiente manera, copiamos lo que tenemos en alert.blade.php

Tendríamos que ir ahora a dashboard y poner en lugar de alert, alert2

```
<x-alert2 :type="$type" id="alerta" class="mb-4">
    <x-slot name="title">
        Este es el título desde el slot
    </x-slot>
    <p>Este es un texto de prueba</p>
</x-alert2>
```

```
app > View > Components > Alert2.php > Alert2
  8
  9  class Alert2 extends Component
 10 {
 11     /**
 12      * Create a new component instance.
 13      */
 14     public $clases;
 15     public function __construct($type='info')
 16     {
 17         switch($type) {
 18             case 'info':
 19                 $clases = "bg-blue-100 border-blue-500 text-blue-700";
 20                 break;
 21             case 'danger':
 22                 $clases = "bg-red-100 border-red-500 text-red-700";
 23                 break;
 24             default:
 25                 $clases = "bg-orange-100 border-orange-500 text-orange-700";
 26                 break;
 27         }
 28         $this->clases = $clases;
 29     }
 30
 31     /**
 32      * Get the view / contents that represent the component.
 33      */
 34     public function render(): View|Closure|string
 35     {
 36         return view('components.alert2');
 37     }
}
```

2. Componentes blade

Para ver todos los componentes creados en la instalación de Jetstream, nos vamos a: views/components

Por ejemplo, si yo quiero llamar al componente que se llama welcome.blade.php

En dashboard.blade.php borro las alertas que teníamos pongo:

```
<x-welcome/>
```

```
resources > views > 🗂️ dashboard.blade.php > ⚒️ x-app-layout > ⚒️ div.py-12 > ⚒️ div.max-w-7xl

1  <x-app-layout>
2    <x-slot name="header">
3      <h2 class="font-semibold text-xl text-gray-800 leading-tight">
4        {{ __('Dashboard') }}
5      </h2>
6    </x-slot>
7
8    <div class="py-12">
9      <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
10
11      <x>Welcome/>
12
13    </div>
14  </div>
15 </x-app-layout>
16
```



3. Modificar plantilla JetStream

Vamos a ver cómo modificar y personalizar la plantilla que viene por defecto en nuestro proyecto Jetstream

Vemos que hay definidas dos rutas, la principal que devuelve una vista welcome y otra que devuelve la vista dashboard que es la que muestra cuando iniciamos sesión.

Observamos que lo está llamando a través de un middleware, que veremos más adelante. Pero por ahora nos vamos a quedar con que un middleware es un filtro que le aplicamos a nuestras rutas. Es decir, que solo me va a mostrar el contenido de esa ruta si se cumple la condición establecida en el middleware. Que en este caso, es autenticación.

Este middleware auth redirige a dashboard si estamos autenticados y en caso contrario nos redirige a la URL de login.

```
15   Route::get('/', function () {
16     return view('welcome');
17   });
18
19
20  Route::middleware([
21    'auth:sanctum',
22    config('jetstream.auth_session'),
23    'verified'
24  ])->group(function () {
25    Route::get('/dashboard', function () {
26      return view('dashboard');
27    })->name('dashboard');
28  });
29
```

3. Modificar plantilla JetStream

Nos dirigimos a dashboard.blade.php (si pinchamos sobre dashboard nos redirige) pero se encuentra en resource/views/dashboard

Observamos que está llamando al componente x-app-layout

```
resources > views > 🐾 dashboard.blade.php > ⚙️ x-app-layout
1   <x-app-layout>
2     <x-slot name="header">
3       <h2 class="font-semibold text-xl">
4         {{ __('Dashboard') }}
5       </h2>
6     </x-slot>
7
8     <div class="py-12">
9       <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
10
11       <x>Welcome/>
12
13     </div>
14   </div>
15 </x-app-layout>
16
```

3. Modificar plantilla JetStream

Entonces nos dirigimos: app/view/AppLayout.php

Lo que hace el método render es mostrar la vista que se encuentra en layouts.app que si hacemos ctrl click nos redirige.

(app/resource/views/layouts/app.blade.php)

```
app > View > Components > AppLayout.php > ...
1  <?php
2
3  namespace App\View\Components;
4
5  use Illuminate\View\Component;
6  use Illuminate\View\View;
7
8  class AppLayout extends Component
9  {
10     /**
11      * Get the view / contents that represents the component.
12      */
13     public function render(): View
14     {
15         return view('layouts.app');
16     }
17 }
18
```



3. Modificar plantilla JetStream

En app.blade.php tenemos la **estructura básica** que tendrán todas las páginas de nuestro sitio web.

Analicemos la estructura:

```
1 <!DOCTYPE html>
2 <html lang="{{ str_replace('_', '-', app()->getLocale()) }}>
3 |   <head>
```

El método app() está sacando el valor de idioma del archivo de configuración que se encuentra en config\app.php :

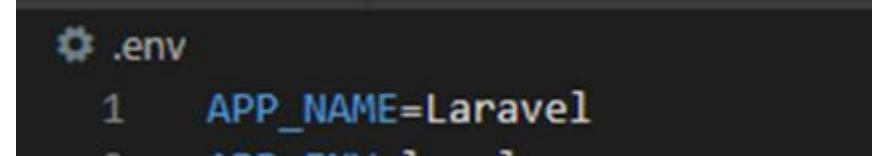
```
config > 🐾 app.php
73 /*
74 |
75 | Application Locale Configuration
76 |
77 | -----
78 |
79 | The application locale determined by the translation service is set to any of the locales which
80 | are defined in the configuration file.
81 |
82 */
83 */
84
85 'locale' => 'en',
86 /*
87 */
```

3. Modificar plantilla JetStream

En la línea 8:

```
7
8      <title>{{ config('app.name', 'Laravel') }}</title>
9
```

Vemos que se usa el método config y nos dice que obtiene el valor del <title> de la variable app.name que está en el archivo de configuración .env



.env

```
1 APP_NAME=Laravel
2 APP_ENV=local
```

En la línea 10:

```
10     <!-- Fonts -->
11     <link rel="preconnect" href="https://fonts.bunny.net">
12     <link href="https://fonts.bunny.net/css?family=figtree:400,500,600&display=swap" rel="stylesheet" />
13
```

Está llamando a una fuente

3. Modificar plantilla JetStream

```
25      <!-- Page Heading -->
26      @if (isset($header))
27          <header class="bg-white shadow">
28              <div class="max-w-7xl mx-auto py-6 px-4 sm:px-6 lg:px-8">
29                  {{ $header }}
30              </div>
31          </header>
32      @endif
33
34
35      <!-- Page Content -->
36      <main>
37          {{ $slot }}
38      </main>
39  </div>
40
```

Si nos fijamos, está llamando a header.

Todo lo que se modifique en dashboard.blade.php se va a modificar.

3. Modificar plantilla JetStream

```
resources > views > 🛡 dashboard.blade.php > ✉ x-app-layout > ✉ div.py-12 > ✉ div.max-w-7xl.mx-auto.sm-px-6  
1   <x-app-layout>  
2     <x-slot name="header">  
3       <h2 class="font-semibold text-xl text-gray-800 leading-tight">  
4         {{ __('Dashboard') }}  
5       </h2>  
6     </x-slot>  
7  
8     <div class="py-12">  
9       <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">  
10      <x>Welcome/>  
11  
12      </div>  
13    </div>  
14  </div>  
15 </x-app-layout>  
16
```

Todo lo que esté dentro del slot con nombre header se va a colocar en la cabecera y lo que no, en main.

3. Modificar plantilla JetStream

Si lo probamos:

```
resources > views > 🌐 dashboard.blade.php > ⚙️ x-app-layout > ⚙️ x-slot > ⚙️ h2.font-semibold.text-xl.text-g
 1 <x-app-layout>
 2   <x-slot name="header">
 3     <h2 class="font-semibold text-xl text-gray-800 leading-tight">
 4       DWES
 5     </h2>
 6   </x-slot>
 7 
```



 Dashboard
Dashboard



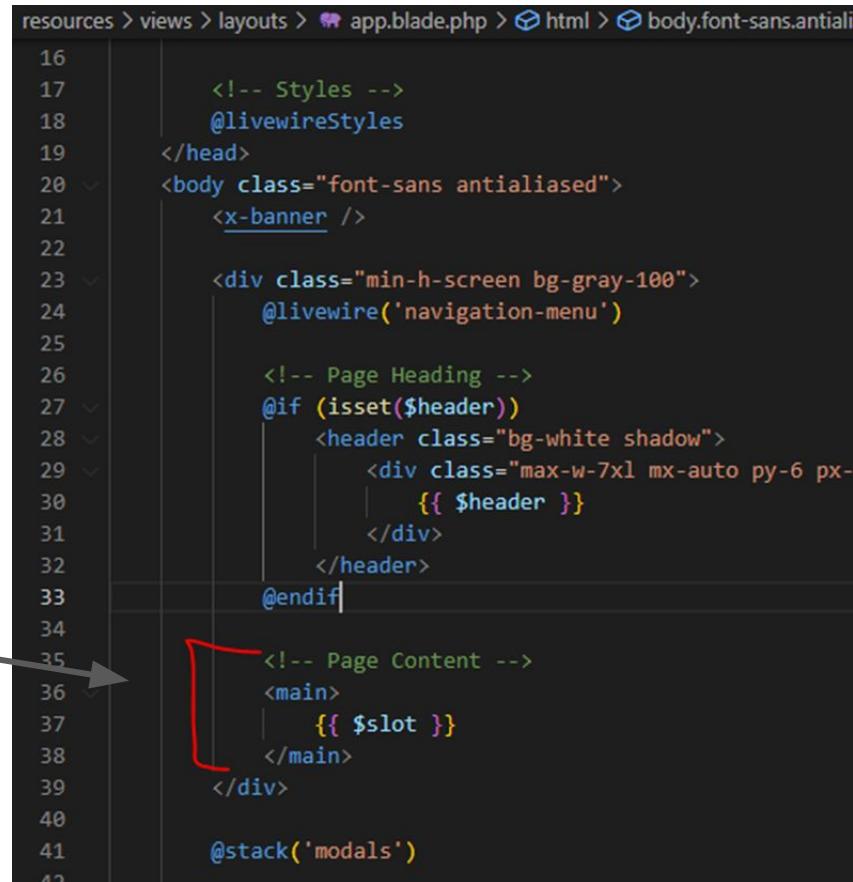
DWES

3. Modificar plantilla JetStream

Vamos a modificar el contenido que tenemos debajo.

El contenido principal irá según nuestra plantilla app.blade

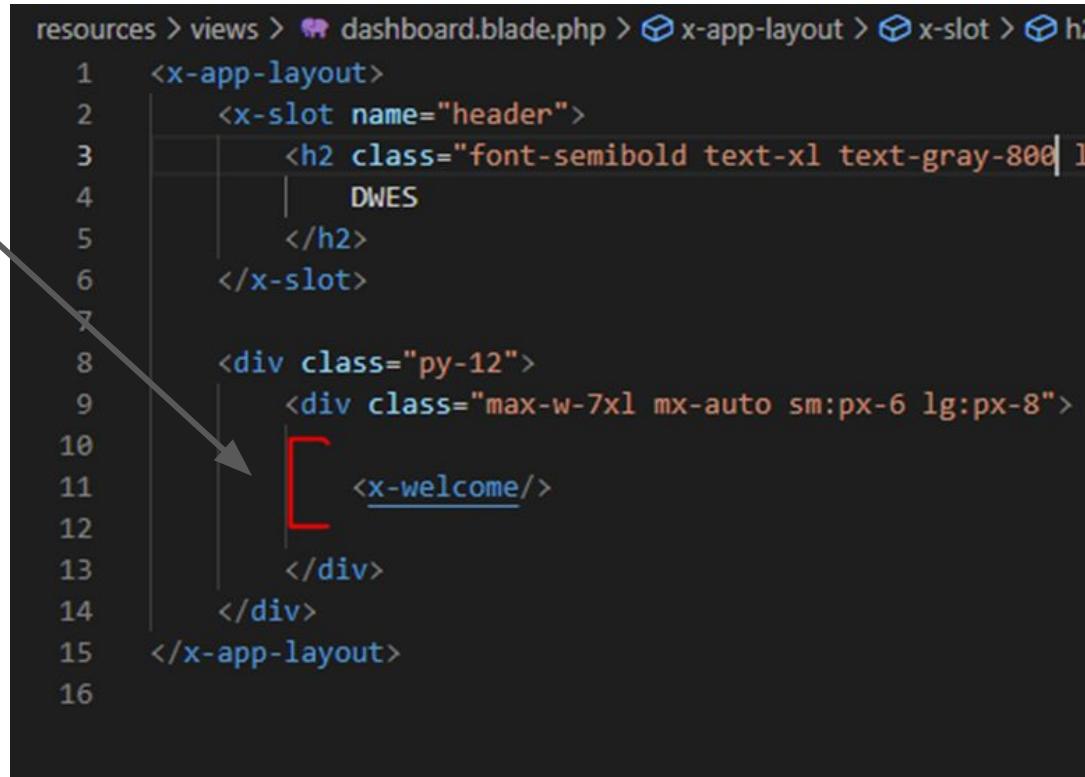
En todo lo que esté en los slots que no sean la cabecera.



```
resources > views > layouts > app.blade.php > html > body.font-sans.antialiased
16
17     <!-- Styles -->
18     @livewireStyles
19     </head>
20
21     <body class="font-sans antialiased">
22         <x-banner />
23
24         <div class="min-h-screen bg-gray-100">
25             @livewire('navigation-menu')
26
27             <!-- Page Heading -->
28             @if (isset($header))
29                 <header class="bg-white shadow">
30                     <div class="max-w-7xl mx-auto py-6 px-6">
31                         {{ $header }}
32                     </div>
33                 </header>
34             @endif
35
36             <!-- Page Content -->
37             <main>
38                 {{ $slot }}
39             </main>
40
41             @stack('modals')
42
```

