

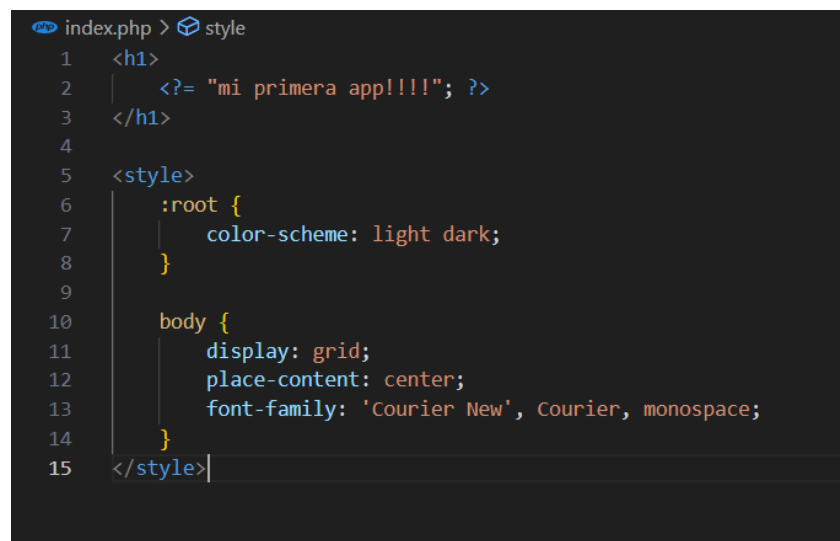
# PHP y Laravel

**PHP** es un lenguaje de tipado dinámico, débil y gradual. Es tipado dinámico porque no es necesario declarar el tipo de la variable y puede cambiar en tiempo de ejecución. Es débil porque va intentar cambiar los tipos automáticamente. Es gradual porque puedes indicar el tipo de las variables (depende del contexto).

## Servidor local en el cmd

Se ejecuta el siguiente comando: **php -S localhost:8000**

- En php puedes escribir tanto html, css y javascript en el mismo archivo .php de la siguiente manera **<?= "Texto"; ?>** (es como escribir **echo**). Ejemplo:

A screenshot of a code editor with a dark background. The editor shows a file named 'index.php' with a 'style' icon. The code is as follows:

```
1 <h1>
2   <?= "mi primera app!!!!"; ?>
3 </h1>
4
5 <style>
6   :root {
7     color-scheme: light dark;
8   }
9
10  body {
11    display: grid;
12    place-content: center;
13    font-family: 'Courier New', Courier, monospace;
14  }
15 </style>
```

## Variables

- Para declarar variables tenemos que utilizar el signo dólar (\$)

```
$name = "Samuel";
```

```
$isDev = true;
```

```
$age = 39;
```

- **Constantes locales:** no se puede usar el signo de dólar.

```
const NOMBRE = 'MIGUEL';
```

- **Constantes globales:**

```
define('IMAGEN_LOGOS', 'https://cdn-assets-eu.frontify.com/s3,
```

## Operadores

- **El operador `+`:** Este operador se utiliza solamente para realizar operaciones aritméticas. Convierte automáticamente cadenas de texto que representan números en valores numéricos para realizar la suma. Por ejemplo, `"10" + 5` devolverá `15`.
- **El operador `.`:** Este operador se usa exclusivamente para concatenar cadenas de texto. A diferencia del operador `+`, no realiza conversión de tipos, sino que combina directamente las cadenas. Por ejemplo, `"Hola" . " Mundo"` devolverá `"Hola Mundo"`.
- **El operador `==`:** Compara dos valores para verificar si son iguales, sin tener en cuenta los tipos de datos. Por ejemplo, `"5" == 5` devolverá `true`.
- **El operador `===`:** Realiza una comparación estricta que evalúa tanto el valor como el tipo de dato. Por ejemplo, `"5" === 5` devolverá `false`.
- **El operador `!=` o `<>`:** Verifica si dos valores son diferentes. Por ejemplo, `10 != 20` devolverá `true`.
- **El operador `!==`:** Realiza una comparación estricta de desigualdad, evaluando también el tipo de dato. Por ejemplo, `"10" !== 10` devolverá `true`.
- **Los operadores lógicos:** Como `&&` (AND), `||` (OR) y `!` (NOT), se utilizan para realizar operaciones lógicas entre expresiones. Por ejemplo, `(true && false)` devolverá `false`.

