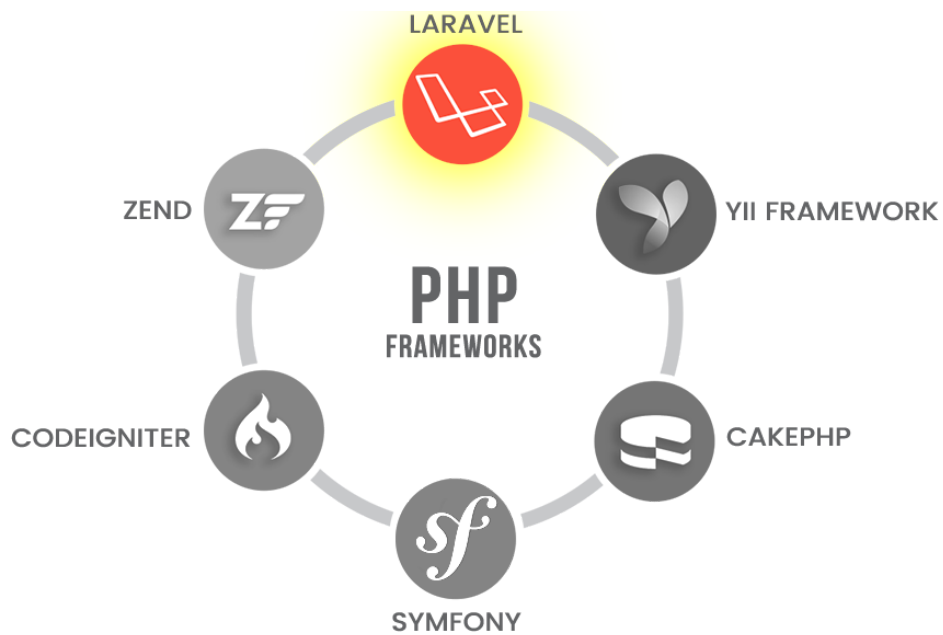


unidad didáctica 7

Framework Laravel



1. duración y criterios de evaluación

2. consideraciones previas

- 2. 1. MVC
 - 2. 1. 1. Modelo
 - 2. 1. 2. Vista
 - 2. 1. 3. Controlador
 - 2. 1. 4. Router
- 2. 2. Artisan

3. instalar docker bitnami/Laravel

- 3. 1. VSCode extensiones

4. carpetas en Laravel

- 4. 1. app/Http/Controllers
- 4. 2. Models
- 4. 3. public
- 4. 4. resources
- 4. 5. routes
- 4. 6. vendor
- 4. 7. .env

5. rutas

- 5. 1. alias
 - 5. 1. 1. forma corta para una vista
- 5. 2. parámetros

6. plantillas o templates

- 6. 1. directivas
- 6. 2. separando código
- 6. 3. estructuras de control

7. controladores

- 7. 1. convenciones

8. tipos de Request

- 8. 1. En HTTP y API's

9. validación de formularios

- 9. 1. mostrar errores de forma dinámica
- 9. 2. mensajes en castellano
- 9. 3. mantener el valor en el campo después de un error
- 9. 4. confirmar password en otro campo

10. migraciones y la base de datos

- 10. 1. qué son migraciones
 - 10. 1. 1. rollback de la migración

11. modelos

- 11. 1. ORM Eloquent
- 11. 2. convenciones en Laravel

11.2.1. [en Modelos](#)

11.3. [crear registros con Eloquent ORM](#)

11.3.1. [cambiar el campo 'username' a único](#)

11.4. [redireccionar al usuario al Muro una vez la cuenta es creada](#)

11.5. [autenticar un usuario que ha creado su cuenta](#)

12. [anexo I - instalación de Tailwind CSS](#)

13. [anexo II - reinstalación de node](#)

14. [ejemplos](#)

14.1. [ejemplo 01. Hola Mundo](#)

14.2. [ejemplo 02. Otras vistas](#)

14.3. [ejemplo 03. Paso de parámetros](#)

14.4. [ejemplo 04. Uso de directivas](#)

14.5. [ejemplo 05. Vista registrarse](#)

14.6. [ejemplo 06. Pasar datos a una vista](#)

14.7. [ejemplo 07. Controlador RegisterController y su formulario](#)

14.8. [ejemplo 08. Petición post](#)

14.8.1. [problema a la "vista"!!](#)

14.9. [ejemplo 09. Validación de campos](#)

14.10. [ejemplo 10. Mensajes de errores dinámicos](#)

14.11. [ejemplo 11. Mantener valor después de un error](#)

14.12. [ejemplo 12. Confirmar valor del password](#)

14.13. [ejemplo 13. Crear usuario en la app](#)

15. [ejercicios propuestos](#)

15.1. [Ejercicio 1](#)

15.1.1. [crear controlador y modelo](#)

15.1.2. [crear tabla](#)

15.1.3. [crear seeder](#)

15.1.4. [crear ruta](#)

15.1.5. [crear vista](#)

15.2. [Ejercicio 2](#)

15.2.1. [crear formulario](#)

15.2.2. [crear página](#)

16. [referencias](#)

1. duración y criterios de evaluación

Duración estimada: ∞ sesiones.

Resultado de aprendizaje y criterios de evaluación:

4. Desarrolla aplicaciones Web embebidas en lenguajes de marcas analizando e incorporando funcionalidades según especificaciones.

a) *Se han identificado los mecanismos disponibles para el mantenimiento de la información que concierne a un cliente web concreto y se han señalado sus ventajas.*

b) *Se han utilizado sesiones para mantener el estado de las aplicaciones Web.*

c) *Se han utilizado cookies para almacenar información en el cliente Web y para recuperar su contenido.*

d) *Se han identificado y caracterizado los mecanismos disponibles para la autenticación de usuarios.*

e) *Se han escrito aplicaciones que integren mecanismos de autenticación de usuarios.*

f) *Se han realizado adaptaciones a aplicaciones Web existentes como gestores de contenidos u otras.*

g) *Se han utilizado herramientas y entornos para facilitar la programación, prueba y depuración del código.*

2. consideraciones previas

2.1. MVC

MVC (Model View Controller o Modelo Vista Controlador) es un patrón de arquitectura de software que permite la separación de obligaciones de cada pieza de tu código.

Este paradigma de la programación enfatiza la separación de la lógica de programación con la presentación.

Ventajas:

- MVC no mejora el performance del código, tampoco da seguridad; pero **tu código tendrá un mejor orden y será fácil de mantener**.
- En un grupo de trabajo, el tener el código ordenado permite que más de una persona pueda entender que es lo que hace cada parte de él.
- Aprender MVC, te hará que otras tecnologías como *Nest, Rails, Django, Net Core, Spring Boot* te serán más sencillas de aprender.

2.1.1. Modelo

Encargado de todas las interacciones en la base de datos (obtener datos, actualizarlos y eliminar).

El Modelo se encarga de consultar una base de datos, obtiene la información pero no la muestra (esa tarea es para las vistas).

El Modelo tampoco se encarga de actualizar la información directamente (esa tarea es del Controlador, que es quien decide cuándo llamarlo).

2.1.2. Vista

Se encarga de todo lo que se ve en pantalla (HTML).

Laravel tiene un Template Engine llamado **Blade** para mostrar los datos.

Si utilizas *React, Vue, Angular, Svelte*, etc. estos serían tu vista.

El Modelo consulta la base de datos, pero es por medio del Controlador que se decide qué Vista hay que llamar y qué datos presentar.

2.1.3. Controlador

Es el que comunica *Modelo* y *Vista*; antes de que el *Modelo* consulte la base de datos el *Controlador* es el encargado de llamar un *Modelo* en específico.

Una vez consultado el *Modelo*, el *Controlador* recibe esa información, manda llamar a la *Vista* y le pasa la información.

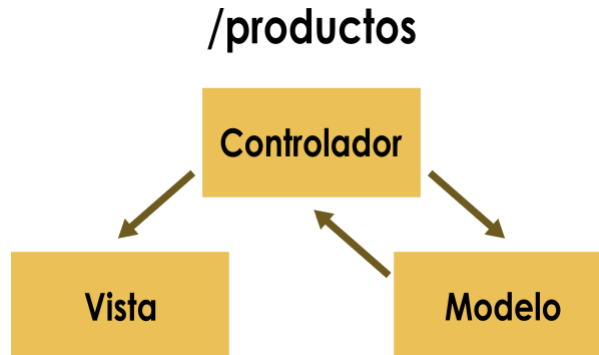
El *Controlador* es el que manda llamar la *Vista* y *Modelos*, que se requieren en cada parte de tu aplicación.

2.1.4. Router

Es el encargado de registrar todas las URL's o Endpoints que va a soportar nuestra aplicación.

Ejemplo:

Si el Usuario accede a `/productos` el router ya tiene registrada esa ruta y un controlador con una función que sabe que Modelo debe llamar y que vista mostrar cuando el usuario visita esa URL.



2.2. Artisan

Artisan es el CLI (Command Line Interface) incluido en Laravel.

Artisan es un script que existe en la base de tu proyecto de Laravel y cuenta con una gran cantidad de scripts disponibles.

Estos comandos te permiten crear *migraciones*, *controladores*, *modelos*, *policies* y mucho más.

Todos los comando podemos encontrarlos ejecutando:

```
1 # php artisan
2 # ó, si no funciona:
3 sudo docker-compose exec myapp php artisan
```

Por ejemplo, si quisiéramos crear un controlador (como veremos más adelante) la orden sería:

```
1 # php artisan make:controller RegisterController
2 # ó, si no funciona:
3 sudo docker-compose exec myapp php artisan make:controller RegisterController
```

3. instalar docker bitnami/Laravel

1. Lo primero de todo es crear una carpeta con el nombre del proyecto y accedemos ella.

Por ejemplo, creamos el proyecto *prularavel* dentro de nuestra carpeta de proyectos:

```
1 $ mkdir ~/dwes/proyectos/prularavel
```

2. Accedemos dentro de la carpeta de este nuevo proyecto.

3. En este punto tenemos dos opciones:

A. Utilizar la imagen de Bitnami ya preparada, así que lo que hacer ahora es [descargar el archivo docker-compose.yml](#) del repositorio de Github oficial.

```
1 curl -LO
https://raw.githubusercontent.com/bitnami/containers/main/bitnami/laravel/docker-compose.yml
```

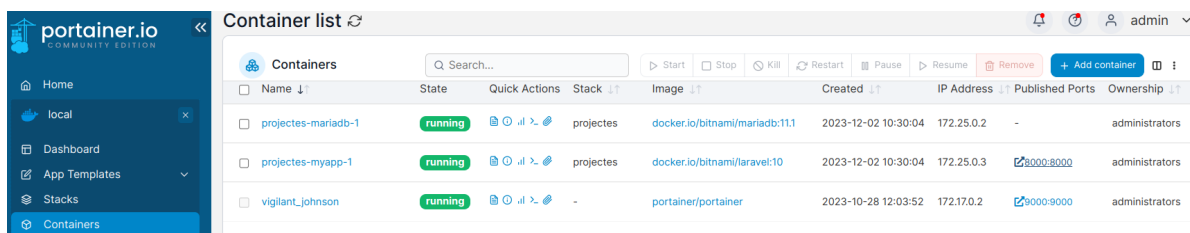
B. Utilizar el fichero `docker-compose.yml` que tenemos en nuestra carpeta del curso (en el que se añade el contenedor para *phpMyadmin*).

