



# Análisis y desarrollo de software.

**ID: 2556456**



[www.sena.edu.co](http://www.sena.edu.co)

@SENAComunica

# Instructor

---

Diego Fernando Calderón Silva  
Correo: dfcalderon@sena.edu.co

# Enlaces para crear Posts



Vamos a comenzar por crear los Posts, por lo tanto, pondremos un botón en el dashboard. Para esto nos dirigimos al app.Blade.php e iremos al bloque de @auth ya que el usuario debe estar autenticado para poder crear un post.

```
@auth()  
  <nav class="flex gap-2 items-center">  
    <a class="flex items-center gap-2 bg-white border p-2 text-gray-600 rounded text-sm uppercase font-bold cursor-pointer" href="">  
      Crear  
    </a>
```

Y tendremos un botón como este:

**CREAR** HOLA DIEGOCALDERON

# Enlaces para crear Posts



Junto a ese botón quiero poner un icono, por lo haremos una búsqueda en la web:

<https://heroicons.com>

Aquí encontraremos bastantes iconos de los mismos creadores de TailwindCSS los cuales son compatibles con react y vueJS y se pueden instalar vía npm o dando click sobre Copy svg.

A screenshot of a search results page from heroicons.com. A search bar at the top contains the text "photo". Below it, a card for the "photo" icon is shown. The card has a title "Outline 24x24, 1.5px stroke", a description "For primary navigation and marketing sections, with an outlined appearance.", and two buttons: "Copy SVG" with a copy icon and "Copy JSX" with a copy icon. To the right of this card is another card for the "camera" icon, which is represented by a camera lens icon.

# Enlaces para crear Posts



Una vez copiado, pegamos en el código del proyecto antes del texto del botón.

```
@auth()
<nav class="flex gap-2 items-center">
  <a class="flex items-center gap-2 bg-white border p-2 text-gray-600 rounded text-sm uppercase font-bold cursor-pointer" href="">
    <svg xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24" stroke-width="1.5" stroke="currentColor" class="w-6 h-6">
      <path stroke-linecap="round" stroke-linejoin="round" d="M6.827 6.175A2.31 2.31 0 015.186 7.23c-.38.054-.757.112-1.134.175C2.999 7.58 2.25 8.507 2.25 9.574V18a2 .25 2.25 0 002.25 2.25h15A2.25 2.25 0 0021.75 18V9.574c0-1.067-.75-1.994-1.802-2.169a47.865 47.865 0 00-1.134-.175 2.31 2.31 0 01-1.64-1.055l-.822-1.316a2.192 2.192 0 00-1.736-1.039 48.774 48.774 0 00-5.232 0 2.192 2.192 0 00-1.736 1.039l-.821 1.316z" />
      <path stroke-linecap="round" stroke-linejoin="round" d="M16.5 12.75a4.5 4.5 0 11-9 0 4.5 4.5 0 019 0zM18.75 10.5h.008v.008h-.008V10.5z" />
    </svg>
    Crear
  </a>
```

# Enlaces para crear Posts



Por ahora el enlace no hace nada por que no tiene href, por lo tanto, lo redireccionaremos hacia un controlador teniendo en cuenta que el Verb es GET, url es /photos, Action es index y RouteName es photos.index

# Actions Handled By Resource Controller			
Verb	URI	Action	Route Name
GET	/photos	index	photos.index
GET	/photos/create	create	photos.create
POST	/photos	store	photos.store
GET	/photos/{photo}	show	photos.show
GET	/photos/{photo}/edit	edit	photos.edit
PUT/PATCH	/photos/{photo}	update	photos.update
DELETE	/photos/{photo}	destroy	photos.destroy

# Enlaces para crear Posts



Pero para crear un nuevo post el url debe ser /photos/create.

## # Actions Handled By Resource Controller

Verb	URI	Action	Route Name
GET	/photos	index	photos.index
GET	/photos/create	create	photos.create
POST	/photos	store	photos.store
GET	/photos/{photo}	show	photos.show
GET	/photos/{photo}/edit	edit	photos.edit
PUT/PATCH	/photos/{photo}	update	photos.update
DELETE	/photos/{photo}	destroy	photos.destroy

# Enlaces para crear Posts



Por esta razón iremos a web.php a crear la nueva ruta que nos lleva al controlador PostController.

```
Route::get('uri: '/posts/create', [PostController::class, 'create'])->name('name: post.create');
```

Ahora en el controlador crearemos el método 'create' así:

```
public function create(){
    dd(...vars: 'Creando posts');
}
```

Y en el app.Blade.php podremos dar href al botón de crear así:

```
<a class="flex items-center gap-2 bg-white border p-2 text-gray-600 rounded text-sm uppercase font-bold cursor-pointer" href="{{ route('post.create') }}>
```

Recargando y dando click sobre el botón crear verificamos la utilidad del endpoint.

# Template de creación de publicaciones



Continuando con el proyecto, creamos un formulario para agregar las publicaciones. Para esto crearemos un directorio llamado `posts` y dentro una vista `create.blade.php`. Lo bueno de esto es que todo tiene el nombre de create, el controlador, el método, la vista, por lo tanto, será mas fácil identificar cada paso que queramos dar. Ahora, cambiaremos el dd por el retorno y nos dirigiremos hacia `posts.create`.

```
public function create(){
    return view( view: 'posts.create');
}
```

Seguidamente, iremos a heredar de `layouts.app` todo, y escribiremos en la sección de título, Crear una nueva publicación.

Recargamos la página, observamos y nos disponemos a crear el código que creara los posts.

# Template de creación de publicaciones



Creare un par de div y copiare el formulario de register.Blade.php para que nuestro diseño se vea igual y se tiene el siguiente resultado:

```
app.blade.php x web.php x PostController.php x create.blade.php x register.blade.php x

1 @extends('layouts.app')
2
3 @section('titulo')
4     Crea una nueva publicacion
5 @endsection
6
7 @section('contenido')
8     <div class="md:flex md:items-center" >
9         <div class="md:w-6/12 px-10 ">
10            Imagen Aqui
11        </div>
12
13         <div class="md:w-6/12 px-10 bg-white p-6 rounded-lg shadow-xl">
14             <form action="{{route('register')}}" method="POST" novalidate>
15                 @csrf
16                 <div class="mb-5">
17                     <label for="name" class="mb-2 block uppercase text-gray-500 font-bold" >Nombre</label>
18                     <input type="text" id="name" name="name" placeholder="Nombre" class="border p-3 w-full rounded-lg @error('name') border-red-500 @enderror" value="{{ old('name') }}>
19                     @error('name')
20                         <p class="bg-red-500 text-white my-2 rounded-lg text-sm p-2 text-center">{{ $message }}</p>
21                     @enderror
22
23                 </div>
24             </form>
25         </div>
26     </div>
27 @endsection
```

# Template de creación de publicaciones



Anteriormente, la imagen esta tal cual como se copió de register, ahora modificaremos el formulario para adecuarlo a nuestra necesidad.

Ahora crearemos un textarea debajo de este div teniendo en cuenta que acá no existe el value.

```
<div class="mb-5">
  <label for="titulo" class="mb-2 block uppercase text-gray-500 font-bold" >Titulo</label>
  <input type="text" id="titulo" name="titulo" placeholder="Titulo" class="border p-3 w-full rounded-lg @error('titulo') border-red-500 @enderror" value="{{ old('titulo') }}>
  @error('name')
    <p class="bg-red-500 text-white my-2 rounded-lg text-sm p-2 text-center">{{ $message }}</p>
  @enderror
</div>
```

```
<div class="mb-5">
  <label for="descripcion" class="mb-2 block uppercase text-gray-500 font-bold" >Descripcion de la publicacion</label>

  <textarea id="descripcion" name="descripcion" placeholder="Descripcion de la publicacion" class="border p-3 w-full rounded-lg @error('titulo') border-red-500 @enderror" > {{ old('titulo') }} </textarea>
  @error('name')
    <p class="bg-red-500 text-white my-2 rounded-lg text-sm p-2 text-center">{{ $message }}</p>
  @enderror
</div>

<input type="submit" value="Publicar" class="bg-sky-600 hover:bg-sky-700 transition-colors cursor-pointer uppercase font-bold w-full p-3 text-white rounded-lg">
```

# Template de creación de publicaciones



**Cierre sesión y compruebe el funcionamiento completo de su app.**

# Template de creación de publicaciones



The screenshot shows a Laravel application error page. The URL in the browser is `127.0.0.1:8000/login`. The error message is: `Missing required parameter for [Route: post.index] [URI: {user}] [Missing parameter: user].` The error is categorized under `Illuminate\Routing\Exceptions\UrlGenerationException`. The page includes navigation links for `STACK`, `CONTEXT`, `DEBUG`, and `FLARE`, along with sharing and documentation options. The stack trace on the right shows the following code from `app/Http/Controllers/LoginController.php:24`:

```
1 <?php
2
3 namespace App\Http\Controllers;
4
5 use Illuminate\Http\Request;
6 use function Laravel\Prompts\password;
7
8 class LoginController extends Controller
9 {
```

# Solucionando inicio de sesión.



Recordando un poco, en LoginController, tenemos que, si el usuario se autentica, lo enviamos a post.index pero si vemos web.php nos encontramos que index requiere el username y laravel no sabe en realidad cual es, por eso debemos pasarle esa información.

En

LoginController enviamos:

A screenshot of a code editor showing a portion of a Laravel application. The tabs at the top include 'app.blade.php', 'web.php', 'PostController.php', 'create.blade.php', and 'LoginController.php'. The 'LoginController.php' tab is active, highlighted with a yellow bar. The code itself is a PHP file with syntax highlighting. It defines a class 'LoginController' extending 'Controller'. The 'index()' method returns a view for 'auth.login'. The 'store(Request \$request)' method validates the 'email' and 'password' fields. If validation fails, it returns a back() response with an error message. If authentication succeeds, it attempts to remember the user and then redirects to 'post.index' with the authenticated user's 'username'.

# Solucionando inicio de sesión.



Antes de avanzar, vamos a app.Blade, en el span de hola donde imprimimos el nombre de usuario y vamos a crear la funcionalidad que lo lleve a su propio muro.

```
    Crear
  </a>
  <a class="font-bold uppercase text-gray-600" href="{{ route('post.index', auth()>user()>username) }}>
    Hola <span class="font-bold" > {{ auth()>user()>username }} </span>
  </a>

  <form action="{{ route('logout') }}" method="post">
    @csrf
    <button type="submit" class="font-bold uppercase text-gray-600" >
      Cerrar sesión
    </button>
  </form>
```

# Solucionando inicio de sesión.



Antes de avanzar, vamos a app.Blade, en el span de hola donde imprimimos el nombre de usuario y vamos a crear la funcionalidad que lo lleve a su propio muro.