## Condicionales

Funciona igual que en javascript, pero hay una particularidad aprovechando las plantillas que te proporciona php. Y **elseif es mejor que vaya pegado**. Ejemplo:

```
<?php if ($isOld): ?>
    <h2>Eres viejo, lo siento</h2>
<?php elseif ($isDev): ?>
    <h2>No eres viejo, pero eres Dev. Ta' bien</h2>
<?php else: ?>
    <h2>Eres un pelaito</h2>
<?php endif; ?>
```

## Ternarias

Funcionan como un condicional, si es true se ejecuta lo primero, sino lo segundo (después de los dos puntos). Ejemplos:

```
$edad = 25;
$mensaje = ($edad >= 18) ? "Eres mayor de edad" : "Eres menor de edad";
echo $mensaje; // Imprimirá: Eres mayor de edad
```

```
$numero = 10;
$resultado = ($numero % 2 == 0) ? "El número es par" : "El número es impar";
echo $resultado;
```

## Match

Funciona como if, elseif y else, pero se ve mucho mejor y son menos líneas de código. Ejemplo:

```
$outputAge = match (true){
    $age < 2 => "Eres un bebe",
    $age < 10 => "Eres un niño",
    $age < 18 => "Eres un adolescente",
    $age == 18 => "Eres mayor de edad",
}
```

```
$age <40 => "Eres un adulto joven",  
$age >= 40 => "Ya estas viejito",  
default => "Huele a madera",  
}
```

## Array

```
//Creando un array  
$bestLanguages = ["PHP","JacaScript","Python"];  
  
//Ingresar datos a la variable en un indice especifico  
$bestLanguages[3] = "Java";  
  
//Anadir datos en la ultima posicion  
$bestLanguages[] = "TypeScript";
```

## Array asociativo (diccionarios)

Un **array asociativo** en PHP es una estructura de datos que, en lugar de utilizar índices numéricos para acceder a sus elementos, utiliza **claves** (o keys) que pueden ser cadenas de texto o números. Estas claves sirven como identificadores únicos para cada valor almacenado en el array.

```
$datos_persona = [  
    "nombre" => "Juan Pérez", // String  
    "edad" => 30, // Entero  
    "hobbies" => ["leer", "programar", "viajar"], // Array simple  
    "altura" => 1.75, // Flotante  
    "casado" => false // Booleano  
];
```

## Ciclos

## 1. **while**

El ciclo **while** ejecuta un bloque de código mientras una condición sea verdadera. Es útil cuando no sabes de antemano cuántas iteraciones se necesitan.

### **Sintaxis:**

```
php
CopiarEditar
while (condición) {
    // Código a ejecutar
}
```

### **Ejemplo:**

```
php
CopiarEditar
$i = 1;
while ($i <= 5) {
    echo "Iteración $i<br>";
    $i++;
}
```

### **Resultado:**

Imprime las líneas:

```
CopiarEditar
Iteración 1
Iteración 2
Iteración 3
Iteración 4
Iteración 5
```

## 2. `do...while`

El ciclo `do...while` ejecuta el bloque de código al menos una vez, incluso si la condición no se cumple, ya que la condición se evalúa después de ejecutar el código.

### Sintaxis:

```
php
CopiarEditar
do {
    // Código a ejecutar
} while (condición);
```

### Ejemplo:

```
php
CopiarEditar
$i = 1;
do {
    echo "Iteración $i<br>";
    $i++;
} while ($i <= 5);
```

### Resultado:

Imprime lo mismo que el ciclo `while`, pero asegura que el bloque de código se ejecute al menos una vez.

## 3. `for`

El ciclo `for` se utiliza cuando se sabe cuántas veces debe repetirse un bloque de código. Es más compacto que `while` y `do...while`.

### Sintaxis:

```
php
CopiarEditar
for (inicialización; condición; incremento/decremento) {
    // Código a ejecutar
}
```

### Ejemplo:

```
php
CopiarEditar
for ($i = 1; $i <= 5; $i++) {
    echo "Iteración $i<br>";
}
```

### Resultado:

Produce el mismo resultado que los ejemplos anteriores.