En caso ejecutaremos la opción B.

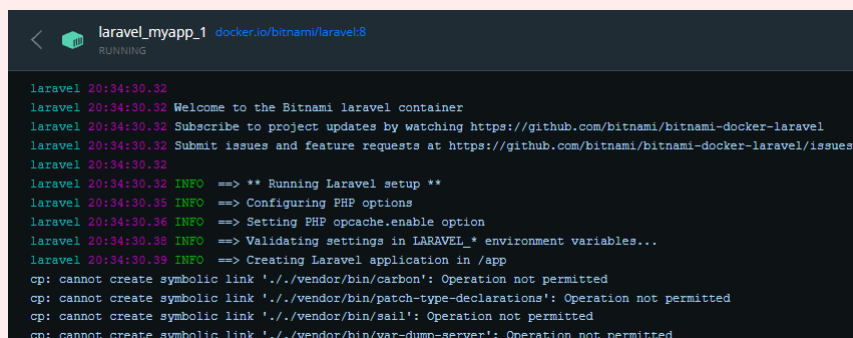
4. Una vez descargado el archivo en nuestra carpeta que acabamos de crear con el nombre del proyecto, lanzamos el siguiente comando por consola para instalar todas las dependencias y crear las imágenes de Docker correspondientes.

```
1 sudo docker-compose up -d
```

5. Si utilizamos el contenedor `Portainer` para la gestión de nuestros contenedores, podremos observar que estarán en marcha nuestros dos contenedores (pertenecientes al servidor web y servidor de bases de datos):



Si por alguna extraña razón estás en Windows y no te funciona una de las 2 imágenes, puede ser debido a la instalación de composer dentro de la imagen de Laravel.



Para solucionarlo, nos vamos a la carpeta del proyecto que se te habrá creado por defecto al hacer docker-compose; en este caso, y si no has modificado el archivo .yml, la carpeta del proyecto sera `my-proyect` y dentro de ella eliminamos la carpeta vendor.

Acto seguido instalar Composer de manera global en nuestro sistema Windows (bájate el instalador [desde este enlace](#)).

Una vez lo instales ya serás capaz de lanzar el comando composer desde cualquier consola de Windows.

3.1. VSCode extensiones

Recomendable instalar los siguientes plugins para Visual Studio Code.

Referentes a PHP:

- *PHP Intelephense*
- *PHP IntelliSense*
- *PHP Namespace Resolver*

Referentes a Laravel:

- *Laravel Blade Snippets*
- *Laravel Snippets*
- *Laravel goto view*
- *Laravel Extra Intellisense*

Referentes a API:

- *Thunder Client*

Referentes a CSS:

- *Tailwind CSS IntelliSense*

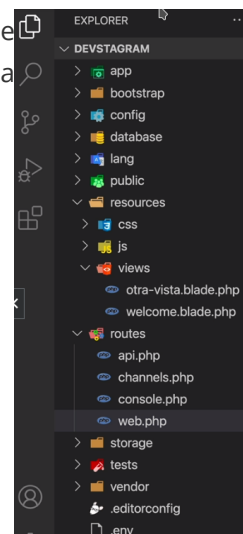
Aporte

Un aporte, o instalación, a tener en cuenta, podría ser la de instalar `Tailwind CSS`. Este software nos va a proporcionar, de manera sencilla y cómoda, una opción de utilizar CSS.

Para ello, seguir las instrucciones del [anexo I - instalación de Tailwind CSS](#).

4. carpetas en Laravel

Al crear un nuevo proyecto con este framework, Laravel crea una serie de carpetas por defecto. Esta estructura de carpetas es la recomendada para utilizar Laravel.



4.1. app/Http/Controllers

En esta carpeta es donde se van a introducir nuestros controladores (cuando se cree un controlador, es aquí donde va a ubicarse).

4.2. Models

Para seguir el paradigma de **MVC**, los modelos van a introducirse en esta carpeta.

4.3. public

Esta es la carpeta más importante ya que es donde se ponen todos los archivos que el cliente va a mostrar al usuario cuando introduzcamos la URL de nuestro sitio web (por ejemplo, dentro de esta carpeta crear otra carpeta `img`). Normalmente se carga el archivo `index.php` por defecto.

4.4. resources

Esta es nuestra carpeta de recursos donde guardaremos los siguientes archivos, que también, están separados por sus carpetas... como cada nombre indica:

- `css` Archivos CSS (archivos originales que son procesados y se colocan las versiones compiladas en `public` mediante al archivo `webpack.mix.js`).
- `js` Archivos JS o JavaScript (archivos originales que son procesados y se colocan las versiones compiladas en `public` mediante al archivo `webpack.mix.js`).
- `lang` Archivos relacionados con el idioma del sitio (variables & strings).
- `views` Archivos de nuestras vistas, lo que las rutas cargan (comúnmente: lo que se ve en pantalla).

Ejemplo: Podemos observar todo esto en el [ejemplo 01](#).

4.5. routes

Otra de las carpetas que más vamos a usar a lo largo de esta unidad dedicada a Laravel es `routes`. En ella se albergan todas las **rut**as (redirecciones web) de nuestro proyecto, pero más concretamente en el archivo `web.php`:

1 Dada una ruta → se cargará una vista

4.6. vendor

En esta carpeta se colocan todas las dependencias de `Composer` que son necesarias tanto para que Laravel funcione o si quieres agregar alguna dependencia extra (por ejemplo: agregar pagos con *Paypal* . Laravel tiene un paquete llamado *cashier* que permite pagos en línea y también podemos instalar una dependencia extra mediante Composer).

Podemos observar en el archivo `composer.json` las dependencias instaladas.

4.7. .env

Aunque `.env` no es una carpeta, sino un archivo, también merece especial atención por ser un fichero de configuración de nuestro proyecto. Por ejemplo, en nuestro caso, la conexión a base de datos sería:

```
1 # ...
2 DB_CONNECTION=mysql
3 DB_HOST=127.0.0.1
4 DB_PORT=3306
5 DB_DATABASE=dwes
6 DB_USERNAME=dwes
7 DB_PASSWORD=dwes
8 # ...
```

5. rutas

Las rutas en Laravel (y en casi cualquier Framework) sirven para redireccionar al cliente (o navegador) a las vistas que nosotros queramos.

Estas rutas se configuran en el archivo `routes/web.php` donde se define la ruta que el usuario pone en la URL después del dominio y se retorna la vista que se quiere cargar al introducir dicha dirección en el navegador.

En este ejemplo podemos observar la clase `Route` y un método estático `get`, este método estático toma la URL ('/' en este caso) y también un *closure* o *callback* `function () ...`

```
1 <?php
2 // Ruta por defecto para cargar la vista welcome (sin extensión
  .blade.php) cuando el usuario introduce simplemente el dominio
3 Route::get('/', function () {
4     return view('welcome');
5 });
```

En el ejemplo de arriba vamos a cargar la vista llamada *welcome* que hace referencia a la vista `resources/views/welcome.blade.php`.

closure VS controlador

Se puede definir en el segundo parámetro un closure o un controlador (veremos más adelante este caso).

Por ejemplo:

```
1 Route::get('/register', [RegisterController::class, 'index']) ->
  name('register');
```

5.1. alias

Es interesante darle un alias o un nombre a nuestras rutas para poder utilizar dichos alias en nuestras plantillas de Laravel que veremos más adelante.

Para ello, basta con utilizar la palabra `name` al final de la estructura de la ruta y darle un nombre que queramos; normalmente descriptivo y asociado a la vista que tiene que cargar el enrutador de Laravel.

```
1 <?php
2 Route::get('/users', function () {
3     return view('users');
4 }) -> name('usuarios');
```

Después veremos que es muy útil ya que a la hora de refactorizar o hacer un cambio, si tenemos enlaces o menús de navegación que apuntan a esta ruta, sólo tendríamos que cambiar el parámetro dentro del `get()` y no tener que ir archivo por archivo.

5.1.1. forma corta para una vista

Laravel nos proporciona una manera más cómoda a la hora de cargar una vista si no queremos parámetros ni condiciones. Tan sólo definiremos la siguiente línea que hace referencia la ruta datos en la URL y va a cargar el archivo `usuarios.php` de nuestra carpeta views como le hemos indicado en el segundo parámetro.

```
1 <?php
2     /* http://localhost/datos/ */
3     Route::view('datos', 'usuarios');
```

Pero no sólo podemos retornar una vista, sino, desde un simple string, a módulos propios de Laravel.

Ejemplo: Podemos observar todo esto en el [ejemplo 02](#).

5.2. parámetros

Ya hemos visto que con PHP podemos pasar parámetros a través de la URL, como si fueran variables, que las recuperábamos a través del método GET o POST.

Con Laravel también podemos introducir parámetros pero de una forma más vistosa y ordenada, de tal manera que sea visualmente más cómodo de recordar y de indexar por los motores de búsqueda como Google.

```
1 http://localhost/cliente/324
```

Para configurar este tipo de rutas en nuestro archivo de rutas `public/routes/web.php` haremos lo siguiente.

```
1 <?php
2     Route::get('cliente/{id}', function($id) {
3         return('Cliente con el id: ' . $id);
4     });
```

¿Qué pasa si no introducimos un id y sólo navegamos hasta cliente/ ? ... que nos va a devolver un `404 | NOT FOUND`.

Para resolver esto, podemos definir una ruta por defecto en caso de que el id (o parámetro) no sea pasado. Para ello usaremos el símbolo `?` en nuestro nombre de ruta e inicializaremos la variable con el valor que queramos.

```
1 <?php
2     Route::get('cliente/{id?}', function($id = 1) {
3         return ('Cliente con el id: ' . $id);
4     });
```

Ahora tenemos otro problema, porque estamos filtrando por id del cliente que, normalmente es un número; pero si introducimos un parámetro que no sea un número, vamos a obtener un resultado no deseado.

Para resolver este caso haremos uso de la cláusula `where` junto con una expresión regular numérica:

```

1 <?php
2     Route::get('cliente/{id?}', function($id = 1) {
3         return ('Cliente con el id: ' . $id);
4     }) -> where('id', '[0-9]+');

```

Además, podemos pasarle variables a nuestra URL para luego utilizarlas en nuestros archivos de plantillas o en archivos .php haciendo uso de un array asociativo. Veamos un ejemplo con la forma reducida para ahorrarnos código:

```

1 <?php
2     Route::view('datos', 'usuarios', ['id' => 5446]);

```

... y el archivo `resources/views/usuarios.blade.php` debe tener algo parecido a esto:

```

1 <!-- Estructura típica de un archivo HTML5 -->
2 <!-- ... -->
3 <p>Usuario con id: <?= $id ?></p>
4 <!-- ... -->

```

Con las plantillas de Laravel **blade.php** veremos cómo simplificar aún más nuestro código.

Para más información acerca de las rutas, parámetros y expresiones regulares en las rutas puedes echar un vistazo a la [documentación oficial de rutas](#) que contiene numerosos ejemplos.