3. Modificar plantilla JetStream

Nosotros en dashboard.blade.php
tenemos que ahora muestra:

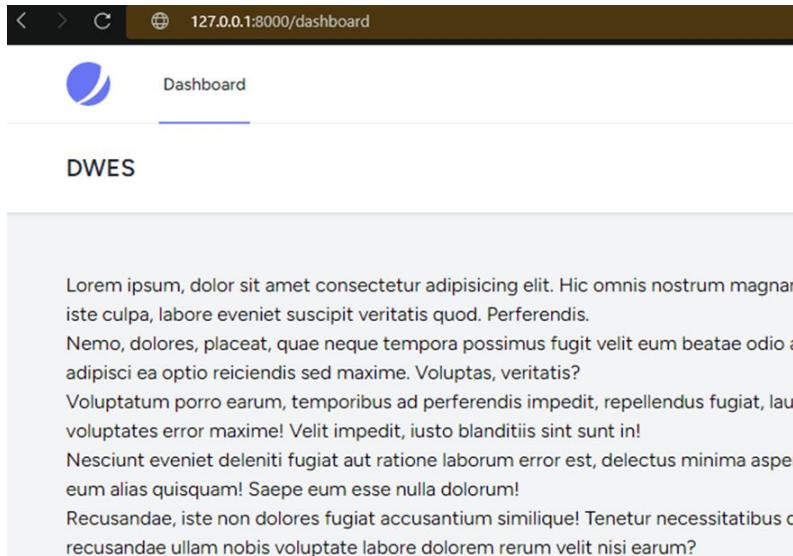


```
resources > views > dashboard.blade.php > x-app-layout > x-slot > h2
1  <x-app-layout>
2    <x-slot name="header">
3      <h2 class="font-semibold text-xl text-gray-800">
4        DWES
5      </h2>
6    </x-slot>
7
8    <div class="py-12">
9      <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
10     <x-welcome/>
11
12   </div>
13 </div>
14 </x-app-layout>
15
16
```

3. Modificar plantilla JetStream

Si lo eliminamos y ponemos, por ejemplo, texto de relleno:

```
<div class="max-w-7xl">  
    p*5>lorem  
</div>
```



The screenshot shows a browser window with the URL 127.0.0.1:8000/dashboard. The page title is "Dashboard". Below it, the heading "DWES" is underlined. A large text area contains five paragraphs of placeholder text: "Lorem ipsum, dolor sit amet consectetur adipisicing elit. Hic omnis nostrum magnam iste culpa, labore eveniet suscipit veritatis quod. Preferendis. Nemo, dolores, placeat, quae neque tempora possimus fugit velit eum beatae odio as adipisci ea optio reiciendis sed maxime. Voluptas, veritatis? Voluptatum porro earum, temporibus ad preferendis impedit, repellendus fugiat, laud voluptates error maxime! Velit impedit, iusto blanditiis sint sunt in! Nesciunt eveniet deleniti fugiat aut ratione laborum error est, delectus minima aspernatur alias quisquam! Saepe eum esse nulla dolorum! Recusandae, iste non dolores fugiat accusantium similique! Tenetur necessitatibus ducimus recusandae ullam nobis voluptate labore dolorem rerum velit nisi earum?"

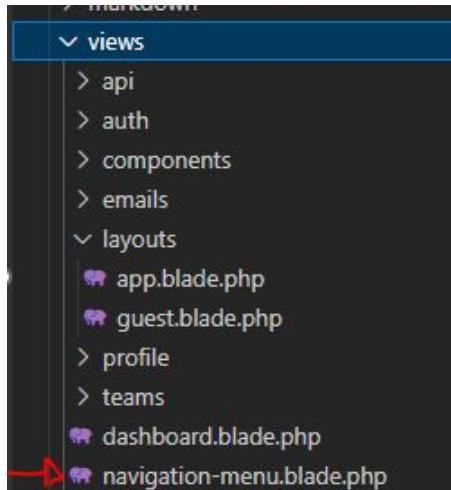
Y lo probamos:



1.1 Agregando links al menú de navegación

El menú de navegación se encuentra:

Este componente renderiza la vista
navigation-menu.blade.php
que se encuentra en
resources/views/.



```
resources > views > layouts > app.blade.php > html > body.html
8         <title>{{ config('app.name', 'Laravel') }}</title>
9
10        <!-- Fonts -->
11        <link rel="preconnect" href="https://fonts.googleapis.com">
12        <link href="https://fonts.googleapis.com/css2?family=Nunito:wght@400;600;700;800&display=swap" rel="stylesheet">
13
14        <!-- Scripts -->
15        @vite(['resources/css/app.css', 'resources/js/app.js'])
16
17        <!-- Styles -->
18        @livewireStyles
19    </head>
20    <body class="font-sans antialiased">
21        <x-banner />
22
23        <div class="min-h-screen bg-gray-100">
24            @livewire('navigation-menu')
```

1.1 Agregando links al menú de navegación

Los links se imprimen en `navigation-menu.blade.php` están siendo impresos a través del componente `<x-nav-link>` y lo encontramos en `views/components`

```
resources > views > components > 🐾 nav-link.blade.php > ...
1  @props(['active'])
2
3  @php
4  $classes = ($active ?? false)
5  |
6  |     ? 'inline-flex items-center px-1 pt-1 border-b-2 border-indigo-400 text-sm +'
7  |     : 'inline-flex items-center px-1 pt-1 border-b-2 border-transparent text-sm'
8  @endphp
9
10 <a {{ $attributes->merge(['class' => $classes]) }}>
11   {{ $slot }}
12 </a>
```

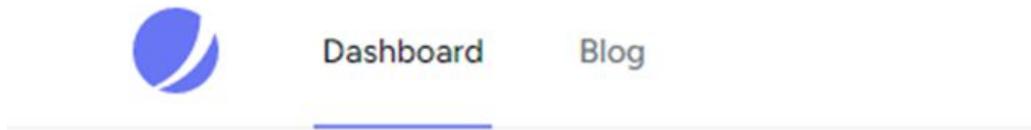
Lo primero que apreciamos es que se ha definido que debemos pasarle un atributo por el componente, ese atributo debe llevar el nombre **active** y puede ser true o false.

1.1 Agregando links al menú de navegación

Para agregar otro link, en
navigation-menu.blade.php

```
11 |         </div>
12 |
13 |         <!-- Navigation Links -->
14 |         <div class="hidden space-x-8 sm:-my-px sm:ml-10 sm:flex">
15 |             <x-nav-link href="{{ route('dashboard') }}" :active="re
16 |             | {{ __('Dashboard') }}"
17 |             </x-nav-link>
18 |
19 |             <x-nav-link :active="false">
20 |             | Blog
21 |             </x-nav-link>
22 |         |
```

Si lo probamos:



1.1 Agregando links al menú de navegación

Para que los links sean dinámicos, es decir, cuando estemos en la página blog, se ponga en oscurita ponemos:

```
<x-nav-link :active="request()->routeIs('blog')">
|   Blog
</x-nav-link>
```

NOTA: En resources/components están todos los archivos que se pueden modificar visualmente. Estos archivos los ha creado Jetstream para que no nos tengamos que preocupar del diseño si no del desarrollo web.

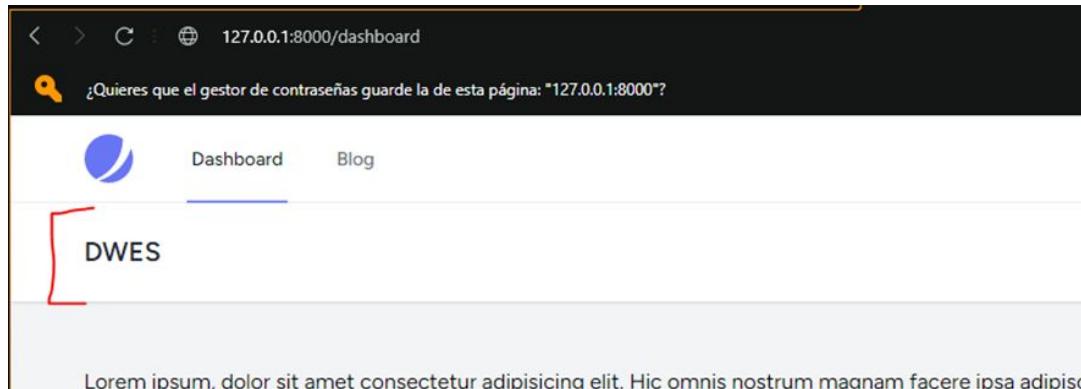
4. Reutilizar una plantilla JetStream

Veamos cómo convertir la plantilla de Jetstream en nuestra plantilla principal y poder extenderla a todas las vistas de nuestra aplicación.

La plantilla la encontramos en
resources/views/layouts
app.blade.php

En la sesión Page Heading se
encuentra lo correspondiente a:

```
26      <!-- Page Heading -->
27      @if (isset($header))
28          <header class="bg-white shadow">
29              <div class="max-w-7xl mx-auto py-6 px-4 sm:px-6 lg:px-8">
30                  {{ $header }}
31              </div>
32          </header>
33      @endif|
```



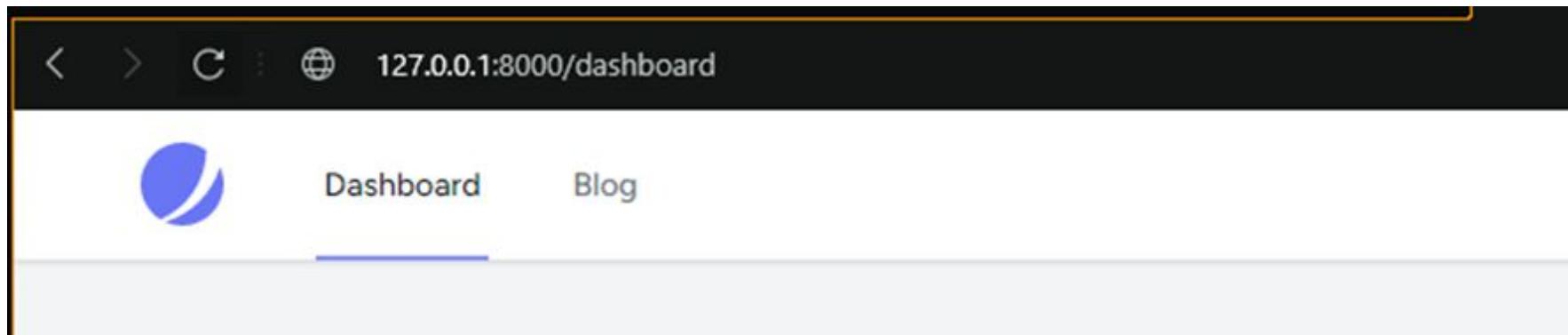
4. Reutilizar una plantilla JetStream

Podemos eliminar la sección Page Heading de esta plantilla y luego agregamos el resaltado que había al nav de la plantilla `navigation-menu.blade.php`

