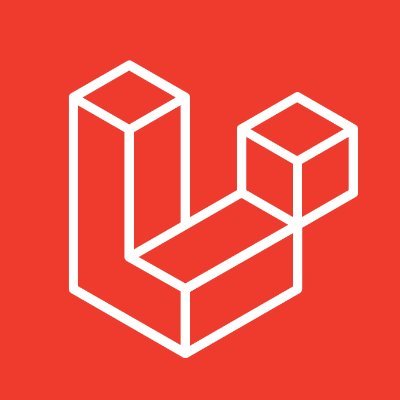
**CURSO DE LARAVEL 10**



**Por**

**MTI. José Arturo Bustamante Lazcano**

Contenido

[INTRODUCCIÓN 3](#_Toc156977011)

[Comando de Instalación del proyecto 3](#_Toc156977012)

[Comando de Instalación Auth 3](#_Toc156977013)

[Soporte de multi lenguaje 3](#_Toc156977014)

[RUTAS 4](#_Toc156977015)

[Middleware 7](#_Toc156977016)

[Layouts y Navegación 7](#_Toc156977017)

[Heredar Estilo 9](#_Toc156977018)

[Links de Navegación 10](#_Toc156977019)

[Link de Navegación móvil 11](#_Toc156977020)

[FORMULARIOS EN LARAVEL 12](#_Toc156977021)

[Casos de envío de datos 12](#_Toc156977022)

[INSERCIÓN EN DATOS EN BD 14](#_Toc156977023)

[Modelos en Laravel 10 14](#_Toc156977024)

[Migraciones 15](#_Toc156977025)

[Eloquent 16](#_Toc156977026)

[Mandar un mensaje de comprobación sobre acción de sesión 17](#_Toc156977027)

[Controladores 19](#_Toc156977028)

[Validación de Formularios 21](#_Toc156977029)

[Base de datos relacional con Eloquent 22](#_Toc156977030)

# INTRODUCCIÓN

## Comando de Instalación del proyecto

\*/ composer créate-project laravle/laravel bootcamp

Ruta del proyecto comandos

\*/ cd bootcamp

\*/ code .

\*/ php artisan serve

## Comando de Instalación Auth

\*/ composer require laravel/breeze –dev

\*/ php artisan breeze:install blade --dark

Abrimos el archivo env para colocar BD y corremos la migración

\*/ php artisan migrate

Ahora Creamos un registro de usuario

## Soporte de multi lenguaje

\*/ composer require laravel-lang/common –dev

\*/ php artisan lang:add es

El efecto del cambio de idioma se debe configurar en la siguiente ruta: config/app.pp Línea 86 ‘locale’ => ‘es’ refrescamos y listo.

Observamos que cuando cerramos la sesión el archivo wellcome que esta por default de laravel aun dice Login y Register esto es por que esta escrito y no tomado como un valor de una variable para eso cambiamos el archivo en la views welcome por:

{{\_\_('Dashboard')}}

{{\_\_('Login')}}

{{\_\_('Register')}}

# RUTAS

Cuando iniciamos un proyecto en laravel por default tenemos una ruta llamada welcome, en este momento tenemos mas rutas por que instalamos la auth de usuarios, las rutas son la activación de las paginas que despliega laravel. Ejemplo:

Route::get('/', function () {

    return view('welcome');

});

Route::get('/dashboard', function () {

    return view('dashboard');

})->middleware(['auth', 'verified'])->name('dashboard');

Route::middleware('auth')->group(function () {

    Route::get('/profile', [ProfileController::class, 'edit'])->name('profile.edit');

    Route::patch('/profile', [ProfileController::class, 'update'])->name('profile.update');

    Route::delete('/profile', [ProfileController::class, 'destroy'])->name('profile.destroy');

});

require \_\_DIR\_\_.'/auth.php';

Las rutas incorporan un método dependiendo de la operación que se realiza por ejemplo get es una método que recolecta información del servidor.

Ahora vamos a crear unos ejemplos con rutas para ello pondré las rutas creadas como un comentario y empezamos a crear nuestros casos:

**Caso 1 queremos mandar en la ruta principal un mensaje de bienvenida**

Route::get('/', function(){

    return 'Bienvenidos al caso 1';

});

**http://localhost/bootcamp/public/**

**Los tipos de ruta como se comento son:**

* **Route::get();**
* **Route::post();**
* **Route::put();**
* **Route::delete();**

**Caso 2 queremos hacer una ruta a servicios que imprima bienvenidos a servicios**

Route::get('/servicios', function(){

    return 'Bienvenidos al caso 2 Servicios';

});

**http://localhost/bootcamp/public/servicios**

**Caso 3 queremos hacer una ruta a chirps y mandar un valor de una variable**

Route::get('chirps/{chirp}', function ($chirp) {

    return 'Bienvenidos al chirps page ' . $chirp;

}

**http://localhost/bootcamp/public/chirps/8**

**Caso 4 Si queremos usar el código anterior tenemos el detalle de que forzosamente debemos mandar un valor pero que pasa si queremos que ese valor se opcional. Para ello le colocamos un ? a la variable y un valor null a la función.**

Route::get('chirps/{chirp?}', function ($chirp = null) {

    return 'Bienvenidos al chirps page ' . $chirp;

});