Verifiquemos que todo esté funcionando correctamente y verifiquemos que pasa si no ponemos auth()->user()->username

```
    Crear
  </a>
<a class="font-bold uppercase text-gray-600" href="{{ route('post.index', auth()>user()>username) }}>
  Hola <span class="font-bold" > {{ auth()>user()>username }} </span>
</a>

<form action="{{ route('logout') }}" method="post">
  @csrf
  <button type="submit" class="font-bold uppercase text-gray-600" >
    Cerrar sesión
  </button>
</form>
```

# Instalemos Dropzone

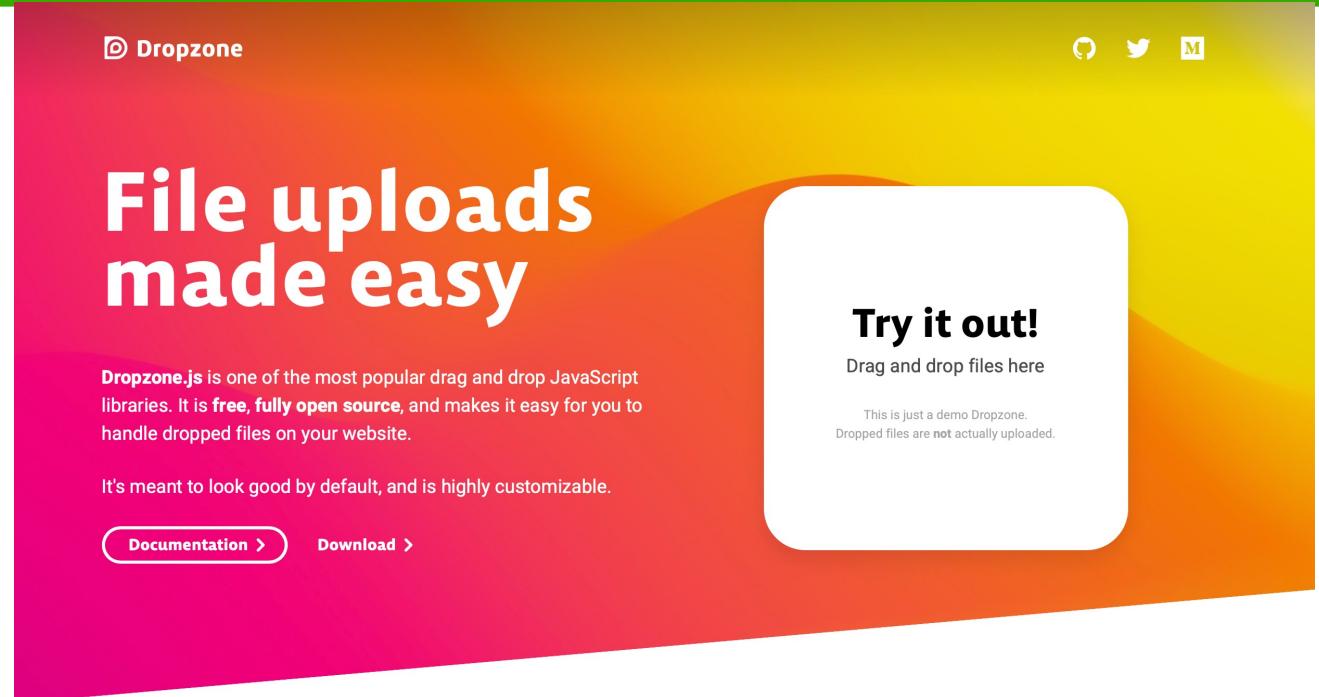


## ¿Qué es dropzone?

Es una biblioteca o una característica de interfaz de usuario que permite al cliente arrastrar y soltar archivos desde su computadora a una zona específica de una página web.

La documentación la encontraremos en:

<https://www.dropzone.dev>



[Source code on GitHub](#)

You can get all the source code on GitHub, as well as installation instructions. If you encounter an issue with this library, this is the place to create an issue.

[GitHub >](#)



[Documentation](#)

All the documentation about Dropzone, and the multiple ways to configure and customise it, can be found on GitBook.

[Docs >](#)



[Questions and Support](#)

If you need help, there are GitHub Discussions and Stackoverflow. Use the tag dropzonejs and there'll be plenty of people helping you out.

[Stack Overflow >](#)

# Instalemos Dropzone



Para instalar vamos a la documentación y en el apartado de instalación nos dice que con npm podemos instalar:

```
npm yarn  
$ npm install --save dropzone
```

Bajando un poco mas vemos que viene la opción de css **Stand-alone file**. y copiaremos el primer link que encontramos al inicio para copiarlo. Primero eliminaremos /resources/js/Bootstrap.js y en /resources/js/app.js, le daremos forma al dropzone seteando los valores que queremos por defecto. Recuerda agregar en app.Blade.php **@vite('resources/js/app.js')**

```
app.blade.php x app.js x create.blade.php x web.php x PostController.php  
import Dropzone from "dropzone";  
  
Dropzone.autoDiscover = false;  
  
const dropzone : $3ed269f2f0fb224b$export$2e2bc... = new Dropzone( el: '#dropzone', options: {  
    dictDefaultMessage: 'Sube aquí tu imagen',  
    acceptedFiles: '.png, .jpg, .jpeg, .gif',  
    addRemoveLinks: true,  
    dictRemoveFile: 'Borrar archivo',  
    maxFiles: 1,  
    uploadMultiple: false,  
})
```

# Instalemos Dropzone



Ahora para visualizar el dropzone, debemos dirigirnos a `create.blade.php` y agregar:

```
@section('contenido')
    <div class="md:flex md:items-center" >
        <div class="md:w-6/12 px-10 ">
            <form action="" id="dropzone" class="dropzone border-dashed border-2 w-full h-96 rounded flex-col justify-center items-center">
                </form>
            </div>
```

Si recargamos la página no observaremos los valores por defecto y esto se debe a que no hay una ruta en `action`, la pondremos con fines demostrativos (`/img`) y tendremos:

The screenshot shows a user interface for creating a new post on a platform called Devstagram. At the top right, there are buttons for 'CREAR' (Create) and 'CERRAR SESIÓN' (Close Session). The main area has a title 'Crea una nueva publicación'. On the left, there is a dashed box labeled 'Sube aquí tu imagen' (Upload your image here). On the right, there are input fields for 'TITULO' (Title) containing 'Título' and 'DESCRIPCION DE LA PUBLICACION' (Post Description). A large blue button at the bottom right is labeled 'PUBLICAR' (Publish).

# Instalemos Dropzone



Ahora para visualizar el dropzone, debemos dirigirnos a `create.blade.php` y agregar:

```
@section('contenido')
    <div class="md:flex md:items-center" >
        <div class="md:w-6/12 px-10 ">
            <form action="" id="dropzone" class="dropzone border-dashed border-2 w-full h-96 rounded flex-col justify-center items-center">
                </form>
            </div>
```

Si recargamos la página no observaremos los valores por defecto y esto se debe a que no hay una ruta en `action`, la pondremos con fines demostrativos (`/img`) y tendremos:

The screenshot shows a user interface for creating a new post. At the top, there's a header with the text "Devstagram" and a user profile for "HOLA DIEGOCALDERON". On the right, there are buttons for "CREAR" and "CERRAR SESIÓN". Below the header, the main area has a title "Crea una nueva publicación". In the center, there's a dashed rectangular area with the placeholder text "Sube aquí tu imagen". To the right of this area, there are input fields for "TITULO" (with "Título" typed in) and "DESCRIPCION DE LA PUBLICACION". At the bottom right, there's a large blue button labeled "PUBLICAR". At the very bottom of the page, a footer bar displays the text "DEVSTAGRAM - TODOS LOS DERECHOS RESERVADOS 2023-10-27 03:25:17".

# Practiquemos



Antes de continuar, carguen su proyecto a GitHub y acceda con un token

# Practiquemos



Ahora, clone este proyecto en su disco:

<https://github.com/diegokld30/aQueNoLoLogran.git>

# Practiquemos



Haga que funcione

# Creando routing para imágenes



Seguimos trabajando en la creación del contenido, es por esto que vamos a crear un controlador llamado:  
ImagenController

Dentro de este controlador creamos un método store que retorne un dd("Hola desde imágenes controlador"); y posteriormente crearemos la ruta en web así:

```
Route::post('/imagenes', [ImagenController::class, 'store'])->name('imagenes.store');
```

Ahora nos dirigimos a la vista de create.blade.php y en el action del formulario le daremos el name de la ruta que acabamos de crear:

```
action="{{ route('imagenes.store') }}
```

Y finalmente debemos saber que para subir imágenes existe algo que se llama enctype y existen diferentes tipos, pero en este caso usamos.

```
enctype="multipart/form-data"
```

# Creando routing para imágenes



Por lo tanto, en este momento nuestro form se debe ver así:

```
<form action="{{ route('imagenes.store') }}" enctype="multipart/form-data" method="post" id="dropzone" class="dropzone border-dashed border-2 w-full h-96 rounded flex flex-col justify-center items-center">
</form>
```

Si recargamos y arrastramos una imagen al dropzone veremos lo siguiente:

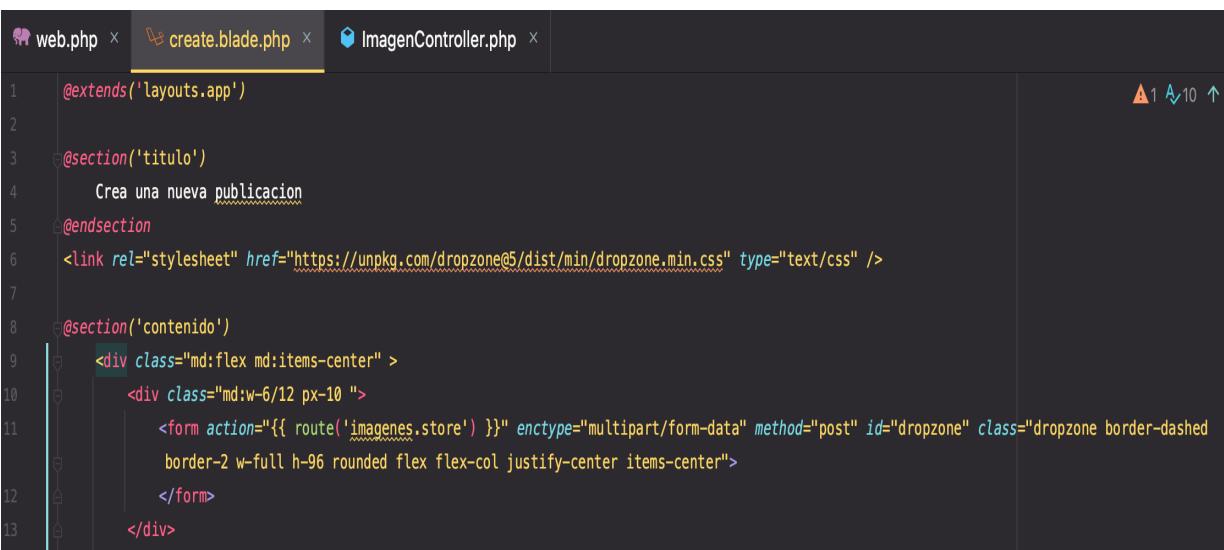
The screenshot shows a user interface for creating a new post. At the top, there's a navigation bar with the Devstagram logo, a camera icon labeled 'CREAR', the user's name 'HOLA DIEGOCALDERON', and a 'CERRAR SESIÓN' button. Below the navigation, the main title is 'Crea una nueva publicacion'. On the left, there's a 'Dropzone' area with the placeholder 'Sube aquí tu imagen'. A file named '8.jpg' (5.8 MB) is shown uploaded. There are also '+' and '-' buttons for managing files. On the right, there's a form with fields for 'TITULO' (Title) and 'DESCRIPCION DE LA PUBLICACION' (Post Description), both currently empty. A large blue 'PUBLICAR' (Publish) button is at the bottom of the form.