Nos vamos a `navigation-menu.blade.php`

Y en la primera línea, agregamos la clase **shadow** de Tailwind, con esto lo que hace es agregar una sombra a la línea entre el menú y el contenido principal de la página

```
resources > views > navigation-menu.blade.php > nav.bg-white.border-b.border-gray-100.shadow ↵
  1 <nav x-data="{ open: false }" class="bg-white border-b border-gray-100 shadow">
```



4. Reutilizar una plantilla JetStream

Centrándonos en el archivo `navigation-menu.blade.php`

Observamos que está dividido en **2 secciones**:

- `<!-- Primary Navigation Menu -->` sería el menú que estamos viendo con nuestro navegador
- `<!-- Responsive Navigation Menu -->` el menú para móviles

```
resources > views > navigation-menu.blade.php > nav.bg-white.border-b.border-gray-100.shadow
1   <nav x-data="{ open: false }" class="bg-white border-b border-gray-100 shadow">
2     <!-- Primary Navigation Menu -->
3   >   <div class="max-w-7xl mx-auto px-4 sm:px-6 lg:px-8">...
141
142
143     <!-- Responsive Navigation Menu -->
144   >   <div :class="{ 'block': open, 'hidden': ! open}" class="hidden sm:hidden">...
220
221   </div>
222 </nav>
```

4. Reutilizar una plantilla JetStream

Centrándonos en Primary Navigation Menu:

LOGO: Lo primero que nos encontramos es el logo.

```
5      <div class="flex">
6          <!-- Logo -->
7          <div class="shrink-0 flex items-center">
8              <a href="{{ route('dashboard') }}>
9                  <x-application-mark class="block h-9 w-auto" />
10             </a>
11         </div>
```



Se encuentra en: resources/views/components/application-mark.blade.php

Si queremos cambiarlo, tendríamos que modificar:

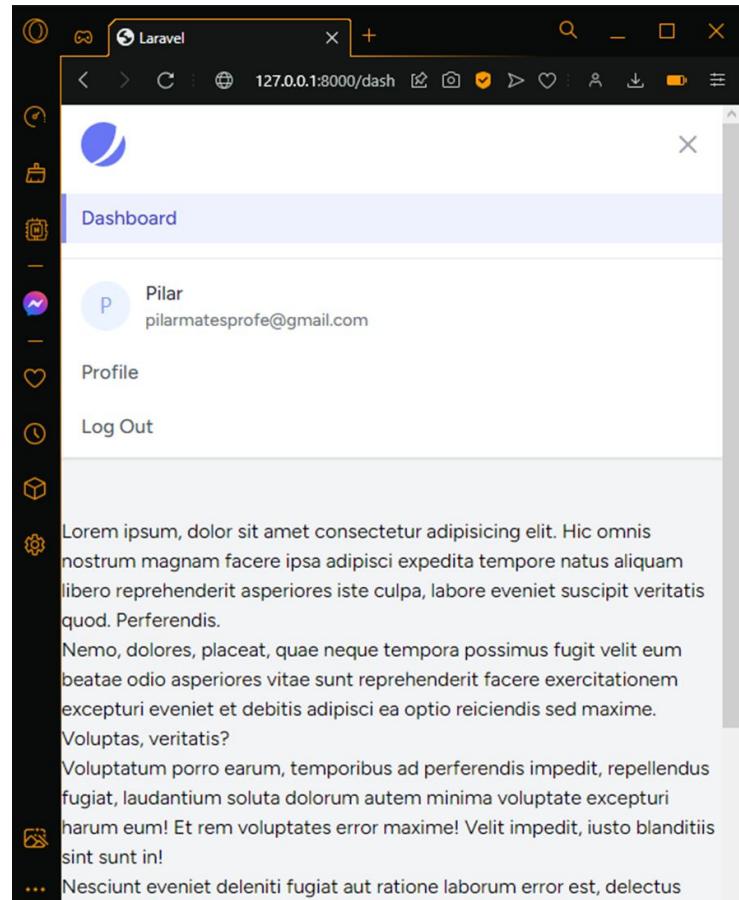
```
resources > views > components > 🐾 application-mark.blade.php > ...
1   <svg viewBox="0 0 48 48" fill="none" xmlns="http://www.w3.org/2000/svg" {{ $attributes }}>
2     <path d="M11.395 44.428C4.557 40.198 0 32.632 0 24 0 10.745 10.745 0 24 0a23.891 23.891 0 0113.997 4.5
3     <path d="M14.134 45.885A23.914 23.914 0 0024 48c13.255 0 24-10.745 24-24 0-3.516-.756-6.856-2.115-9.86
4   </svg>
```

4. Reutilizar una plantilla JetStream

LINKS aquí es donde se colocan los links, como ya hemos visto

Pero qué ocurre. Si nos vamos a la vista móvil:

No aparece el link de blog, solo el de Dashboard



4. Reutilizar una plantilla JetStream

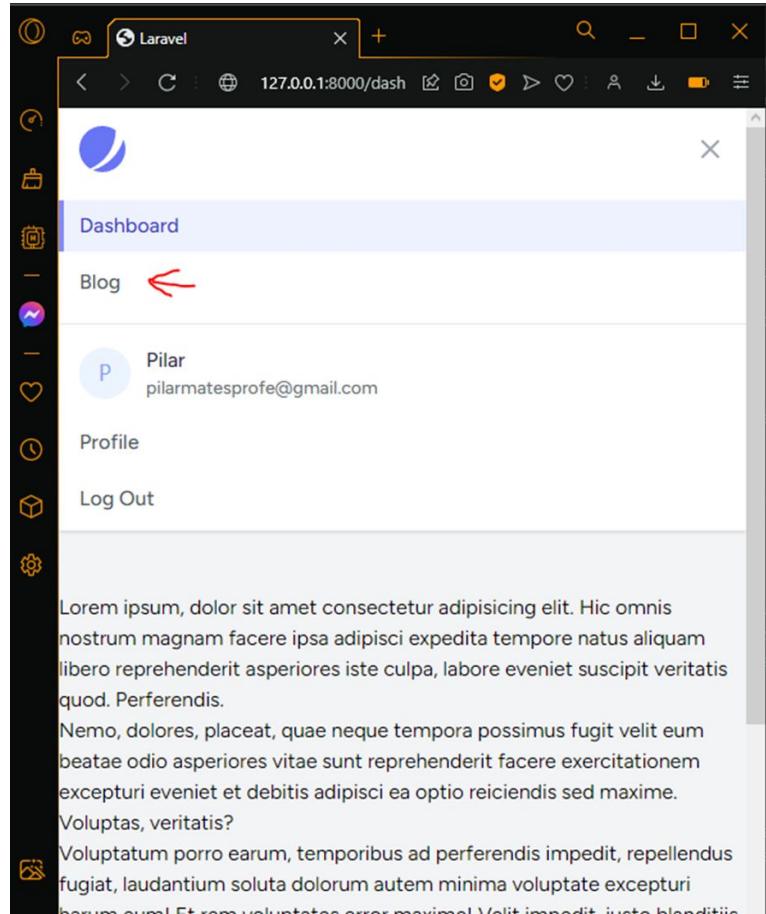
Para solucionarlo: Nos vamos a la sección para móviles: Responsive Navigation Menu y lo agregamos: pero poniendo x-responsive-

```
143    <!-- Responsive Navigation Menu -->
144    <div :class="{'block': open, 'hidden': ! open}" class="hidden sm:hidden">
145        <div class="pt-2 pb-3 space-y-1">
146            <x-responsive-nav-link href="{{ route('dashboard') }}" :active="request()>routeIs('dashbo
147            | {{__('Dashboard')}} }
148            </x-responsive-nav-link>
149
150            <x-responsive-nav-link :active="request()>routeIs('blog')">
151            | Blog
152            </x-responsive-nav-link>
153
154        </div>
155    </div>
```

4. Reutilizar una plantilla JetStream

Si lo probamos:

Por tanto, es importante, siempre que añadamos un link ponerlo también en la vista para móviles.



4. Reutilizar una plantilla JetStream

Modificar el comportamiento del logo

Ahora mismo si clickamos sobre el logotipo, nos redirige a Dashboard. Pero lo normal es que nos redirija a la página principal. Cómo hacemos esto: nos vamos a nuestro archivo de rutas: web.php y a nuestra ruta principal le damos un nombre:

Y ahora en
navigation-menu.blade.php
en la parte de logo,
cambiamos dashboard por
home

```
16  Route::get('/', function () {  
17      |     return view('welcome');  
18  })->name('home');
```

```
5   <div class="flex">  
6       <!-- Logo -->  
7       <div class="shrink-0 flex items-center">  
8           <a href="{{ route('home') }}>  
9               <x-application-mark class="block h-9 w-auto" />  
10          </a>  
11      </div>
```

4. Reutilizar una plantilla JetStream

Extender nuestra plantilla a la vista principal

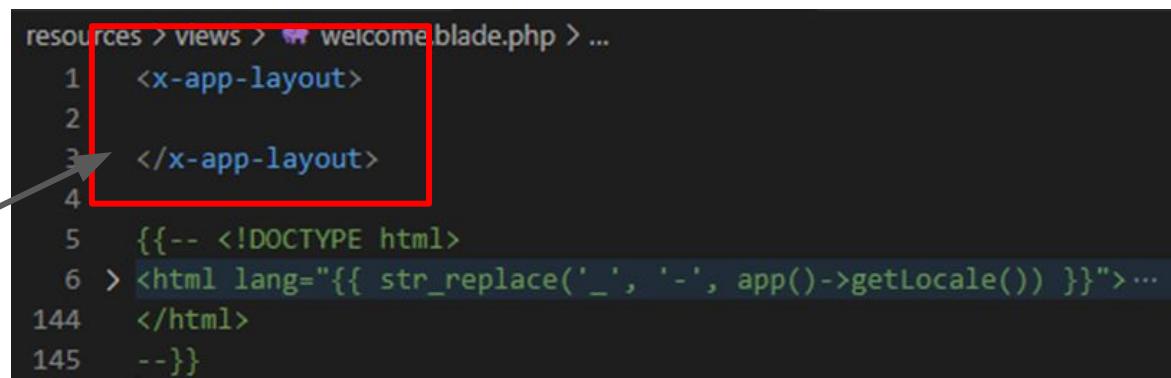
Ahora lo que queremos hacer es extender nuestra plantilla (dashboard) a la vista principal.

Vamos a resources/views welcome.blade.php

Y comentamos todo lo que tenemos {{ -- --}}

Para extender el contenido de la plantilla principal: (app.blade) que hace uso, como vimos, del componente AppLayouts.

Entonces ponemos:



```
resources > views > welcome.blade.php > ...
1   <x-app-layout>
2
3   </x-app-layout>
4
5   {{-- !DOCTYPE html
6   > <html lang="{{ str_replace('_', '-', app()->getLocale()) }}">...
144  </html>
145  --}}

```

4. Reutilizar una plantilla JetStream

Extender nuestra plantilla a la vista principal

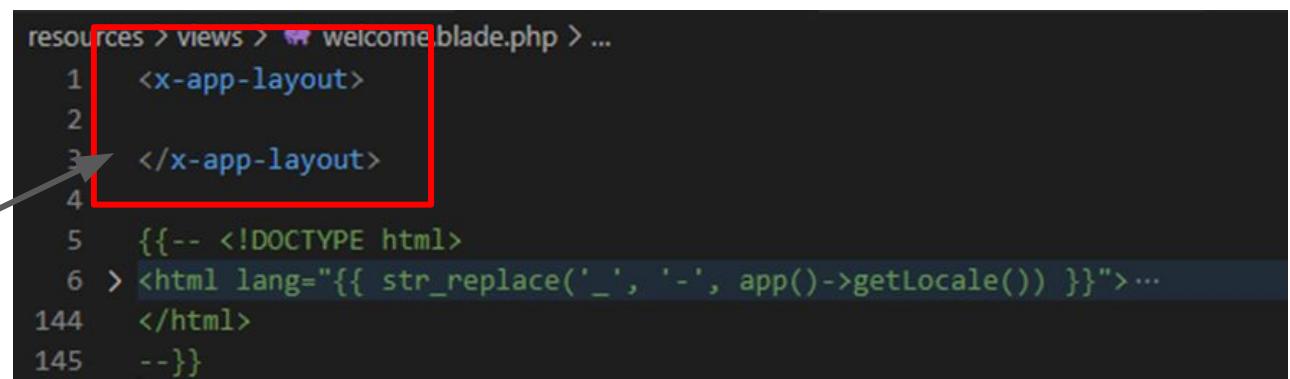
Ahora lo que queremos hacer es extender nuestra plantilla (dashboard) a la vista principal.

