LARAVEL

**CREAR UN PROYECTO NUEVO**

En la terminal de Ubuntu 20.04 escribir el comando: **curl -s** [**https://laravel.build/example-app**](https://laravel.build/example-app) **| bash**

**ARRANCANDO EL SERVIDOR EN DOCKER**

Nuevamente en la terminal de Ubuntu escribir cd nombreDeLaCarpeta y poner el comando: **./vendor/bin/sail up**

Detener los servicios: cuando termine de trabajar voy a powershell, escribo **wsl**, **cd nombreDeLaCarpeta** y por último **./vendor/bin/sail down**

**SAIL**

Sail es un CLI para comunicarte, interactuar con los archivos de Docker para arrancar tus servicios, llamar artisan, o instalar dependencias de npm.

**Como utilizarlo:** visitar la documentación oficial para ver todos sus comandos.

**INICIAR UN SERVIDOR**

Ubicarnos dentro de la carpeta que contiene nuestros archivos de proyecto, abrir la terminal y escribir el comando: **php -S localhost: numeroDePuerto**

**PROGRAMACIÓN ORIENTADA A OBJETOS**

**CREANDO UNA CLASE**

La creamos con la palabra reservada **class** seguido del nombre que le pongamos con la primera letra en **mayúscula**.

Las clases te permiten crear objetos, y para crearlo necesitamos **instanciar una clase**, con la palabra reservada **new** seguido del **nombre de la clase**.

(**var\_dump()** proporciona información sobre el tamaño y tipo de datos de la variable y, en el caso de objetos y arrays, de los elementos que la componen)

class MenuRestaurant {

}

$bebida = new MenuRestaurant();

$postre = new MenuRestaurant();

// var\_dump($bebida);

**PROPIEDADES**

Las variables de las clases se conocen como **propiedades**. Una propiedad contiene información que pertenece a un objeto.

class MenuRestaurant {

    public $nombre = "";

    public $precio = 0;

}

$bebida = new MenuRestaurant();

$bebida->nombre = "Jugo de naranja"; //sintaxis de asignar un valor

$bebida->precio = 30;

echo $bebida->nombre."<br/>"; //sintaxis para mostrar un valor

var\_dump($bebida);

// Jugo de naranja

// object(MenuRestaurant)#1 (2) {

//     ["nombre"]=>

//     string(15) "Jugo de naranja"

//     ["precio"]=>

//     int(30)

//   }

$postre = new MenuRestaurant();

$postre->nombre = "Helado";

$postre->precio = 100;

var\_dump($postre);

**MÉTODOS**

Las function que se encuentran dentro de una **class** se llaman **métodos**.

class MenuRestaurant {

    public $nombre = 'Nombre del Producto';

    public $precio = 0;

    public function obtenerProductoyNombre () {

        echo "El producto ".$this->nombre." tiene un precio de ".$this->precio;

    }

}

// Instanciar una clase

$bebida = new MenuRestaurant();

$bebida->nombre = "Jugo de naranja";

$bebida->precio = 100;

$bebida->obtenerProductoyNombre();

echo "<br/>";

$postre = new MenuRestaurant();

$postre->nombre = "Tiramisú";

$postre->precio = 500;

$postre->obtenerProductoyNombre();

**GETTERS**

Los getters son una convención que existen en la OOP para obtener el valor que existe en un objeto. Son funciones que te permiten acceder a las propiedades de los objetos, utilizamos la palabra **get** seguida del nombre de la variable/propiedad.

Estos retornan un valor, NO lo imprimen al valor.

Se suelen hacer un getter por cada propiedad que tenga el objeto.

class MenuRestaurant {

        public $nombre = 'Nombre del Producto';

        public $precio = 0;

        public function getNombre() {

            return $this->nombre;

        }

        public function getPrecio() {

            return $this->precio;

        }

    }

$bebida = new MenuRestaurant();

$bebida->nombre = "Jugo de naranja";

$bebida->precio = 30;

echo $bebida->getNombre();//Jugo de naranja

echo "<br>";

echo $bebida->getprecio();// 100

**CONSTRUCTORES**

Es una function que se ejecuta automáticamente en el momento en el que creas un objeto con un class, es decir, en el momento que la instancias.