# Creando routing para imágenes

Hasta el momento sabemos que se logra arrastrar un archivo de imagen, pero no se ve bien, hace falta darle estilo. Para esto vamos a ir a la web de dropzone, en el apartado documentación en la opción instalación y seguidamente en Stand-alone file encontraremos la información que necesitamos.

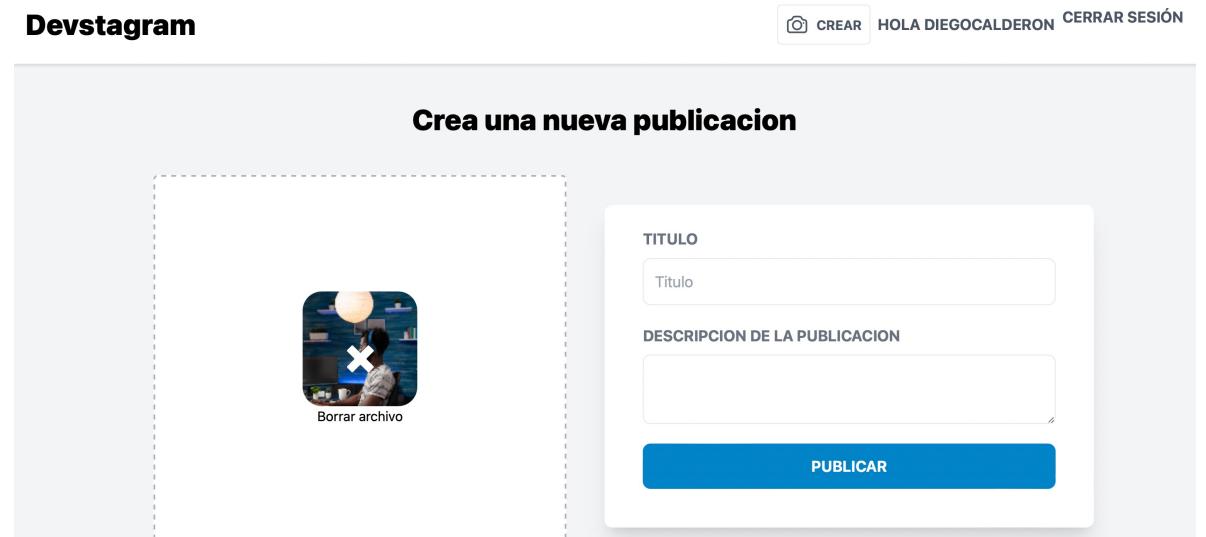
Vamos a hacer una prueba copiando el link dentro de créate.

```
<link rel="stylesheet" href="https://unpkg.com/dropzone@5/dist/min/dropzone.min.css" type="text/css" />
```



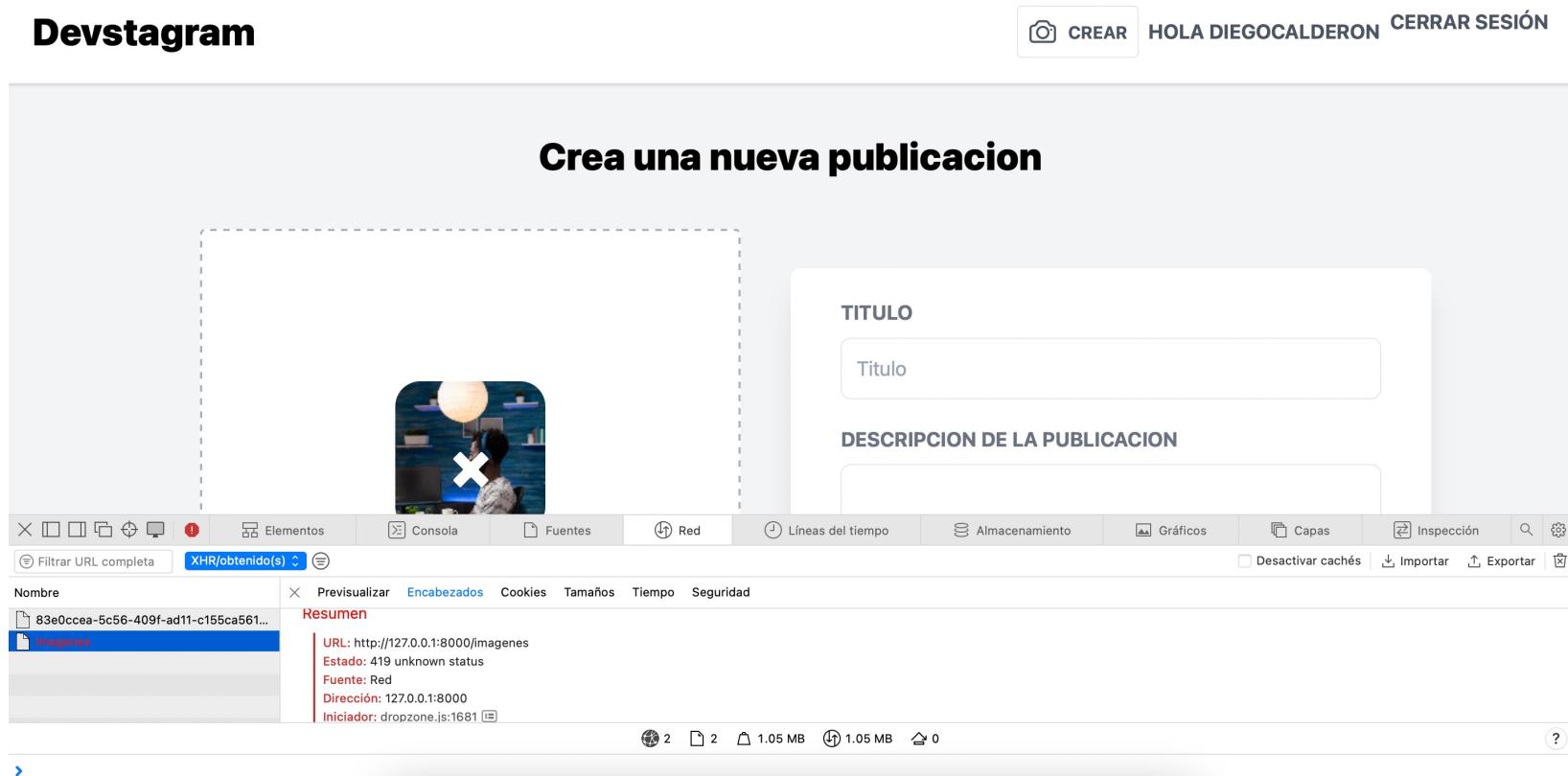
```
1 @extends('layouts.app')
2
3 @section('titulo')
4     Crea una nueva publicacion
5 @endsection
6 <link rel="stylesheet" href="https://unpkg.com/dropzone@5/dist/min/dropzone.min.css" type="text/css" />
7
8 @section('contenido')
9     <div class="md:flex md:items-center" >
10         <div class="md:w-6/12 px-10 " >
11             <form action="{{ route('imagenes.store') }}" enctype="multipart/form-data" method="post" id="dropzone" class="dropzone border-dashed border-2 w-full h-96 rounded flex-col justify-center items-center">
12                 </form>
13         </div>

```



# Creando routing para imágenes

Como ya sabemos, es buena práctica revisar el inspector cada que agreguemos componentes js a nuestro proyecto. Y en este caso el componente nos trae un error 419



# Creando routing para imágenes

Y analizando la respuesta o causa, se evidencia que falta un token del tipo csrf



The screenshot shows a browser developer tools interface with the Network tab selected. A request labeled "XHR/obtenido(s)" is highlighted, showing a failed response. The response body contains the following JSON error message:

```
1: {
  2:   "message": "CSRF token mismatch.",
  3:   "exception": "Symfony\\Component\\HttpKernel\\Exception\\HttpException",
  4:   "file": "/Applications/XAMPP/xamppfiles/htdocs/devstagram/vendor/laravel/framework/src/Illuminate/Foundation/Exceptions/Handler.php",
  5:   "line": 451,
  6:   "trace": [
  7:     {
  8:       "file": "/Applications/XAMPP/xamppfiles/htdocs/devstagram/vendor/laravel/framework/src/Illuminate/Foundation/Exceptions/Handler.php",
  9:       "line": 422,
 10:       "function": "prepareException",
 11:       "class": "Illuminate\\Foundation\\Exceptions\\Handler",
 12:       "type": "->"
 13:     },
 14:     {
 15:       "file": "/Applications/XAMPP/xamppfiles/htdocs/devstagram/vendor/laravel/framework/src/Illuminate/Routing/Pipeline.php",
 16:       "line": 51,
 17:       "function": "render",
 18:       "class": "Illuminate\\Foundation\\Exceptions\\Handler",
 19:       "type": "->"
 20:     },
 21:   ],
 22: }
```

The status bar at the bottom indicates 2 requests made, totaling 1.05 MB.

# Creando routing para imágenes



Para solucionar el problema del token vamos a ir a `create.blade.php` y dentro del formulario que está dentro div que genera el dropzone agregamos la directiva del csrf.

```
<div class="md:w-6/12 px-10 ">
    <form action="{{ route('imagenes.store') }}" enctype="multipart/form-data" method="post" id="dropzone" class="dropzone border-dashed border-2 w-full h-96 rounded flex flex-col justify-center items-center">
        @csrf
    </form>
</div>
```

Ahora en nuestro controlador le daremos funcionalidad al request así:

```
class ImagenController extends Controller
{
    public function store(Request $request){
        $input = $request->all();

        return response()->json($input);
    }
}
```

# Creando routing para imágenes

Y si vamos al inspector nos encontraremos con la información que tiene request.

