
UD10.7 – Laravel

Formularios y validación de datos

2º CFGS
Desarrollo de Aplicaciones Web
2022-23

1.- Formularios

Por el funcionamiento de Laravel, el uso de formularios implica tener en cuenta los diferentes **métodos** permitidos para cada ruta:



GET HEAD	posts
POST	posts
GET HEAD	posts/crear
GET HEAD	posts/{post}
PUT PATCH	posts/{post}
DELETE	posts/{post}
GET HEAD	posts/{post}/editar ...

1.- Formularios

Método	Ruta	Uso
GET	posts	Se puede utilizar desde enlaces normales. La vista mostrará el listado de todos los posts de la base de datos.
POST	posts	Se debe utilizar desde un formulario usando el método post. Recibe los datos de un post y los almacena en la base de datos.
GET	posts/create	Se puede utilizar desde enlaces normales. La vista mostrará un formulario con todos los campos para guardar un post. El action del formulario tendrá la ruta de la línea anterior de esta tabla.
GET	posts/{post}	Se puede usar desde enlaces normales. La vista mostrará todos los datos del post.
PUT	posts/{post}	Se debe utilizar desde un formulario usando el método post y @method('put'). Recibe los nuevos datos de un post y los actualiza en la base de datos.
GET	posts/{post}/edit	Se puede utilizar desde enlaces normales. La vista mostrará un formulario con todos los campos para guardar un post. El action del formulario tendrá la ruta de la línea anterior de esta tabla.
DELETE	posts/{post}	Se debe utilizar desde un formulario usando el método post y @method('delete'). Elimina el post de la base de datos.

1.- Formularios

Cada vez que se realizan acciones que modifican registros de la base de datos se debe mostrar feedback al usuario.

Las rutas que modifican registros son de los tipos: POST, PUT y DELETE.

Una manera rápida de ofrecer ese feedback es que los métodos que gestionan esas rutas devuelvan una vista con un mensaje del tipo:

"Elemento guardado correctamente"

"Elemento eliminado correctamente"

1.- Formularios

Otra manera es que esos métodos devuelvan una de las vistas del controlador y que se le pase el mensaje como argumento extra.

Esta acción ya se estudio en el apartado de las rutas.

Por ejemplo:

- Al almacenar un elemento se puede devolver la vista show de ese controlador.
- Al eliminar un elemento se puede devolver la vista index de ese controlador.
- Al modificar un elemento se puede devolver la vista show de ese controlador.
- ...

1.- Formularios

Para ver el funcionamiento de los formularios se utilizará como ejemplo el siguiente escenario:

- Una tabla llamada Books con los campos: id, titulo, isbn, precio y timestamps.
- Un modelo llamado Book.
- Un controlador tipo recurso llamado BookController.
- Rutas creadas para el controlador BookController.

2.- Guardando datos nuevos en la base de datos

Para guardar un registro nuevo en la tabla Books de la base de datos se necesitará una vista: **resources/views/books/create.blade.php**.

Esta vista mostrará un formulario para introducir los datos del libro a **almacenar**.

La estructura básica del formulario será la siguiente.

```
<form action="{{route('books.store')}}" method="post">
    @csrf

    {{--
    | Campos HTML para almacenar los datos de un libro
    --}}

    <input type="submit" value="Guardar">
</form>
```

2.- Guardando datos nuevos en la base de datos

Vista: **resources/views/books/create.blade.php**.

```
resources > views > books > create.blade.php > ...
1  @extends('layout')
2
3  @section('titulo', 'Nuevo libro')
4
5  @section('contenido')
6      <h1>Introduce un libro nuevo</h1>
7
8      <form action="{{route('books.store')}}" method="post">
9          @csrf
10         <label for="titulo">Título: </label>
11         <input type="text" id="titulo" name="titulo">
12         <br>
13         <label for="isbn">ISBN: </label>
14         <input type="text" id="isbn" name="isbn">
15         <br>
16         <label for="precio">Precio: </label>
17         <input type="text" id="precio" name="precio">
18         <br>
19         <input type="submit" value="Guardar">
20     </form>
21 @endsection
```


2.- Guardando datos nuevos en la base de datos

El formulario anterior enviará los datos a la ruta **/books/store** mediante el método **post** y será el método **store** del controlador el que recibirá los datos del formulario.