**http://localhost/bootcamp/public/chirps**

**Caso 5 Condiciones para redirección, en algunos casos necesitamos que apartir de un datos mandemos a una u otra ruta o por ejemplo mandar con permisos o no para esto ejemplificamos que si el valor de chirps es 2 lo mande a una pagina de lo contrario se continue ejecutando.**

Route::get('chirps/{chirp?}', function ($chirp = null) {

    if ($chirp === '2'){

        return redirect('/chirps');

    }

    return 'Chirps page: ' . $chirp;

});

**http://localhost/bootcamp/public/chirps/80**

**http://localhost/bootcamp/public/chirps/2**

**Caso 6 una practica recomendable es usar rutas con nombre estas nos ayudan para que en caso de que la ruta cambie de ubicación esta continue funcionando por que tiene un alias. En este ejemplo cambiare la ruta original de caso chirps por tailwind y especificare el método name y usaremos el index, probamos con los mismos links y estos funcionan.**

Route::get('/tailwind', function(){

    return 'Bienvenidos al Modelo Chirps';

})->name('chirps.index');

Route::get('chirps/{chirp?}', function ($chirp = null) {

    if ($chirp === '2'){

        return to\_route('chirps.index');

    }

    return 'Chirps page: ' . $chirp;

});

**http://localhost/bootcamp/public/chirps/80**

**http://localhost/bootcamp/public/chirps/2**

Para finalizar regresamos nuestro archivo web.php como estaba solo que agregaremos dos rutas editando la de welcome.

Route::view('/', 'welcome')->name('welcome');

Route::get('/chirps', function () {

    return 'Welcome to our chirps page';

})->name('chirps.index');

Route::get('/dashboard', function () {

    return view('dashboard');

})->middleware(['auth', 'verified'])->name('dashboard');

Route::middleware('auth')->group(function () {

    Route::get('/profile', [ProfileController::class, 'edit'])->name('profile.edit');

    Route::patch('/profile', [ProfileController::class, 'update'])->name('profile.update');

    Route::delete('/profile', [ProfileController::class, 'destroy'])->name('profile.destroy');

});

require \_\_DIR\_\_.'/auth.php';

## Middleware

Es una capa intermedia que se sitúa entre la solicitud HTTP entrante y la aplicación. Se utiliza para realizar acciones específicas antes o después de que la solicitud alcance la lógica de la aplicación principal. Los middlewares son una parte fundamental del sistema de manejo de solicitudes HTTP en Laravel y ofrecen una manera conveniente de filtrar, modificar o manipular la solicitud y la respuesta.

**Ejemplos:**

**Caso 1 nosotros tenemos una ruta llamada chirps si queremos que el usuario la pueda ver si esta logeado usaremos el middleware para indicarlo metiendo el código dentro de el.**

Route::view('/', 'welcome')->name('welcome');

Route::get('/dashboard', function () {

    return view('dashboard');

})->middleware(['auth', 'verified'])->name('dashboard');

Route::middleware('auth')->group(function () {

    Route::get('/profile', [ProfileController::class, 'edit'])->name('profile.edit');

    Route::patch('/profile', [ProfileController::class, 'update'])->name('profile.update');

    Route::delete('/profile', [ProfileController::class, 'destroy'])->name('profile.destroy');

    Route::get('/chirps', function () {

        return 'Welcome to our chirps page';

    })->name('chirps.index');

});

require \_\_DIR\_\_.'/auth.php';

**ahora si desea entrar a chirps deberá estar logeado de lo contrario lo mandara al inicio de sesión.**

## Layouts y Navegación

Ahora vamos a crear un elemento aun más sofisticado las vistas que son la capa encargada de mostrar la interfaz de usuario de una aplicación web. En la ruta Resourses/View/ crearemos una carpeta llamada chirps y dentro un archivo index.blade.php.

Nota la carpeta se crea para tener un control de las vistas de nuestro modulo existe en caso de ser un modulo que trabaje con base de datos créate, show, index y delete, además se usa el esquema de plantilla de blade.

Ahora editemos la ruta para que nos mande al archivo en el web.php ahora la ruta se llamara como una vista y dentro de la vista chirps.index escribimos un h1 con algún texto.

Web.php

Route::view('/', 'welcome')->name('welcome');

Route::get('/dashboard', function () {

    return view('dashboard');

})->middleware(['auth', 'verified'])->name('dashboard');

Route::middleware('auth')->group(function () {

    Route::get('/profile', [ProfileController::class, 'edit'])->name('profile.edit');

    Route::patch('/profile', [ProfileController::class, 'update'])->name('profile.update');

    Route::delete('/profile', [ProfileController::class, 'destroy'])->name('profile.destroy');

    Route::get('/chirps', function () {

        return view('chirps.index');

    })->name('chirps.index');

});

require \_\_DIR\_\_.'/auth.php';

Viwe chirps - index.blade.php

<h1>Bienvenidos al primer ejemplo de una vista.</h1>

## Heredar Estilo

Ahora vamos a darle el estilo de plantilla que tiene nuestro sitio para ello hay que tomar las clases de Tailwind que usa laravel. Esta la tomare en layout/dashboard.blade.php y se las pondré a mi archivo chirps / index.blade.php

<x-app-layout>

<h1>Bienvenidos al primer ejemplo de una vista.</h1>

</x-app-layout>

Ya se ve igual pero me gustaría seguir tomando la estructura del dashboard completa así que copiare todo el código del dashboard a mi archivo index de chirps