Filtrar URL completa	XHR/obtenido(s)	Previsualizar	Encabezados	Cookies	Tamaños	Tiempo	Seguridad
 94b56414-afe2-4fff-bb91-a0314ee87d...		1 { 2    "_token": "L2Mn0DLr1VbEQxzgYtVgXvBtFs9QxeDTxAr1KhCt", 3    "file": {} 4 }					
 imágenes							

Si nos fijamos nos muestra el token que generamos y también un "file". Este file esta vacio pero ahí es donde debería estar la información del archivo que estamos intentando subir, entonces los que vamos a hacer es modificar el controlador nuevamente de la siguiente forma:

```
class ImagenController extends Controller
{
    public function store(Request $request){
        // $input = $request->all();
        $imagen = $request->file('file');

        return response()->json(['imagen' => $imagen->extension()]);
    }
}
```

# Creando routing para imágenes

Recargando la página y cargando nuevamente una imagen observamos:

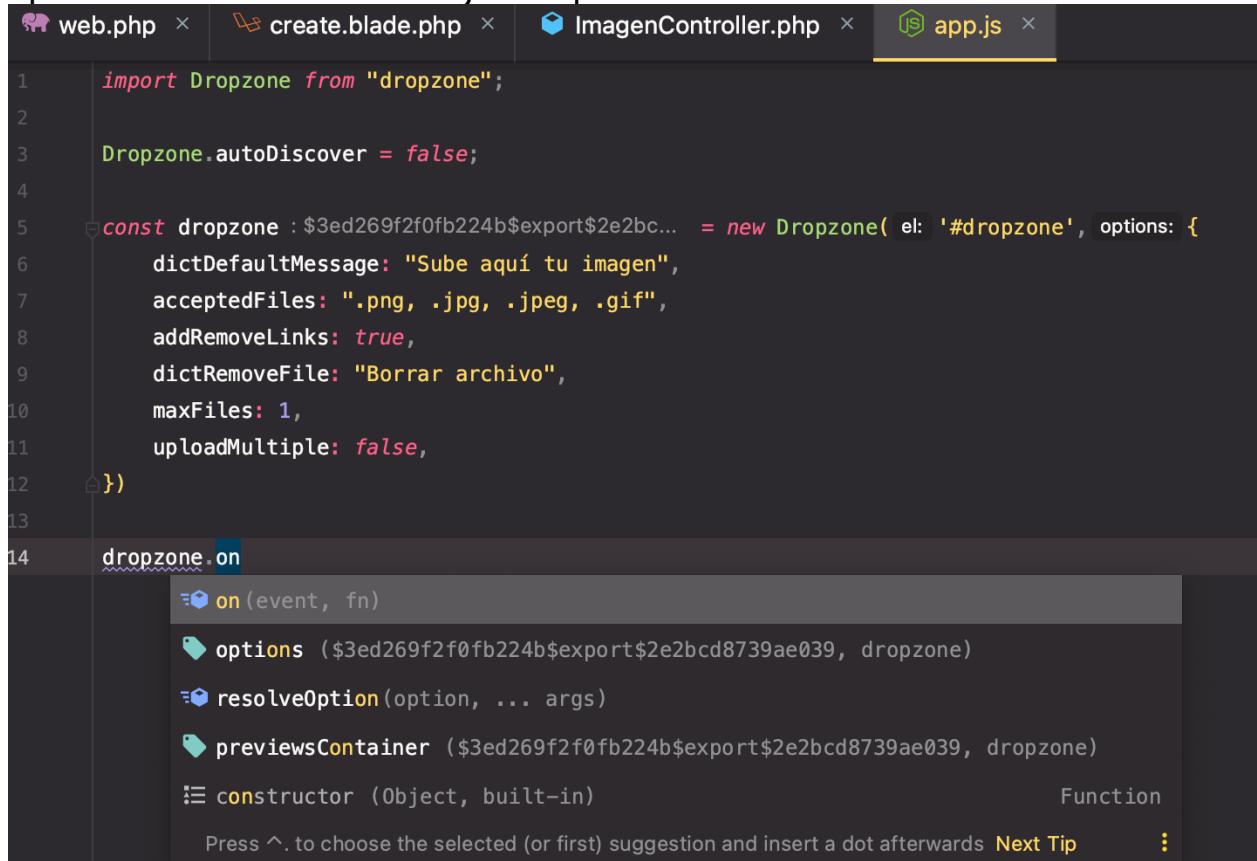
Nombre	X	Previsualizar	Encabezados	Cookies	Tamaños	Tiempo	Seguridad
 ed59eca2-559e-4582-84ec-8ea3350c...	1	{					
 imágenes	2	"imagen": "jpg"					
	3	}					

Como podemos observar nos esta retornando la extensión tal cual como lo pedimos, pero y... ¿Qué pasa con el nombre del archivo?

Resulta que en PHP, se debe trabajar la super variable global FILE, la cual nos ayuda a guardar temporalmente en memoria el nombre y la información completa de nuestro archivo. Para lograr trabajar debemos instalar una dependencia que nos permita modificar las imágenes y que en el diseño tengamos una cuadricula perfecta así como también lograr subir las imágenes al servidos.

# Eventos de dropzone

Dropzone tiene una serie de eventos que permite visualizar el estado de los archivos. Vamos a volver al archivo app.js y vamos a añadir algo de código así: Si observamos al momento de usar la constante dropzone y añadir .on, el editor nos pide que le pasemos un evento y después un callback:



The screenshot shows a code editor with several tabs at the top: web.php, create.blade.php, ImagenController.php, and app.js. The app.js tab is active. The code is as follows:

```
1 import Dropzone from "dropzone";
2
3 Dropzone.autoDiscover = false;
4
5 const dropzone = new Dropzone( el: '#dropzone', options: {
6   dictDefaultMessage: "Sube aquí tu imagen",
7   acceptedFiles: ".png, .jpg, .jpeg, .gif",
8   addRemoveLinks: true,
9   dictRemoveFile: "Borrar archivo",
10  maxFiles: 1,
11  uploadMultiple: false,
12})
13
14 dropzone.on
```

A tooltip is displayed over the 'on' keyword in line 14, listing the available event handlers:

- on (event, fn)
- options (\$3ed269f2f0fb224b\$export\$2e2bcd8739ae039, dropzone)
- resolveOption(option, ... args)
- previewsContainer (\$3ed269f2f0fb224b\$export\$2e2bcd8739ae039, dropzone)
- constructor (Object, built-in) Function

At the bottom of the tooltip, it says: "Press ^ to choose the selected (or first) suggestion and insert a dot afterwards".

# Eventos de dropzone

Voy a usar como evento sending, ya que es lo que estoy buscando hacer y cómo callback lo siguiente:

```
dropzone.on( event: 'sending', fn: function (file, xhr, formData) : void {
    console.log(file);
})
```

Recargo mi página, subo una imagen y reviso la consola de mi inspector y me encuentro con los siguientes datos:

# Eventos de dropzone

Como vimos anteriormente tenemos mucha información tanto para file, como para formData y xhr la cual podemos usar de necesitarla. También hay más eventos como podemos observar en la siguiente imagen:

```
dropzone.on( event: 'sending', fn: function (file, xhr, formData) : void {
    console.log(file);
})
dropzone.on( event: 'success', fn: function (file, response) : void {
    console.log(response);
})
dropzone.on( event: 'error', fn: function (file, message) : void {
    console.log(message);
})
dropzone.on( event: 'removedfile', fn: function () : void {
    console.log('Archivo eliminado');
})
```

Cargue una imagen, elimínela y observe que pasa en el inspector.

# Eventos de dropzone

Miremos ahora que si eliminamos o comentamos el contenido del evento eliminar y volvemos a cargar una imagen, nos muestra que en la línea 18 hay un evento. Y si revisamos que evento es, encontramos success.

```
client.ts:150
[vite] connected.

app.js:15
File {upload: Object, status: "uploading", previewElement: <div class="dz-preview dz-file-preview dz-processing">, previewTemplate: <div class="dz-preview dz-file-preview dz-processing">, _removeLink: <a class="dz-remove">, ...}
  {imagen: "jpg"} app.js:18

17   '',
18   dropzone.on('success', function (file, response){
19     console.log(response);
})
```

Hay que tener en cuenta, que la respuesta la podemos modificar en el controlador, en este momento esta para que retorne la extensión del archivo cargado, pero nosotros podemos modificarla a nuestro antojo así:

```
class ImagenController extends Controller
{
    public function store(Request $request){
        // $input = $request->all();
        $imagen = $request->file('file');

        return response()->json(['imagen' => "Esta es la respuesta de imagen"]);
    }
}
```

{imagen: "Esta es la respuesta de imagen"}

# Intervention



Veamos el material de apoyo llamado publicaciones, y evidenciamos como hay muchas imágenes con diferentes tamaños tal como en Instagram. Pero si analizamos también Instagram, notaremos que todas las imágenes son cuadradas. PHP provee demasiadas formas de manipular imágenes, en esta ocasión trabajaremos con intervention (<https://image.intervention.io/v2>).

Instalamos con:

```
composer require intervention/image
```

Lo hacemos así para que laravel descubra los paquetes que necesita y los instale. Revisar composer.js

Esa es la primer parte, pero si vemos la documentación nos dice que debemos integrarla con laravel. Para esto debemos abrir el archivo config/app.php y buscamos los providers y al final de todo el provider copiamos:

```
Intervention\Image\ImageServiceProvider::class
```

# Intervention



```
'providers' => ServiceProvider::defaultProviders()->merge([
    /*
     * Package Service Providers...
     */

    /*
     * Application Service Providers...
     */

    App\Providers\AppServiceProvider::class,
    App\Providers\AuthServiceProvider::class,
    // App\Providers\BroadcastServiceProvider::class,
    App\Providers\EventServiceProvider::class,
    App\Providers\RouteServiceProvider::class,

    Intervention\Image\ImageServiceProvider::class
])->toArray(),
```

# Intervention

Lo anterior automáticamente irá a vendor, lo agregará y hará parte del proyecto.

Ahora sigamos con el paso a paso donde debemos buscar un alias o '\$aliases' y pegar:

'Image' => Intervention\Image\Facades\Image::class

```
'aliases' => Facade::defaultAliases()->merge([
    // 'Example' => App\Facades\Example::class,
    'Image' => Intervention\Image\Facades\Image::class
])->toArray(),
```

# Almacenando imágenes en el servidor

Veamos ahora como dar uso a la herramienta que acabamos de instalar. Veamos como en el controlador ImagenController estamos obteniendo el archivo, pero antes de obtenerlo usemos intervention image así:

```
$nombreImagen = Str::uuid() . ".$imagen->extension();
```



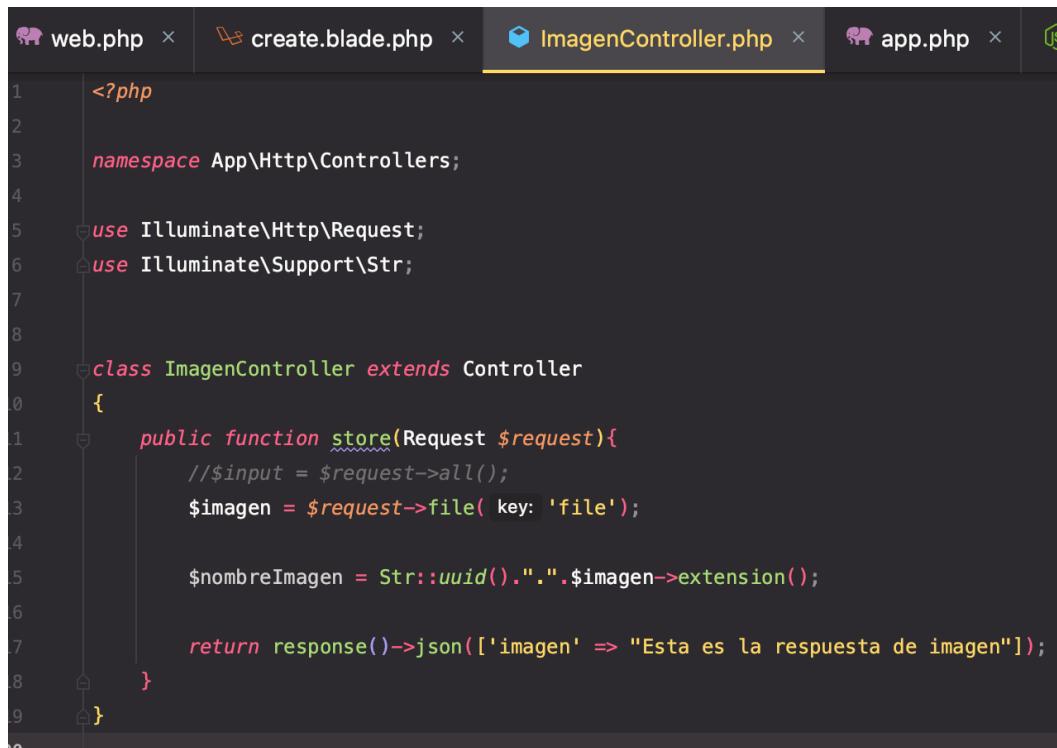
The screenshot shows a code editor with multiple tabs. The active tab is 'ImagenController.php'. The code in this tab is as follows:

```
<?php  
namespace App\Http\Controllers;  
  
use Illuminate\Http\Request;  
use Psy\Util\Str;  
  
class ImagenController extends Controller  
{  
    public function store(Request $request){  
        // $input = $request->all();  
        $imagen = $request->file('file');  
  
        $nombreImagen = Str::uuid() . ".$imagen->extension();  
  
        return response()->json(['imagen' => "Esta es la respuesta de imagen"]);  
    }  
}
```