Así, el siguiente paso es guardar los datos en la base de datos desde el método store:

```
public function store(BookRequest $request)
{
    $book = new Book();
    $book->titulo = $request->get('titulo');
    $book->isbn = $request->get('isbn');
    $book->precio = $request->get('precio');
    $book->save();

    return redirect()->route('books.show', ['book' => $book->id]);
}
```

2.- Guardando datos nuevos en la base de datos

Para almacenar los datos en la base de datos **se necesita un objeto nuevo del modelo** y sobre él se deben ir guardando sus valores obteniéndolos de la variable **\$request**.

Una vez guardados todos los campos se debe usar el método **save**.

Al ejecutar el método **save** se guarda el registro en la base de datos y además está disponible el **id del nuevo registro**.

Gracias al nuevo **id** se puede redirigir a la página web del nuevo elemento.

```
public function store(Request $request)
{
    $book = new Book();
    $book->titulo = $request->get('titulo');
    $book->isbn = $request->get('isbn');
    $book->precio = $request->get('precio');
    $book->save();

    return redirect()->route('books.show', ['book' => $book->id]);
}
```

2.- Guardando datos nuevos en la base de datos

Si existen relaciones entre las tablas de la base de datos, a las vistas con formularios se les puede mandar los datos de las tablas relacionadas para usarlos en el formulario:

```
public function create()  
{  
    $autores = Author::all();  
    return view('books.create', compact('autores'));  
}
```

Al recibir los datos de la tabla relacionada se puede añadir un campo seleccionable:

```
<select name="autor" id="autor">  
    <option value="0">Selecciona un autor</option>  
    @foreach ($autores as $autor)  
        <option value="{{ $autor->id }}">{{ $autor->nombre }}</option>  
    @endforeach  
</select>
```

2.- Guardando datos nuevos en la base de datos

Para relacionar el nuevo registro **Book** con su **Author** en el método **store** se debe asociar de la siguiente manera:



```
public function store(Request $request)
{
    $book = new Book();
    $book->titulo = $request->get('titulo');
    $book->isbn = $request->get('isbn');
    $book->precio = $request->get('precio');
    $book->author()->associate(Author::findOrFail($request->get('autor')));
    $book->save();

    return redirect()->route('books.show', ['book' => $book->id]);
}
```

3.- Actualizando datos nuevos en la base de datos

El procedimiento es similar al de guardar un registro nuevo.

se necesitará una vista: **resources/views/books/edit.blade.php**.

Esta vista mostrará un formulario con los datos del libro a **modificar**.

La estructura básica del formulario será la siguiente.

```
<form action="{{route('books.update', $book->id)}}" method="post">
    @csrf
    @method('put')

    {{--
        Campos HTML para modificar Los datos de un Libro
    --}}

    <input type="submit" value="Guardar">
</form>
```

3.- Actualizando datos nuevos en la base de datos

Vista: **resources/views/books/edit.blade.php**.

Como se recibe el objeto Book se puede usar en la propiedad value de los campos para rellenar los datos.

```
resources > views > books > edit.blade.php > ...
1  @extends('layout')
2
3  @section('titulo', 'Nuevo libro')
4
5  @section('contenido')
6      <h1>Introduce un libro nuevo</h1>
7
8      <form action="{{route('books.update', $book->id)}}" method="post">
9          @csrf
10         @method('put')
11
12         <label for="titulo">Título: </label>
13         <input type="text" id="titulo" name="titulo" value="{{ $book->titulo }}">
14         <br>
15         <label for="isbn">ISBN: </label>
16         <input type="text" id="isbn" name="isbn" value="{{ $book->isbn }}">
17         <br>
18         <label for="precio">Precio: </label>
19         <input type="text" id="precio" name="precio" value="{{ $book->precio }}">
20         <br>
21         <input type="submit" value="Guardar">
22     </form>
23 @endsection
```

3.- Actualizando datos nuevos en la base de datos

Si se envían datos de una tabla relacionada, se puede seleccionar directamente el registro que es clave ajena del elemento que se está editando:

```
<select name="autor" id="autor">
  <option value="0">Selecciona un autor</option>
  @foreach ($autores as $autor)
    <option value="{ $autor->id }" {{ $autor->id==$book->author_id?'selected':'' }}>
      {{ $autor->nombre }}
    </option>
  @endforeach
</select>
```

También se pueden marcar los checkbox y radiobutton si se usaron en el formulario de creación del registro:

```
<input type="checkbox" name="visibilidad" id="visibilidad" {{ $post->visibilidad==1?'checked':'' }}>
```

3.- Actualizando datos nuevos en la base de datos

El formulario anterior enviará los datos a la ruta **/books/update** mediante el método **post** y será el método **update** del controlador el que recibirá los datos del formulario.