Vamos a resources/views welcome.blade.php

Y comentamos todo lo que tenemos {{ -- --}}

Para extender el contenido de la plantilla principal: (app.blade) que hace uso, como vimos, del componente AppLayouts.

Entonces ponemos:



```
resources > views > welcome.blade.php > ...
1  <x-app-layout>
2
3  </x-app-layout>
4
5  <!-- !DOCTYPE html-->
6  > <html lang="{{ str_replace('_', '-', app()->getLocale()) }}">...
144 </html>
145 -->
```

Parece que está funcionando perfecto pero cuando cerramos sesión, **nos da un error**.

4. Reutilizar una plantilla JetStream

Extender nuestra plantilla a la vista principal

Nos vamos a navigation-menu y nos centramos en:

Tanto Teams Dropdown como Setting dropdown son configuraciones para cuando el usuario esté autenticado.

Para cuando no: vamos a usar la directiva de Blade @auth @endauth que va a mostrar ese contenido sólo cuando el usuario esté autenticado.



```
27
28     <div class="hidden sm:flex sm:items-center sm:ml-6">
29         <!-- Teams Dropdown -->
30     > @if (Laravel\Jetstream\Jetstream::hasTeamFeatures())
31         @endif
32
33         <!-- Settings Dropdown -->
34
35     > <div class="ml-3 relative">...
36
37         </div>
38
39     </div>
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
```

```
28     <div class="hidden sm:flex sm:items-center sm:ml-6">
29
30         @auth
31             <!-- Teams Dropdown -->
32         > @if (Laravel\Jetstream\Jetstream::hasTeamFeatures())
33             @endif
34
35             <!-- Settings Dropdown -->
36
37         > <div class="ml-3 relative">...
38
39             </div>
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
59
60
61
62
63
64
65
66
67
68
69
69
70
71
72
73
74
75
76
77
78
79
79
80
81
82
83
84
85
86
87
88
89
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
109
110
111
112
113
114
115
116
117
118
119
119
120
121
122
123
124
125
126
127
128
129
129
130
131
132
133
134
135
136
137
```

4. Reutilizar una plantilla JetStream

Extender nuestra plantilla a la vista principal

Podemos poner también algunas condiciones para cuando no esté autenticado.
Por ejemplo, unos links que vayan a register y login

Estos links están en welcome.blade.php :

```
    <a href="{{ url('/dashboard') }}" class="font-semibold text-gray-600 hover:text-gray-900 fo
@else
    ↗<a href="{{ route('login') }}" class="font-semibold text-gray-600 hover:text-gray-900 fo
        @if (Route::has('register'))
            ↗<a href="{{ route('register') }}" class="ml-4 font-semibold text-gray-600 hover:text
            @endif
@endauth
```

Los copiamos y los pegamos en navigation-menu.blade de tal forma:

4. Reutilizar una plantilla JetStream

Extender nuestra plantilla a la vista principal

```
>           @auth
>             <!-- Teams Dropdown -->
>             @if (Laravel\Jetstream\Jetstream::hasTeamFeatures())
>               @endif

>             <!-- Settings Dropdown -->

>             <div class="ml-3 relative">...
>               </div>

@else
  ↗  <a href="{{ route('login') }}" class="font-semibold text-gray-600 hover:text-gray-900 focus:outline-none" type="button">Iniciar sesión
  ↗  <a href="{{ route('register') }}" class="ml-4 font-semibold text-gray-600 hover:text-gray-900 focus:outline-none" type="button">Registrarse

@endauth
```

4. Reutilizar una plantilla JetStream

Extender nuestra plantilla a la vista principal

Y hacemos lo mismo para el navegador de móvil: Responsive Setting Options

```
72
73     <!-- Responsive Settings Options -->
74     @auth
75
76     >         <div class="pt-4 pb-1 border-t border-gray-200">...
77         </div>
78
79     @else
80
81         <div class="py-1 border-t border-gray-200">
82             <x-responsive-nav-link href="{{ route('login') }}" :active="request()>routeIs('login')">
83                 Login
84             </x-responsive-nav-link>
85
86             <x-responsive-nav-link href="{{ route('register') }}" :active="request()>routeIs('register')">
87                 Register
88             </x-responsive-nav-link>
89         </div>
90
91     @endauth
92
93     </div>
94
95 </nav>
```

Si lo comprobamos, no nos da ningún error.

4. Reutilizar una plantilla JetStream

Cambiar el nombre del Dashboard

Por último, para cambiar el nombre de Dashboard por Home por ejemplo, nos vamos a:

```
12
13          <!-- Navigation Links -->
14  <div class="hidden space-x-8 sm:-my-px sm:ml-10 sm:flex">
15
16      <x-nav-link href="{{ route('home') }}" :active="request()->routeIs('home')">
17          | {{ __('Home') }}
18      </x-nav-link>
19
```

Y en la parte móvil:

```
159     <!-- Responsive Navigation Menu -->
160     <div :class="{'block': open, 'hidden': ! open}" class="hidden sm:hidden">
161         <div class="pt-2 pb-3 space-y-1">
162             <x-responsive-nav-link href="{{ route('home') }}" :active="request()->routeIs('home')">
163                 | {{ __('Home') }}
164             </x-responsive-nav-link>
165
```

5. Middleware

Middleware es un software que proporciona un mecanismo para filtrar las solicitudes que ingresan a nuestra aplicación.

Por ejemplo, Laravel incluye un middleware que verifica que el usuario de su aplicación esté autenticado. Si no lo está, el middleware redirige al usuario a la pantalla de inicio de sesión, y si lo está, permitirá que la solicitud continue en la aplicación.

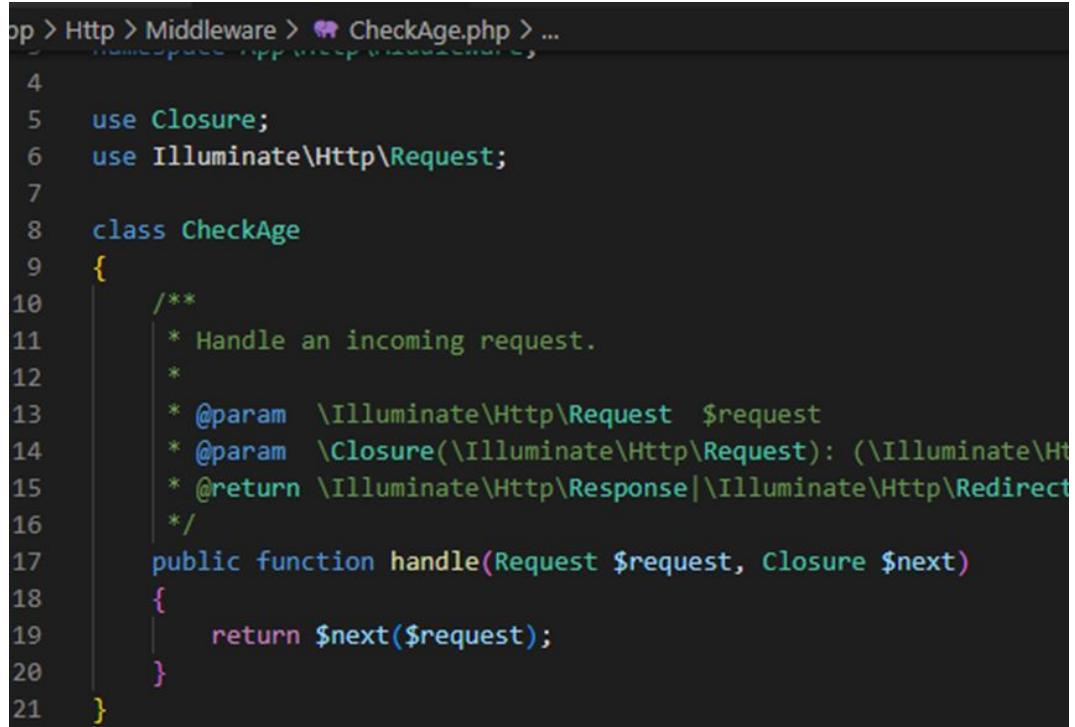
Vamos a crear un middleware, por ejemplo, que nos deje o no acceder según la edad que tengamos. Hay que seguir una serie de pasos:

5. Middleware

Crear un middleware: Escribimos `php artisan make:middleware CheckAge`

Se crea en
app/Http/Middleware/CheckAge.php

Este archivo contiene una clase `CheckAge` con un método `handle` que se encargará de filtrar las solicitudes HTTP.



The screenshot shows a code editor with the file `CheckAge.php` open. The file is located in the `app/Http/Middleware` directory. The code defines a middleware class `CheckAge` with a `handle` method. The code is annotated with PHPDoc comments and uses the `Illuminate\Http\Request` and `Closure` interfaces.

```
 4
 5  use Closure;
 6  use Illuminate\Http\Request;
 7
 8  class CheckAge
 9  {
10      /**
11       * Handle an incoming request.
12       *
13       * @param \Illuminate\Http\Request $request
14       * @param Closure(\Illuminate\Http\Request): (\Illuminate\Http\Response|\Illuminate\Http\RedirectResponse) $next
15       */
16
17      public function handle(Request $request, Closure $next)
18      {
19          return $next($request);
20      }
21  }
```

5. Middleware

Registrar un middleware:

Para usar nuestro nuevo middleware debemos registrarlo en `App/Http/Kernel.php`

```
55     protected $middlewareAliases = [
56         'auth' => \App\Http\Middleware\Authenticate::class,
57         'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
58         'auth.session' => \Illuminate\Session\Middleware\AuthenticateSession::class,
59         'cache.headers' => \Illuminate\Http\Middleware\SetCacheHeaders::class,
60         'can' => \Illuminate\Auth\Middleware\Authorize::class,
61         'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
62         'password.confirm' => \Illuminate\Auth\Middleware\RequirePassword::class,
63         'signed' => \App\Http\Middleware\ValidateSignature::class,
64         'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
65         'verified' => \Illuminate\Auth\Middleware\EnsureEmailIsVerified::class,
66         'age' => \App\Http\Middleware\CheckAge::class,
67     ];
68 }
```

5. Middleware

Asignar el middleware a la ruta

En web.php creamos dos rutas:

```
22 Route::get('prueba', function () {
23     return "Has accedido correctamente a esta ruta.");
24     ->middleware('age');
25
26 Route::get('no-autorizado', function () {
27     return "Ud. no es mayor de edad.";
28});|
```

5. Middleware

Probando el middleware

Volvemos a nuestro middleware CheckAge.php

Y escribimos en nuestra función handle

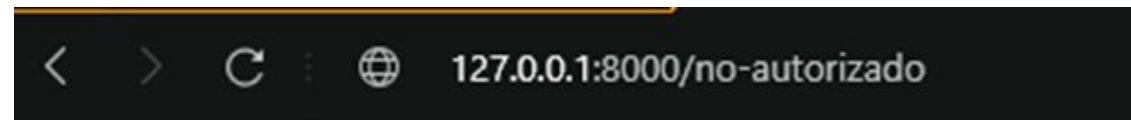
```
17     public function handle(Request $request, Closure $next)
18     {
19         if($request->age >= 18) {
20             return $next($request);
21         } else {
22             return redirect('no-autorizado');
23         }
24     }
25 }
```

5. Middleware

Probando el middleware

Si lo probamos:

127.0.0.1:8000/prueba?age=5



127.0.0.1:8000/prueba?age=20



5. Middleware

Cambiando el tipo de comprobación

Para no depender de un parámetro pasado por la URL y utilizar la información del usuario actualmente logueado hacemos lo siguiente en el método handle de CheckAge.php:

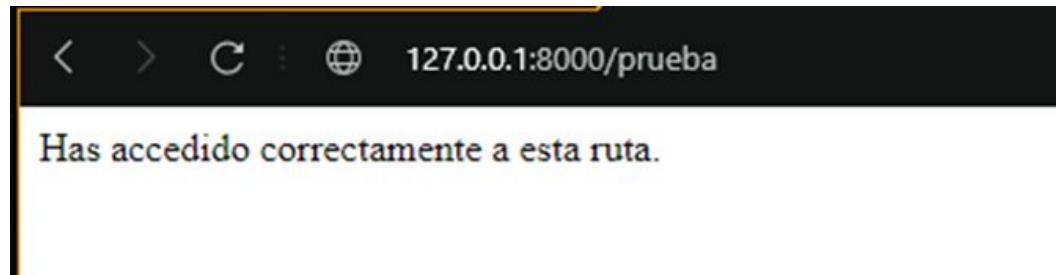
Pasamos el email porque no tenemos registrado la edad.

```
10
11     />
12
13     public function handle(Request $request, Closure $next)
14     {
15         if( auth()->user()->email == "pilar.matesprofe@gmail.com" ) {
16             return $next($request);
17         } else {
18             return redirect('no-autorizado');
19         }
20     }
21 }
```

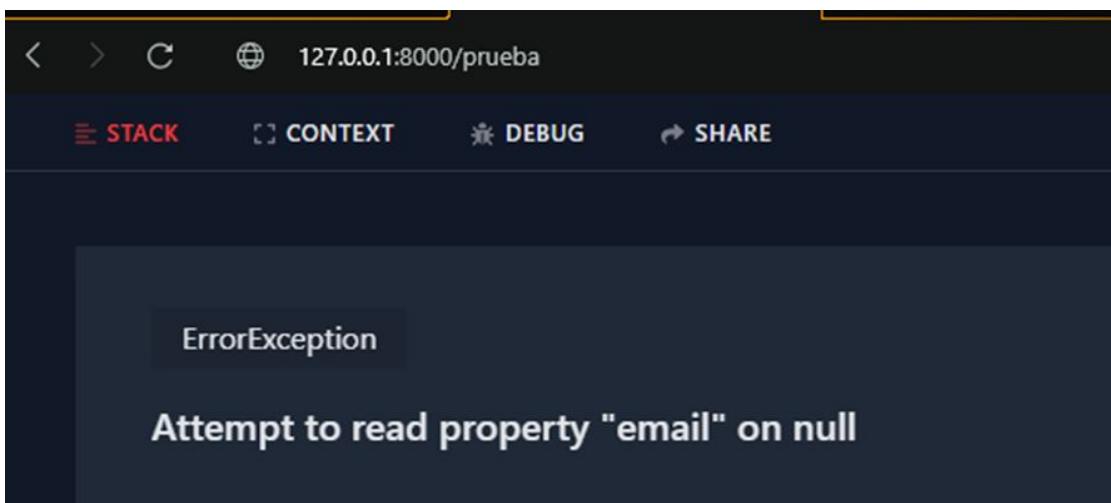
5. Middleware

Cambiando el tipo de comprobación

Si lo probamos estando autenticado:



Pero si lo probamos sin estar autenticado, nos da un error:



5. Middleware

Cambiando el tipo de comprobación

Para el caso de no estar autenticado el anterior código produce un error ya que no encuentra la propiedad email.

Para evitar este error actualizamos la ruta en web.php:

```
21
22 Route::get('prueba', function () {
23     return "Has accedido correctamente a esta ruta.";
24     ->middleware(['auth:sanctum', 'age']);
25 }
```

De esta forma la ruta debe cumplir con los dos middlewares y si no estamos autenticados nos redirige a la pantalla de login.

Unidad 7

Framework Laravel 9 - Laravel Jetstream
Páginas web dinámicas con Laravel Livewire

1. Laravel Livewire

Laravel Livewire es un framework full-stack escrito en PHP para Laravel, que permite crear aplicaciones web dinámicas e interactivas sin necesidad de escribir código JavaScript

En nuestro proyecto jetstream agregamos un nuevo componente:

```
Php artisan make:livewire ShowPosts
```



Como ya hemos visto, esto nos crea dos archivos:

La lógica: que se encuentra en app/Http/Livewire -> ShowPosts.php

Y la vista que se accede a través de la lógica: show-posts.blade.php

1. Laravel Livewire

Para crear páginas web dinámicas, lo que hacemos es usar los componentes de Livewire como si fueran **controladores**. Para ello:

Nos vamos a `web.php` e importamos el componente que acabamos de crear:



```
routes > 🌐 web.php > ...
1   <?php
2
3
4   use Illuminate\Support\Facades\Route;
5   use App\Http\Livewire>ShowPosts;
6
```

Ahora en las rutas, borramos la función anónima y ponemos:

```
21
22   Route::middleware([
23     'auth:sanctum',
24     config('jetstream.auth_session'),
25     'verified'
26   ])->group(function () {
27     Route::get('/dashboard', ShowPosts::class)->name('dashboard');
28   });

```

2. Tabla dinámica

Para poner un ejemplo de página web dinámica, vamos a crear una tabla de datos que sea dinámica.

Creamos la tabla vamos a crear una tabla y la vamos a llenar de datos de prueba como ya hemos visto

Creamos un modelo que se llame Post con migraciones y factories:

Php artisan make:model Post -mf

Nos vamos a database/migrations y abrimos la migración que se ha creado y escribimos dos columnas: title y content



```
database > migrations > 2023_03_06_171346_create_posts_table.php > class
4  use Illuminate\Database\Schema\Blueprint;
5  use Illuminate\Support\Facades\Schema;
6
7  return new class extends Migration
8  {
9      /**
10      * Run the migrations.
11      */
12      public function up(): void
13      {
14          Schema::create('posts', function (Blueprint $table)
15          {
16              $table->id();
17
18              $table->string('title');
19              $table->string('content');
20
21              $table->timestamps();
22          });
23      }
24 }
```

2. Tabla dinámica

Agregamos datos

Abrimos el Factories para agregar
datos falsos database/factories
PostFactory.php



```
17     public function definition(): array
18     {
19         return [
20             'title'=>$this->faker->sentence(),
21             'content'=>$this->faker->text()
22         ];
23     }
24 }
```

Abrimos el Seeder
database/seeder
DataBaseSeeder.php



```
public function run(): void
{
    // \App\Models\User::factory(10)->create();

    // \App\Models\User::factory()->create([
    //     'name' => 'Test User',
    //     'email' => 'test@example.com',
    // ]);

    \App\Models\Post::factory(100)->create();
}
```

2. Tabla dinámica

Migramos

Nos vamos al Modelo Post y escribimos:

Una vez tengamos todo esto hecho,
migramos

php artisan migrate:fresh -seed

¡Son dos guiones!

```
app > Models > Post.php > ...
1  <?php
2
3  namespace App\Models;
4
5  use Illuminate\Database\Eloquent\Factories\HasFactory;
6  use Illuminate\Database\Eloquent\Model;
7
8  class Post extends Model
9  {
10     use HasFactory;
11
12     protected $fillable = ['title', 'content'];
13 }
14 |
```

2. Tabla dinámica

The screenshot shows the phpMyAdmin interface for the 'jetstream' database. The left sidebar lists various databases and tables. The 'posts' table under the 'jetstream' database is selected. The main area displays the results of the query `SELECT * FROM `posts``. The results are shown in a table with columns: id, title, and content. There are four rows of data, each with edit, copy, and delete options.

	id	title	content
<input type="checkbox"/> Editar Copiar Borrar	1	Velit error ea itaque non.	Eligendi sed ipsum vel quos. P...
<input type="checkbox"/> Editar Copiar Borrar	2	Atque accusamus quaerat fugiat dolores blanditiis ...	Ducimus qui ratione do molest...
<input type="checkbox"/> Editar Copiar Borrar	3	Ex doloremque est ut odit ut velit et.	Deleniti in reprehenderi minus...
<input type="checkbox"/> Editar Copiar Borrar	4	Cum autem rerum ut laudantium.	Id ratione explicabo a l vol...

2. Tabla dinámica

Nos centramos en nuestro componente ShowPosts, escribimos como si fuera un controlador

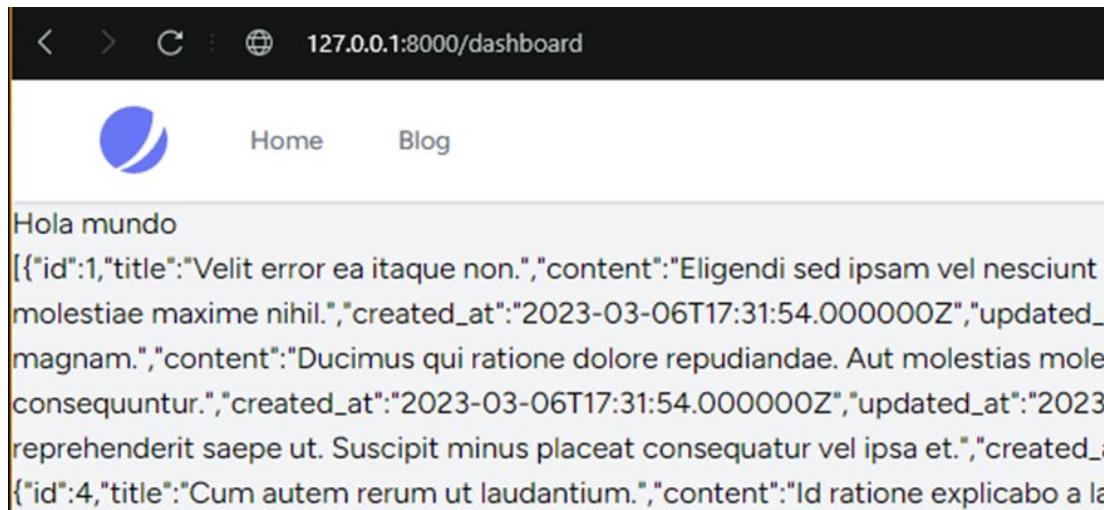
```
app > Http > Livewire > 📄 ShowPosts.php > 🎨 ShowPosts > ⚡ render
1  <?php
2
3  namespace App\Http\Livewire;
4
5  use Livewire\Component;
6  use App\Models\Post;
7
8
9  class ShowPosts extends Component
10 {
11     public $title;
12
13     public function render()
14     {
15
16         $posts = Post::all(); ↗
17
18         return view('livewire.show-posts', compact('posts'));
19     }
20 }
```

2. Tabla dinámica

Si nos vamos a nuestra vista:

Si lo comprobamos, como el usuario no existe porque hemos realizado migraciones, tenemos que volver a crearlo. Logueamos y:

```
resources > views > livewire > 🌈 show-posts.blade.php > ...
1  <div>
2
3      <h1> Hola mundo </h1>
4
5
6      {{ $posts }} //
7
8  </div>
9
```



2. Tabla dinámica

En primer lugar, para centrar este contenido:

Me voy a navigation.menu.blade y copio:

```
resources > views > navigation-menu.blade.php > nav.bg-white.border-b.border-gray
1   <nav x-data="{ open: false }" class="bg-white border-b border-gray">
2     |   <!-- Primary Navigation Menu -->
3     →><div class="max-w-7xl mx-auto px-4 sm:px-6 lg:px-8">
```

Y lo pego en la vista show-posts.blade, agregándole algo de margen: py-12

```
4
5   [ <div class="max-w-7xl mx-auto px-4 sm:px-6 lg:px-8 py-12">
6     {{ $posts }}
7   </div>
8 ]
```

2. Tabla dinámica

Vamos a poner nuestros datos en una **tabla**

Buscamos una tabla: **(os la doy yo)**

<https://flowbite.com/docs/components/tables/>

Este código lo pegamos en lugar
de \$posts de show.blade.php

Creamos un componente Blade: en
la carpeta de views/component
creamos una que se llame:

Table.blade.php

Y copiamos los div de inicio y de
cierre:

Extensión: Laravel Blade formatter
ordena con Alt + Shift + f

```
resources > views > components > 🗂 table.blade.php > ⚙️ div.rela
1   <div class="relative overflow-x-auto">
2     {{$slot}}
3   </div>
```

No tiene mucho sentido con esta tabla pero es mejor
tenerlos separados por si usamos tablas con muchos div

2. Tabla dinámica

Y en show-posts.blade.php llamamos al componente table que acabamos de crear:

```
resources > views > livewire > 🚀 show-posts.blade.php > ⚒ div > ⚒ div.max-w-7xl.mx-auto.px-4.sm:px-6.lg:px-8.py-12
1  <div>
2
3      {{ $title }}
4
5      <div class="max-w-7xl mx-auto px-4 sm:px-6 lg:px-8 py-12">
6
7
8          ➡️ <x-table>
9          |     <table class="w-full text-sm text-left text-gray-500 dark:text-gray-400"> ...
73         |     </table>
74
75          ➡️ </x-table>
```