<x-app-layout>

    <x-slot name="header">

        <h2 class="font-semibold text-xl text-gray-800 dark:text-gray-200 leading-tight">

            {{ \_\_('Chrps') }}

        </h2>

    </x-slot>

    <div class="py-12">

        <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">

            <div class="bg-white dark:bg-gray-800 overflow-hidden shadow-sm sm:rounded-lg">

                <div class="p-6 text-gray-900 dark:text-gray-100">

                    {{ \_\_("You're logged in!") }}

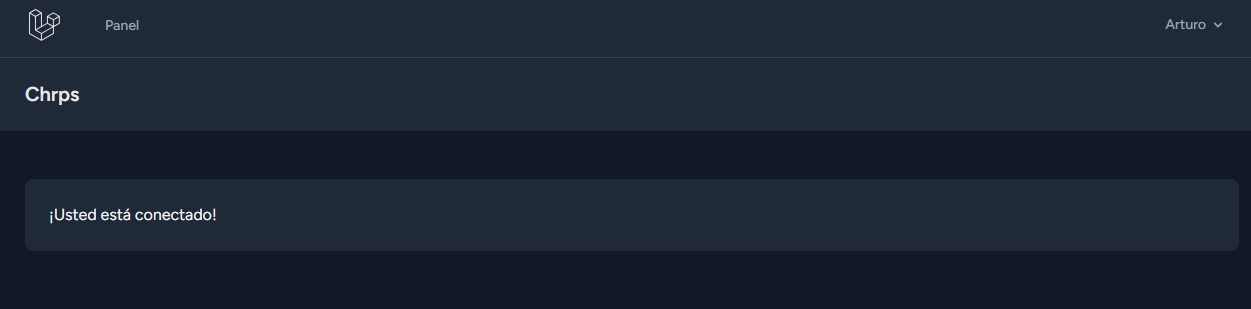
                </div>

            </div>

        </div>

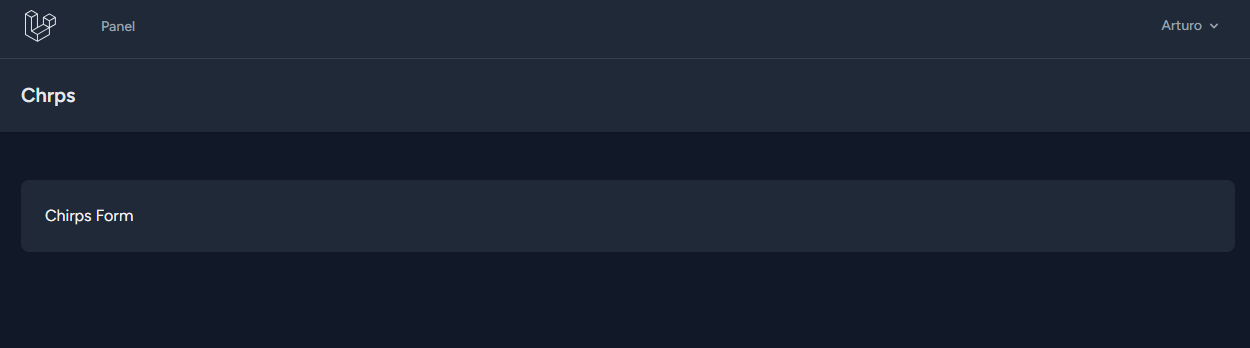
    </div>

</x-app-layout>



## Links de Navegación

Agregaremos unos Links de navegación para pasar entre el dashboard y los chirps y para que se diferencien aun mas le agregare un forms al chirps.



Nos vamos al archivo de navegación que esta en layouts/navigation.blade.php en la sección de Navigation Links y agregamos a chirps ahora hay que recordar que en mis rutas yo le coloque un name así que es hora de usarlo:

<!-- Navigation Links -->

                <div class="hidden space-x-8 sm:-my-px sm:ms-10 sm:flex">

                    <x-nav-link :href="route('dashboard')" :active="request()->routeIs('dashboard')">

                        {{ \_\_('Dashboard') }}

                    </x-nav-link>

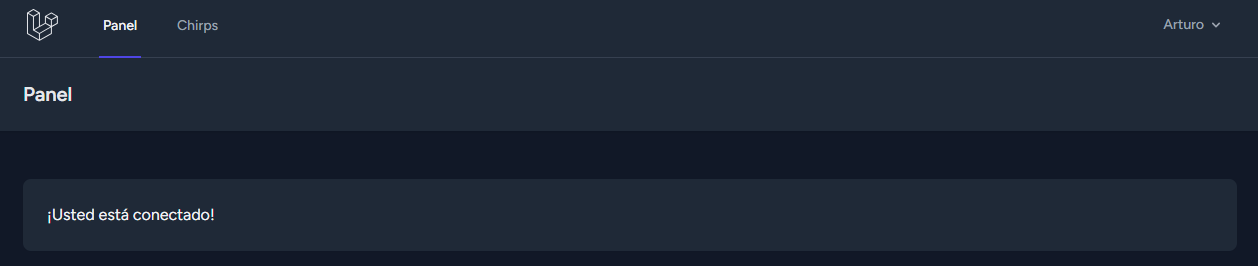
                    <x-nav-link :href="route('chirps.index')" :active="request()->routeIs('chirps.index')">