Así, el siguiente paso es guardar los datos en la base de datos desde el método **update**.

Como este método ya recibe el objeto **Book**, solo hay que realizar los cambios y salvar.

```
public function update(Request $request, Book $book)
{
    $book->titulo = $request->get('titulo');
    $book->isbn = $request->get('isbn');
    $book->precio = $request->get('precio');
    $book->author()->associate(Author::findOrFail($request->get('autor')));
    $book->save();

    return redirect()->route('books.show', ['book' => $book->id]);
}
```


4.- Slug

El **slug** es una herramienta que permite mejorar el posicionamiento SEO de una aplicación web haciendo uso de URL'S amigables legibles para las personas y con información para los motores de búsqueda.

Si se analizan las URL vistas en el caso de estudio del blog:

127.0.0.1:8000/posts/1

Aun siendo una URL amigable no aporta mucha información sobre el contenido de la página web.

4.- Slug

127.0.0.1:8000/posts/1

Imaginemos que la URL anterior muestra un post cuyo título es:

'Hoy, por primera vez en la historia, hay un hombre vivo con el corazón de un cerdo latiendo en su interior'

Es bastante improbable que exista otro post con el mismo título así que se puede usar ese mismo campo título para que forme parte de la URL amigable.

Aunque hay que realizar algunos cambios porque a los motores de búsqueda no les gustan ni los espacios en blanco ni los caracteres especiales como las tildes y si somos exquisitos ni las mayúsculas.

4.- Slug

Sabiendo lo anterior se podría cambiar la URL anterior:

127.0.0.1:8000/posts/1

Por esta:

127.0.0.1:8000/posts/hoy-por-primera-vez-en-la-historia-hay-un-hombre-vivo-con-el-corazon-de-un-cerdo-latiendo-en-su-interior

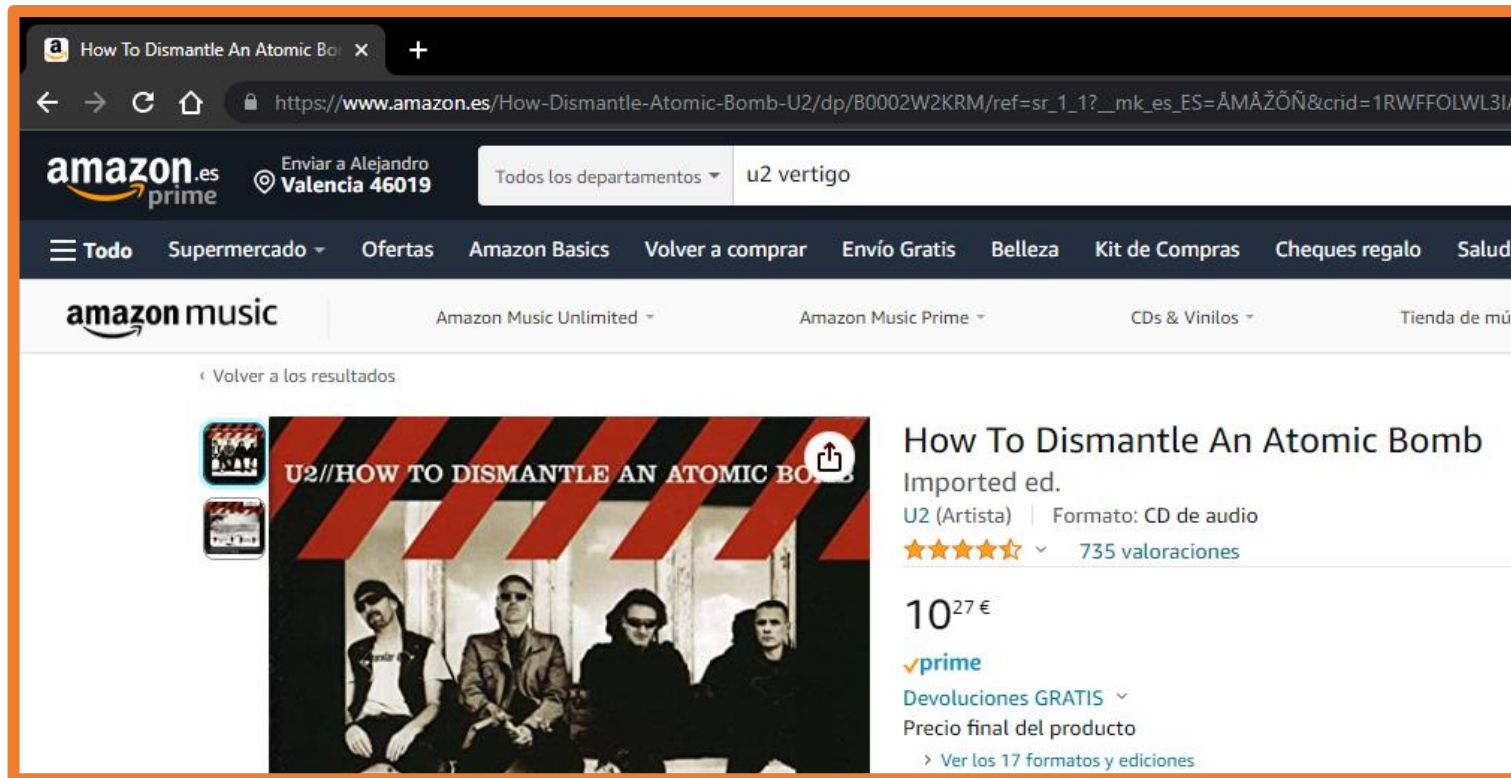
Esta nueva URL además de ser amigable, aporta mucha información sobre el contenido de la página web que va a mostrar.