2. Tabla dinámica

Modificando la tabla (os la doy modificada)

```
8 <x-table>
9     <table class="w-full text-sm text-left text-gray-500 dark:text-gray-400">
0         <thead class="text-xs text-gray-700 uppercase bg-gray-50 dark:bg-gray-700 dark:text-gray-400">
1             <tr>
2                 <th scope="col" class="px-6 py-3">
3                     Id
4                 </th>
5                 <th scope="col" class="px-6 py-3">
6                     Title
7                 </th>
8                 <th scope="col" class="px-6 py-3">
9                     Content
0                 </th>
1                 <th scope="col" class="px-6 py-3">
2                     Action
3                 </th>
4             </tr>
5         </thead>
```



2. Tabla dinámica

Modificando la tabla

Y para que se vayan llenando con datos, creamos un foreach

```
<tbody>
    @foreach ($posts as $post)
        {
            <tr class="bg-white border-b dark:bg-gray-900 dark:border-gray-700">
                <th scope="row"
                    class="px-6 py-4 font-medium text-gray-900 whitespace nowrap dark:text-white">
                    {{$post->id}}
                </th>
                <td class="px-6 py-4">
                    {{$post->title}}
                </td>
                <td class="px-6 py-4">
                    {{$post->content}}
                </td>
                <td class="px-6 py-4">
                    <a href="#" 
                        class="font-medium text-blue-600 dark:text-blue-500 hover:underline">Edit</a>
                </td>
            </tr>
        }
    </tbody>
    @endforeach
```

2. Tabla dinámica

Creamos un buscador en la tabla

En primer lugar, en el **componente**

En la **vista**, lo que vamos a hacer es crear un input:

```
app > Http > Livewire > ShowPosts.php > ...
1  <?php
2
3  namespace App\Http\Livewire;
4
5  use Livewire\Component;
6  use App\Models\Post;
7
8
9  class ShowPosts extends Component
10 {
11
12     public $search = "Esto es lo que se buscará";
13
14     public function render()
15     {
16
17         $posts = Post::all();
18
19         return view('livewire.show-posts', [
20             'posts' => $posts,
21             'search' => $this->$search
22         ]);
23     }
24 }
```

```
<x-table>  
  <div class="px-6 py-3">  
    <input type="text" wire:model="search">  
  </div>  
  
  <table class="w-full text-sm text-left text-gray-500 dark:  
    <thead>
```

La propiedad `wire:model` lo que hace es llamar al `search` que acabamos de crear.

2. Tabla dinámica

Creamos un buscador en la tabla

The screenshot shows a web application interface. At the top, there is a dark header bar with navigation icons (back, forward, refresh) and a URL field displaying "127.0.0.1:8000/dashboard". Below the header, the page has a light gray header section featuring a blue circular logo on the left, followed by "Home" and "Blog" links. The main content area contains a search input field with a blue border and the placeholder text "Esto es lo que se buscar". Below the search field is a table with a dark header row containing columns labeled "ID" and "TITLE". The first data row visible shows an ID of 1 and a title of "Velit error ea itaque non.". To the right of the table, there are some partially visible columns with text like "co", "Eli", and "es".

ID	TITLE	co
1	Velit error ea itaque non.	Eli

2. Tabla dinámica

Creamos un buscador en la tabla

Si modificamos ShowPosts, de la siguiente manera:

Lo que hacemos con where es filtrar las palabras del título.
Entonces en nuestra tabla, buscamos palabras por el título.

```
9  class ShowPosts extends Component
10 {
11
12     public $search;
13
14     public function render()
15     {
16
17         $posts = Post::where('title', 'like', '%' . $this->search . '%')->get();
18         return view('livewire.show-posts', compact('posts'));
19     }
20 }
```

Si queremos filtrarlos por contenido o por título:

```
class ShowPosts extends Component
{
    public $search;

    public function render()
    {
        $posts = Post::where('title', 'like', '%' . $this->search . '%')
                    ->orwhere('content', 'like', '%' . $this->search . '%')->get();
        return view('livewire.show-posts', compact('posts'));
    }
}
```

2. Tabla dinámica

Creamos un buscador en la tabla

Para darle estilos a este buscador, hacemos uso del componente Blade (que recordamos está en la carpeta: `resources/views/components`).

En particular: `input.blade.php` porque es un input lo que estoy usando.

Para usarlo: nos vamos a `show-posts.blade.php`

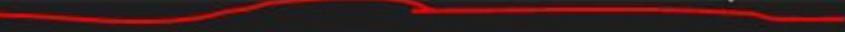
```
6
7      <x-table>
8
9          <div class="px-6 py-3">
10             ↗ <x-input type="text" wire:model="search"/>
11
12         </div>
13
14             <table class="w-full text-sm text-left text-gray-500 dark:text-te
```

2. Tabla dinámica

Creamos un buscador en la tabla

Para que ocupe toda la tabla y poner un texto dentro:

```
<div class="px-6 py-3">
    <x-input class="w-full" placeholder="Escriba lo que quiera buscar" type="text" wire:model="search"/>
</div>
```



2. Tabla dinámica

Creamos un buscador en la tabla

Para no mostrar la tabla cuando busquemos algo que no está.

Comprimimos la tabla y creamos el siguiente if else

```
<x-table>
  <div class="px-6 py-3">
    <x-input class="w-full" placeholder="Escriba lo que quiera buscar" type="text" />
  </div>

  @if($posts->count())
    <table class="w-full text-sm text-left text-gray-500 dark:text-gray-400">
      <thead>
        <tr>
          <th>Title</th>
          <th>Category</th>
          <th>Author</th>
          <th>Published At</th>
        </tr>
      </thead>
      <tbody>
        @foreach($posts as $post)
          <tr>
            <td>{$post->title}</td>
            <td>{$post->category}</td>
            <td>{$post->author}</td>
            <td>{$post->published_at}</td>
          </tr>
        @endforeach
      </tbody>
    </table>
  @else
    <div class="px-6 py-3">
      No existe ningún registro que coincida
    </div>
  @endif
</x-table>
```



2. Tabla dinámica

Creamos un buscador en la tabla

The screenshot shows a web browser window with the URL `127.0.0.1:8000/dashboard` in the address bar. The page itself has a dark header with navigation icons (back, forward, refresh) and a search bar. Below the header, there is a logo icon and two menu items: "Home" and "Blog". The main content area contains a search input field with the placeholder text "Search" and a result card below it. The result card displays the number "123123" in a large font. At the bottom of the page, a message states "No existe ningún registro que coincida".

2. Tabla dinámica

Para hacer que la tabla se ordene según id, nombre, contenido.

Vamos a crear en ShowPosts.php

Dos variables que iniciamos en id y en desc

Y usamos el método orderBy que vimos con anterioridad.

Y además, creamos también la función order

```
class ShowPosts extends Component
{
    public $search;
    public $sort = 'id';
    public $direction = 'desc';

    public function render()
    {
        $posts = Post::where('title', 'like', '%' . $this->search . '%')
            ->orwhere('content', 'like', '%' . $this->search . '%')
            ->orderBy($this->sort, $this->direction)
            ->get();
        return view('livewire.show-posts', compact('posts'));
    }
}
```

```
public function order($sort){
    $this->sort = $sort;
}
```

2. Tabla dinámica

Para hacer que la tabla se ordene según id, nombre, contenido.

Modificamos un poco la cabecera de la tabla para que podamos clickar sobre ella y para que nos ordene. Nos vamos a la vista shows-post.blade.php

Y ponemos:

La clase `cursor-pointer` nos permite clicar y `wire:click` nos hace la acción `order` que hemos creado en ShowPosts

```
@if($posts->count())
  <table class="w-full text-sm text-left text-gray-500 dark:text-gray-400">
    <thead class="text-xs text-gray-700 uppercase bg-gray-50 dark:bg-gray-700 dark:text-gray-400">
      <tr>
        <th scope="col" class="cursor-pointer px-6 py-3" wire:click="order('id')">
          Id
        </th>
        <th scope="col" class="cursor-pointer px-6 py-3" wire:click="order('title')">
          Title
        </th>
        <th scope="col" class="cursor-pointer px-6 py-3" wire:click="order('content')">
          Content
        </th>
        <th scope="col" class="px-6 py-3">
          Action
        </th>
      </tr>
    </thead>
    <tbody>
```

2. Tabla dinámica

Para hacer que la tabla se ordene según id, nombre, contenido.

Para que al volver a clicar, nos lo vuelva a ordenar:

Modificamos la función order metiéndole un if.

```
15  
16     public function order($sort){  
17         if ($this->sort==$sort) {  
18             if ($this->direction=='desc') {  
19                 $this->direction='asc';  
20             } else {  
21                 $this->direction='desc';  
22             }  
23         } else {  
24             $this->sort=$sort;  
25             $this->direction='asc';  
26         }  
27     }  
28  
29 }  
30  
31 **C
```

2. Tabla dinámica

Para hacer que la tabla se ordene según id, nombre, contenido.

Para **agregar iconos** que me indiquen cómo está ordenado.

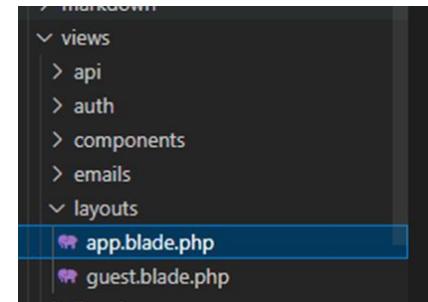
Nos vamos a nuestro proyecto C:\xampp\htdocs\jetstream\public

Y creamos una carpeta que se llame vendor

Y dentro, copiamos la librería que os he dejado en el Classroom: fontawesome

Nos dirigimos a la plantilla app.blade
views/layouts/app.blade.php

Y llamamos a la librería con el método asset



```
18
19      <!-- Styles -->
20      → <link rel="stylesheet" href="{{ asset('vendor/fontawesome/css/all.min.css') }}">
21          @livewireStyles
22
```

**C

2. Tabla dinámica

Para hacer que la tabla se ordene según id, nombre, contenido.

Nos dirigimos a la vista componente.Y copiamos tanto en **Id**, title y content:

```
<th scope="col" class="cursor-pointer px-6 py-3" wire:click="order('id')">
    Id
    @if ($sort=='id')
        @if ($direction == 'asc')
            <i class="fas fa-sort-alpha-up float-right mt-1"></i>
        @else
            <i class="fas fa-sort-alpha-down float-right mt-1"></i>
        @endif
    @else
        <i class="fas fa-sort float-right mt-1"></i>
    @endif
```

2. Tabla dinámica

Para hacer que la tabla se ordene según id, nombre, contenido.

Nos dirigimos a la vista componente.Y copiamos tanto en Id, **title** y content:

```
<th scope="col" class="cursor-pointer px-6 py-3" wire:click="order('title')">
    Title

    @if ($sort=='title')

        @if ($direction == 'asc')
            <i class="fas fa-sort-alpha-up float-right mt-1"></i>
        @else
            <i class="fas fa-sort-alpha-down float-right mt-1"></i>
        @endif

        @else
            <i class="fas fa-sort float-right mt-1"></i>
        @endif |
```

2. Tabla dinámica

Para hacer que la tabla se ordene según id, nombre, contenido.

Nos dirigimos a la vista componente.Y copiamos tanto en Id, title y content:

```
<th scope="col" class="cursor-pointer px-6 py-3" wire:click="order('conte  
Content  
  
@if ($sort=='content')  
  
    @if ($direction == 'asc')  
        <i class="fas fa-sort-alpha-up float-right mt-1"></i>  
    @else  
        <i class="fas fa-sort-alpha-down float-right mt-1"></i>  
    @endif  
  
    @else  
        <i class="fas fa-sort float-right mt-1"></i>  
    @endif
```

2. Tabla dinámica

Agregar botón para crear un registro (Métodos mágicos y modales)

Lo que vamos a hacer es agregar un botón para que se nos abra un formulario y poder agregar un nuevo registro.