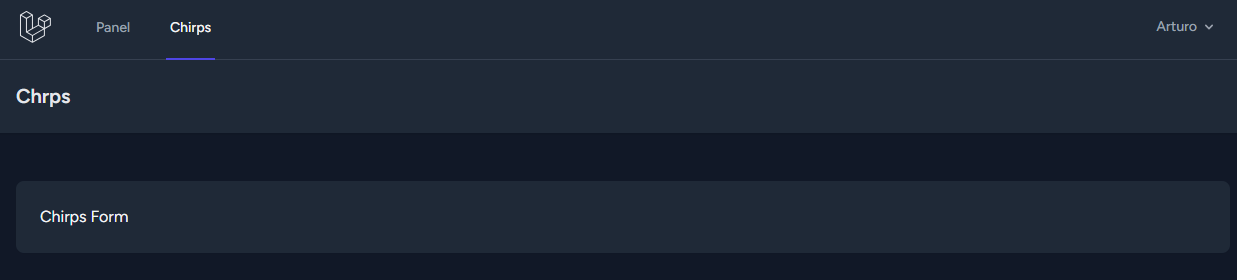
                        {{ \_\_('Chirps') }}

                    </x-nav-link>

                </div>

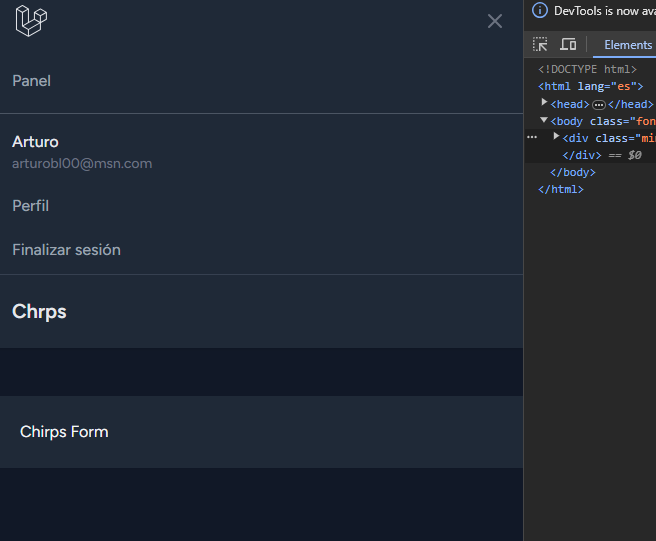
            </div>





## Link de Navegación móvil

Si hacemos responsivo el sitio nos damos cuenta que Chirps no aparece



Editamos la sección

<!-- Responsive Navigation Menu -->

    <div :class="{'block': open, 'hidden': ! open}" class="hidden sm:hidden">

        <div class="pt-2 pb-3 space-y-1">

            <x-responsive-nav-link :href="route('dashboard')" :active="request()->routeIs('dashboard')">

                {{ \_\_('Dashboard') }}

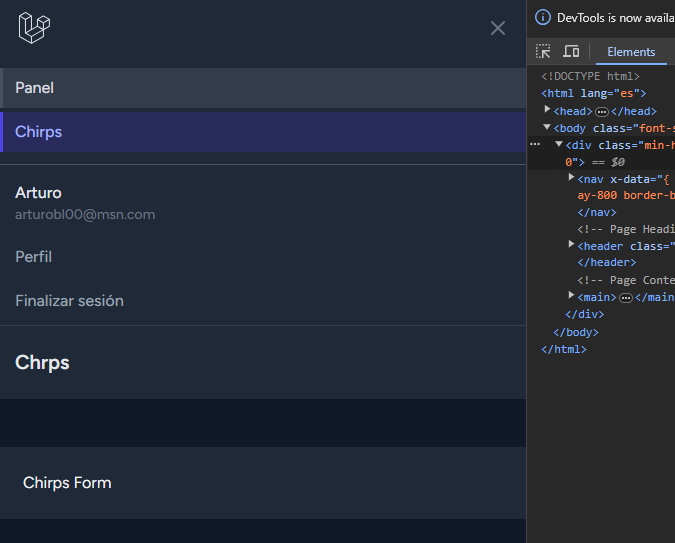
            </x-responsive-nav-link>

            <x-responsive-nav-link :href="route('chirps.index')" :active="request()->routeIs('chirps.index')">

                {{ \_\_('Chirps') }}

            </x-responsive-nav-link>

        </div>



# FORMULARIOS EN LARAVEL

Ahora vamos a agregar un formulario a nuestra pagina chips para eso editamos un poco nuestro archivo index.blade.php

<div class="p-6 text-gray-900 dark:text-gray-100">

                    {{ \_\_("Chirps Form") }}

                    <form>

                        <textarea class="bg-gray-800 text-white border border-gray-700 p-2 w-full h-24"></textarea>

                        <button>Enviar</button>

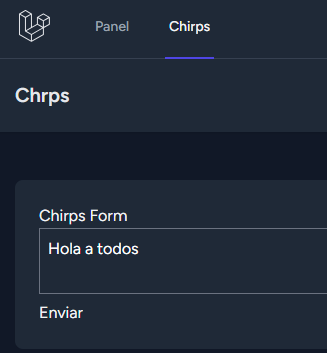
                    </form>

                </div>

Las clases son de Tailwind pregunte a chatGTP como poner un textarea en transparente.