4.- Slug

Ejemplos de slugs en aplicaciones web:

https://www.amazon.es/How-Dismantle-Atomic-Bomb-U2/dp/B0002W2KRM/ref=sr_1_1?keywords=u2+vertigo

En ocasiones el slug solo aporta información y acompaña al id del elemento a mostrar.



4.- Slug

Ejemplos de slugs en aplicaciones web:

<https://portal.edu.gva.es/iesserpis/2023/01/31/informacio-proves-dacces-cicles-formatius/>

A veces se añaden a las URL información como la fecha solo de manera informativa ya que es el slug el que permite encontrar en la base de datos el elemento a mostrar.



4.- Slug

Los cambios a realizar en la aplicación para poder usar un slug en una tabla de la base de datos son los siguientes:

- Añadir un campo nuevo a la tabla llamado slug que sea único (no se podrá repetir).
- Indicar en la ruta dentro de web.php que se va a usar ese slug en la URL.
- Indicar en el modelo de la tabla que el slug va a hacer las funciones de identificador.
- Cuando se guarda el registro se deberá generar el slug (Laravel ofrece una herramienta para realizar esta tarea de manera sencilla).

Práctica

Actividad 12:

Guardar y modificar registros.

Uso de slug.

5.- Validación de formularios

Como ya se ha visto anteriormente durante el curso cuando se trabaja con formularios HTML es recomendable realizar la validación de datos que envía el usuario en diferentes momentos:

- En el navegador → HTML5
- En el navegador antes de enviar los datos → JavaScript
- En el servidor al recibir los datos → PHP, Python, JSP...

5.- Validación de formularios

Laravel ofrece **un sistema de validación de datos** que se ejecuta antes de guardar los datos en la base de datos.

Este sistema de guardado **además proporciona información sobre los errores** que se puedan producir.

Lo aconsejable es crear una validación por cada formulario. Si existen dos formularios idénticos se puede reutilizar la validación creada.

Por ejemplo, para la tabla Books se usará un formulario en la vista create y otro similar en la vista edit por lo que se podrá reutilizar la validación.

5.- Validación de formularios

Para crear un **archivo de validación** se usa el comando:

```
# php artisan make:request nombre_validador
```

Por ejemplo:

```
# php artisan make:request BookRequest
```

Con el comando anterior se crea el archivo:

app/Requests/BookRequest.php

5.- Validación de formularios

Los **archivos de validación** contienen **2 métodos**, suficientes para validar y mostrar los errores en inglés, pero si se quiere mostrar los errores en otro idioma se puede añadir un tercer método.

- **authorize()**: indica si se necesitan privilegios o no.
- **rules()**: reglas a aplicar a los campos del formulario.
- **messages()**: mensajes para cada una de las reglas definidas.

5.- Validación de formularios

Veamos un ejemplo:

Para cada campo del formulario se pueden indicar ninguna, una o más reglas.