Se escribe así: **\_\_construct(){}**

class MenuRestaurant {

    public $nombre;

    public $precio;

    public function \_\_construct($nombre, $precio) {

        $this->nombre = $nombre;

        $this->precio = $precio;

    }

    public function getNombre()

    {

        return $this->nombre;

    }

    public function getPrecio()

    {

        return $this->precio;

    }

}

$bebida = new MenuRestaurant("Jugo de naranja", 30);

echo $bebida->getNombre();

**HERENCIA**

Al heredar una clase todas sus propiedades y métodos se le pasan a la nueva clase.

Utilizamos la palabra reservada **extends nombreDeLaClasePadre**.

Para pedir algo del componente padre utilizamos **parent::**

class MenuRestaurant {

    public $nombre;

    public $precio;

    public function \_\_construct( $nombre, $precio ) {

        $this->nombre = $nombre;

        $this->precio = $precio;

    }

    public function getNombre()

    {

        return $this->nombre;

    }

    public function getPrecio()

    {

        return $this->precio;

    }

}

class Bebida extends MenuRestaurant{

    public $medida;

    public function \_\_construct( $nombre, $precio, $medida ) {

        parent::\_\_construct($nombre, $precio);

        $this->medida = $medida;

    }

    public function getMedida()

    {

        return $this->medida;

    }

}

$bebida = new Bebida("Jugo de naranja", 30, "300ml");

var\_dump($bebida);

**MODIFICADORES DE ACCESO: public, private y protected**

Todos las propiedades y métodos que estén en una clase deben tener un modificador de acceso. Esto le va a permitir al código, acceder a ciertos valores o restringir su acceso.

**Public:** se puede acceder en cualquier lugar, ya sea en la clase o en el objeto.

**Protected:** significa que solo se puede acceder por medio de la clase.

**Private:** no puede ser accedido por los hijos, solamente la clase que lo tienen puede acceder. La única forma de acceder a ella es a través una function que este **public**.

class MenuRestaurant {

    public $nombre;

    protected $precio;

    public function \_\_construct( $nombre, $precio ) {

        $this->nombre = $nombre;

        $this->precio = $precio;

    }

    public function getNombre()

    {

        return $this->nombre;

    }

    public function getPrecio()

    {

        return $this->precio;

    }

}

class Postre extends MenuRestaurant {

    public $peso;

    public function \_\_construct( $nombre, $precio, $peso ) {

        parent::\_\_construct( $nombre, $precio );

        $this->peso = $peso;

    }

    public function getPeso()

    {

        return $this->peso;

    }

}

$postre = new Postre("Helado", 100, "150g");

echo $postre->nombre;//Helado. es public: accedemos por medio del objeto

echo "<br>";

echo $postre->getPrecio();//100. es protected: accedemos por medio de la clase.

**SETTERS**

Se utilizan para añadir o modificar un valor.

Se escribe **setNombreDeLaPropiedad(){}**

class MenuRestaurant {

        public $nombre;

        public $precio;

        protected $categoria;

        public function \_\_construct( $nombre, $precio ) {

            $this->nombre = $nombre;

            $this->precio = $precio;

        }

        public function getNombre()

        {

            return $this->nombre;

        }

        public function getPrecio()

        {

            return $this->precio;

        }

        public function getCategoria()

        {

            return $this->categoria;

        }

        public function setCategoria($categoria) {

            $this->categoria= $categoria;

        }

    }

class Postre extends MenuRestaurant {

        public $peso;

        public function \_\_construct( $nombre, $precio, $peso ) {

            parent::\_\_construct( $nombre, $precio );

            $this->peso = $peso;

        }

        public function getPeso()

        {

            return $this->peso;

        }

    }

    $postre = new Postre("Tiramisu", 100, "150g");

    $postre->setCategoria("dulce");

    var\_dump($postre);

**MÉTODOS ESTÁTICOS**

Los métodos estáticos tienen como característica que no requieren instanciar la clase y puedes acceder directamente a ellos por medio del nombre de la clase en lugar de crear un objeto.

Utilizamos la palabra reservada **static** entre el **modificador de acceso** y la palabra **function**.