Ejemplo: Podemos observar un ejemplo en [ejemplo 03](#).

6. plantillas o templates

A través de las plantillas de Laravel vamos a escribir **menos código** PHP y vamos a tener nuestros archivos **mejor organizados**.

Blade es el sistema de plantillas que trae Laravel, por eso los archivos de plantillas que guardamos en el directorio de `views` llevan la extensión `blade.php`.

De esta manera sabemos inmediatamente que se trata de una plantilla de Laravel y que forma parte de una vista que se mostrará en el navegador.

6.1. directivas

Laravel tiene un gran número de directivas que podemos utilizar para ahorrarnos mucho código repetitivo entre otras funciones.

Digamos que las directivas son pequeñas funciones ya escritas que aceptan parámetros y que cada una de ellas hace una función diferente dentro de Laravel. Por ejemplo:

- `@yield` define el contenido dinámico que se va a cargar. Se usa conjuntamente con `@section`.
- `@section` y `@endsection` es un bloque de código dinámico.
- `@extends` importa el contenido de una plantilla ya creada.

6.2. separando código

Veamos sobre un ejemplo cómo separar el código para no repetirlo.

Ejemplo: Podemos observar todo esto en el [ejemplo 04](#).

Ejemplo: Realiza también el [ejemplo 05](#).

6.3. estructuras de control

Como en todo buen lenguaje de programación, en Laravel también tenemos estructuras de control.

En Blade (plantillas de Laravel) siempre que iniciemos un bloque de estructura de control **DEBEMOS** cerrarla.

- `@foreach` ~ `@endforeach` lo usamos para recorrer arrays.
- `@if` ~ `@endif` para comprobar condiciones lógicas.
- `@switch` ~ `@endswitch` en función del valor de una variable ejecutar un código.
- `@case` define la casuística del switch.
- `@break` rompe la ejecución del código en curso.
- `@default` si ninguna casuística se cumple.

```

1 <?php
2     $equipo = ['María', 'Alfredo', 'William', 'Verónica'];
3
4     @foreach ($equipo as $elemento)
5         <p> {{ $elemento }} </p>
6     @endforeach
7     // si no funciona la estructura anterior:
8     // foreach ($equipo as $elemento) {
9     //     echo "<p>" . $elemento . "</p>";
10    // }

```

Acordaros que podemos pasar variables a través de las rutas como si fueran parámetros. Pero en este caso, vamos a ver otra directiva más; el uso de `@compact`.

```

1 <?php
2     // Uso de @compact
3     $equipo = ['María', 'Alfredo', 'William', 'Verónica'];
4
5     // Route::view('nosotros', ['equipo' => 'equipo']);
6     Route::view('nosotros', @compact('equipo'));

```

Ejemplo: Veamos sobre un ejemplo pasar información a la vista a través del [ejemplo 06](#).

7. controladores

Los controladores son el lugar perfecto para definir la **lógica de negocio** de nuestra aplicación o sitio web.

Hace de intermediario entre la *Vista* (lo que vemos con nuestro navegador o cliente) y el servidor donde la app está alojada.

Por defecto, los controladores se guardan en una carpeta específica situada en `app/Http/Controllers` y tienen extensión `.php`.

Para crear un controlador nuevo debemos hacer uso de nuestro querido CLI *artisan* donde le diremos que cree un controlador con el nombre que nosotros queramos.

Abrimos la consola y nos situamos en la raíz de nuestro proyecto:

```
1 php artisan make:controller PagesController
2 # ó, si no funciona:
3 # sudo docker-compose exec myapp php artisan make:controller PagesController
```

Si todo ha salido bien, recibiremos un mensaje por consola con que todo ha ido bien y podremos comprobar que, efectivamente, se ha creado el archivo `PagesController.php` con una estructura básica de controlador, dentro de la carpeta `Controllers` que hemos descrito anteriormente.

Ahora podemos modificar nuestro archivo de rutas `web.php` para dejarlo limpio de lógica y trasladar ésta a nuestro nuevo controlador. La idea de esto es dejar el archivo `web.php` tan limpio como podamos para que, de un vistazo, se entienda todo perfectamente.

recuerda

sólo movemos la lógica, mientras que las cláusulas como `where` y `name` las seguimos dejando en el archivo de rutas `web.php`.

7.1. convenciones

Laravel tiene una convención a la hora de nombrar los métodos de tus controllers conocida como Resource Controllers.

Esta convención ayuda bastante para tener todo mejor organizado:

verbo HTTP	URI	acción	ruta
GET	/clientes	index	clientes.index
POST	/clientes	store	clientes.store
DELETE	/clientes/{cliente}	destroy	clientes.destroy

En este enlace a la documentación de Laravel vemos las acciones que son controladas (o manejadas) por el controlador [enlace](#).

Veamos cómo quedaría un *refactor* del archivo de rutas utilizando un Controller como el que acabamos de crear.

Ahora nos quedaría de la siguiente manera:

```

1  <?php
2      // web.php (v2.0) Refactorizado
3      use App\Http\Controllers\PagesController;
4      use Illuminate\Support\Facades\Route;
5
6      Route::get('/', [ PagesController::class, 'inicio' ]);
7      Route::get('datos', [ PagesController::class, 'datos' ]);
8      Route::get('cliente/{id?}', [ PagesController::class, 'cliente' ]) ->
where('id', '[0-9]+' );
9      Route::get('nosotros/{nosotros?}', [ PagesController::class, 'nosotros' ]) ->
name('nosotros');
```

y en nuestro archivo controlador lo dejaríamos de la siguiente manera:

```

1  <?php
2      // PagesController.php
3      namespace App\Http\Controllers;
4
5      class PagesController extends Controller {
6          public function inicio() {
7              return view('welcome');
8          }
9
10         public function datos() {
11             return view('usuarios', ['id' => 56]);
12         }
13
14         public function cliente($id = 1) {
15             return ('Cliente con el id: ' . $id);
16         }
17
18         public function nosotros($nombre = null) {
19             $equipo = ['Paco',
20                     'Enrique',
21                     'Maria',
22                     'Veronica'];
23             return view('nosotros', @compact('equipo', 'nombre'));
24         }
25     }
```

Ejemplo: Podemos observar todo esto en el [ejemplo 07](#).

8. tipos de Request

8.1. En HTTP y API's

En HTTP existen diferentes tipos de Request o tipos de Petición: GET, POST, PUT, PATCH y DELETE.

- **GET** es el más simple; cuando visitas un sitio web por default es un GET, y el método solo se utiliza para recuperar datos pero nunca debe enviar datos.
- **POST** se utiliza cuando mandas datos a un servidor; esto incluye información que llenas en un formulario o buscador.
- **PUT** es utilizado para actualizar un elemento; pero si no existe crea uno nuevo; PUT es un reemplazo total de un registro.
- **PATCH** es utilizado para actualizar parcialmente un elemento o recurso.
- **DELETE** se utiliza para eliminar un recurso o elemento.

Ejemplo: Podemos observar todo esto en el [ejemplo 08](#).

9. validación de formularios

Para validar los campos de un formulario podemos utilizar las reglas de validación; éstas se colocarán en la función `store` del controlador.

Todas las reglas de validación de Laravel podemos observarlas en la documentación oficial, en este [enlace](#).

Ejemplo: Podemos observar todo esto en el [ejemplo 09](#).

9.1. mostrar errores de forma dinámica

Si utilizamos la directiva `@error` ... `@enderror` de esta forma:

```
1 @error('name')
2     <p class="bg-red-500 text-white my-2 rounded-lg text-sm p-2 text-center">
3         el nombre es obligatorio
4     </p>
5 @enderror
```

solo mostraría si el nombre es obligatorio; pero podemos tener más validaciones (en el mismo *input*) que chequear.

Este tipo de errores **con texto estático no es la mejor opción**. Laravel ya tiene una serie de mensajes para dicho caso. Deberemos imprimir un mensaje con el mensaje de error `$message`.

Ejemplo: Podemos observar todo esto en el [ejemplo 10](#).

9.2. mensajes en castellano

Los mensajes de errores están en inglés, ¿cómo podemos **mostrarlos en castellano**? Existen paquetes en Laravel en castellano, por ejemplo [MarcoGomesr/laravel-validation-en-espanol](#).

En el terminal, a nivel de tu proyecto, ejecuta la clonación:

```
1 git clone https://github.com/MarcoGomesr/laravel-validation-en-espanol.git
   resources/lang
```

Para terminar el cambio accede al fichero `config/app.php` y en la línea 86 (más o menos) cambiar el idioma a español:

```
1 'locale' => 'es',
```

Ejemplo: Poner nuestro proyecto en idioma español.

9.3. mantener el valor en el campo después de un error

Muchas veces es frustrante volver a un formulario después de un error y observar que los valores de todos los campos se han borrado y que necesitas volver a introducirlos. Para evitar este caso podemos poner en los `input` el atributo `value` y pasarle `"{{ old('username') }}"`.

Ejemplo: Podemos observar todo esto en el [ejemplo 11](#).

9.4. confirmar password en otro campo

Para confirmar el password notamos que habíamos puesto anteriormente al campo de confirmación de password esta etiqueta:

Una convención en Laravel para comprobar si dos campos son iguales es asignarle al segundo el prefijo `_confirmation`.

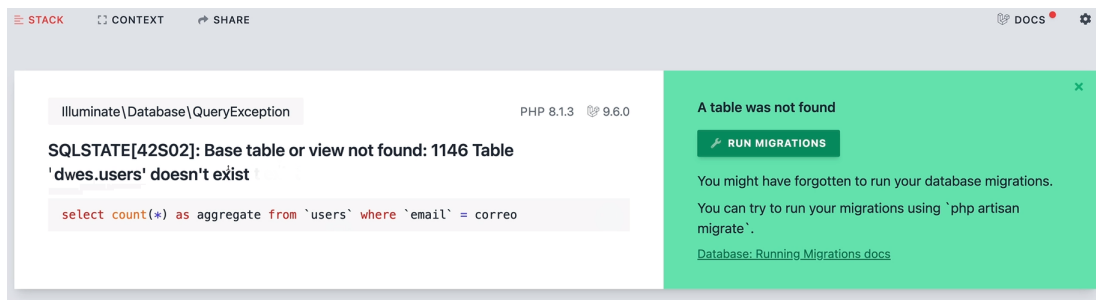
Pasos a seguir:

1. Poner de nombre del segundo *input* `password_confirmation` (y **debe nombrarse así**)
2. Poner la validación `confirmed` para el campo `password` en el controlador.

Ejemplo: Podemos observar todo esto en el [ejemplo 12](#).

10. migraciones y la base de datos

Si has seguido los ejemplos anteriores correctamente habrás comprobado que cuando ponemos un correo e intentamos acceder muestra el siguiente error:



Nos indica que está revisando el valor de `email` en la tabla de usuarios y esta no existe. Para completar el formulario habrá que crear dicha tabla `users`. Para esto debemos realizar nuestra primera **migración**.

10.1. qué son migraciones

Las Migraciones se les conoce como el control de versiones de tu base de datos; de esta forma se puede crear la base de datos y compartir el diseño con el equipo de trabajo.