Para cada regla se puede indicar un mensaje (si no se indica saldrá en inglés).

```
public function authorize()
{
    return true;
    //return false;
}

public function rules()
{
    return [
        'titulo' => 'required|min:10',
        'isbn' => 'required|min:13|max:13|unique:books',
        'precio' => 'required|numeric'
    ];
}

public function messages()
{
    return [
        'titulo.required' => 'El título es obligatorio',
        'titulo.min' => 'El título debe tener al menos 10 caracteres',
        'isbn.required' => 'El isbn es obligatorio',
        'isbn.min' => 'El ISBN debe tener 13 caracteres',
        'isbn.max' => 'El ISBN debe tener 13 caracteres',
        'isbn.unique' => 'El ISBN introducido ya se encuentra en el sistema',
        'precio.required' => 'El precio es obligatorio',
        'precio.numeric' => 'El precio debe ser un número',
    ];
}
```

5.- Validación de formularios

Algunas reglas de validación:

bail

required

unique:nombre_tabla

max

min

...

email

numeric

integer

alpha

url

En la documentación se pueden consultar todas:
[reglas validación](#)

Accepted	Enum	Multiple Of
Accepted If	Exclude	Not In
Active URL	Exclude If	Not Regex
After (Date)	Exclude Unless	Nullable
After Or Equal (Date)	Exclude With	Numeric
Alpha	Exclude Without	Password
Alpha Dash	Exists (Database)	Present
Alpha Numeric	File	Prohibited
Array	Filled	Prohibited If
Ascii	Greater Than	Prohibited Unless
Bail	Greater Than Or Equal	Prohibits
Before (Date)	Image (File)	Regular Expression
Before Or Equal (Date)	In	Required
Between	In Array	Required If
Boolean	Integer	Required Unless
Confirmed	IP Address	Required With
Current Password	JSON	Required With All
Date	Less Than	Required Without
Date Equals	Less Than Or Equal	Required Without All
Date Format	Lowercase	Required Array Keys
Decimal	MAC Address	Same
Declined	Max	Size
Declined If	Max Digits	Sometimes
Different	MIME Types	Starts With
Digits	MIME Type By File Extens...	String
Digits Between	Min	Timezone
Dimensions (Image Files)	Min Digits	Unique (Database)
Distinct	Missing	Uppercase
Doesnt Start With	Missing If	URL
Doesnt End With	Missing Unless	ULID
Email	Missing With	UUID
Ends With	Missing With All	

5.- Validación de formularios

Cuando dos formularios son iguales pero las reglas de validación cambian entonces se deben crear dos archivos de validación diferentes.

Por ejemplo, en un formulario de registro de usuarios el nombre de usuario debe ser único.

Si posteriormente no se permite cambiar el nombre de usuario pero sí cambiar otros datos de la cuenta, en ese formulario no será necesario comprobar el nombre de usuario.

5.- Validación de formularios

El siguiente paso es indicar al controlador y a sus métodos que reciben los datos del formulario que se va a usar la validación:

- Se debe importar el archivo de validación en la parte superior.
use App\Http\Requests\BookRequest;
- Se debe cambiar el parámetro Request que reciben.

```
public function store(Request $request)
```



```
public function store(BookRequest $request)
```

```
public function update(Request $request, Book $book)
```



```
public function update(BookRequest $request, Book $book)
```

5.- Validación de formularios

Al usar los archivos de validación cuando se producen errores validando los datos recibidos Laravel vuelve a cargar la vista del formulario.

Laravel inyecta la variable **\$errors** donde se almacenan los errores producidos en la validación

Consultando la variable \$errors se podrá mostrar los mensajes correspondientes a los errores de validación.

La forma de mostrar los errores dependerá de las características de la aplicación y del gusto del diseñador/desarrollador.

5.- Validación de formularios

Mostrar el primer error:

```
@if($errors->any())  
    Hay errores en el formulario: <br>  
    {{$errors->first()}}  
@endif
```

Hay errores en el formulario:
El título es obligatorio

5.- Validación de formularios

Mostrar todos los errores seguidos:

```
@if($errors->any())
    Hay errores en el formulario: <br>
    <ul>
        @foreach($errors->all() as $error)
            <li>{{$error}}</li>
        @endforeach
    </ul>
@endif
```

Hay errores en el formulario:

- El título es obligatorio
- El ISBN introducido ya se encuentra en el sistema
- El precio es obligatorio

5.- Validación de formularios

Mostrar cada error bajo el campo en el que se produce:

```
<label for="titulo">Título: </label>  
<input type="text" id="titulo" name="titulo">  
@error('titulo') <br>Error: {{$message}} @enderror  
<br>
```

Título:

Error: El título es obligatorio

5.- Validación de formularios

Como se ha indicado anteriormente cuando se producen errores en la validación Laravel vuelve a cargar la vista del formulario.