**CONSTANTES**

Se escriben con la palabra reservada **const** y el nombre todo en **mayúsculas**.

Tampoco deben ser instanciadas para usarlas, solo debemos poner el **nombreDeLaClase::nombreDeLaConstante**.

class Bebida extends MenuRestaurant {

    public $medida;

    // Constante

    const CON\_ALCOHOL = 1;

    const SIN\_ALCOHOL = 1;

    public function \_\_construct( $nombre, $precio, $medida ) {

        parent::\_\_construct( $nombre, $precio  );

        $this->medida = $medida;

    }

    public function getPrecio()

    {

        return "El precio es: " . $this->precio;

    }

    public function getMedida()

    {

        return $this->medida;

    }

}

echo Bebida::CON\_ALCOHOL;//1

**CLASSES ABSTRACTAS**

Es una clase que no se puede instanciar. En cambio, sirve como base para otras clases, es decir, que sus clases hijas heredan todo su contenido y estas sí se pueden instanciar.

Se crean utilizando la palabra reservada **abstract**, seguido de **class NombreDeLaClase{}**

abstract class MenuRestaurant {}

**INTERFACES**

Las interfaces son plantillas para tus clases. Se crean con la palabra reservada **interface nombreInterface {}**

Conectamos la interfaz con la clase mediante la palabra reservada **implements** seguido del nombre de la interface que creamos.

SOLO FUNCIONA CON MÉTODOS**.**

interface RestauranteInterface {

    public function getNombre();

    public function getPrecio();

}

class MenuRestaurant implements RestauranteInterface {

    public $nombre;

    protected $precio;

    public function \_\_construct( $nombre, $precio ) {

        $this->nombre = $nombre;

        $this->precio = $precio;

    }

    public function getNombre()

    {

        return $this->nombre;

    }

    public function getPrecio()

    {

        return $this->precio;

    }

}

**NAMESPACE**

Se utiliza para que dos clases puedan tener el mismo nombre. Se utiliza la palabra reservada **namespace**.

namespace empleado\_1;

class Empleado {

    public function \_\_construct(){

        echo "desde Empleado.php";

    }

}

$empleado = new Empleado();

OTRO ARCHIVO:

require\_once 'Empleado.php';

echo "<br>";

class Empleado {

    public function \_\_construct(){

        echo "desde 14-namespaces.php";

    }

}

$empleado = new Empleado();

**DEVSTAGRAM: PRIMEROS PASOS CON LA APP**

**CREANDO UN LAYOUT**

Para hacer el contenido dinámico crearemos un archivo dentro de una capeta layout dentro de views, y utilizaremos en los demás archivos la palabra reservada **@extends(ruta de donde esta el archivo del layout)**.

Y para que el contenido sea dinámico utilizaremos **@yield(“nombre del yield a identificar”)**

**Layout:**

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <meta http-equiv="X-UA-Compatible" content="ie=edge">

    <title>Index</title>

</head>

<body>

    <h1>@yield("title")</h1>

</body>

</html>

**Index:**

@extends("layouts.app")

@section("title")

    Inicio

@endsection

**LARAVEL MIX**

Nos permite copilar frameworks CSS y JS en nuestras aplicaciones de Laravel sin esfuerzo.

Soporta Tailwind, Boostrap, SASS para CSS.

También puedes añadir React o VueJS a tus aplicaciones.

Solo necesitamos instalar las dependencias y ejecutar un watch.

**Instalar Tailwind:**

* Paso 1: Escribir en la carpeta del proyecto el comando:

**./vendor/bin/sail npm install -D tailwindcss postcss autoprefixer**

* Paso 2: Luego de realizar ese paso, escribimos el siguiente comando.