## Casos de envío de datos

Caso 1 enviar los datos sin un name y un método



En este caso el resultado de la url queda así: <http://127.0.0.1:8000/chirps>? No hay datos que se estén recibiendo.

Caso 2: Enviar los datos con name pero sin especificar un método que por default laravel usa get.

Resultado: <http://127.0.0.1:8000/chirps?messege=Arturo>

Caso 3: especificar el método al formulario en post

Para que funcione con el método post no solo se debe poner el método en el formulario también debemos crear una ruta post en nuestro web.php. Aparte al formulario hay que poner un token @csfr.

Paso uno la ruta

Route::post('/chirps', function(){

        return 'Procesando el chirps';

    });

Paso dos form en post y token

<form method="post">

@csrf

      <textarea class="bg-gray-800 text-white border border-gray-700 p-2 w-full h-24" name="messege"></textarea>

      <button>Enviar</button>

</form>

Caso 4: Acceder a los datos que ingreso el usuario. En laravel tenemos un método que se llama request() este se coloca en la función de la ruta a esta le pasamos el name del formulario y la almacenamos dentro de una variable.

 Route::post('/chirps', function(){

        $messege = request('messege');

        //Insertar en la base de datos

        return 'Procesando el chirps - '. $messege;

    });

Voy a dar un poco de formato a mi formulario

<div class="py-12">

        <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">

            <div class="bg-white dark:bg-gray-800 overflow-hidden shadow-sm sm:rounded-lg">

                <div class="p-6 text-gray-900 dark:text-gray-100">

                    {{ \_\_("Chirps Form") }}

                    <form method="post">

                        @csrf

                        <textarea

                        placeholder="{{\_\_('Whats on you mind?')}}"

                        class="bg-gray-800 text-white border border-gray-700 p-2 w-full h-24" name="messege"></textarea>

                        <x-primary-button>Enviar</x-primary-button>

                    </form>

                </div>

            </div>

        </div>

    </div>

# INSERCIÓN DE DATOS EN BD

Lo que necesitamos para que laravel envíe datos de un formulario a una tabla de la base de datos es crear un modelo.

## Modelos en Laravel 10

Los modelos se refieren a una clase de PHP que interactúa con una tabla de la base de datos. Los modelos en Laravel están diseñados para facilitar el acceso y la manipulación de datos en la base de datos. Cada modelo está asociado con una tabla específica en la base de datos y se utiliza para realizar operaciones como la recuperación de registros, la creación de nuevos registros, la actualización de registros existentes y la eliminación de registros.

Comando

\*/ php artisan make:model Chirp -mrc

Para nombrar un modelo debemos usar PascalCase y deben estar escritos en singular los modelos trabajan con una migración y un controlador la migración manda o crea la tabla y el controlador contiene las opciones crud.

Después de ejecutar el comando nos dicen las rutas de nuestro archivo en mi caso:

INFO Model [C:\laragon\www\bootcamp\app\Models\Chirp.php] created successfully.

INFO Migration [C:\laragon\www\bootcamp\database\migrations/2024\_01\_19\_145952\_create\_chirps\_table.php] created successfully.

INFO Controller [C:\laragon\www\bootcamp\app\Http\Controllers\ChirpController.php] created successfully.

La razón por la cual el modelo se crea en singular es por que las entidades “Migraciones” se usan en plural por ejemplo este modelo creara una tabla llamada Chirps. Por ejemplo si yo quiero crear una tabla que se llame Productos mi modelo se llamaría Producto, si quiero una llamada Categorías mi modelo se llamaría Categoria.

## Migraciones

Las migraciones se crean y gestionan mediante la línea de comandos de Artisan, el sistema de comandos de Laravel. Al generar una migración, se crea un archivo en el directorio database/migrations que contiene dos métodos: up y down.

**up:** Este método especifica las operaciones que deben realizarse cuando se ejecuta la migración. Por lo general, se utiliza para crear tablas, agregar columnas o realizar otros cambios en la estructura de la base de datos.

**down:** Este método especifica las operaciones que deben realizarse cuando se revierte la migración. Es útil para deshacer los cambios realizados en el método up.

El comando para ejecutar las migraciones una vez creadas y modificadas método **UP**:

\*/ php artisan migrate

El comando para revertir las migraciones método **down:**

\*/ php artisan migrate:rollback

Para nuestro ejemplo agregaremos un campo a la tabla que crearemos llamado message el código queda así

public function up(): void

    {

        Schema::create('chirps', function (Blueprint $table) {

            $table->id();

            $table->string('message');

            $table->timestamps();

        });

    }

Ahora necesitamos vincular el mensaje con quien lo escribió para eso vamos a realizar una relación con foreignId()

Detallamos los datos

foreugnId: establece el nombre del campo que se va a relacionar o llave foránea

constrained: restringe la llave foránea para asegurar la integridad referencial

cascadeOnDelete: para eliminar los registros que acompañan al registro de la llave foránea.

public function up(): void

    {

        Schema::create('chirps', function (Blueprint $table) {

            $table->id();

            $table->string('message');

            $table->foreignId('user\_id')->constrained()->cascadeOnDelete();

            $table->timestamps();

        });

    }

## Eloquent

Eloquent es el ORM (Object-Relational Mapping) incluido en Laravel. Proporciona una implementación activa de los patrones ActiveRecord para trabajar con bases de datos relacionales. En otras palabras, Eloquent permite interactuar con la base de datos utilizando objetos en lugar de escribir consultas SQL directamente.