## 4. **foreach**

Gracias al sistema de plantillas podemos iterar un array para distintas funciones como por ejemplo:

```
<ul>
    <?php foreach ($bestLenguajes as $language): ?>
        <li><?= $language ?></li>
    <?php endforeach; ?>
</ul>
```

## Llamadas a APIs

1. **Usando curl (sin dependencias):** es la forma mas básica que hay.

```

<?php

const API_URL = "https://whenisthenextnucufilm.com/api";
# Inicializar una nueva sesión de cURL; ch = cURL handle
$ch = curl_init(API_URL);
// Indicar que queremos recibir el resultado de la petición y no mostrarla en
pantalla
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
/* Ejecutar la petición
   y guardamos el resultado
*/
$result = curl_exec($ch);

* una alternativa sería utilizar file_get_contents
// $result = file_get_contents(API_URL); // si solo quieres hacer un GET de una API

$data = json_decode($result, true);

curl_close($ch);
?>

```

## Funciones

Las funciones en PHP son bloques de código reutilizables que realizan una tarea específica. Permiten organizar y estructurar el código, haciéndolo más legible y mantenible. Aunque no hay información específica sobre funciones en el contexto proporcionado, puedo ofrecer un ejemplo básico de una función en PHP:

```

//Sin indicar de que tipo es el parametro de entrada y el de salida
function saludar($nombre) {
    return "¡Hola, " . $nombre . "!";
}
/*
//Indicando el parametro de entrada y salida
function saludar(string $nombre): string {
    return "¡Hola, " . $nombre . "!";
}
*/

// Uso de la función
$mensaje = saludar("Samuel");
echo $mensaje; // Imprimirá: ¡Hola, Samuel!

```

En este ejemplo, la función 'saludar' toma un parámetro '\$nombre' y devuelve un saludo personalizado. Puedes llamar a esta función pasándole diferentes nombres



como argumento.

- Funcion con match pattern:

Aquí tienes un ejemplo de una función en PHP que utiliza el patrón match para devolver un string:

```
function clasificarEdad(int $edad): string {
    return match (true) {
        $edad < 18 => "Menor de edad",
        $edad >= 18 && $edad < 65 => "Adulto",
        $edad >= 65 => "Adulto mayor",
        default => "Edad no válida"
    };
}

// Uso de la función
echo clasificarEdad(25); // Imprimirá: Adulto
echo clasificarEdad(70); // Imprimirá: Adulto mayor
```

Esta función utiliza el patrón match para clasificar la edad de una persona en diferentes categorías. El patrón match es similar al ejemplo que se muestra en el contexto, pero en este caso, lo hemos adaptado para devolver un string basado en la edad proporcionada.

## Modo Estricto en PHP

PHP ofrece un modo estricto que ayuda a detectar errores comunes y mejora la seguridad del código. Para activar este modo, se utiliza la declaración

`declare(strict_types=1)` al inicio del archivo PHP.

El modo estricto afecta principalmente a la comprobación de tipos en las funciones. Aquí tienes un ejemplo:

```
<?php
declare(strict_types=1);

function sumar(int $a, int $b): int {
```

```

        return $a + $b;
    }

    echo sumar(5, 10); // Funciona correctamente, imprime 15
    echo sumar(5, "10"); // Lanza un TypeError, ya que "10" es un string
    ?>

```

En este ejemplo, la función `sumar` espera dos parámetros de tipo `int` y devuelve un `int`. Con `strict_types=1`, PHP lanzará un error si intentamos pasar un tipo de dato incorrecto, como un string en lugar de un int.

## Importar archivos

En PHP, hay varias formas de importar archivos. **Importante entender que al importar un archivo es como si estuviera copiando y pegando el código de ese archivo al actual.** Las más comunes son:

- **require:** Importa y ejecuta el archivo especificado. Si el archivo no se encuentra, se produce un error fatal y se detiene la ejecución del script.
- **require\_once:** Similar a require, pero PHP verifica si el archivo ya ha sido incluido. Si es así, no lo incluye de nuevo. Esto es útil para evitar redefiniciones de funciones o clases.
- **include:** Similar a require, pero si el archivo no se encuentra, se genera una advertencia y el script continúa ejecutándose.
- **include\_once:** Combina el comportamiento de include con la verificación de require\_once. Incluye el archivo si no ha sido incluido antes, y si no lo encuentra, genera una advertencia.

Ejemplo de uso:

```

// Importar un archivo una sola vez
require_once 'config.php';

// Importar un archivo que es crucial para la aplicación
require 'functions.php';

```

```
// Importar un archivo que no es crítico  
include 'optional_features.php';
```

En general, se recomienda usar `require_once` para archivos que contienen definiciones de funciones o clases, y `require` para archivos críticos que deben estar presentes para que la aplicación funcione correctamente.