**./vendor/bin/sail npx tailwindcss init**

* Paso 3: el tercer paso es abrir **webpack.mix.js** y escribir en los corchetes require(“tailwindcss”).

mix.js('resources/js/app.js', 'public/js')

    .postCss('resources/css/app.css', 'public/css', [

        require("tailwindcss")

    ]);

* Paso 4: en la consola escribir el comando **./vendor/bin/sail npm run watch**
* Paso 5: en el archivo tailwind.config.js en **content** escribir la ruta donde queremos que se escaneen los estilos que requiera el archivo que especificamos.

module.exports = {

  content: [

    "./resources/views/layouts/app.blade.php"

  ],

  theme: {

    extend: {},

  },

  plugins: [],

}

Si tenemos más layout la mejor forma de poner la ruta es la siguiente:

module.exports = {

  content: [

    "./resources/\*\*/\*/\*.blade.php", "./resources/\*\*/\*/\*.js"

  ],

  theme: {

    extend: {},

  },

  plugins: [],

}

Esto lo que va a hacer es de forma recursiva entrar a todas las carpetas que esten en views y que tengan como extensión .blade.php para que se les aplique tailwind.

* Paso 6:en el archivo **app.css** de la carpeta resourcer/css escribir: **@tailwind base; @tailwind components; @tailwind utilities;**
* Paso 7: en el archivo app.blade.php escribimos la etiqueta link y en el href pasamos la ruta con la function **asset**.

<link rel="stylesheet" href="{{ asset("css/app.css") }}">

<script src="{{ asset("js/app.js") }}" defer></script>

**REGISTRO Y LOGIN DE USUARIO**

**MVC**

Son las iniciales de Model View Controller.

Es un patrón de arquitectura de software que permite la separación de obligaciones de cada pieza de tu código.

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 una mejor organización 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.

**Modelo:**

* Encargado de todas las interacciones en la base de datos, obtener datos, actualizarlos y eliminarlos.
* Se encarga de consultar una base de datos, obtiene la información, pero no la muestra, eso es trabajo de la vista.
* El modelo tampoco se encarga de actualizar la información directamente; es el Controlador quien decide cuando llamarlo.

**Vista:**

* Se encarga de todo lo que se ve en pantalla (HTML).
* Laravel tiene un template engine llamado Blade para mostrar los datos.
* El modelo consulta la base de datos, pero es por medio del controlador que se decide que Vista hay que llamar y que datos presentar.

**Controlador:**

* Es el que comunica el modelo y vista; antes de que el Modelo consulte la base de datos, el Controlador es el encargado de llamar un Modelo específico.
* Una vez consultado el Modelo, el controlador recibe esa información, manda a llamar a la vista y le pasa la información.
* El controlador es el que manda a llamar a la vista y modelos, que se requieren en cada parte de tu aplicación.

**Router en MVC:**

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

**ARTISAN**

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

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

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

**Comandos:**

**./vendor/bin/sail php artisan** o **./vendor/bin/sail artisan**

Este comando instala artisan y nos permite ejecutar ciertos comandos más, como por ejemplo crear un **controller**:

**./vendor/bin/sail artisan make:controller –help** (esto nos permite ver cómo crearlo)

El comando para crear en sí un controller es el mismo pero agregando el nombre de la clase que vamos a crear acompañado de la palabra controller, ejemplo:

**./vendor/bin/sail artisan make:controller RegisterController**

Luego de ejecutar ese comando, en la carpeta HTTP/Controllers se encontrará el archivo creado.

**Crear controller dentro de una subcarpeta:**

**./vendor/bin/sail artisan make:controller auth\\RegisterController**

**TIPOS DE REQUEST**

En HTTP existen diferentes tipos de request: GET, POST, PUT, PATCH y DELETE.

**GET:** 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.

**IMPLEMENTANDO SEGURIDAD PARA CROSS SITE REQUEST**

Utilizamos **@csrf** para que se pueda comunicar correctamente el controller por **POST**.

<form action="/register" method="POST">

 @csrf

… Resto del codigo

Este **csrf** nos creara en el código fuente este input:

<input type="hidden" name="\_token" value="FqeR95D8cibkLQbMupQlTrYtHWGsYUwaKBYt7s8z">

**CONTROLLERS y CONVENCIONES**

Los controllers van a ayudarte a tener un código mejor organizado, además de una separación mayor en la funcionalidad de tus aplicaciones y sitios web.

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

|  |  |  |  |
| --- | --- | --- | --- |
| **Verbo HTTP** | **URI** | **Acción** | **Ruta** |
| GET | /clientes | Index | Clientes.index |
| POST | /clientes | Store | Clientes.store |
| DELETE | /clientes/{cliente} | destroy | Clientes.destroy |

**NOMBRES A LAS RUTAS Y SUS VENTAJAS**

Podemos darles nombres predeterminados a las rutas para que no tener que en un futuro andar corrigiendo todos los enlaces.