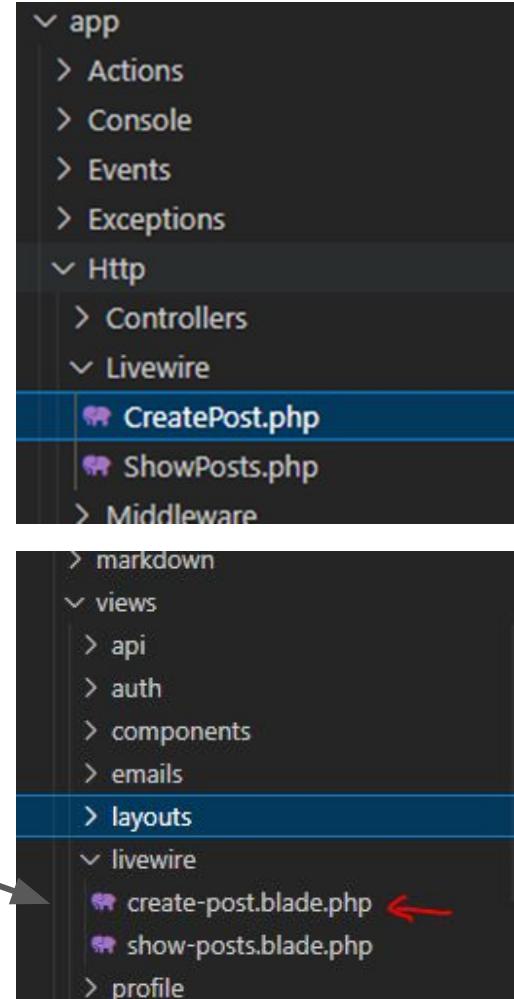
Para ellos creamos un componente:

```
Php artisan make:livewire create-post
```

Como ya sabemos, nos ha creado 2 archivos: la lógica y la vista.

La lógica: App/Http/Livewire CreatePost

Y la vista, que accedemos a través de la lógica pero que se encuentra en: views/livewire create-post.blade.php



2. Tabla dinámica

Agregar botón para crear un registro (Métodos mágicos y modales)

En la vista agregamos el botón, para ello hacemos uso de los componentes de Blade, en este caso el botón rojo que se llama: danger-button

Lo colocamos en la vista de la tabla: show.post.blade

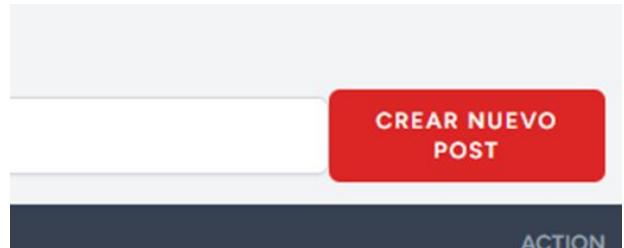
```
resources > views > livewire > 🌐 create-post.blade.php > ...
1  <div>
2    <x-danger-button>
3      Crear nuevo post
4    </x-danger-button>
5  </div>
6
```

```
<div class="px-6 py-3 flex items-center">
  <x-input class="w-full" placeholder="Escriba lo que quiera buscar" type="text" wire:model="search"/>
  ➔ @livewire('create-post')
</div>
```

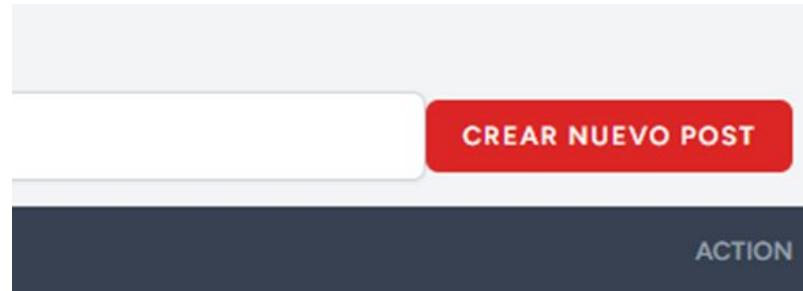
2. Tabla dinámica

Agregar botón para crear un registro (Métodos mágicos y modales)

Para que se vea en una única línea:



```
<div class="px-6 py-3 flex items-center">
  <x-input class="flex-1" placeholder="Escriba lo que quiera buscar" type="text" wire:>
    @livewire('create-post')
```



2. Tabla dinámica

Agregar botón para crear un registro (Métodos mágicos y modales)

Ahora para que al pulsar el botón se nos abra un modal.

También vamos a hacer uso de componente de Blade:

Dialog-modal.blade.php

Nos vamos a la vista de nuestro componente: create-post.blade

```
resources > views > livewire > 🐾 create-post.blade.php > ...
1  <div>
2    <x-danger-button>
3      Crear nuevo post
4    </x-danger-button>
5
6    <x-dialog-modal>
7
8      <x-slot name="title">
9        </x-slot>
10
11      <x-slot name="content">
12        </x-slot>
13
14      <x-slot name="footer">
15        </x-slot>
16
17    </x-dialog-modal>
18
19
20  </div>
```

2. Tabla dinámica

Agregar botón para crear un registro (Métodos mágicos y modales)

Para hacer funcionarlo, en la lógica de esa vista:

Y ahora en la vista, vamos a hacer uso de un método mágico ya que al ser una función sencilla no hace falta meter tanto código en la lógica.

The screenshot shows a code editor with two files. On the left is a blade template named `create-post.blade.php`. It contains the following code:

```
1 <div>
2   <x-danger-button wire:click="$set('open', true)">
3     Crear nuevo post
4   </x-danger-button>
5
6   <x-dialog-modal wire:model="open">
7     <x-slot name="title">
8
9     </x-slot>
```

An arrow points from the `wire:click` attribute in the first `x-danger-button` to the `$open` property in the component class. Another arrow points from the `wire:model` attribute in the `x-dialog-modal` tag to the `render` method.

On the right is the component file `CreatePost.php` located at `app/Http/Livewire/CreatePost.php`. It contains the following code:

```
1 <?php
2
3 namespace App\Http\Livewire;
4
5 use Livewire\Component;
6
7 class CreatePost extends Component
8 {
9   public $open = true;
10
11   public function render()
12   {
13     return view('livewire.create-post');
```

Esto lo que hace es que al hacer click sobre el botón cambia el método de open, mostrando así el diálogo.

2. Tabla dinámica

Agregar botón para crear un registro

Y ahora, rellenamos los slots.
Usamos componentes de Blade
menos en text-area porque no hay
componentes para ello.

```
<x-dialog-modal wire:model="open">  
  <x-slot name="title">  
    Crear nuevo post  
  </x-slot>  
  
  <x-slot name="content">  
    <div class="mb-4">  
      <x-label value="Titulo del post"/>  
      <x-input type="text" class="w-full"/>  
    </div>  
  
    <div class="mb-4">  
      <x-label value="Contenido del post"/>  
      <textarea class="w-full" rows="6"></textarea>  
    </div>  
  </x-slot>  
  
  <x-slot name="footer">  
    <x-secondary-button wire:click="$set('open', false)">  
      Cancelar  
    </x-secondary-button>  
  
    <x-danger-button>  
      Crear post  
    </x-danger-button>  
  </x-slot>
```

2. Tabla dinámica

Agregar botón para crear un registro

Si lo probamos:

Crear nuevo post

Título del post

Contenido del post

CANCELAR

CREAR POST

2. Tabla dinámica

Agregar botón para crear un registro

Vamos a hacer que cuando pulsemos el botón crear se nos cree un nuevo registro:

Nos vamos a CreatePost.php y creamos un método que se encargue de guardar lo que vaya a poner en el formulario.

```
4      use Livewire\Component;
5      use App\Models\Post;
6
7
8  class CreatePost extends Component
9  {
10
11     public $open = false;
12
13     public $title, $content;
14
15     public function save(){
16         Post::create([
17             'title' => $this->title,
18             'content' => $this->content,
19
20         ]);
21     }
22 }
```

**C

2. Tabla dinámica

Agregar botón para crear un registro

Y ahora en la vista:

Si lo probamos, para quitar el diálogo tenemos que **refrescar la página**.

Para que, una vez rellenado, al hacer click en crear post, se nos cierre el diálogo y no tengamos que refrescar la página. Vamos a establecer una conexión entre las dos vistas.

Vamos a hacer uso de **eventos y oyentes**.

```
<x-slot name="content">
  <div class="mb-4">
    <x-label value="Título del post"/>
    <x-input type="text" class="w-full" wire:model.defer="title"/>
  </div>

  <div class="mb-4">
    <x-label value="Contenido del post"/>
    <textarea wire:model.defer="content" class="w-full" rows="6"></textarea>
  </div>
</x-slot>

<x-slot name="footer">
  <x-secondary-button wire:click="$set('open', false)">
    Cancelar
  </x-secondary-button>

  <x-danger-button wire:click="save">
    Crear post
  </x-danger-button>
</x-slot>
```

2. Tabla dinámica

Agregar botón para crear un registro

Entonces, nos vamos a CreatePost y en nuestra función save:

Estamos haciendo que el método CreatePost sea el **emisor** y que emita la función render.



```
app > Http > Livewire > 🐾 CreatePost.php > 🏷 CreatePost > ⚙ save
```

```
4
5  use Livewire\Component;
6  use App\Models\Post;
7
8  class CreatePost extends Component
9  {
10
11     public $open = false;
12
13     public $title, $content;
14
15     public function save()
16     {
17         Post::create([
18             'title' => $this->title,
19             'content' => $this->content,
20         ]);
21
22         $this->emit('render');
23     }
24
25     public function render()
26     {
27         return view('livewire.create-post');
28     }
}
```

The code shows a Livewire component named CreatePost. It has properties \$open, \$title, and \$content. The save() method creates a new Post record with the title and content from the properties. After saving, it emits the 'render' event using \$this->emit('render'). A red arrow and a red circle highlight this line of code.

2. Tabla dinámica

Agregar botón para crear un registro

El oyente será, ShowPost: que lo que hace es que cuando escuche el método render que ha emitido el CreatePost, imprima su render.

```
app > Http > Livewire > 🐘 ShowPosts.php > 📁 ShowPosts
11
12     public $search;
13     public $sort = 'id';
14     public $direction = 'desc';
15
16     protected $listeners = ['render' => 'render'];
17
18     public function render() ↗
```

Si lo probamos ahora, ya se nos crea sin la necesidad de refrescar la página.

2. Tabla dinámica

Agregar botón para crear un registro

Ahora vamos a hacer que el diálogo se cierre cuando lo creemos. Para ello, nos vamos a CreatePost:

```
3           'content' => $this->content,  
4  
5       ]);  
6  
7       $this->reset('open','title','content');|  
8  
9       $this->emit('render');
```

2. Tabla dinámica

Creación de diálogos

Otro uso de los emisores y los oyentes es el uso de scripts. Entonces, por ejemplo, si queremos que no salga un diálogo cuando hayamos creado el post.