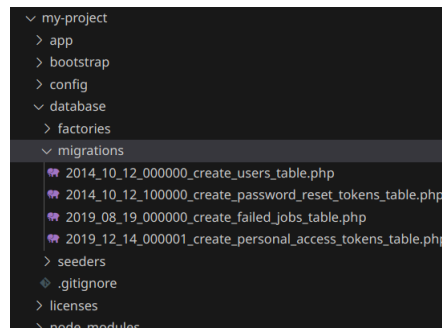
Si deseas agregar nuevas tablas o columnas a una tabla existente, puedes hacerlo con una nueva migración; si el resultado no fue el deseado, puedes revertir esa migración.

Lanzar desde la línea de comandos:

```
1 // ejecuta las migraciones
2 php artisan migrate
3 // ó, si no funciona:
4 // sudo docker-compose exec myapp php artisan migrate
```

```
1 // en caso de querer deshacer el cambio:
2 php artisan migrate:rollback
3 // ó, si no funciona:
4 // sudo docker-compose exec myapp php artisan migrate:rollback
5
6 // regresar las últimas 5 (por ejemplo) migraciones
7 php artisan migrate:rollback --step=5
8 // ó, si no funciona:
9 // sudo docker-compose exec myapp php artisan migrate:rollback --step=5
```

Las migraciones se van a ir colocando, siempre, en la carpeta del proyecto `database/migrations`:



Laravel tiene unas migraciones por defecto, sobre todo para la creación de usuarios.

Consideraciones previas

Recuerda tener el fichero `.env` configurado para acceder a tu bd en cuestión y con usuario y contraseña adecuados:

```
10
11 DB_CONNECTION=mysql
12 DB_HOST=127.0.0.1
13 DB_PORT=3306
14 DB_DATABASE=dwes
15 DB_USERNAME=dwes
16 DB_PASSWORD=dwes
```

Abrimos el terminal, dentro del proyecto:

```
1 php artisan migrate
2 // ó, si no funciona:
3 // sudo docker-compose exec myapp php artisan migrate
```

```
• abc@jolly-wright:~/DWES/proyectos/pru-udemy/my-project$ sudo docker-compose exec myapp php artisan migrate

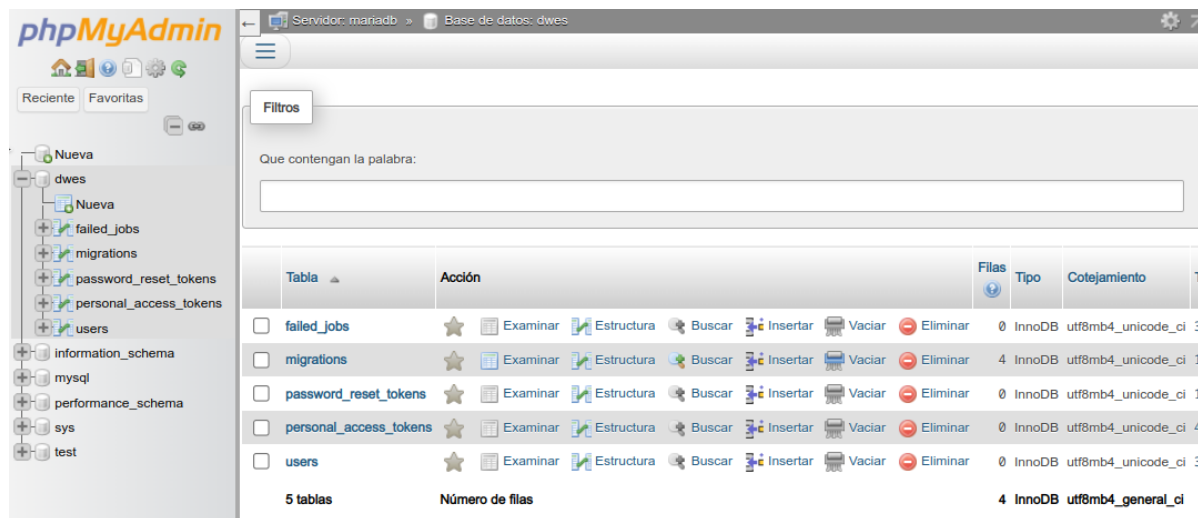
INFO Preparing database.

Creating migration table ..... 10ms DONE

INFO Running migrations.

2014_10_12_000000_create_users_table ..... 11ms DONE
2014_10_12_100000_create_password_reset_tokens_table ..... 6ms DONE
2019_08_19_000000_create_failed_jobs_table ..... 6ms DONE
2019_12_14_000001_create_personal_access_tokens_table ..... 10ms DONE
```

Si, después de ejecutar, accedemos a nuestra base de datos, por ejemplo desde *phpMyAdmin*:



10.1.1. rollback de la migración

Si, quisiéramos echar para atrás en la migración:

```
1 php artisan migrate:rollback
2 // ó, si no funciona:
3 // sudo docker-compose exec myapp php artisan migrate:rollback
```

```
INFO Rolling back migrations.

2019_12_14_000001_create_personal_access_tokens_table ..... 8ms DONE
2019_08_19_000000_create_failed_jobs_table ..... 2ms DONE
2014_10_12_100000_create_password_reset_tokens_table ..... 4ms DONE
2014_10_12_000000_create_users_table ..... 4ms DONE
```

Vemos que las tablas dejan de existir (solo queda la tabla migraciones) en nuestra bd:



Ejemplo: Podemos observar todo esto en el [ejemplo 13](#).

11. modelos

11.1. ORM Eloquent

Laravel incluye su propio **ORM** (Object Relacional Mapper) que hace muy sencillo interactuar con tu base de datos.

En **Eloquent** cada tabla tiene su propio **modelo**; ese modelo interactúa únicamente con esa tabla y tiene las funciones necesarias para crear registros, obtenerlos, actualizarlos y eliminarlos.

```
1 sudo docker-compose exec myapp php artisan make:model Cliente
```

Laravel tiene el modelo `users` creado por defecto.

11.2. convenciones en Laravel

11.2.1. en Modelos

Cuando creas el Modelo **Cliente**, Eloquent asume que la tabla se va a llamar **clientes**.

Si el Modelo se llama **Producto**; Eloquent espera una tabla llamada **productos**.

Problema

Puede ser un problema llamar tu modelo **Proveedor**, porque Eloquent espera la tabla llamada **provedors**, pero se puede reescribir en el modelo.

11.3. crear registros con Eloquent ORM

Para **insertar una fila** en nuestra tabla `users`, debemos insertar el siguiente código en nuestro controlador `RegisterController.php`:

```
1 //...
2 use Illuminate\Support\Facades\Hash
3 use App\Models\User // si no se encuentra se añade
4 //...
5 public function store(Request $request) {
6     //...
7     User::create([
8         'name' => $request->name,
9         'username' => $request->username,
10        'mail' => $request->email,
11        'password' => Hash::make($request->password)
12    ]);
13 }
14 //...
```



```

my-project > app > Http > Controllers > RegisterController.php > ...
1  <?php
2  |
3  namespace App\Http\Controllers;
4
5  use App\Models\User;
6  use Illuminate\Http\Request;
7  use Illuminate\Support\Facades\Hash;
8
9  class RegisterController extends Controller
10 {
11     public function index() {
12         return view('auth.register');
13     }
14
15     public function store(Request $request) {
16         //dd($request);
17         //dd($request->get('email'));
18
19         //validación
20         $this->validate($request, [
21             'name' => ['required', 'min:5'],
22             'username' => ['required', 'unique:users', 'min:3', 'max:20'],
23             'email' => ['required', 'unique:users', 'email', 'max:60'],
24             'password' => ['required', 'confirmed', 'min:6']
25         ]);
26
27         // dd('Creando usuario');
28
29         User::create([
30             'name' => $request->name,
31             'username' => $request->username,
32             'email' => $request->email,
33             'password' => Hash::make($request->password)
34         ]);
35     }
36 }
37

```

A tener en cuenta:

1. cuando introducimos `User` arriba en el código se va a importar `use App\Models\User;`
2. el método `create` corresponde a un `insert into ...`
3. podemos utilizar un *helper* (en Laravel encontramos una gran variedad) relacionado con los string; por ejemplo para que no introduzcamos espacios no deseados en el campo username.
4. vemos que, para *hashear* el password y que no se vea la cadena literal, podemos utilizar la clase `Hash::make(cadena)`. Si no importa directamente `use Illuminate\Support\Facades\Hash` le damos *botón derecho-import class*.

cuidado

Si no modificamos nada más, el campo `username` obtendrá un error. Esto es debido a que este campo lo hemos introducido nosotros después de la primera migración; y Laravel tiene un sistema de seguridad por el que no permite creaciones de campos tan fácilmente (así prevee posibles ataques de inserción de código en nuestra base de datos).

5. Para ello, además del código anterior, modificaremos el modelo `User.php` que se encuentra en la carpeta `app/Models` como medida de seguridad:

```


11 class User extends Authenticatable
12 {
13     use HasApiTokens, HasFactory, Notifiable;
14
15     /**
16      * The attributes that are mass assignable.
17      *
18      * @var array<int, string>
19      */
20     protected $fillable = [
21         'name',
22         'email',
23         'password',
24         'username'
25     ];

```

4. Probamos insertar un usuario en la app:

Regístrate en la APP

LOGIN CREAR CUENTA



NOMBRE

NOMBRE

EMAIL

PASSWORD

REPETIR PASSWORD

CREAR CUENTA

MIPRIMERAWEB - TODOS LOS DERECHOS RESERVADOS 2023

Y vemos que se ha insertado en la base de datos:

Examinar	Estructura	SQL	Buscar	Insertar	Exportar	Importar	Privilegios	Operaciones	Disparadores
Mostrando filas 0 - 1 (total de 2, La consulta tardó 0.0002 segundos.)									
SELECT * FROM `users`									
<input type="checkbox"/> Perfilando <input type="button" value="Editar en línea"/> <input type="button" value="Editar"/> <input type="button" value="Explicar SQL"/> <input type="button" value="Crear código PHP"/> <input type="button" value="Actualizar"/>									
<input type="checkbox"/> Mostrar todo Número de filas: 25 Filtrar filas: Buscar en esta tabla Ordenar según la clave: Ninguna									
Opciones extra									
id	name	email	email_verified_at	password	remember_token	created_at	updated_at	username	
2	prueba	prueba@pruebadw.es	NULL	\$2y\$12\$2MvUjCRC/G5dMxUD5uHjX17V01WxymQvT1o...	NULL	2023-12-07 11:36:40	2023-12-07 11:36:40	prueba	
<input type="checkbox"/> Seleccionar todo Para los elementos que están marcados: <input type="button" value="Editar"/> <input type="button" value="Copiar"/> <input type="button" value="Borrar"/> <input type="button" value="Exportar"/>									
<input type="checkbox"/> Mostrar todo Número de filas: 25 Filtrar filas: Buscar en esta tabla Ordenar según la clave: Ninguna									

11.3.1. cambiar el campo 'username' a único

1. Echar para atrás la última migración:

```

1 php artisan migrate:rollback
2 // ó, si no funciona:
3 // sudo docker-compose exec myapp php artisan migrate:rollback

```

INFO Rolling back migrations.