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

Y después en todos los enlaces **a** y en los **actions** de los form utilizamos el helper **route(“”)**:

<form action="{{ route("register") }}" method="POST">

<a href="{{ route("register") }}">Crear cuenta</a>

**LEER DATOS INGRESADOS EN UN FORMULARIO**

La function **dd();** en un controlador sirve para imprimir lo que pase como argumento.

Debemos pasarle al controller del POST un Request:

public function store (Request $request)

    {

        // dd($request); //accedo a todos los valores

        dd($request->get("username")); //accedo a un valor determinado

    }

**VALIDACIÓN DE FORMULARIOS**

Utilizamos el método **validate($request, [name del input => validacion]);**

// Validación

$this->validate($request, [

   "name" => ["required", "max:30"]

]);

Luego en el **form** para mostrar el mensaje utilizamos la directiva **@error(“”)** y **@enderror(“”)**

En esta directiva podemos poner una etiqueta <p> y usar mensajes predeterminados de Laravel poniendo {{ $message }}

<p>{{ $message }}</p>

Podemos usar el **@error** dentro de un **class** y de un **value** de una etiqueta para que haga algo si hay error:

<input

   class="border p-3 w-full rounded-lg @error("name") border-red-500 @enderror"

   type="text"

   name="name"

   id="name"

   placeholder="Escribí tu nombre..."

   value= {{ old("name") }}

>

**MIGRACIONES y la BASE DE DATOS en LARAVEL**

**MIGRACIONES**

Se las 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.

**Comandos útiles:**

**./vendor/bin/sail artisan migrate** : este comando corre las migraciones, las ejecuta

**./vendor/bin/sail artisan migrate:rollback** : esto revierte la migración.

**./vendor/bin/sail artisan migrate:rollback –step=5** : revertimos las últimas 5 migraciones.

**Otros comandos para crear las migraciones:**

**./vendor/bin/sail artisan make:migration agregar\_imagen\_user**

Luego de haber migrado la base de datos, al estar mysql dentro de Docker corremos el siguiente comando:

**./vendor/bin/sail mysql -u**

A continuación, escribimos: **show databases;**

Después elegimos la base de datos a usar, ejemplo: **use devstagram;**

Y ahora, podemos ver las tablas: **show tables;**

Si queremos ver una tabla en particular escribimos, por ejemplo: **describe users;**

**CONECTARSE A LA BASE DE DATOS CON DBeaver**

Nos podemos conectar a la base de datos con la información que aparece en el archivo **.env** en la raíz del proyecto:

DB\_CONNECTION=mysql

DB\_HOST=mysql

DB\_PORT=3306

DB\_DATABASE=devstagram

DB\_USERNAME=sail

DB\_PASSWORD=password

En **DBeaver** arriba a la izquierda hay un logo de conexión que nos permite conectar a diversas bases de datos. En este caso vamos a usar MySQL.

Si al poner los datos de la abse de datos (nombre, usuario, contraseña) nos tira error, debemos hacer click derecho en el nombre de la base de datos/Editar información, y el error en sí lo tenemos en el puerto, porque es el mismo puerto que usa **Docker**.

Entonces para arreglar esto vamos al proyecto, buscamos en la raíz del proyecto el archivo **Docker-compose.yml** y buscamos una variable que se llama **FORWARD\_DB\_PORT:-3306**, copiamos esa variable y en el archivo **.env** la pegamos y le decimos otro puerto:

DB\_CONNECTION=mysql

DB\_HOST=mysql

DB\_PORT=3306

DB\_DATABASE=devstagram

DB\_USERNAME=sail

DB\_PASSWORD=password

FORWARD\_DB\_PORT=3307

Entonces ahora si volvemos a **DBeaver** y colocamos el puerto que elegimos.