# Almacenando imágenes en el servidor

En este momento nuestro código a través de ::uuid() está generando un id único para cada una de las imágenes por que en el directorio de nuestro servidor no podemos tener dos archivos que se llamen igual. Ahora importemos la clase de Str en la cabecera del archivo:

```
use Illuminate\Support\Str;
```



The screenshot shows a code editor with several tabs at the top: 'web.php', 'create.blade.php', 'ImagenController.php' (which is the active tab), and 'app.php'. The 'ImagenController.php' tab has a yellow underline. The code in the editor is as follows:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Str;

class ImagenController extends Controller
{
    public function store(Request $request){
        // $input = $request->all();
        $imagen = $request->file('key: 'file');

        $nombreImagen = Str::uuid().".". $imagen->extension();

        return response()->json(['imagen' => "Esta es la respuesta de imagen"]);
    }
}
```

# Almacenando imágenes en el servidor



Y retornemos el nombre de la imagen para validar que todo este bien. Para esto modificamos el return y escribimos:

```
return response()->json(['imagen' => $nombreImagen]);
```

Cargamos nuevamente una imagen, y al revisar el inspector del navegador notamos que imagen tiene un nombre diferente al que veníamos viendo.

A screenshot of a browser's developer tools console. The top bar shows 'Consola abierta a la(s) 19:57:35'. On the left, there are several log entries from 'vite': '[vite] connecting...', '[vite] connected.', and 'Elemento seleccionado'. In the center, there is a detailed log entry for a file upload: 'File {upload: Object, status: "uploading", previewElement: <div class="dz-preview dz-file-preview dz-processing">, previewTemplate: <div class="dz-preview dz-file-preview dz-processing">, \_removeLink: app.js:15 <a class="dz-remove">, ...}'. On the right, there is a 'Código de módulo' section with 'client.ts:18' and 'client.ts:150'. At the bottom, there is another log entry: '{imagen: "73b122cb-2d2a-49cf-948a-f435ca847247.jpg"} app.js:18'.

```
[vite] connecting...
[vite] connected.
> Elemento seleccionado
< ► <body class="bg-gray-100">...</body> = $1
El > File {upload: Object, status: "uploading", previewElement: <div class="dz-preview dz-file-preview dz-processing">, previewTemplate: <div class="dz-preview dz-file-preview dz-processing">, _removeLink: app.js:15 <a class="dz-remove">, ...}
El {imagen: "73b122cb-2d2a-49cf-948a-f435ca847247.jpg"} app.js:18
Código de módulo — client.ts:18
client.ts:150
app.js:15
app.js:18
```

# Almacenando imágenes en el servidor



Ya que tenemos el nombre de la imagen vamos a generar la imagen que se guardara en el servidor asi:

```
$imagenServidor = Image::make($imagen);
```

Y agregamos también la clase Image en la cabecera:

```
use Intervention\Image\Facades\Image;
```

En este momento, `imagenServidor` es una instancia de `image`, por lo que podemos usarla y agregarle elementos así:

```
$imagenServidor->fit(1000,1000);
```

Donde le indico al servidor que tendremos una imagen con esas medidas. `Fit()` es un api de `intervention` y si revisamos la documentación encontraremos todas las que están disponibles junto con su forma de uso.

# Almacenando imágenes en el servidor



Luego de eso, tendremos que ubicar la imagen a algún lugar del servidor ya que la imagen temporal se elimina después de un tiempo determinado, por esto en un directorio upload que vamos a crear, guardaremos las imágenes, y para guardarlas usaremos:

```
class ImagenController extends Controller
{
    public function store(Request $request){
        // $input = $request->all();
        $imagen = $request->file('key: file');

        $nombreImagen = Str::uuid() . ".$imagen->extension();

        $imagenServidor = Image::make($imagen);
        $imagenServidor->fit(width: 1000, height: 1000);

        $imagenPath = public_path(path: 'uploads') . '/' . $nombreImagen;
        $imagenServidor->save($imagenPath);

        return response()->json(['imagen' => $nombreImagen]);
    }
}
```

# Almacenando imágenes en el servidor



Intente cargar una imagen...

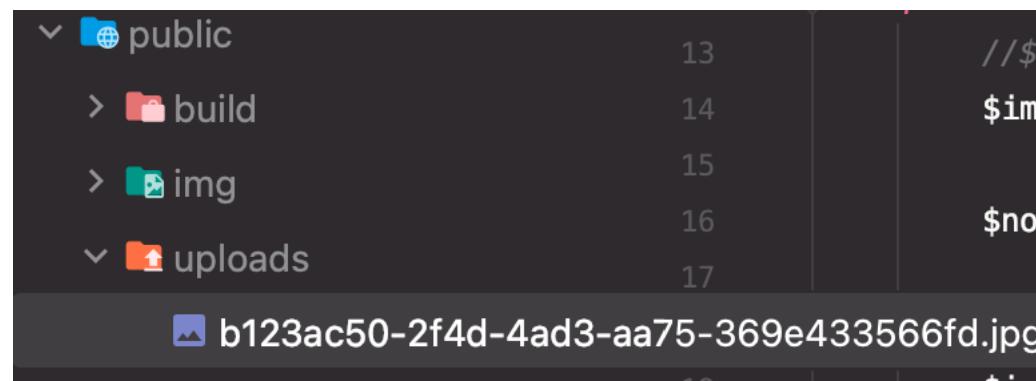
Si le funciona, perfecto, continue, pero a mi no me funciona.

```
Consola borrada a la(s) 20-13-32
```

```
[vite] connecting...
[vite] connected.
File {upload: Object, status: "uploading", previewElement: <div class="dz-preview dz-file-preview dz-processing">, previewTemplate: <div class="dz-preview dz-file-preview dz-processing">, _removeLink: app.js:15
<a class="dz-remove">...</a>}
Failed to load resource: the server responded with a status of 500 (Internal Server Error)
http://127.0.0.1:8000/imagenes
{message: "Can't write image data to path (/Applications/XAMP/uploads/b315262d-e720-43e6-a6eb-a66d3a9cf6d2.jpg)", exception: "Intervention\Image\Exception\NotWritableException", file: "/Applications/XAMPP/xamppfiles/htdocs/devstagram/v Intervention\Image\Image.php", line: 150, trace: Array}
S Código de módulo — client.ts:18
client.ts:150
app.js:15
app.js:21
```

Me dice que no se puede escribir nada en esa ruta, por lo que debo ir a public y crear manualmente el directorio upload en public.

Notaremos que el error desaparece y si revisamos el directorio que acabamos de crear, encontraremos al fin la imagen cargada.



# Modelos y migraciones para los post



Vamos a ver un poco las publicaciones y notaremos que cada una tiene un nombre, una descripción y el usuario que la ha creado. Vamos a crear un nuevo modelo y con esto una nueva migración, y si vamos a los controladores, notaremos que ya existe el PostController pero no un modelo.

Vamos a crear un modelo llamado Post:

```
php artisan make:model Post
```

Y vamos a crear su respectiva migración:

```
php artisan make:migration create_posts_table
```

Vimos como crear ambas cosas por separado, pero ya que van de la mano veamos cómo crearlas a la vez:  
Eliminemos los script anteriormente creados y demos el comando:

```
Php artisan make:model --migration --controller --factory Post
```

Como ya tengo el controlador, lo elimino del comando y envio:

```
php artisan make:model --migration --factory Post
```

# Modelos y migraciones para los post



Como ya tengo el controlador, lo elimino del comando y envío:

```
php artisan make:model --migration --factory Post
```

```
ingdiegoc@MacBook-Pro-de-Diego devstagram % php artisan make:model --migration --factory Post

[INFO] Model [app/Models/Post.php] created successfully.

[INFO] Factory [database/factories/PostFactory.php] created successfully.

[INFO] Migration [database/migrations/2023_11_03_013648_create_posts_table.php] created successfully.
```

Nota: factory es una forma de hacer un testing a la base de datos (lo veremos mas adelante).

# Añadiendo propiedades a la migración



Veamos un poco la migración de user y notemos que la escritura de código no es igual a sql:

```
return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('users', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('email')->unique();
            $table->timestamp('email_verified_at')->nullable();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();
        });
    }
}
```

# Añadiendo propiedades a la migración



Ahora vamos a ver un poco la documentación de las migraciones (<https://laravel.com/docs/10.x/migrations>) podemos ver que en columns hay varios tipos de columnas disponibles.

Observando el id nos encontramos que es un alias de bigIncrements el cual crea un autoincrement y es de tipo unsigned

Analicemos:

- String
- Varchar
- Text
- rememberToken
- foreingId
- Boolean
- dateTime
- Date

## # Available Column Types

The schema builder blueprint offers a variety of methods that correspond to the different types of columns you can add to your database tables. Each of the available methods are listed in the table below:

<a href="#">bigIncrements</a>	<a href="#">jsonb</a>	<a href="#">string</a>
<a href="#">bigInteger</a>	<a href="#">lineString</a>	<a href="#">text</a>
<a href="#">binary</a>	<a href="#">longText</a>	<a href="#">timeTz</a>
<a href="#">boolean</a>	<a href="#">macAddress</a>	<a href="#">time</a>
<a href="#">char</a>	<a href="#">mediumIncrements</a>	<a href="#">timestampTz</a>
<a href="#">dateTimeTz</a>	<a href="#">mediumInteger</a>	<a href="#">timestamp</a>
<a href="#">dateTime</a>	<a href="#">mediumText</a>	<a href="#">timestampsTz</a>
<a href="#">date</a>	<a href="#">morphs</a>	<a href="#">timestamps</a>
<a href="#">decimal</a>	<a href="#">multiLineString</a>	<a href="#">tinyIncrements</a>
<a href="#">double</a>	<a href="#">multiPoint</a>	<a href="#">tinyInteger</a>
<a href="#">enum</a>	<a href="#">multiPolygon</a>	<a href="#">tinyText</a>
<a href="#">float</a>	<a href="#">nullableMorphs</a>	<a href="#">unsignedBigInteger</a>
<a href="#">foreignKey</a>	<a href="#">nullableTimestamps</a>	<a href="#">unsignedDecimal</a>
<a href="#">foreignKeyFor</a>	<a href="#">nullableUlidMorphs</a>	<a href="#">unsignedInteger</a>
<a href="#">foreignUlid</a>	<a href="#">nullableUuidMorphs</a>	<a href="#">unsignedMediumInteger</a>
<a href="#">foreignUuid</a>	<a href="#">point</a>	<a href="#">unsignedSmallInteger</a>
<a href="#">geometryCollection</a>	<a href="#">polygon</a>	<a href="#">unsignedTinyInteger</a>
<a href="#">geometry</a>	<a href="#">rememberToken</a>	<a href="#">ulidMorphs</a>
<a href="#">id</a>	<a href="#">set</a>	<a href="#">uuidMorphs</a>
<a href="#">increments</a>	<a href="#">smallIncrements</a>	<a href="#">ulid</a>
<a href="#">integer</a>	<a href="#">smallInteger</a>	<a href="#">uuid</a>

# Añadiendo propiedades a la migración



Volviendo a la migración que acabamos de crear la configuramos de la siguiente manera teniendo en cuenta que cada posts esta relacionado con el id de la tabla de usuarios.