2023_12_07_105013_add_username_to_users_table 8ms **DONE**

2. Hacer cambios en el fichero `...add_username_to_users_table.php`:

```
1 //...
2 Schema::table('users', function(Blueprint $table){
3     $table->string('username')-> unique() -> after('name');
4 });
5 //...
```

```
return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::table('users', function (Blueprint $table) {
            $table->string('username')-> unique();
        });
    }
}
```

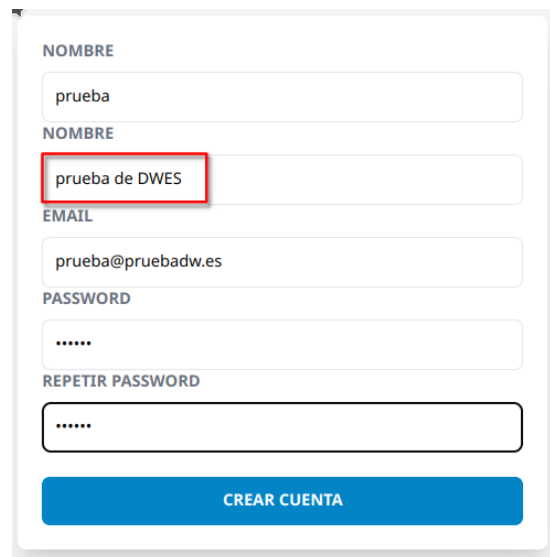
3. Volver a migrar:

```
1 php artisan migrate
2 // ó, si no funciona:
3 // sudo docker-compose exec myapp php artisan migrate
```

4. Ahora podemos hacer un cambio en `RegisterController.php`, que convierte la cadena a una URL (minúscula y los espacios los sustituye por un guión medio):

```
User::create([
    'name' => $request->name,
    'username' => Str::slug($request->username),
    'email' => $request->email,
    'password' => Hash::make($request->password)
]);
```

5. Insertamos un valor en `username` con mayúsculas y espacios:



	id	name	email	email_verified_at	password	remember_token	created_at	updated_at	username
	1	prueba	prueba@pruebadw.es	NULL	\$2y\$12\$[gwy]TIZBMGcJvuQIHq38eb4q9Wbww2G0fELKobBrMx...	NULL	2023-12-07 12:18:48	2023-12-07 12:18:48	prueba-de-dwes

6. Para que no aparezca un mensaje de error al introducir dos usuarios con el `username` iguales, lo que podemos hacer es modificar el Request (cuando es nuestra última opción):

```
// Modificar el Request:
$request->add(['username' => Str::slug($request->username)]);

//validación
$this->validate($request, [
    'name' => ['required', 'min:5'],
    'username' => ['required', 'unique:users', 'min:3', 'max:20'],
    'email' => ['required', 'unique:users', 'email', 'max:60'],
    'password' => ['required', 'confirmed', 'min:6']
]);

// dd('Creando usuario');

User::create([
    'name' => $request->name,
    'username' => $request->username,
    'email' => $request->email,
    'password' => Hash::make($request->password)
]);
```

11.4. redireccionar al usuario al Muro una vez la cuenta es creada

1. Crear un controlador de nombre `PostController`:

```
1 | sudo docker-compose exec myapp php artisan make:controller PostController
```

2. Crear un controlador de nombre `LoginController`:

```
1 | sudo docker-compose exec myapp php artisan make:controller LoginController
```

```
web.php x register.blade.php RegisterController.php PostController.php 2014_10_12_0
my-project > routes > web.php > ...
15 |
16 | */
17 |
18 | // mediante closure
19 | Route::get('/', function () {
20 |     return view('inicio');
21 | });
22 |
23 | // mediante controlador
24 | Route::get('/register', [RegisterController::class, 'index']) -> name('register');
25 | Route::post('/register', [RegisterController::class, 'store']) -> name('register');
26 |
27 |
28 | Route::get('/muro', [PostController::class, 'index'])->name('posts.index');
```

```
my-project > app > Http > Controllers > PostController.php > PostController > index
1 | <?php
2 |
3 | namespace App\Http\Controllers;
4 |
5 | use Illuminate\Http\Request;
6 |
7 | class PostController extends Controller
8 | {
9 |     //
10 |     public function index() {
11 |         dd('Muro');
12 |     }
13 | }
14 |
```

```

web.php  register.blade.php  RegisterController.php x  PostController.php  20
my-project > app > Http > Controllers > RegisterController.php > RegisterController > store
25     $this->validate($request, [
26         'name' => ['required', 'min:5'],
27         'username' => ['required', 'unique:users', 'min:3', 'max:20'],
28         'email' => ['required', 'unique:users', 'email', 'max:60'],
29         'password' => ['required', 'confirmed', 'min:6']
30     ]);
31
32     // dd('Creando usuario');
33
34     User::create([
35         'name' => $request->name,
36         'username' => $request->username,
37         'email' => $request->email,
38         'password' => Hash::make($request->password)
39     ]);
40
41     // Redireccionar con helper redirect
42     return redirect()->route('posts.index');
43
44 }

```

11.5. autenticar un usuario que ha creado su cuenta

```

web.php  register.blade.php  RegisterController.php  PostController.php x
my-project > app > Http > Controllers > PostController.php > ...
1  <?php
2
3  namespace App\Http\Controllers;
4
5  use Illuminate\Http\Request;
6  use Illuminate\Support\Facades\Auth;
7
8  class PostController extends Controller
9  {
10     //
11     public function index() {
12         dd(Auth()->user());
13     }
14 }

```

```

web.php  register.blade.php  RegisterController.php x  PostController.php
my-project > app > Http > Controllers > RegisterController.php > RegisterController > store
32     // dd('Creando usuario');
33
34     User::create([
35         'name' => $request->name,
36         'username' => $request->username,
37         'email' => $request->email,
38         'password' => Hash::make($request->password)
39     ]);
40
41     // autenticar un usuario
42     // auth()->attempt([
43     //     'email' => $request->email,
44     //     'password' => $request->password
45     // ]);
46
47     // otra forma de autenticar
48     auth()->attempt($request->only('email', 'password'));
49
50     // Redireccionar con helper redirect
51     return redirect()->route('posts.index');
52
53 }

```

12. anexo I - instalación de Tailwind CSS

Si hemos decidido instalar `Tailwind CSS` para que nos eche una mano con nuestro css, deberemos de seguir estos pasos:

1. Hay que comprobar la versión de npm y node:

```
1 | npm -v
2 | node -v
```

2. Si la versión de nodejs es inferior a la versión 14 (a la hora de crear este documento) **antes de seguir habremos de reinstalarlo**. Para ello habrá que ir al [anexo II - reinstalación de node](#) en este mismo documento.

no continuar si la versión de node es inferior a 14.

3. Si nuestra versión de nodejs es correcta (o hemos procedido a reinstalar node en el punto 2), lo primero será desinstalar bootstrap:

```
1 | npm uninstall bootstrap
```

4. Instalaremos en desarrollo estas tres dependencias:

```
1 | npm install -D tailwindcss postcss autoprefixer
```

5. Generamos ahora el fichero `tailwindcss.config.js` que aparecerá en la raíz del proyecto:

```
1 | npx tailwindcss init -p
```

6. Editar el fichero del proyecto Laravel `tailwindcss.config.js` que se ha generado en el directorio raíz del proyecto y donde indicaremos dónde vamos a utilizarlo:

```
1 | /** @type {import('tailwindcss').Config} */
2 | export default {
3 |   content: [
4 |     "./resources/**/*.blade.php",
5 |     "./resources/**/*.js",
6 |     "./resources/**/*.vue",
7 |   ],
8 |   theme: {
9 |     extend: {},
10 |   },
11 |   plugins: [],
12 | }
```

7. Ahora, en el fichero `/resources/css/app.css` agregar las siguientes líneas:

```
1 @tailwind base;
2 @tailwind components;
3 @tailwind utilities;
```

8. Desde el terminal (y siempre dentro de nuestro proyecto), vamos a ejecutar:

```
1 npm run dev
2 # ó, si no funciona:
3 # npm run dev -- --host
```

```
abc@abc-pc:~/Escritorio/IES/DWES/proyectos/my-project$ npm run dev
> dev
> vite
VITE v4.5.0 ready in 470 ms
→ Local: http://localhost:5173/
→ Network: use --host to expose
→ press h to show help
LARAVEL v10.34.2 plugin v0.8.1
→ APP_URL: http://localhost
```

9. En el fichero `/resources/views/layouts/app.blade.php` hay que indicarle que va a utilizar el fichero `/resources/css/app.css`, para ello hay que añadirlo en:

```
1 @vite('resources/css/app.css')
```

```
resources > views > layouts > app.blade.php
1 <!DOCTYPE html>
2 <html lang="{{ str_replace('_', '-', app()->getLocale()) }}">
3 <head>
4 <meta charset="utf-8">
5 <meta name="viewport" content="width=device-width, initial-scale=1">
6
7 <title>Mi proyecto - @yield('titulo')</title>
8
9 @vite(resources/css/app.css)
10
11 </head>
12 <body>
13 <h1>@yield('titulo')</h1>
14 <hr>
15 @yield('contenido')
16 </body>
17 </html>
18
```

A partir de ahora, y con este ejemplo, podemos observar que se nos muestra el css:

```
<body>
  <nav>
    <a href="/">Principal</a>
    <a href="/nosotros">Nosotros</a>
    <a href="/tienda">Tienda virtual</a>
  </nav>

  <h1 class="">@yield('titulo')</h1>
  <hr>
  @yield('contenido')
</body>
</html>
```

{} first-letter: &::first-letter
 {} first-line:
 {} marker:
 {} selection:
 {} file:
 {} placeholder:
 {} backdrop:
 {} before:
 {} after:
 {} first:
 {} last:
 {} only:

13. anexo II - reinstalación de node

Para reinstalar nodejs:

1. Desinstalar node por completo:

```
1 | sudo apt-get purge --auto-remove nodejs
```

2. Eliminar todo resto de node y npm:

```
1 | a. Antes que nada, debe ejecutar el siguiente comando desde el terminal:
```

```
1 | sudo rm -rf /usr/local/bin/npm /usr/local/share/man/man1/node*
   /usr/local/lib/dtrace/node.d ~/.npm ~/.node-gyp /opt/local/bin/node
   opt/local/include/node /opt/local/lib/node_modules
```

```
1 | b. Eliminar los directorios node o node_modules de /usr/local/lib con la ayuda del siguiente comando:
```

```
1 | sudo rm -rf /usr/local/lib/node*
```

```
1 | c. Eliminar los directorios node o node_modules de /usr/local/include con la ayuda del siguiente comando:
```

```
1 | sudo rm -rf /usr/local/include/node*
```

```
1 | d. Eliminar cualquier archivo de nodo o directorio de /usr/local/bin con la ayuda del siguiente comando:
```

```
1 | sudo rm -rf /usr/local/bin/node
```

3. Instalar otra vez nvm:

```
1 | a. Instalar NVM (Node Version Manager), desde el directorio de usuario ~` :
```

```
1 | curl -o- https://raw.githubusercontent.com/nvm-
   sh/nvm/v0.39.0/install.sh | bash
```

```
1 | b. Actualiza el archivo .bashrc:
```

```
1 | source .bashrc
```

```
1 | c. Confirma que el directorio local está configurado:
```



```
1 | echo $NVM_DIR
2 | /home/username/.nvm
```