**FINALIZANDO EL REGISTRO DE USUARIOS**

**AÑADIR LA MIGRACION PARA USERNAME**

Debemos ejecutar el comando:

**./vendor/bin/sail artisan make:migration add\_username\_to\_users\_table**

Luego especificar que vamos a agregar en el archivo que se creó:

**$table->string(“username”);**

public function up()

    {

        Schema::table('users', function (Blueprint $table) {

            $table->string("username");

        });

    }

    /\*\*

     \* Reverse the migrations.

     \*

     \* @return void

     \*/

    public function down()

    {

        Schema::table('users', function (Blueprint $table) {

            $table->dropColumn("username");

        });

    }

Y por último volvemos a correr:

**./vendor/bin/sail artisan migrate**

**CREANDO REGISTROS con ELOQUENT ORM**

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.

**Comando para crear un modelo:**

**./vendor/bin/sail php artisan make:model NombreDelModelo**

El nombre del modelo va en singular.

**Convenciones en modelos:**

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

Ya posicionado en el archivo del controller, después de las validaciones escribimos **User** y le damos **enter**, esto nos va a crear la importación del modelo en el controller.

Ahora creamos el usuario:

User::create([

   "name" => $request->name,

   "username" => $request->username,

   "email" => $request->email,

   "password" => $request->password

]);

**INSERTAR EL USUARIO CON EL MODELO**

Lo anterior esta bien, pero nos va a dar error porque el “username” lo creamos nosotros. Entonces en el modelo **User** en la variable **$fillable** agregamos **“username”**. Esto es para que Laravel sepa que valores se van a insertar en la base de datos:

protected $fillable = [

'name',

  'email',

  'password',

  'username'

];

**HASHEAR PASSWORD**

Escribimos **Hash** antes del **$request** y le damos **enter** para que nos importe el Hash.

Luego utilizamos **Hash:make()**

User::create([

  "name" => $request->name,

  "username" => $request->username,

  "email" => $request->email,

  "password" => Hash::make($request->password)

]);

**ALMACENAR EL USERNAME DE FORMA QUE SE PUEDA UTILIZAR EN LA URL**

En el caso que el usuario en un campo de **username** escriba con espacios, debemos modificar este username para que podamos usarlo en la URL. Esto lo hacemos con el **helper** denominado **Str::slug()**. Este helper transforma los espacios en guiones.

User::create([

 "name" => $request->name,

 "username" => Str::slug($request->username),

 "email" => $request->email,

 "password" => Hash::make($request->password)

]);

**EVITAR USERNAME DUPLICADOS**

En el archivo de username creado en **migrations** debemos especificarle que sea **unique()**.

Schema::table('users', function (Blueprint $table) {

    $table->string("username")-> unique();

});

Si esto no lo habíamos hecho antes vamos a tener que escribir el comando:

**./vendor/bin/sail php artisan migrate:rollback –step=1**

Para retroceder en lo migrado, y volver a aplicar la migración con:

**./vendor/bin/sail php artisan migrate**

Luego debemos modificar el **$request**:

//Modificando el Request

$request->request->add(["username" => Str::slug($request->username)]);

User::create([

   "name" => $request->name,

   "username" => $request->username,

   "email" => $request->email,

   "password" => Hash::make($request->password)

]);

**REDIRECCIONAR AL USUARIO AL MURO, UNA VEZ CREADA LA CUENTA**

Vamos a crear un **PostController** y un **LoginController** con el comando:

**./vendor/bin/sail artisan make:controller PostController**

**./vendor/bin/sail artisan make:controller LoginController**

Luego en el **RegisterController** vamos a hacer el redireccionamiento con un helper **redirect()**, y a este helper le pasamos **route(**nombre de como le pusimos en el archivo de ruta**)**:

**Archivo web.php:**

Route::get('/feed', [PostController::class, "index"])->name("posts.index");

**Controller:**

return redirect()->route("posts.index");

**AUTENTICACIÓN DE USUARIOS, REDIRECCIONAMIENTO Y CREAR EL MURO**

**AUTENTICAR A UN USUARIO**

Con el helper **auth()** y su function **attempt()** :

**Controller de register:**

// Autenticar un usuario

// auth()->attempt([

// "email" => $request->email,

//  "password" => $request->password

// ]);

// Otra forma de autenticar

auth()->attempt($request->only("email", "password"));

**PostController:**

class PostController extends Controller

{

    public function index ()