```
git commit -am "Añadiendo propiedades a la migración"
git push origin main

diegokld30 *
public function up(): void
{
    Schema::create('posts', function (Blueprint $table) {
        $table->id();
        $table->string('column: 'titulo');
        $table->text('column: 'descripcion');
        $table->string('column: 'imagen');
        $table->foreignId('column: 'user_id')->constrained()->onDelete('cascade');
        $table->timestamps();
    });
}
```

Realicemos la migración.

Nota: si en algún momento no pueden hacer migraciones prueben con:  
php artisan migrate:refresh

# Factories para testing en BD



Los Factory permiten hacer pruebas a las bases de datos lo cual es bastante conveniente en la fase de desarrollo. En este caso, acabamos de crear una migración por lo que queremos saber si se está guardando la información correctamente.

Resumen: En lugar de llenar el formulario de crear post mil veces, crea un factory para automatizar ese proceso.

Vamos a abrir el factories creado anteriormente (PostFactory) y retornaremos lo siguiente haciendo uso de la librería faker

A screenshot of a code editor showing two files: Post.php and PostFactory.php. The PostFactory.php file is currently active and displays the following PHP code:

```
namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;
use Illuminate\Database\Eloquent\Factories\Factory as diegokld30;

/**
 * @extends \Illuminate\Database\Eloquent\Factories\Factory<\App\Models\Post>
 */
no usages diegokld30
class PostFactory extends Factory
{
    /**
     * Define the model's default state.
     *
     * @return \Illuminate\Database\Eloquent\Model|mixed
     */
    diegokld30
no usages diegokld30
public function definition(): array
{
    return [
        'titulo'=>$this->faker->sentence( nbWords: 5),
        'descripcion'=>$this->faker->sentence( nbWords: 20),
        'imagen'=>$this->faker->uuid().'jpg',
        'user_id'=>$this->faker->randomElement( [1,2,3])
    ];
}
```

The code uses the Faker library to generate sample data for a Post model, defining default values for attributes like title, description, image, and user ID.

# Factories para testing en BD



Para correr el anterior factory, vamos a utilizar una herramienta llamada “tinker”. Para esto abriremos una nueva terminal y escribimos

Php artisan tinker

Tal como observamos, podremos interactuar con la base de datos, en este caso usaremos el modelo User y le diremos que busque un id que no existe y notaremos como nos devuelve un null.

```
ingdiegoc@MacBook-Pro-de-Diego devstagram % php artisan tinker
Psy Shell v0.11.21 (PHP 8.2.11 - cli) by Justin Hileman
> $usuario = User::Find(4)
[!] Aliasing 'User' to 'App\Models\User' for this Tinker session.
= null
```

# Factories para testing en BD



Ahora enviamos un id que si existe en la bd

```
> > $usuario = User::Find(1)
= App\Models\User {#6301
    id: 1,
    name: "diego",
    email: "diego@info.com",
    email_verified_at: null,
    #password: "$2y$10$GC6X.xHKrd77LnpjLeLax0jQ/UZwaHRHziEd3ybKMzcF//9x9m5TS",
    #remember_token: null,
    created_at: "2023-11-06 21:55:00",
    updated_at: "2023-11-06 21:55:00",
    username: "diegocalderon",
}
```

# Factories para testing en BD

Ahora vamos a correr el factory el cual esta directamente relacionado con el posts ya que asi lo creamos:  
Para correrlo en la terminal que estamos trabajando escribimos:

```
App\Models\Post::factory()
```

Tambien podríamos escribir App\Models\Post::factory();

Ahora le diremos la cantidad de veces que queremos crear el comando de forma automática, teniendo en cuenta que va a elegir aleatoriamente 3 id:

```
App\Models\Post::factory()->times(200)->create();
```

Con esto le estoy diciendo que cree 200 repeticiones de tal prueba que voy a ejecutar. Si sale algún error, es por qué no se ingresaron id válidos y las pruebas solo se pueden hacer sobre datos existentes:

```
'user_id'=>$this->faker->randomElement([1,2,3])
```

Si dentro de randomElement, enviara 4,6,7 que son datos no existentes me debe mostrar error. Para correr nuevamente debe salir con exit y volver a enviar el comando.

# Factories para testing en BD



Vimos que se crearon todos esos en terminal, pero verifiquemos en la bd que esta pasando.

✓ Mostrando filas 0 - 24 (total de 400, La consulta tardó 0.0001 segundos.)

**SELECT \* FROM `posts`**

Perfilando [ Editar en línea ] [ Editar ] [ Explicar SQL ] [ Crear código PHP ] [ Actualizar ]

	id	título	descripción	imagen	user_id	created_at	updated_at
<input type="checkbox"/>	1	Tenetur consequatur consequatur non eligendi exped...	Maiores ratione repellendus quasi consequatur rati...	ff67729-b340-32aa-9d8d-0559850155ccjpg	3	2023-11-06 22:08:17	2023-11-06 22:08:17
<input type="checkbox"/>	2	Minus consequatur placeat quia dolorem.	Cum eos molestiae dicta voluptatem tempora qui qua...	d5ea0bb6-5909-351d-84ea-09bd8a7eeb23jpg	2	2023-11-06 22:08:17	2023-11-06 22:08:17
<input type="checkbox"/>	3	Incidunt fugiat vero repudiandae aut.	Totam vero blanditiis non aliquid molestias maiore...	4f299c60-d46b-302a-b194-f329ab014451.jpg	2	2023-11-06 22:08:17	2023-11-06 22:08:17
<input type="checkbox"/>	4	Soluta repellendus nam sit quam laudantium.	Asperiores quia vitae qui architecto nulla expedit...	22a6b51a-c3f8-3a39-b065-86d105de5827.jpg	3	2023-11-06 22:08:17	2023-11-06 22:08:17
<input type="checkbox"/>	5	Temporibus voluptate eum in odit laudantium volupt...	Eveniet ut hic quidem perspicatis facere ea rem c...	0c49beb7-001f-38f4-9ce6-411089f5fcf2.jpg	2	2023-11-06 22:08:17	2023-11-06 22:08:17
<input type="checkbox"/>	6	Incident labore perspicatis iste deleniti nobis.	Reiciendis doloribus porro non sint ut culpa eos v...	811d7aba-b9c8-3a54-b517-5c8574431fb9.jpg	2	2023-11-06 22:08:17	2023-11-06 22:08:17
<input type="checkbox"/>	7	Itaque ut maxime cupiditate.	Consequuntur qui minus ullam dicta preferendis off...	debb1606-d641-3dbc-9fb7-b75efd362566.jpg	2	2023-11-06 22:08:17	2023-11-06 22:08:17
<input type="checkbox"/>	8	Harum debitis laudantium adipisci dolor sequi labo...	Minima nemo et ullam tempora quam ullam esse provi...	66b70a54-ec2a-39b8-97bd-9d66921139f6.jpg	1	2023-11-06 22:08:17	2023-11-06 22:08:17

Inserto tantos usuarios como le dijimos, pero en el campo de imagen, vemos que al final no está ".jpg" como le dijimos, sino que esta sin el punto. También notamos que relaciona perfectamente los usuarios en el campo user\_id y si pinchamos sobre el dato nos lleva al perfil correcto.

# Factories para testing en BD



Volviendo al PostFactory agregare el punto en la extensión y revertire la migración para volverla a hacer:

```
'imagen'=>$this->faker->uuid() . '.jpg',
```

```
ingdiegoc@MacBook-Pro-de-Diego devstagram % php artisan migrate:rollback --step=1

INFO Rolling back migrations.

2023_11_03_013648_create_posts_table ..... 14ms DONE

ingdiegoc@MacBook-Pro-de-Diego devstagram % php artisan migrate

INFO Running migrations.

2023_11_03_013648_create_posts_table ..... 35ms DONE
```

No uso refresh ya que no quiero perder la información guardada en la bd.

Ahora volveremos a tinker y volveré a ejecutar ese factory

```
ingdiegoc@MacBook-Pro-de-Diego devstagram % php artisan tinker
Psy Shell v0.11.21 (PHP 8.2.11 - cli) by Justin Hileman
> App\Models\Post::factory()->times(200)->create();
= Illuminate\Database\Eloquent\Collection {#6313
    all: [
        App\Models\Post {#6342
            id: 1
            title: "Post 1"
            content: "Content of Post 1"
            created_at: "2023-11-03T10:00:00Z"
            updated_at: "2023-11-03T10:00:00Z"
        }
    ]
}
```

# Validación a las publicaciones



Continuando con el proyecto, miremos como almacenar las imágenes, asi que abrire web.php, postController y create.blade.php.

Lo primero es definir la ruta de tipo post:

```
Route::post('uri: '/posts', [PostController::class, 'store'])->name('name: posts.store');
```

Ahora creamos el método con un dd para confirmar el funcionamiento del endpoint

```
public function store(){
    dd(...vars: 'Hola desde el posts del formulario');
}
```

Y en el formulario cambiare el action de register a posts.store

```
16 <div class="md:w-6/12 px-10 bg-white p-6 rounded-lg shadow-xl">
17     <form action="{{route('posts.store')}}" method="POST" novalidate>
```

Pinchamos el botón y vemos que el endpoint funciona. Es por esto que ahora si crearemos la validación.

# Validación a las publicaciones

La validación en el método stores será:

```
public function store(Request $request){  
    $this->validate($request,[  
        'titulo' => 'required|max:250',  
        'descripcion' => 'required|max:250',  
    ]);  
}
```

Y ahora nos falta crear el campo que almacenara la imagen:

Vamos a créate Blade.php y debajo del error de la descripción se cierra un div. Debajo de ese div, crearemos un nuevo div así:

# Validación a las publicaciones



Vamos a creáte Blade.php y debajo del error de la descripción se cierra un div. Debajo de ese div, crearemos un nuevo div así:

```
        @error('descripcion')
        <p class="bg-red-500 text-white my-2 rounded-lg text-sm p-2 text-center">{{ $message }}</p>
        @enderror
    </div>

    <div class="mb-5">
        <input type="hidden" name="imagen">
        @error('imagen')
        <p class="bg-red-500 text-white my-2 rounded-lg text-sm p-2 text-center">{{ $message }}</p>
        @enderror
    </div>
```

Y registramos la validación de imagen en el controlador