Vamos a hacer uso de un plugin SweetAlert2: <https://sweetalert2.github.io/#download>

Para usarlo, copiamos el código cdn que está en su página web: **(os lo doy)**

Y lo colocamos en nuestra plantilla principal, para poder usarlo más veces: Resource/view/layouts / app.blade

```
13
14      <!-- Scripts -->
15      @vite(['resources/css/app.css', 'resources/js/app.js'])
16       <script src="https://cdn.jsdelivr.net/npm/sweetalert2@11"></script> |
17
18
```

2. Tabla dinámica

Creación de diálogos

Ahora nos vamos a CreatePost y **emitimos** otro evento que se va a llamar alert

```
14  
15     public function save(){  
16         Post::create([  
17             'title' => $this->title,  
18             'content' => $this->content,  
19         ]);  
20  
21         $this->reset('open','title','content');  
22  
23         $this->emit('render');  
24         $this->emit('alert');  
25     }  
26  
27
```



2. Tabla dinámica

Creación de diálogos

Y este evento lo voy a escuchar desde un script. Entonces en nuestra plantilla principal (app.blade), casi al final colocamos:

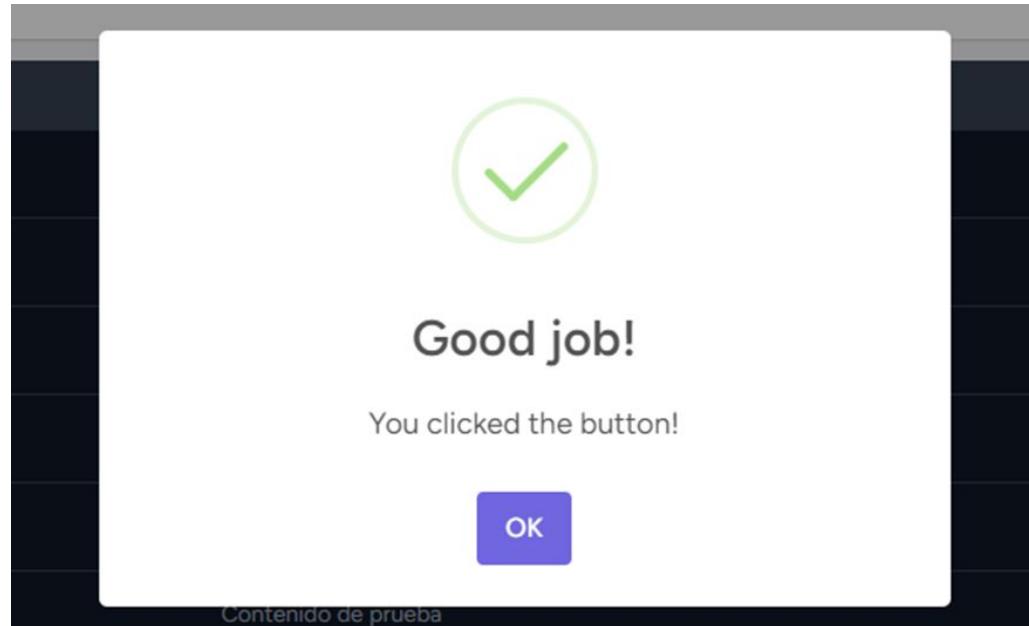
~~Swal.fire~~ está sacado de la página del plugin.

```
41
42     @livewireScripts
43
44     <script>
45         Livewire.on('alert', function () {
46             Swal.fire(
47                 'Good job!',
48                 'You clicked the button!',
49                 'success'
50             )
51         })
52     </script>
53
54
55     </body>
```

**C

2. Tabla dinámica

Creación de diálogos



2. Tabla dinámica

Creación de diálogos

Para personalizarlo: nos vamos a `create.post`

```
1
2
3
4
5
6
7
```

```
    ],
    $this->reset('open','title','content');

    $this->emit('render');
    $this->emit('alert','El post se creó satisfactoriamente');
}
```

Y en `post.blade.php`:

```
<script>
Livewire.on('alert', function (message) {
  Swal.fire(
    '¡Enhorabuena!', message, 'success'
  )
})</script>
```

2. Tabla dinámica

Creación de diálogos



¡Enhorabuena!

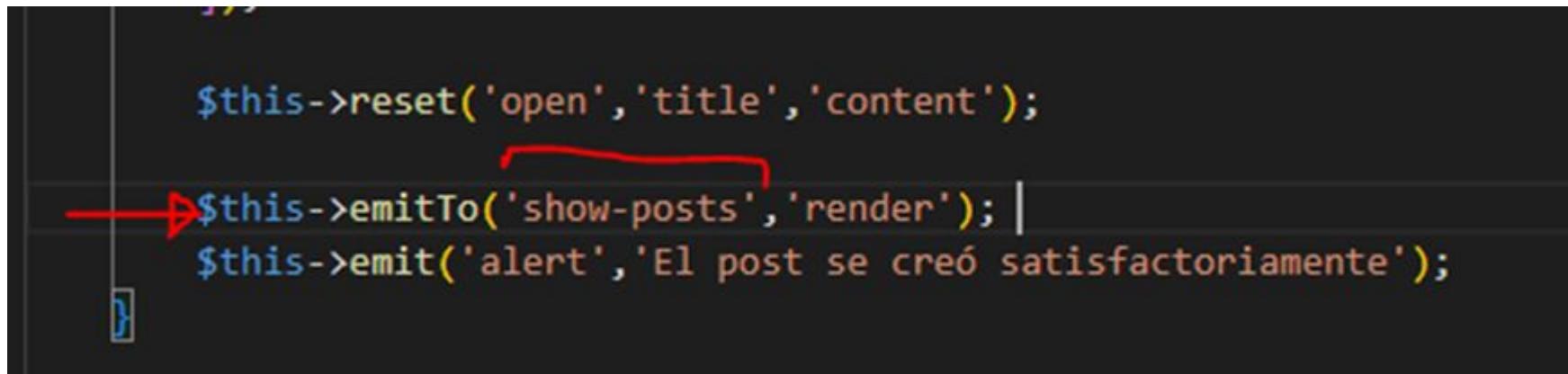
El post se creó satisfactoriamente

OK

2. Tabla dinámica

Creación de diálogos

Para personalizar quién escucha el mensaje, para evitar que todos los render sean emitidos, tenemos que volver a CreatePost:



```
$this->reset('open','title','content');
$this->emitTo('show-posts', 'render'); |
$this->emit('alert','El post se creó satisfactoriamente');
}
```

Así solo lo escucha el método show-posts

2. Tabla dinámica

Validaciones

Ahora mismo, si dejamos algo en blanco, al darle a crear nos da un error. Para eso, como ya vimos, hay que hacer uso de validaciones.

En CreatePost: escribimos las reglas de validación:

Y en la función save colocamos:

```
app > Http > Livewire > 🌸 CreatePost.php > 📁 CreatePost > ⚙️ save
  6  use App\Models\Post;
  7
  8  class CreatePost extends Component
  9  {
 10
 11      public $open = false;
 12
 13      public $title, $content;
 14
 15      protected $rules = [
 16          'title' => 'required|min=10',
 17          'content' => 'required|min=10',
 18      ];
 19
 20      public function save(){
 21
 22          $this->validate();
 23
 24          Post::create([
 25              'title' => $this->title,
 26              'content' => $this->content,
```

2. Tabla dinámica

Validaciones

Y ahora en la vista, nosotros lo hacíamos con `error` y `enderror` pero vamos a usar **componentes de Blade** que ya tienen sus estilos propios:

```
</x-slot>

<x-slot name="content">
  <div class="mb-4">
    <x-label value="Título del post"/>
    <x-input type="text" class="w-full" wire:model.defer="ti

    ↗ <x-input-error for='title' />

  </div>

  <div class="mb-4">
    <x-label value="Contenido del post"/>
    <textarea wire:model.defer="content" class="w-full" row

    ↗ <x-input-error for='content' />

  </div>

</x-slot>
```

2. Tabla dinámica

Validaciones

Crear nuevo post

Título del post

The title field is required.

Contenido del post

The content field is required.

2. Tabla dinámica

Paginar nuestra tabla ¡Sencillísimo!

Nos vamos a ShowPost:

```
17
18     protected $listeners = ['render'=> 'render'];
19
20
21     public function render()
22     {
23
24         $posts = Post::where('title', 'like', '%' . $this->search . '%')
25             ->orwhere('content', 'like', '%' . $this->search . '%')
26             ->orderBy($this->sort, $this->direction)
27             → ->paginate(10);
28
29         return view('livewire.show-posts', compact('posts'));
30     }
31
```

2. Tabla dinámica

Paginar nuestra tabla ¡Sencillísimo!

Nos vamos a nuestra vista: show.posts

```
111
112      @endif
113
114      [  <div class="px-6 py-3">
115          {$posts->links()}
116      </div>
117
118
119      </x-table>
120
```



2. Tabla dinámica

Paginar nuestra tabla ¡Sencillísimo!

Si lo comprobamos y damos click a cualquier número, vemos que la página se recarga cada vez que clicko.

Y esto no nos interesa porque quiero que sea **dinámica**.

Para ello, volvemos a ShowPosts:

Con esto tengo ya que es dinámico.

```
1 <?php
2
3 namespace App\Http\Livewire;
4
5 use Livewire\Component;
6 use App\Models\Post;
7 use Livewire\WithPagination;
8
9
10 class ShowPosts extends Component
11 {
12
13     use WithPagination;
14 }
```

PEEERO

2. Tabla dinámica

Paginar nuestra tabla ¡Sencillísimo!

Si buscamos algo que está en la página 1 y estamos en la página 10, nos dice que no existe.

Para arreglar esto:

En ShowPost, agregamos el siguiente método:

```
32 }  
33  
34 [ public function updatingSearch(){  
35     $this->resetPage();  
36 }  
37  
38 public function order($sort){
```

2. Tabla dinámica

Recuadro con el número de páginas que quiero mostrar

En show.post.blade.php

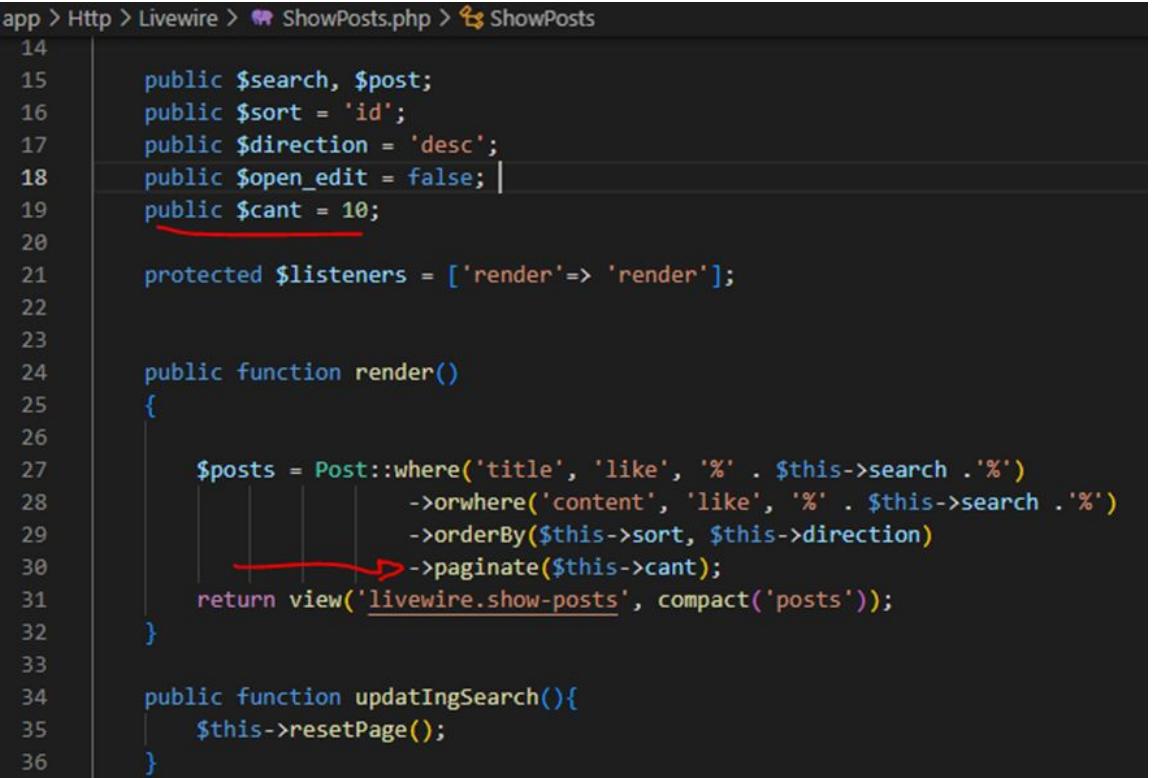
```
<x-table>
  <div class="px-6 py-3 flex items-center">
    <div class="flex items-center">
      <span> Mostrar </span>
      <select wire:model="cant" class="mx-2">
        <option value="10">10</option>
        <option value="25">25</option>
        <option value="50">50</option>
        <option value="100">100</option>
      </select>
      <span>entradas</span>
    </div>
  </div>
```

**C

2. Tabla dinámica

Recuadro con el número de páginas que quiero mostrar

Y como estamos haciendo uso de un wire:model **cant**, tenemos que **crearlo** en Showpost



```
app > Http > Livewire > ShowPosts.php > ShowPosts
14
15     public $search, $post;
16     public $sort = 'id';
17     public $direction = 'desc';
18     public $open_edit = false;
19     public $cant = 10;
20
21     protected $listeners = [ 'render'=> 'render'];
22
23
24     public function render()
25     {
26
27         $posts = Post::where('title', 'like', '%' . $this->search . '%')
28                 ->orwhere('content', 'like', '%' . $this->search . '%')
29                 ->orderBy($this->sort, $this->direction)
30                 ->paginate($this->cant);
31
32         return view('livewire.show-posts', compact('posts'));
33
34     public function updatingSearch(){
35         $this->resetPage();
36 }
```