Lo modelos pertenecen a Eloquent y me permiten

* User::get()
* User::where()
* User::orderBy()
* User::créate()
* User:;update()
* User::delete()

Para usar el modelo nos vamos al archivo de rutas web.php y lo importamos

use App\Http\Controllers\ProfileController;

use Illuminate\Support\Facades\Route;

use App\Models\Chirp;

Ahora en la ruta post de nuestro formulario vamos a usar el modelo créate de eloquent

Route::post('/chirps', function(){

        //Insertar en la base de datos

        Chirp::create([

            'message' => request('messege'),

            'user\_id' => auth()->id(),

        ]);

        return to\_route('chirps.index');

    });

Explicamos mandamos a llamar al modelo Chirp con su propiedad créate dento le pusimos un arreglo que nos incluye el nombre del campo de la tabla y se le asigna el request que ya explicamos el cual lo toma del name del elemento de nuestro formulario y el método auth que toma el id del usuario.

Si ahorita ejecutamos nos mandara un error esto por que no hemos activado en el Modelo el método fillable.

Para que esto funcione además en el modelo Chirp debemos agregar el método fillable que me permitirá identificar cuales son los campos permitidos.

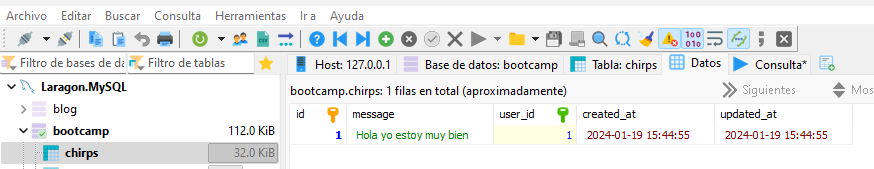
protected $fillable = [

        'message',

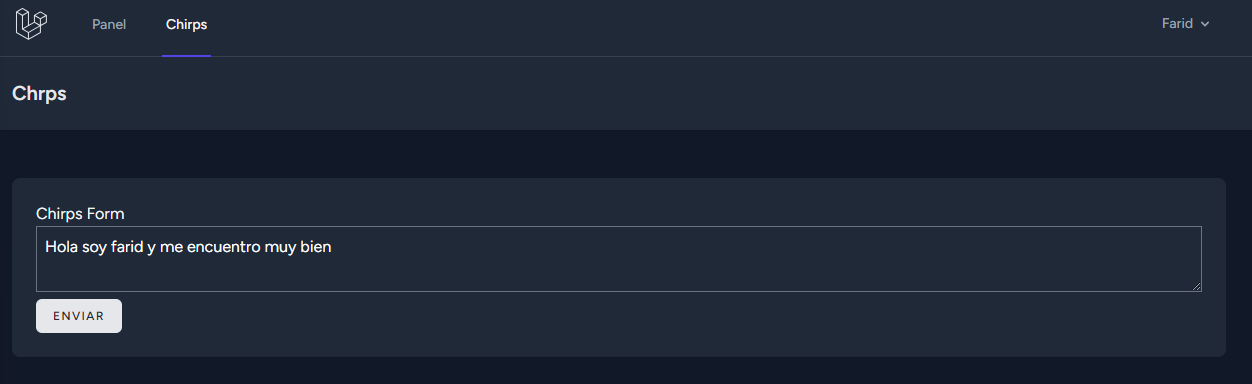
        'user\_id',

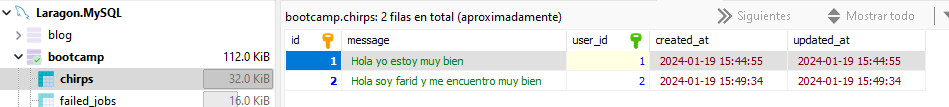
    ];

Ejecutamos nuestro formulario y notamos que el registro fue enviado



Ahora probemos creando otro usuario para que el registro tenga un id = 2 y probemos nuestros formulario.





Ya Tenemos un Formulario, una ruta, una migración, un modelo y podemos insertar datos en una tabla.

## Mandar un mensaje de comprobación sobre acción de sesión

Cuando ejecutamos el registro en nuestra base de datos sabemos que funciona todo pero es una buena práctica indicarlo al usuario y esto puede ser mediante un mensaje emergente.

Paso 1: En la ruta web.php después de guardar el dato y antes de mandar la redirección ponemos el mensaje de sesión flash esto hace que en cada actualización de pagina o sitio este se borre.

Route::post('/chirps', function(){

        //Insertar en la base de datos

        Chirp::create([

            'message' => request('messege'),

            'user\_id' => auth()->id(),

        ]);

        //Mensaje de sesion

        session()->flash('status', 'Se ha añadido el registro');

        return to\_route('chirps.index');

    });

Ahora pondremos el mensaje en el archivo de navegación ya que este se encuentra presente en todo nuestro sitio se coloca hasta arriba en el inicio.