```
public function store(Request $request){
    $this->validate($request, [
        'titulo' => 'required|max:250',
        'descripcion' => 'required|max:250',
        'imagen'=> 'required'
    ]);
}
```

# Validación a las publicaciones

Al recargar la pagina y darle al botón, tenemos el siguiente resultado:

**TITULO**

El campo titulo es requerido.

**DESCRIPCION DE LA PUBLICACION**

El campo descripcion es requerido.

El campo imagen es requerido.

**PUBLICAR**

# Validación a las publicaciones

Se debe a que tenemos un campo oculto. Para solucionar esto, iremos a app.js en el directorio resources el cual es el que se encarga de hacer reponsivo cualquier objeto que estemos trabajando. En el apartado de dropzone, le daremos la respuesta a la imagen así:

# Validación a las publicaciones



Veamos como recuperar la imagen del usuario si la validación falla, esto, para evitar duplicidad de imágenes en el servidor.

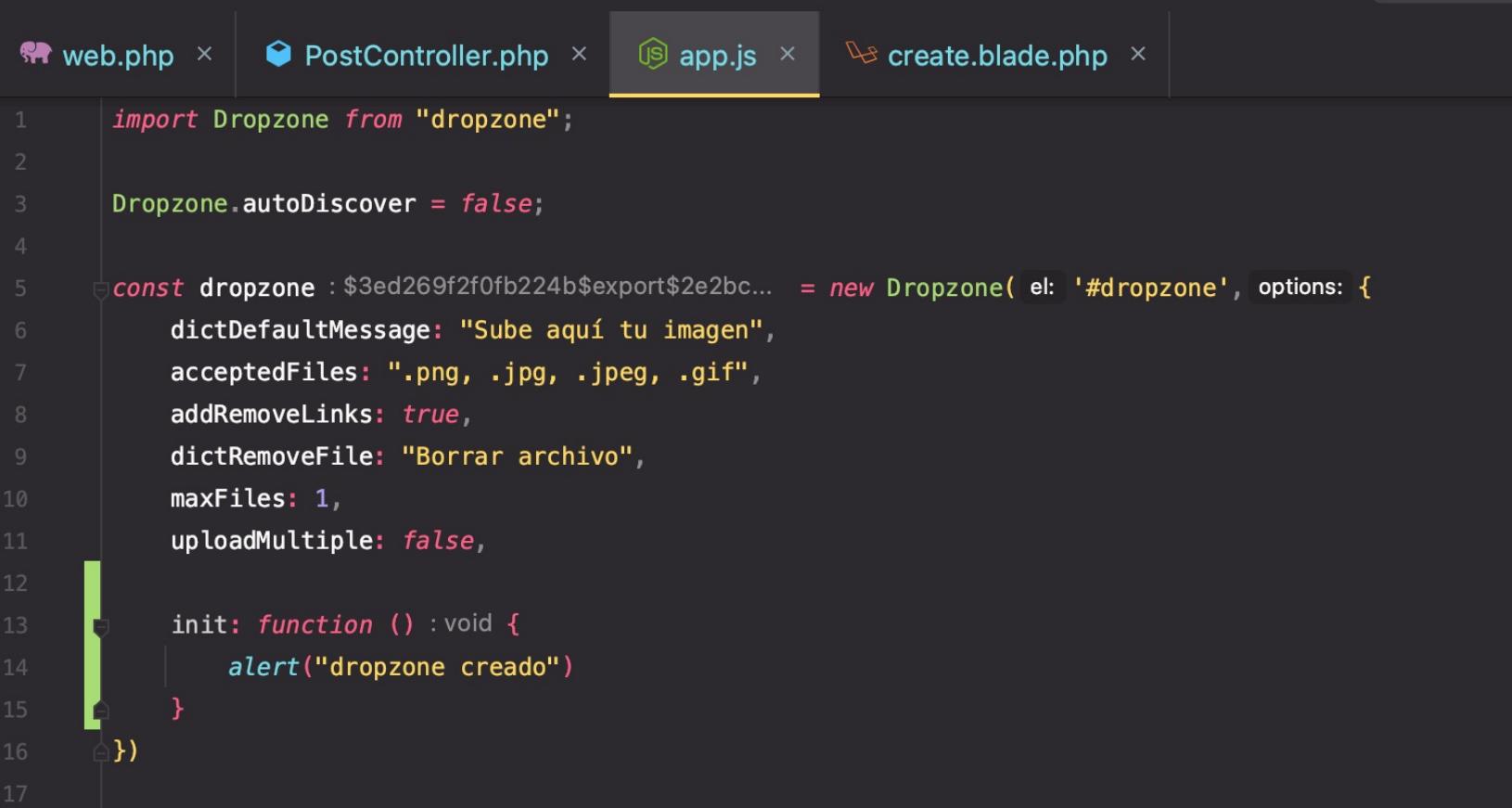
Lo primero es ir a `creáte.blade.php` y crear un value en el último div que creamos.

```
<div class="mb-5">
    <input type="hidden" name="imagen" value="{{ old('imagen') }}>
    @error('imagen')
        <p class="bg-red-500 text-white my-2 rounded-lg text-sm p-2 text-center">{{ $message }}</p>
    @enderror
</div>
```

Si intenta cargar otra imagen luego de recargar la página, dará la impresión de que no se hizo nada ni que se subió nada, así que, para eso, iremos al `app.js` y agregaremos en el dropzone un init:

# Validación a las publicaciones

Init:



```
web.php × PostController.php × app.js × create.blade.php ×
import Dropzone from "dropzone";

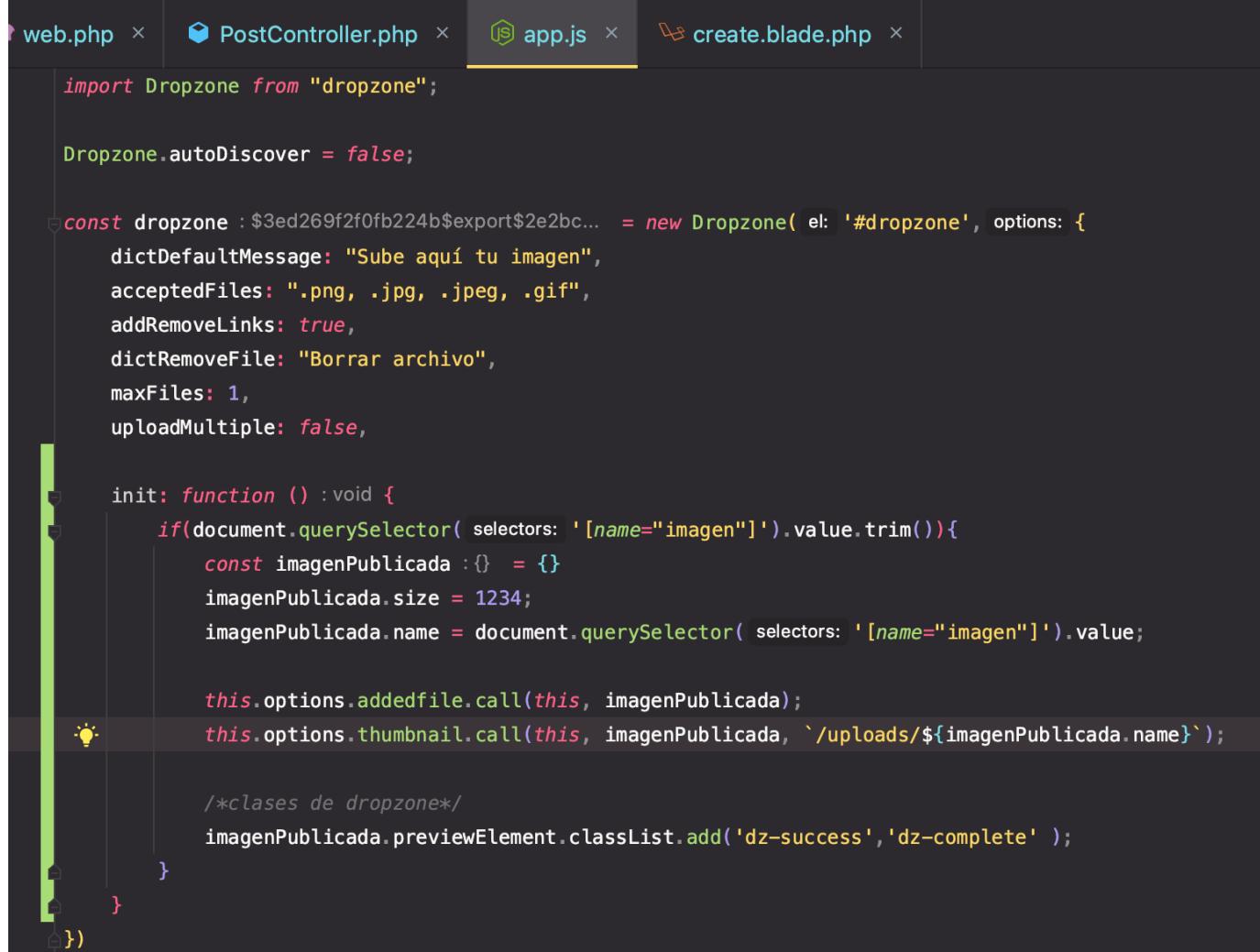
Dropzone.autoDiscover = false;

const dropzone : $3ed269f2f0fb224b$export$2e2bc... = new Dropzone( el: '#dropzone', options: {
    dictDefaultMessage: "Sube aquí tu imagen",
    acceptedFiles: ".png, .jpg, .jpeg, .gif",
    addRemoveLinks: true,
    dictRemoveFile: "Borrar archivo",
    maxFiles: 1,
    uploadMultiple: false,
    init: function () : void {
        alert("dropzone creado")
    }
})
```

Recarga la pagina y valida. Después de la validación cambiaremos el código del init

# Validación a las publicaciones

Vamos a hacer que pase algo solo si value tiene algún dato en el:



```
web.php × PostController.php × app.js × create.blade.php ×
import Dropzone from "dropzone";

Dropzone.autoDiscover = false;

const dropzone : $3ed269f2f0fb224b$export$2e2bc... = new Dropzone( el: '#dropzone', options: {
    dictDefaultMessage: "Sube aquí tu imagen",
    acceptedFiles: ".png, .jpg, .jpeg, .gif",
    addRemoveLinks: true,
    dictRemoveFile: "Borrar archivo",
    maxFiles: 1,
    uploadMultiple: false,