4. Instalar node:

```
1 | a. Revisar qué versiones de Node.js están disponibles:
```

```
1 | nvm ls-remote
```

```
1 | b. Instalar la versión que desees (elige la v20.10.0):
```

```
1 | nvm install v20.10.0
```

5. Comprobar que la nueva versión de node es superior a 14:

```
1 | node -v
```

14. ejemplos

14.1. ejemplo 01. Hola Mundo

Vamos a eliminar todo el `style` que viene por defecto y a vaciar de contenido de la etiqueta `<body>` de la vista **resources/views/welcome.blade.php** y creamos etiquetas:

```
1 <h1>Página principal</h1>
2 <h2>Hola Mundo.</h2>
```

[\[volver ^\]](#)

14.2. ejemplo 02. Otras vistas

Creamos dos ficheros `nosotros.blade.php`, `tienda.blade.php` en **resources/views**:

Además, añadimos en el fichero `web.php` las siguientes rutas:

```
1 Route::get('/nosotros', function () {
2     return view('nosotros');
3 });
4
5 Route::view('/tienda-virtual', 'tienda'); // forma corta para la ruta a una
6     vista
```

[\[volver ^\]](#)

14.3. ejemplo 03. Paso de parámetros

En el fichero `web.php` modifica la ruta a tienda-virtual para que se le pase por parámetro la variable `$id`. Préviamente se ha inicializado a un valor; controlar si se ha introducido sin valor y si se ha introducido algún valor no numérico:

```
1 Route::get('tienda/{id?}', function($id = 1) {
2     return view('tienda', ['id' => $id]);
3 }) -> where('id', '[0-9]+');
4
5 // otra forma:
6 // Route::view('tienda/{id?}', 'tienda', ['id' => 1]);
```

Después, mostrar este valor en la vista `tienda.blade.php`:

```
1 <h1>tienda virtual</h1>
2 <p>producto con id: <?= $id ?></p>
```

[\[volver ^\]](#)

14.4. ejemplo 04. Uso de directivas

Cómo hacer uso del poder de Laravel para crear plantillas y no repetir código.

Supongamos que tenemos ciertas estructuras HTML repetidas como puede ser una cabecera *header*, un menú de navegación *nav* y un par de secciones que hacen uso de este mismo código.

Supongamos que tenemos 3 apartados en la web: *Inicio* | *Blog* | *Fotos*

1. Primero de todo tendremos que generar un archivo que haga de plantilla de nuestro sitio web.

Para ello creamos el archivo `app.blade.php` dentro del nuevo directorio de plantillas `resources/views/layouts`.

Dicho archivo va a contener el típico código de una página simple de HTML y en el body añadiremos nuestros contenido estático y dinámico.

```

1  <!DOCTYPE html>
2  <html lang="{{ str_replace('_', '-', app()->getLocale()) }}">
3  <head>
4      <meta charset="utf-8">
5      <meta name="viewport" content="width=device-width, initial-
scale=1">
6
7      <title>Mi proyecto - @yield('titulo')</title>
8
9      <!-- introducir la siguiente linea para poder utilizar TailwindCSS
-->
10     @vite('resources/css/app.css')
11
12 </head>
13
14 <body class="bg-gray-100">
15     <header class="p-5 border-b bg-white shadow">
16         <div class="container mx-auto flex justify-between items-center">
17             <h1 class="text-3xl font-black">
18                 @yield('titulo')
19             </h1>
20
21             <nav class="flex gap-5 items-center">
22                 <a class="font-bold uppercase text-gray-600 text-sm"
href="#">Login</a>
23                 <a class="font-bold uppercase text-gray-600 text-sm"
href="#">Crear cuenta</a>
24             </nav>
25         </div>
26     </header>
27
28     <!-- BORRAR MÁS ADELANTE este menú en el siguiente ejemplo -->
29     <nav class="flex gap-5 items-center">
30         <a class="font-bold uppercase text-gray-600 text-sm"
href={{ route('inicio') }} >inicio</a> |
31         <a class="font-bold uppercase text-gray-600 text-sm"
href={{ route('noticias') }} >blogs</a> |
32         <a class="font-bold uppercase text-gray-600 text-sm"
href={{ route('galeria') }} >fotos</a>
33     </nav>
34
35 
```

```

36 </nav>
37
38 <!-- CONTENIDO PRINCIPAL -->
39 <main class="container mx-auto mt-10">
40     <h2 class="font-black text-center text-3xl mb-10">
41         @yield('titulo')
42     </h2>
43     @yield('contenido')
44 </main>
45
46 <!-- FOOTER -->
47 <footer class="text-center p-5 text-gray-500 font-bold uppercase">
48     MiPrimeraWeb - Todos los derechos reservados @php echo date('Y')
49 @endphp
49     <br>
50 <!-- con helpers -->
51     MiPrimeraWeb - Todos los derechos reservados {{ now()->year }}
52 </footer>
53 </body>
54 </html>

```

Cada sección que haga uso de esta plantilla contendrá un menú de navegación con enlaces a cada una de las secciones y el contenido dinámico de cada sección.

A tener en cuenta:

- `{{ route('inicio') }}` muestra:
 - `{{ ... }}` es una etiqueta para Blade que muestra un echo.
 - `route(...)` lanza la ruta de nombre (o alias) concreto.
- `@php echo date('Y') @endphp` son etiquetas para Blade para contener código php. En este caso un echo que muestra el año `'Y'` de la fecha actual `date()`.
- `now()` es un helper de Laravel (componente del framework diseñado para facilitar alguna tarea típica en el desarrollo de una aplicación web). Tiene distintos atributos y métodos, entre ellos, `year` para mostrar el año.

2. Ahora crearemos los archivos dinámicos de cada una de las secciones, en nuestro caso:

`inicio.blade.php`:

Importamos el contenido de plantilla bajo la directiva `@extends` para que cargue los elementos estáticos que hemos declarado y con la directiva `@section` y `@endsection` definimos el bloque de código dinámico, en función de la sección que estemos visitando.

```

1 @extends('layouts.app')
2
3 @section('titulo')
4     página principal
5 @endsection
6
7 @section('contenido')
8     contenido de la página principal
9 @endsection

```

`blog.blade.php`:

```
1 @extends('layouts.app')
2
3 @section('titulo')
4     noticias
5 @endsection
6
7 @section('contenido')
8     contenido de todas las noticias
9 @endsection
```

`fotos.blade.php`:

```
1 @extends('layouts.app')
2
3 @section('titulo')
4     fotografías
5 @endsection
6
7 @section('contenido')
8     galería de fotografías
9 @endsection
```

3. El último paso que nos queda es configurar el archivo de rutas `routes/web.php`:

```
1 <?php
2 // web.php
3 Route::view('', 'inicio') -> name('inicio');
4 Route::view('blog', 'blog') -> name('noticias');
5 Route::view('fotos', 'fotos') -> name('galeria');
```

De esta manera podremos hacer uso del menú de navegación que hemos puesto en nuestra plantilla y gracias a los alias noticias y galeria, la URL será más amigable.

[\[volver ^\]](#)

14.5. ejemplo 05. Vista registrarse

Antes de continuar con el ejemplo, debes **modificar (guarda los modificados)**:

- elimina el menú de navegación de *inicio* | *blogs* | *fotos* del fichero `layouts/app.blade.php`.
- elimina las rutas en el fichero `web.php` que afecten a este menú (exceptuando la ruta de inicio).
- elimina las dos vistas `blog.blade.php` y `foto.blade.php`.

Este borrado se debe a que NO vamos a continuar con estas vistas; solo eran un ejemplo de uso de directivas y la separación de código.

Como has observado, nuestro anterior ejemplo contenía en el fichero `resources/views/layouts/app.php` un menú de navegación en el que se indicaba *Login* y *Crear cuenta*.

Crea las vistas para estos dos enlaces. Para ello alojarás sus dos vistas en una carpeta nueva `resources/views/auth` con nombre `register.blade.php` para *Crear cuenta*.

```
1 @extends('layouts.app')
2
3 @section('titulo')
4   Regístrate en tu APP
5 @endsection
6
7 @section('contenido')
8
9 @endsection
```

Modificar en el fichero `app.blade.php` el menú de crear cuenta para que se pueda acceder desde el enlace a la vista creada anteriormente:

```
1 // ...
2 <nav class="flex gap-5 items-center">
3   <a class="font-bold uppercase text-gray-600 text-sm" href="#">Login</a>
4   <a class="font-bold uppercase text-gray-600 text-sm" href="{{
5     route('register') }}">Crear cuenta</a>
6 </nav>
7 // ..
```

Añadir al fichero `web.php` la entrada:

```
1 Route::view('/crear-cuenta', 'auth.register') -> name('register'); //la ruta
   contiene .
```

[\[volver ^\]](#)

14.6. ejemplo 06. Pasar datos a una vista

1. Crear en `web.php` un array (de nombre `arrayProductos`) con, al menos, 4 ítems:

```
1 Route::get('listaproductos', function(){
2   $productos = ['rosa', 'clavel', 'orquídea', 'lirio'];
3   return view('pru1', compact('productos'));
4 });
```

Para pasar un array a una vista existen varios métodos:

- `return view('pru1', ['productos' => $productos]);`
- `return view('pru1') -> with ('productos', $productos);`
- **`return view('pru1', compact('productos'));`**

con la función de php `compact` se crea internamente un array con productos.

2. Pasar este array a una vista, de nombre `listaproductos.blade.php` y muéstralos en una tabla:

```

1 <?php
2     foreach ($productos as $item) {
3         echo "<li>".$item."</li>";
4     }
5 ?>

```

Pero como hemos visto, Laravel tiene unas directivas que funcionan en sus vistas y que se pueden utilizar de forma más elegante:

```

1 @foreach ($usuarios as $item)
2     <li>{{ $item }}</li>
3 @endforeach

```

[\[volver ^\]](#)

14.7. ejemplo 07. Controlador RegisterController y su formulario

En vez de usar *closures* (o *callbacks*) en el fichero `web.php` vamos a crear controladores en nuestro ejemplo y trasladar la lógica de negocio a estos últimos. Así, y siguiendo con los ejemplos anteriores, vamos a crear un controlador para gestionar el registro en nuestra APP (vamos a ordenar nuestros ficheros y colocaremos `RegisterController` dentro de la carpeta `Auth` y añadimos doble diagonal inversa `\\`).