Para que veamos en modo producción nuestras clases debemos ejecutar en la consola

\*/ npm run dev

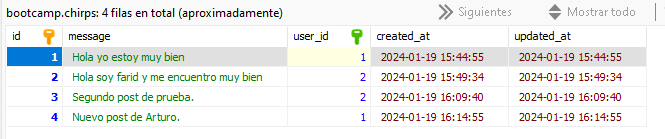
La url es nombre del proyecto.test en nuestro caso bootcamp.test

@if (session('status'))

    <div class="bg-green-600 text-green-100 text-center text-lg p-2">{{ session('status')}}</div>

@endif





## Controladores

En Laravel, los controladores son clases PHP que manejan la lógica de la aplicación. Su propósito principal es recibir las solicitudes HTTP del usuario, procesar esa solicitud y devolver una respuesta adecuada. Los controladores ayudan a organizar la lógica de la aplicación y mantienen el código más modular y fácil de mantener.

Con los controladores evitamos tener en nuestras rutas tanto código, los controladores incluyen las funciones que requerimos por ejemplo e un crud y en las rutas nada mas las llamamos y listo creamos un efectivo sistema de MVC.

Paso 1: Importamos el controlador en el archivo de rutas web.php

use App\Http\Controllers\ChirpController;

Paso 2: Modificamos la ruta con la llamada al controlador

Route::get('/chirps', [ChirpController::class, 'index'])->name('chirps.index');

* Tenemos el método: get
* Tenemos la ruta que se usa en el navegador: /chirps
* Tenemos el Controlador en un array con el nombre y la función: [ChirpController::class, 'index']
* Tenemos el name: 'chirps.index'

Paso 3: Modificamos nuestro controller y ponemos nuestro código

public function index()

    {

        return view('chirps.index');

    }

Lo mismo hacemos para el de agregar registro solo que aquí realizamos

Paso 1: modificamos nuestra ruta y agregamos un name esto para que en el formulario lo coloquemos como su action

Route::post('/chirps', [ChirpController::class, 'store'])->name('chirps.store');

* Tenemos el método: post
* Tenemos la ruta que se usa en el navegador: /chirps
* Tenemos el Controlador en un array con el nombre y la función: [ChirpController::class, 'store']
* Tenemos el name: 'chirps.store'

Paso 2: Modificamos el form del index.blade.php

<form method="post" action="{{route('chirps.store')}}">

Paso 3: Creamos nuestra función en el controller en store pero observamos que utiliza request así que usamos esa variable y le pedimos que obtenga message con get.

public function store(Request $request)

    {

        //Insertar en la base de datos

        Chirp::create([

            'message' => $request->get('messege'),

            'user\_id' => auth()->id(),

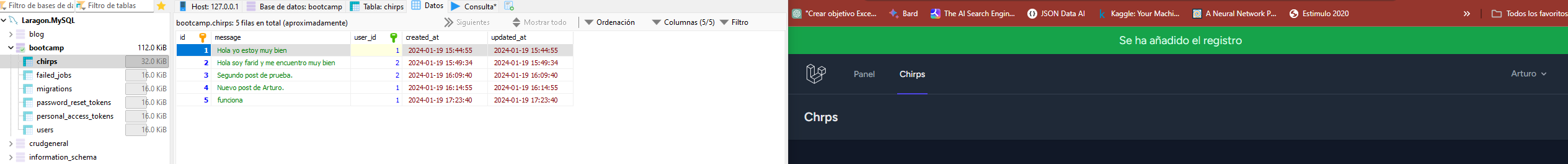
        ]);

        //Mensaje de sesion

        session()->flash('status', 'Se ha añadido el registro');

        return to\_route('chirps.index');