    {

        dd(auth()->user());//vamos a ver la info que llega

    }

}

**CREANDO EL MURO**

Para que no pueda mostrar el **muro** a un usuario no autenticado creamos un **construct** que utilice un middleware de validación:

class PostController extends Controller

{

    public function \_\_construct()

    {

        $this->middleware("auth");

    }

    public function index ()

    {

        // dd(auth()->user());//vamos a ver la info que llega

        return view("dashboard");

    }

}

Para mostrar el nombre del usuario en el muro hacemos así:

<p>{{ auth()->user()->username }}</p>

**REVISAR SI LAS CREDENCIALES DEL LOGIN SON CORRECTAS**

En el controller del post por login vamos a hacer un **if**, que verifique si esta autenticado, y que si no esta autenticado muestre un error con la function **back()-with()**:

if(!auth()->attempt($request->only("email", "password"))){

    return back()->with("mensaje", "Credenciales incorrectas");

}

La function **back()** lo que hace es redirigir a la vista anterior al **procedimiento del POST**, y agregándole el **with** le pasamos un mensaje.

Luego en la **vista** debemos pasar ese if con el mensaje, y redirigir:

@if (session("mensaje"))

   <p>{{ session("mensaje") }}</p>

@endif

return redirect()->route("posts.index");

**CERRAR SESIÓN Y DIRECTIVAS PARA UTILIZARSE EN TEMPLATES**

**REVISAR SI UN USUARIO ESTA AUTENTICADO O NO**

Podemos hacerlo con las **directivas @if** o con **@auth** y **@guest**:

**Con if:**

@if (auth()->user())

   <li><a href="{{ route("posts.index") }}"> Hola: {{ auth()->user()->username }}

</a></li>

   <li><a href="#">Cerrar sesión</a></li>

@else

  <li><a href="{{ route("login") }}">Login</a></li>

   <li><a href="{{ route("register") }}">Crear cuenta</a></li>

@endif

**Con auth y guest:**

@auth

  <li><a href="{{ route("posts.index") }}">Hola: {{ auth()->user()->username }}</a></li>

   <li><a href="#">Cerrar sesión</a></li>

@endauth

@guest

  <li><a href="{{ route("login") }}">Login</a></li>

   <li><a href="{{ route("register") }}">Crear cuenta</a></li>

@endguest

**CERRAR LA SESIÓN**

**Creamos el LogoutController:**

./vendor/bin/sail php artisan make:controller LogoutController

**Creamos la ruta:**

Route::get("/logout", [LogoutController::class, "store"])->name("logout");

**Creamos el método store en el LogoutController:**

Utilizamos el método **logout**

class LogoutController extends Controller

{

    public function store()

    {

        // dd("Cerrando sesión");

        auth()->logout();

        return redirect()->route("login");

    }

}

**AÑADIR SEGURIDAD EN EL CIERRE DE SESIÓN**

En vez de hacer la ruta por **GET** lo hacemos por **POST** y esto nos permite acceder a la directiva **@csrf**. Ya con esto en el **nav** en vez de poner una etiqueta **<a>** lo que hacemos es reemplazarla por un **form** que contenta un **button**.

@auth

   <li><a href="{{ route("posts.index") }}">Hola: {{ auth()->user()->username }}</a></li>

   <form action="{{ route("logout") }}" method="POST">

      @csrf

       <li><button type="submit">Cerrar sesión</button></li>

   </form>

@endauth

**MANTENER LA SESIÓN SIEMPRE ABIERTA**

Creamos el input:checkbox en la vista del login:

<div class="mb-5">

   <label for="rememberMe">Mantener mi sesión abierta</label>

<input type="checkbox" name="remember" id="rememberMe">

</div>

Y ahora en el controller le pasamos el **$request->remember**:

public function store(Request $request)

    {

       // dd($request->remember);//para ver cómo se activa en "on" el checkbox de remember

        $this->validate($request, [

            "email" => ["required", "email"],

            "password" => ["required"]

        ]);

        if(!auth()->attempt($request->only("email", "password"), $request->remember)){

            return back()->with("mensaje", "Credenciales incorrectas");

        }

        return redirect()->route("posts.index");

    }