Este comportamiento hace que lo escrito por el usuario en el formulario desaparezca.

Laravel ofrece una herramienta para solucionar ese problema.

Esta herramienta es el método **old**.

```
<label for="titulo">Título: </label>
<input type="text" id="titulo" name="titulo" value="{{old('titulo')}}">
@error('titulo') <br>Error: {{$message}} @enderror
<br>
```

Práctica

Actividad 13: Validando formularios.

6.- Envío de archivos a formularios

A la hora de añadir archivos en un formulario, la parte de la vista es similar a como se realiza con HTML5.

Al formulario se le debe añadir la propiedad `enctype="multipart/form-data"` para indicar que se van a enviar archivos.

```
<form action="{{route('posts.store')}}" method="post" enctype="multipart/form-data">  
  @csrf  
  
  <label for="titulo">Título: </label>
```

6.- Envío de archivos a formularios

En el controlador que recibe la información del formulario sobre el archivo recibido se debe utilizar el método **store** o el método **storeAs**.

En los dos métodos se debe indicar el lugar donde se guardará el archivo.

En el archivo **config/filesystems.php** están indicados todos los lugares posibles y se pueden indicar más si se necesitan. Por defecto se usará **public**.



```
'disks' => [  
  
    'local' => [  
        'driver' => 'local',  
        'root' => storage_path('app'),  
        'throw' => false,  
    ],  
  
    'public' => [  
        'driver' => 'local',  
        'root' => storage_path('app/public'),  
        'url' => env('APP_URL').'/storage',  
        'visibility' => 'public',  
        'throw' => false,  
    ],  
  
    's3' => [  
        'driver' => 's3',  
        'key' => env('AWS_ACCESS_KEY_ID'),  
        'secret' => env('AWS_SECRET_ACCESS_KEY'),  
        'region' => env('AWS_DEFAULT_REGION'),  
    ],  
]
```

6.- Envío de archivos a formularios

Con el método **store** será Laravel quien cree el nombre de archivo lo cuál es muy útil cuando es el usuario el que sube archivos al servidor para que no se repitan los nombres:

```
$nombreGenerado = $request->file('archivo')->store('public');
```


Este método se puede utilizar cuando se guardan varios archivos en el mismo lugar, y el nombre del archivo se almacena en la base de datos:

```
$nombreGenerado = $request->file('archivo')->store('public');  
$post = new Post();  
$post->title = $request->get('titulo');  
$post->slug = Str::slug($post->title);  
$post->content = $request->get('contenido');  
$post->avatar = $nombreGenerado;   
$post->save();
```


6.- Envío de archivos a formularios


Con el método **storeAs** se debe indicar el **nombre que tendrá el archivo a guardar**:

```
$request->file('archivo')->storeAs('public', 'curriculo.pdf');
```



Esta funcionalidad se puede utilizar, por ejemplo, cuando se quiera almacenar los archivos de un usuario **en una carpeta** para él:


```
$request->file('archivo')->storeAs('public/'. Auth::user()->name, 'curriculo.pdf');
```



6.- Envío de archivos a formularios

Tanto con el método **store** como con el método **storeAs** además de indicar que los archivos se almacenen en **public** se pueden indicar subcarpetas:

```
$nombreGenerado = $request->file('archivo')->store('public/avatars');
```



6.- Envío de archivos a formularios

Ubicación de los archivos almacenados con **store('public')**

Para preservar la privacidad del proyecto, Laravel ubica los archivos almacenados con el método **store** en la carpeta **/storage/app/public**.

Como bien la única carpeta accesible desde el navegador es la carpeta **public** que se encuentra en el raíz del proyecto para poder poner enlaces a archivos almacenados con **store** se debe ejecutar primero el siguiente comando:

```
# php artisan storage:link
```

6.- Envío de archivos a formularios

Una vez ejecutado el comando anterior ya se puede utilizar la URL **/storage** en los enlaces y en las imágenes para mostrar los archivos de esa ubicación y todas las subcarpetas que se hayan creado.

```
<a href="/storage/avatars/{{Auth::user()->avatar}}" alt="image">archivo</a>
```

6.- Envío de archivos a formularios

Si se añaden ubicaciones en zonas privadas del proyecto en el archivo **config/filesystems.php** se deberá volver a ejecutar el comando:

```
# php artisan storage:link
```

Así las nuevas ubicaciones también estarán disponibles.