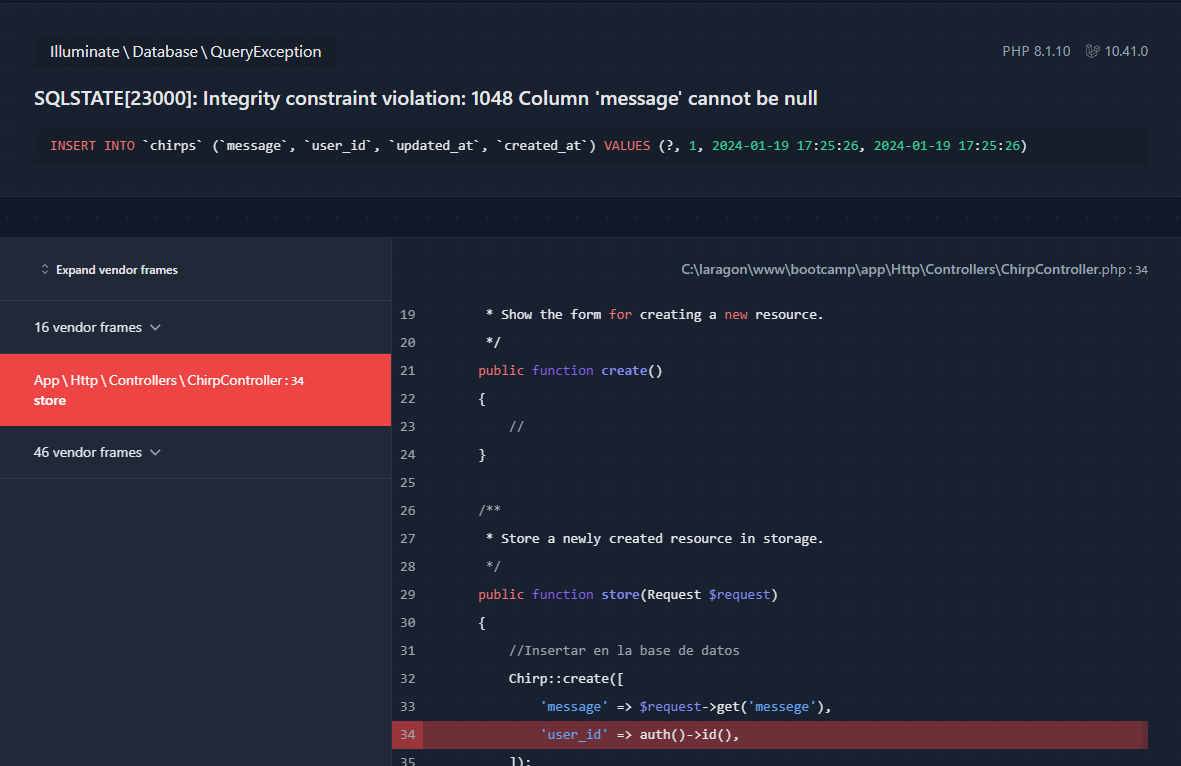
    }



Ahora tenemos algo más limpio y estructurado

# Validación de Formularios

Actualmente si nosotros ejecutamos nuestro formulario sin poner algún dato en la textarea laravel nos mandaría este error eso por que en las reglas de la tabla se indico que message no puede estar en null.



Para validar datos en laravel es muy sencillo se hace en 2 pasos.

Paso 1: Colocamos un array con las variables que queremos validar y las metemos en un requets->validate

 //Validación de las variables

        $request->validate([

'message' => 'required',

]);

En este caso en el array de message se le ponen las validaciones para nuestro caso es que es requerido y que debe tener 3 caracteres mínimo.

Paso 2: Agregamos en nuestro formulario debajo del elemento que vamos a validar el siguiente código

<textarea placeholder="{{\_\_('Whats on you mind?')}}" class="bg-gray-800 text-white border border-gray-700 p-2 w-full h-24" name="messege"></textarea>

<x-input-error : messages = "$errors->get('message')"/>

<x-primary-button>Enviar</x-primary-button>

# BASE DE DATOS RELACIONAL CON ELOQUENT

Para hacer registros en BD relacionadas debemos agregar en los modelos la relación o mejor dicho el tipo, para este ejemplo usaremos hasMany que se refiere de 1 a muchos. Editaremos un poco nuestro ChirpController y el modelo de User, en el modelo User coloaremos la relación y el Controller la consumiremos el código queda así.

Archivo Model -> Chirp

Quitamos el user id ya que se tomará directamente de la relación y hará una asignación masiva

class Chirp extends Model

{

    use HasFactory;

    protected $fillable = [

        'message',

    ];

}

Archivo Model -> User vamos a definir la relación

   //Agregamos la relación hasMany entre Chirp y User

    public function chirps(): HasMany{

        return $this->hasMany(Chirp::class);

    }

Archivo Controller -> ChripController vamos a guardar el registro

public function store(Request $request)

    {

        //Validación de las variables

        $request->validate([

            'message' => 'required',

        ]);

        //Insertar en la base de datos

        /\*Chirp::create([

            'message' => $request->get('message'),

            'user\_id' => auth()->id(),

        ]);\*/

        //Insertar datos en tablas relacionadas

        $request->user()->chirps()->create(['message' => $request->get('message'),]);

        //Mensaje de sesion

        session()->flash('status', 'Se ha añadido el registro');

        return to\_route('chirps.index');

    }

## Listado de chirps

Vamos a listar el contenido de nuestra tabla chirps para esto vamos a ocupar la pagina index de nuestro layout chirps y nuestro controller en el método index y un poquito de estilo

Archivo ChirpController

public function index()

    {

        return view('chirps.index', [

            'chirps' => Chirp::latest()->get()

        ]);

    }

Archivo index.blade.php

<div class="mt-6 bg-white dark:bg-gray-800 shadow-sm rounded-lg divide-y dark:divide-gray-900">

                    @foreach($chirps as $chirp)

                        <div class="p-6 flex space-x-2">

                            <svg class="h-6 w-6 text-gray-600 dark:text-gray-400 -scale-x-100" fill="none" stroke="currentColor" stroke-width="1.5" viewBox="0 0 24 24" xmlns="http://www.w3.org/2000/svg" aria-hidden="true">

                                <path stroke-linecap="round" stroke-linejoin="round" d="M7.5 8.25h9m-9 3H12m-9.75 1.51c0 1.6 1.123 2.994 2.707 3.227 1.129.166 2.27.293 3.423.379.35.026.67.21.865.501L12 21l2.755-4.133a1.14 1.14 0 01.865-.501 48.172 48.172 0 003.423-.379c1.584-.233 2.707-1.626 2.707-3.228V6.741c0-1.602-1.123-2.995-2.707-3.228A48.394 48.394 0 0012 3c-2.392 0-4.744.175-7.043.513C3.373 3.746 2.25 5.14 2.25 6.741v6.018z"></path>

                            </svg>

                            <div class="flex-1">

                                <div class="flex justify-between items-center">

                                    <div>

                                        <span class="text-gray-800 dark:text-gray-200">Comentario</span>

                                        <small class="ml-2 text-sm text-gray-600 dark:text-gray-400">{{ $chirp->created\_at->format('j M Y, g:i a') }}</small>

                                    </div>

                                </div>

                                <p class="mt-4 text-lg text-gray-900 dark:text-gray-100">{{ $chirp->message }}</p>

                            </div>

                        </div>

                    @endforeach

                </div>