Abrimos el CLI artisan (nuevo terminal en VS Code o `Ctrl+``) en la carpeta de nuestro proyecto y ejecutamos:

```

1 php artisan make:controller Auth\\RegisterController
2 # ó, si no funciona:
3 # sudo docker-compose exec myapp php artisan make:controller
  Auth\\RegisterController

```

En `web.php` modificar la ruta de *Crear cuenta* (quitar el *callback* y añadir el controlador y su método). Se recomienda como convención el nombre del método `index`:

```

1 // ...
2 use App\Http\Controllers\Auth\RegisterController;
3 // ...
4 Route::get('/register', [RegisterController::class, 'index']) ->
  name('register');

```

Y en `RegisterController.php`, trasladamos la lógica de negocio que ejecutaba el *callback*, añadiendo la palabra reservada `public` y añadiendo también un nombre a esta función `index()`:

```

1 public function index() {
2     return view('auth.register');
3 }

```

Para la vista de register `register.blade.php` vamos a introducir el código:

```

1  @extends('layouts.app')
2
3  @section('titulo')
4      Regístrate en la APP
5  @endsection
6
7  @section('contenido')
8      <div class="md:flex md:justify-center md:gap-10 md:items-center">
9          <div class="md:w-6/12 p-5">
10             
12         </div>
13
14         <div class="md:w-4/12 bg-white p-6 rounded-lg shadow-xl">
15             <!-- todavía no existe ni método ni enlace en el form -->
16             <form action="">
17                 <div>
18                     <label for="name" class="mb-2 block uppercase text-gray-500 font-
19                     bold">
20                         Nombre
21                     </label>
22                     <input
23                         id="name"
24                         name="name"
25                         type="text"
26                         placeholder="tu nombre"
27                         class="border p-3 w-full rounded-lg"
28                     />
29                 </div>
30
31                 <div>
32                     <label for="username" class="mb-2 block uppercase text-gray-500
33                     font-bold">
34                         Nombre
35                     </label>
36                     <input
37                         id="username"
38                         name="username"
39                         type="text"
40                         placeholder="tu nombre de usuario"
41                         class="border p-3 w-full rounded-lg"
42                     />
43                 </div>
44
45                 <div>
46                     <label for="email" class="mb-2 block uppercase text-gray-500 font-
47                     bold">
48                         Email
49                     </label>
50                     <input
51                         id="email"
52                         name="email"
53                         type="text"
54                         placeholder="tu email de registro"
55                         class="border p-3 w-full rounded-lg"
56                     />
57                 </div>
58             </form>
59         </div>
60     </div>

```



```

53         </div>
54
55         <div>
56             <label for="password" class="mb-2 block uppercase text-gray-500
font-bold">
57                 Password
58             </label>
59             <input
60                 id="password"
61                 name="password"
62                 type="password"
63                 placeholder="tu contraseña de registro"
64                 class="border p-3 w-full rounded-lg"
65             />
66         </div>
67
68         <div>
69             <label for="password_confirmation" class="mb-2 block uppercase
text-gray-500 font-bold">
70                 Repetir password
71             </label>
72             <!-- password_confirmation para validar posteriormente password -->
73             <input
74                 id="password_confirmation"
75                 name="password_confirmation"
76                 type="password"
77                 placeholder="repite la contraseña"
78                 class="border p-3 w-full rounded-lg"
79             />
80         </div>
81
82         <br>
83         <input
84             type="submit"
85             value="Crear cuenta"
86             class="bg-sky-600 hover:bg-sky-700 transition-colors
cursor-pointer uppercase font-bold w-full p-3 text-white rounded-
87             lg"
88         />
89     </form>
90 </div>
91 @endsection

```

A tener en cuenta:

- En `{{asset('images/dwes_registrar.png')}}:`
 - `{{ }}` hace referencia a un echo.
 - `asset()` hace referencia a la ruta `app/public` de Laravel.
- El `input password_confirmation` del formulario tiene este nombre de forma 'obligatoria'; es decir, el prefijo `_confirmation` hará que Laravel, internamente, compruebe que el input de nombre `password` y el input de nombre `password_confirmation` contengan la misma cadena.

[\[volver ^\]](#)

14.8. ejemplo 08. Petición post

Vamos a crear ahora el enlace registrarse. Para ello accedemos a `web.php` e introducimos la línea con el método POST:

```
1 Route::get('/register', [RegisterController::class, 'index']) ->
  name('register');
2 Route::post('/register', [RegisterController::class, 'store']);
```

A tener en cuenta

- No ponemos un alias en la segunda ruta porque va a tomar también el alias del anterior ruta.

Al mismo tiempo, en nuestro controlador `RegisterController.php` agregamos la función `store` :

```
1 public function store() {
2     dd('formulario...');
3 }
```

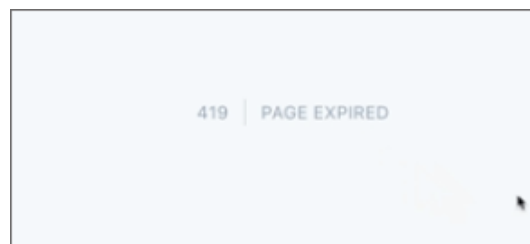
La función `dd` (*dump or die*) en Laravel permite parar la ejecución en ese punto. Es decir, parecida a la función `dump()` pero finaliza la ejecución del script.

Para terminar este punto, en la vista `register.blade.php` vamos a modificar la etiqueta `form` para que redirija al action correspondiente con el método en cuestión. Fíjate que en action ponemos la función `route` y el nombre de la ruta:

```
1 ...
2 <form action="{{ route('register') }}" method="POST">
3 ...
```

14.8.1. problema a la "vista"!!

Si recargamos la página y clicamos en el enlace vemos que nos muestra el siguiente error:



¿Por qué una página expirada?

Laravel es un framework enfocado a la seguridad (en este caso, se asegura que no suframos ataques del tipo XSRF o Cross Site Request Forgery). Así que Laravel tiene consideraciones de seguridad.

Para evitar estos ataques usaremos la directiva `@csrf` justo después de la línea de la etiqueta `<form>`.

```

1 ...
2 <form action="{{ route('register') }}" method="POST">
3     @csrf
4 ...

```

Si pulsamos *F12* para ver el código en el navegador se mostrará un campo oculto con un *token* oculto para validar la cadena y evitar este tipo de ataques:

```

<div class="md:w-4/12 bg-white p-6 rounded-lg shadow-xl">
    <form action="/crear-cuenta" method="POST">
        <input type="hidden" name="_token" value="IArNWYpdGJLCoLw53eD2CGaEhfe8QriFRul3nOng">
        <label for="name" class="mb-2 block uppercase text-gray-500 font-bold">
            Nombre
        </label>
    </form>
</div>

```

Modificamos la función `store` para pasarle el objeto `$request`:

```

1 public function store(Request $request) {
2     dd(request);
3     // dd(request->get('email'));
4 }

```

Si accedemos a la ruta `localhost/register` se observa la información del array `$request`:

```

Illuminate\Http\Request {#37 ▾ // app/Http/Controllers/RegisterController.php:18
  +attributes: Symfony\Component\HttpFoundation\ParameterBag {#42 ▸}
  +request: Symfony\Component\HttpFoundation\InputBag {#38 ▾
    #parameters: array:6 [▾
      "_token" => "LThnoGG85KAmg3Pwgm80Z2Ee21XkQSaHC1LtD3DB"
      "name" => "arturo"
      "username" => "arturo76"
      "email" => null
      "password" => null
      "password_confirmation" => null
    ]
  }
  +query: Symfony\Component\HttpFoundation\InputBag {#45 ▸}
  +server: Symfony\Component\HttpFoundation\ServerBag {#40 ▸}
  +files: Symfony\Component\HttpFoundation\FileBag {#44 ▸}
  +cookies: Symfony\Component\HttpFoundation\InputBag {#43 ▸}
  +headers: Symfony\Component\HttpFoundation\HeaderBag {#39 ▸}
  #content: null
  #languages: null
  #Charsets: null
  #encodings: null
  #acceptableContentTypes: null
  #pathInfo: "/register"
  #requestUri: "/register"
}

```

[\[volver ^\]](#)

14.9. ejemplo 09. Validación de campos

Para validar los campos del formulario del controlador `RegisterController.php` podemos utilizar las reglas de validación siguientes; éstas se colocarán en la función `store` del controlador:

```

1 public function store(Request $request) {
2     //dd($request);
3     //dd($request->get('email'));
4
5     //validación
6     $this->validate($request, [
7         'name' => ['required', 'min:5'],
8         'username' => ['required', 'unique:users', 'min:3', 'max:20'],
9         'email' => ['required', 'unique:users', 'email', 'max:60'],
10        'password' => ['required', 'confirmed', 'min:6']
11    ]);

```

por qué en `unique` se refiere a una tabla `users` que todavía no la hemos creado?

Más adelante lo veremos, pero se puede observar en la carpeta `database/migrations` que tenemos una migración de la tabla `users`. Laravel crea automáticamente estas tablas.

Podemos observar que, en apariencia, no hace nada (o no muestra nada). Es decir, para mostrar un mensaje de error cuando no se cumpla una validación colocaremos la directiva `@error ... @enderror` en nuestro formulario (en el ejemplo `register.blade.php`) justo después del `<input>` en cuestión.

Siguiendo en el ejemplo:

```

1 <input
2     id="name"
3     name="name"
4     type="text"
5     placeholder="tu nombre"
6     class="border p-3 w-full rounded-lg"
7 >
8 @error('name')
9     <p class="bg-red-500 text-white my-2 rounded-lg text-sm p-2 text-center">
10         el nombre es obligatorio
11     </p>
12 @enderror

```

[\[volver ^\]](#)

14.10. ejemplo 10. Mensajes de errores dinámicos

Como se ha comentado, la mejor opción de mostrar un mensaje de error en Laravel es de forma dinámica mediante `$message`:

En el ejemplo anterior, modificaríamos el contenido de la directiva `@error`.

```

1 @error('username')
2     <p class="bg-red-500 text-white my-2 rounded-lg text-sm p-2 text-center">
3         {{ $message }}
4     </p>
5 @enderror

```

A tener en cuenta:

- El párrafo se ha modificado para tener texto blanco sobre fondo rojo.

- Si, además, queremos pintar de color rojo el campo en el que aparece el error, podemos poner en la clase del `input` un `@error` que diga que si existe un error en (por ejemplo) el campo `username` coloree el borde del campo en rojo:

```

1 <input
2     id="username"
3     name="username"
4     type="text"
5     placeholder="tu nombre de usuario"
6     class="border p-3 w-full rounded-lg
7         @error('username') border-red-500 @enderror"
8 >

```

[\[volver ^\]](#)

14.11. ejemplo 11. Mantener valor después de un error

Para 'guardar' del valor de un campo si volvemos al formulario en el atributo `value` del `input` colocaremos `{{ old('username') }}`. En el ejemplo de `username`:

```

1 <input
2     id="username"
3     name="username"
4     type="text"
5     placeholder="tu nombre de usuario"
6     class="border p-3 w-full rounded-lg
7         @error('username') border-red-500 @enderror"
8     value= "{{ old('username') }}"
9 >

```

Entre las validaciones de Laravel también podrá verse la validación de HTML5. Si quieres deshabilitar esta última puedes introducir el atributo `novalidate` en la etiqueta `<form ... novalidate>`.

A partir de ahora, como tarea, puedes validar tú mismo todos los campos del formulario.