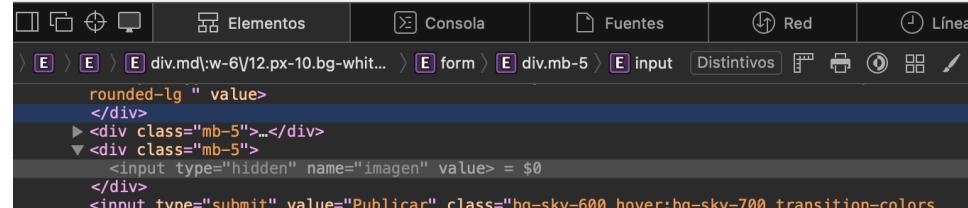
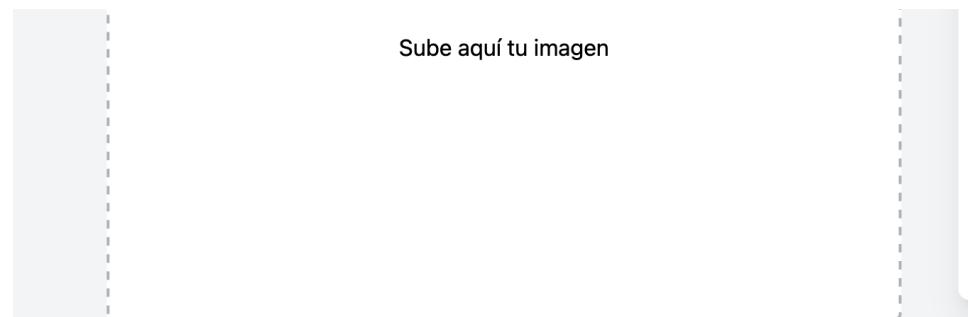
    init: function () :void {
        if(document.querySelector( selectors: '[name="imagen"]' ).value.trim()){
            const imagenPublicada :{} = {}
            imagenPublicada.size = 1234;
            imagenPublicada.name = document.querySelector( selectors: '[name="imagen"]' ).value;

            this.options.addedfile.call(this, imagenPublicada);
            this.options.thumbnail.call(this, imagenPublicada, `/uploads/${imagenPublicada.name}`);
        }
    }
})
```

# Validación a las publicaciones

Ahora, cuando intentamos eliminar la imagen, se borra del dropzone, pero en consola continua el value. Para corregir esto añadimos al final del código:

```
diegokld30 *
dropzone.on( event: 'removedfile', fn: function () : void {
    document.querySelector( selectors: '[name="imagen"]' ).value = ''
})
```



# Almacenando publicaciones

Recordemos un poco que datos debemos enviar a la tabla para guardar una publicación:

```
diegokld30 *  
public function up(): void  
{  
    Schema::create('posts', function (Blueprint $table) {  
        $table->id();  
        $table->string('column: 'titulo');  
        $table->text('column: 'descripcion');  
        $table->string('column: 'imagen');  
        $table->foreignId('column: 'user_id')->constrained()->onDelete('cascade');  
        $table->timestamps();  
    });  
}
```

Tal como se observa, debemos enviar título, descripción y también el id del usuario que inicio sesión. Para esto nos dirigimos al PostController y después de la validación ponemos:

# Almacenando publicaciones

Dentro del método stores agregamos:

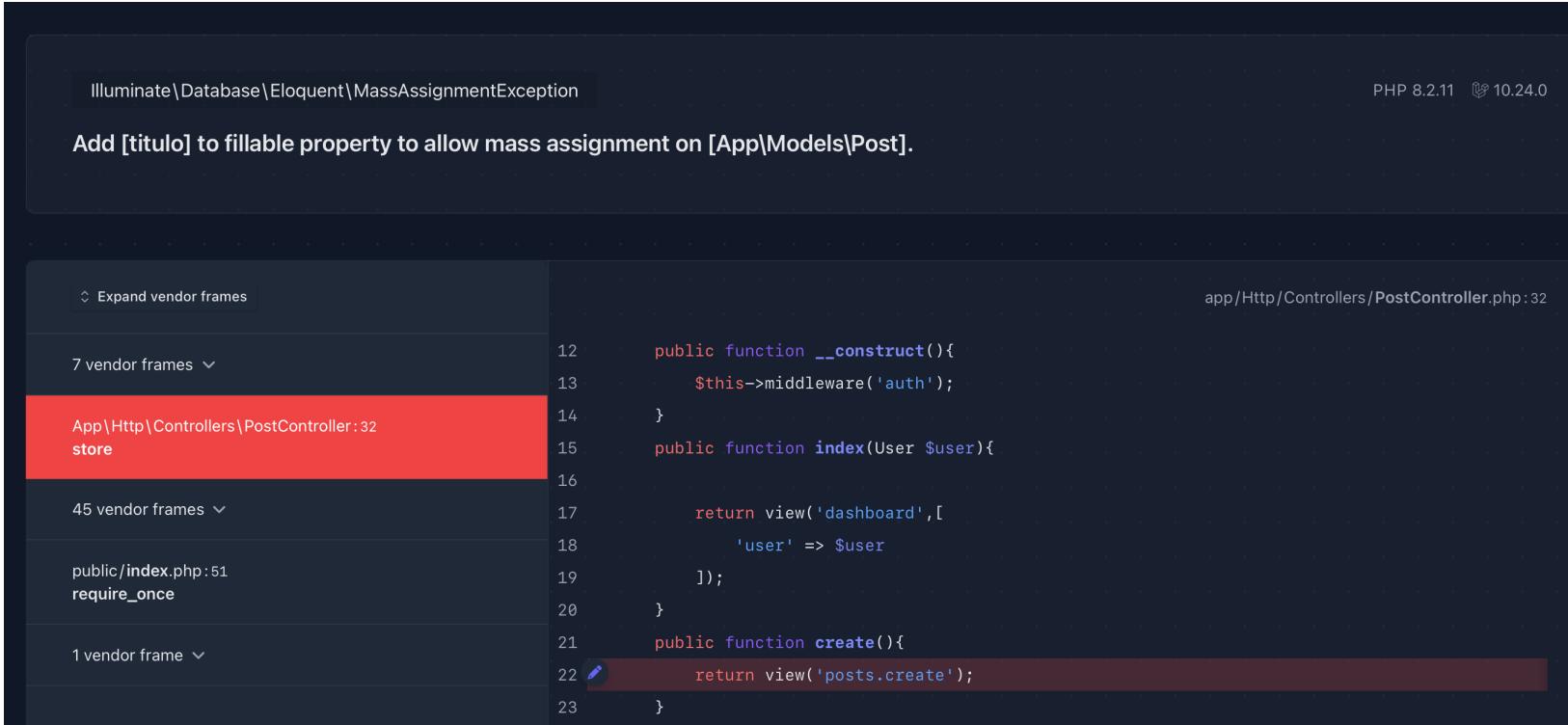
```
public function store(Request $request){
    $this->validate($request, [
        'titulo' => 'required|max:250',
        'descripcion' => 'required|max:250',
        'imagen'=> 'required'
    ]);

    Post::create([
        'titulo'=>$request->titulo,
        'descripcion'=>$request->descripcion,
        'imagen'=>$request->imagen,
        'user_id'=>auth()->user()->id
    ]);
    return redirect()->route( route: 'post.index', auth()->user()->username);
}
```

Llene los datos, cargue una imagen y dele enviar.

# Almacenando publicaciones

Si le funciona el código, maravilloso, si no, como a mi, la solución se presenta a continuación.



The screenshot shows a Laravel error page with the following details:

Exception: Illuminate\Database\Eloquent\MassAssignmentException

Message: Add [titulo] to fillable property to allow mass assignment on [App\Models\Post].

Environment: PHP 8.2.11 | MySQL 10.24.0

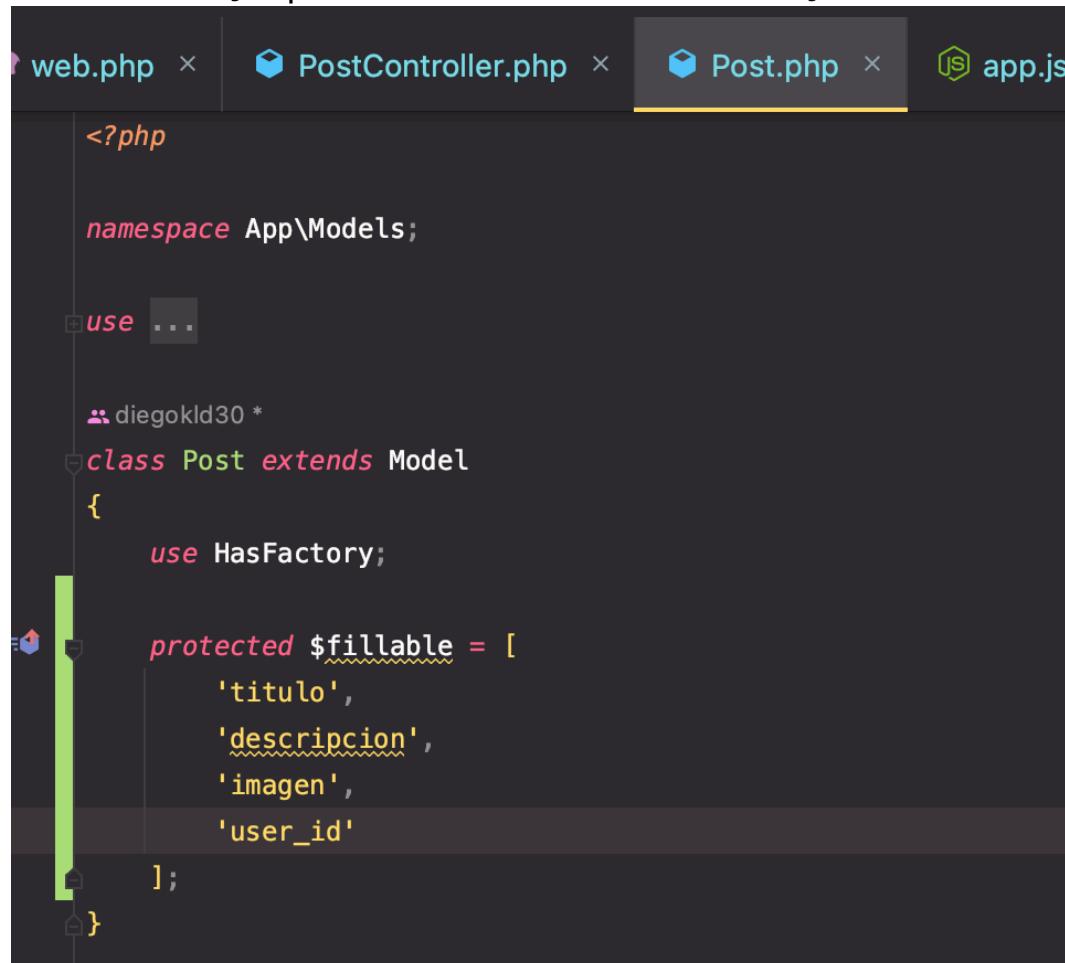
Stack Trace:

```
app/Http/Controllers/PostController.php:32
12     public function __construct(){
13         $this->middleware('auth');
14     }
15     public function index(User $user){
16
17         return view('dashboard', [
18             'user' => $user
19         ]);
20     }
21     public function create(){
22         return view('posts.create');
23     }
```

The stack trace shows the error occurred at line 32 of PostController.php. The code at line 22 is highlighted with a red rectangle, indicating the problematic line.

# Almacenando publicaciones

Este error es debido a que no hemos colocado lo que vamos a enviar en el fillable, es por esto que vamos a agregar el titulo en ese campo. Para eso hay que ir al modelo de Post y crearlo así:



```
<?php

namespace App\Models;

use ...;

use diegokld30 *
class Post extends Model
{
    use HasFactory;

    protected $fillable = [
        'titulo',
        'descripcion',
        'imagen',
        'user_id'
    ];
}
```

# Almacenando publicaciones

Recargamos, y si todo sale bien lo debe dirigir al inicio del muro



Y también en su BD creo una relación.





# G R A C I A S

Línea de atención al ciudadano: 01 8000 910270  
Línea de atención al empresario: 01 8000 910682



[www.sena.edu.co](http://www.sena.edu.co)