[\[volver ^\]](#)

14.12. ejemplo 12. Confirmar valor del password

En el ejemplo, poner en el Blade que lleva el formulario:

```

1 <div>
2     <label for="password_confirmation" class="mb-2 block uppercase text-gray-
500 font-bold">
3         Repetir password
4     </label>
5     <input
6         id="password_confirmation"
7         name="password_confirmation"
8         type="password"
9         placeholder="repite la contraseña"
10        class="border p-3 w-full rounded-lg"
11    />
12 </div>

```

Después, en el controlador en cuestión (`RegisterController.php`), poner la validación `confirmed` (ya se había introducido en anteriores ejemplos):

```

1 //...
2     $this->validate($request, [
3         'name' => ['required', 'min:5'],
4         'username' => ['required', 'unique:users', 'min:3', 'max:20'],
5         'email' => ['required', 'unique:users', 'email', 'max:60'],
6         'password' => ['required', 'confirmed', 'min:6']
7     ]);
8 //...

```

[\[volver ^\]](#)

14.13. ejemplo 13. Crear usuario en la app

Como vemos, si intentamos crear un usuario en nuestro ejemplo de inserción de usuarios, se obtiene un error en el que nos indica que falta el campo `username` en la tabla `users`. Esto es debido a que, cuando se ha ejecutado, por primera vez, la migración este campo no existía. Para que la app funcione deberemos de migrar este campo.

Ejecutamos (el nombre lleva una convención de Laravel):

```

1 php artisan make:migrationadd_username_to_users_table
2 # ó, si no funciona:
3 # sudo docker-compose exec myapp php artisan make:migration
  add_username_to_users_table

```

Si accedemos al fichero generado en la carpeta `migrations` insertaremos el código que se muestra a continuación:

```

1 //...
2 Schema::table('users', function(Blueprint $table){
3     $table->string('username')-> unique() -> after('name');
4 });
5 //...

```

```

my-project > database > migrations > 2023_12_07_105013_add_username_to_users_table.php > class > up
1  <?php
2
3  use Illuminate\Database\Migrations\Migration;
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::table('users', function (Blueprint $table) {
15             $table->string('username');
16         });
17     }
18
19     /**
20      * Reverse the migrations.
21      */
22     public function down(): void
23     {
24         Schema::table('users', function (Blueprint $table) {
25             $table->dropColumn('username');
26         });
27     }
28 };

```

Para que los cambios surjan efecto, volvemos a ejecutar `migrate`:

```

1  php artisan migrate
2  # ó, si no funciona:
3  # sudo docker-compose exec myapp php artisan migrate

```

Aunque se indique el campo `username` como *string* se creará en la base de datos como *varchar*.

Siguiendo el ejemplo anterior ahora no nos dará error la inserción en el formulario.

[\[volver ^\]](#)

15. ejercicios propuestos

15.1. Ejercicio 1

Sobre el proyecto de la sesión anterior, vamos a crear una página (vista) en la que se muestre el contenido de una tabla. a añadir un formulario (parecido) al que hemos estado creando (registro) pero que muestre (por ejemplo) un producto (como por ejemplo libros):

15.1.1. crear controlador y modelo

Crear un **controlador** para la tabla `libros`. Para esto, recuerda:

1. crear un controlador:

```
1 | sudo docker-compose exec myapp php artisan make:controller
   LibroController --model=Libro
```

Este comando generará, a la vez, un controlador y su modelo relacionado. `database/migrations`.

Esto creará el controlador en la carpeta `App\Http\Controllers` y el modelo en la carpeta `App\Models`.

2. Si queremos mostrar todas las filas de la tabla `libros` creamos el siguiente método en el **controlador** anteriormente generado:

```
1 | public function mostrarTodos()
2 | {
3 |     // Obtener todas las filas de la tabla "libro"
4 |     $libros = Libro::all();
5 |
6 |     // Pasar los datos a la vista
7 |     return view('libros.mostrar', ['libros' => $libros]);
8 | }
```

En dicho método se aprecia que guarda en la variable `$libros` todos los libros de la tabla libros. A continuación, lanza una vista de nombre `mostrar.blade.php` almacenada en la carpeta `resources/views/libros` en la que pasa por parámetro la variable `$libros` anteriormente creada.

Por otro lado, en el **modelo** debemos insertar en la clase `Libro`:

```
1 | protected $table = 'libros';
```

15.1.2. crear tabla

Crea una tabla `libros` en tu base de datos. Para esto, recuerda:

1. crear una migración para crear la tabla libros:

```
1 | sudo docker-compose exec myapp php artisan make:migration
   create_libros_table
```


Este comando generará un nuevo archivo de migración en el directorio `database/migrations`.

2. abre el archivo de migración recién creado. Puedes encontrarlo en el directorio `database/migrations` y tendrá un nombre similar a `2024_01_01_000000_create_libros_table.php` (la fecha y la hora pueden variar).
3. dentro del archivo de migración, encontrarás dos métodos: `up` y `down`. En el método `up`, define la estructura de la tabla "libro". Puedes hacerlo utilizando la sintaxis del constructor de esquemas de Laravel. Aquí hay un ejemplo básico:

```

1  use Illuminate\Database\Migrations\Migration;
2  use Illuminate\Database\Schema\Blueprint;
3  use Illuminate\Support\Facades\Schema;
4
5  class CreateLibroTable extends Migration
6  {
7      public function up()
8      {
9          Schema::create('libro', function (Blueprint $table) {
10             $table->id();
11             $table->string('titulo');
12             $table->text('descripcion')->nullable();
13             // Agrega más columnas según sea necesario para tu
14             aplicación
15             $table->decimal('precio', 8, 2);
16             $table->timestamps();
17         });
18     }
19     public function down()
20     {
21         Schema::dropIfExists('libro');
22     }
23 }
```

En este ejemplo, la tabla `libros` tiene una columna de ID, un título, una descripción, un precio (que hemos añadido nosotros pues podemos personalizar la estructura de la tabla según nuestras necesidades) y las marcas de tiempo (`created_at` y `updated_at`).

4. Después de definir la estructura de la tabla, ejecuta las migraciones con el siguiente comando:

```
1 | sudo docker-compose exec myapp php artisan migrate
```

Este comando aplicará la migración y creará la tabla "libro" en tu base de datos.

15.1.3. crear seeder

Crea un `seeder` para introducir información:

```
1 | sudo docker-compose exec myapp php artisan make:seeder LibroSeeder
```

Este comando generará un nuevo archivo de seeder en el directorio `database/seeds`.

1. abre el archivo de seeder recién creado. Puedes encontrarlo en el directorio `database/seeds` y tendrá un nombre similar a `LibroSeeder.php`.

Dentro del archivo de seeder, dentro del método `run`, puedes utilizar el modelo `Libro` para insertar datos en la tabla. Asegúrate de tener el modelo `Libro` creado en tu aplicación. Aquí tienes un ejemplo básico:

```

1  use Illuminate\Database\Seeder;
2  use App\Models\Libro;
3
4  class LibroSeeder extends Seeder
5  {
6      public function run()
7      {
8          // Ejemplo de inserción de datos en la tabla "libro"
9          Libro::create([
10             'titulo' => 'Libro 1',
11             'descripcion' => 'Descripción del Libro 1',
12             'precio' => 15.99,
13         ]);
14
15         Libro::create([
16             'titulo' => 'Libro 2',
17             'descripcion' => 'Descripción del Libro 2',
18             'precio' => 21.50,
19         ]);
20
21         // *****
22         // AGREGA MÁS LIBROS (POR LO MENOS 5)
23         // *****
24     }
25 }
```

2. ejecutar el seeder y insertar los datos en la tabla, utiliza el siguiente comando:

```
1  sudo docker-compose exec myapp php artisan db:seed --class=LibroSeeder
```

Esto ejecutará el seeder que acabas de crear y agregaría los datos a la tabla "libro".

Recuerda que, además de ejecutar un seeder específico, también puedes utilizar el comando `php artisan db:seed` sin especificar un seeder en particular para ejecutar todos los seeders definidos en tu aplicación.

15.1.4. crear ruta

Crear una ruta para acceder a dicho formulario de inserción de libros (productos).

Para ello, recuerda introducir en el fichero `web.php` de la carpeta `routes` la ruta para lanzar el controlador-mostrarTodos (y, si no se introduce de forma automática, poner también el `use` que apunte a dicho controlador).

```
1  Route::post('/libros', [LibroController::class, 'mostrarTodos']) ->
    name('libros');
```

15.1.5. crear vista

Crear la vista `resources/views/libros/mostrar.blade.php` e introducir el código para recorrer la variable `$libros` que se pasa desde el método `mostrarTodos` del controlador `LibroController`. Para esto, recuerda, utilizar la directiva `@foreach` ... `@foreach` y acceder a cada campo mediante `$libro->id` (por ejemplo).

Crea en tu página principal (del ejemplo seguido durante esta unidad) un enlace a dicha ruta para probar el ejercicio (también lo podrás probar añadiendo a la URL de la aplicación `/mostrar-libros`).

15.2. Ejercicio 2

Crear una página (vista) en la que se inserte un registro de la tabla libros. Un formulario (parecido) al que hemos estado creando (registro) pero que muestre un libro:

1. Recuerda poner en el método `store` en el controlador:

```

1      public function store(Request $request) {
2
3          $this->validate($request, [
4              'titulo' => ['required', 'unique:libros'],
5              'precio' => ['required', 'numeric',
6                          'min: 0', 'max:10000',
7                          'regex:/^\d+(\.\d{1,2})?$/']
8          ]);
9      }

```

- o `'max:10000'` para establecer un valor máximo permitido.
- o `'regex:/^\d+(\.\d{1,2})?$/'` para permitir decimales con hasta dos lugares decimales.

2. Por otro lado, en el **modelo** debemos insertar en la clase `Libro` :

```

1      protected $fillable = [
2          'titulo',
3          'descripcion',
4          'precio',
5      ];

```

En Laravel, el atributo `protected $fillable` se utiliza para especificar qué campos de una tabla de base de datos pueden ser asignados masivamente. Cuando se realiza una operación de asignación masiva, como al crear un nuevo modelo utilizando el método `create` o al actualizar un modelo mediante el método `update`, Laravel utiliza el conjunto de campos especificados en `$fillable` para determinar qué datos se deben incluir en la operación.

En este caso, al crear o actualizar un usuario, solo los campos `titulo`, `descripcion` y `precio` serán aceptados en una asignación masiva. Si intentas asignar otros campos no especificados en `$fillable`, Laravel lanzará una excepción `MassAssignmentException` para proteger contra la asignación masiva no autorizada.

Este enfoque ayuda a mejorar la seguridad al limitar qué campos pueden ser modificados de manera masiva, evitando posibles ataques de asignación masiva no autorizada. Es una práctica recomendada utilizar `$fillable` para definir explícitamente los campos que pueden ser asignados masivamente en lugar de permitir la asignación masiva de todos los campos con `$guarded`.

15.2.1. crear formulario

[...]

15.2.2. crear página

[...]

¿Qué entregar?

Como entrega de esta sesión deberás comprimir tu proyecto **laravel** con los cambios incorporados, y eliminando las carpetas `vendor` y `node_modules`. Renombra el archivo comprimido a `proyecto_07.zip`.

16. referencias

- [Tutorial de Composer](#)
- [Web Scraping with PHP – How to Crawl Web Pages Using Open Source Tools](#)
- [PHP Monolog](#)
- [Unit Testing con PHPUnit — Parte 1.](#), de Emiliano Zublena.