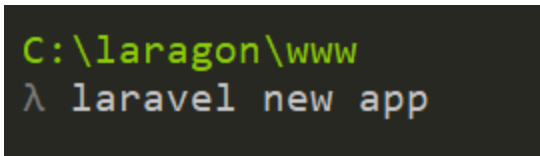
## Laravel

- Laravel es un framework de PHP para el desarrollo web.
- Se caracteriza por su elegante sintaxis y su enfoque en la productividad del desarrollador.
- Ofrece características como ORM (Eloquent), sistema de rutas, motor de plantillas (Blade), y un sistema de migración de bases de datos.
- Laravel sigue el patrón de arquitectura MVC (Modelo-Vista-Controlador).
- Tiene una comunidad activa y una extensa documentación, lo que facilita su aprendizaje y uso.

## Inicializar un proyecto de Laravel (en laragon)

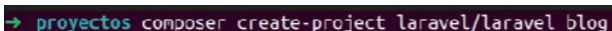
Se debe escribir en la terminal (cmd) de laragon el siguiente comando:

**laravel new** nombre\_del\_proyecto



```
C:\laragon\www  
λ laravel new app
```

En la terminal:



```
→ proyectos composer create-project laravel/laravel blog
```

Para ingresar a la carpeta del proyecto:

```
C:\laragon\www
λ cd app
```

Para ver el proyecto en el navegador encendemos **apache** en laragon (gracias a los host virtuales que nos da laragon) e ingresamos:

⚠ No es seguro app.test

con **ls** vemos las aplicaciones en la terminal y **cls** para limpiar la pantalla

## Rutas

se encuentran en /routes/web.php. Aquí realizo unos ejemplos de lo que se puede hacer en las rutas

```
Route::get('/', function () {
    return 'Home';
    #return view('welcome');
});

#Wildcard o parametro que recibe la funcion anonima
Route::get('/nosotros/{id}',function($id){
    return $id;
});

#para recuperar un segmento de la url
Route::get('/contacto/{id}',function($id){
    return request()->segment(1);
});

#Para agrupar rutas con mismo prefijo en comun
Route::prefix('/company')->group(function(){
```

```
Route::get('/nosotros',function(){
    return 'Nosotros.';
});

Route::get('/contacto',function(){
    return 'Contacto';
});
});
```

**El orden de las rutas importan, si tengo por ejemplo /posts/create y /posts/{id} se debe colocar de primero el de create.**

## Views

Cuando veamos la palabra view sabemos que se esta refiriendo a las plantillas de blade

Las views se encuentran almacenadas en /resources/views. Cuando en las rutas indicamos una vista php va a buscar en ese directorio el archivo. En el siguiente ejemplo pasa eso, busca en el directorio y muestra en el navegador el archivo **welcome.blade.php**

```
Route::get('/', function () {
    return view('welcome');
});
```

## Blade

Es un motor de platillas muy potente

Se utiliza dobles llaves para ejecutar código php. Ejemplo:

```
{{'Hola mundo!'}}
```

**Directivas:** se escriben iniciando con un signo @.

## Pasar variables a las plantillas de blade en las rutas

```

Route::get('/', function () {
    #array en el que suponemos que obtuvimos datos
    $posts = [
        [
            'title' => 'Novedades de laravel 9.',
            'excerpt' => 'Esto es uin texto loco escripto nada r
        ],
        [
            'title' => 'Curso de laravel 9.',
            'excerpt' => 'Esto es uin texto loco escripto nada r
        ],
        [
            'title' => 'Manejo basico de Eloquent',
            'excerpt' => 'Esto es uin texto loco escripto nada r
        ],
    ];

    return view('welcome',[
        'posts' => $posts,
    ]);
});

```

## Manejo Artisan: el CLI de Laravel

**php artisan list** lista todos los comandos que se pueden ejecutar

**php artisan route** muestra todas las rutas que nosotros tengamos en el proyecto

Vamos a necesitar un archivo o modelo de Eloquent por cada tabla de la base de datos con la que nos deseemos comunicar. Para hacer el proceso mas sencillo podemos ejecutar en la terminal el siguiente comando que nos creara un archivo con la clase del modelo.

```
php artisan make:model <nombre_del_modelo>
```

esto creara un archivo en /app/Models con el nombre del modelo especificado

Para crear una clase controladora usamos:

```
php artisan make:controller <nombre_del_controlador>
```

Para crear una clase controladora y generar los métodos al mismo tiempo:

```
php artisan make:controller <nombre_del_controlador> --resource
```

## Controladores.

se llevan la lógica para que no este en las rutas, de la siguiente manera:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class HomeController extends Controller
{
    public function show(){
        $posts = [
            [
                'title' => 'Novedades de laravel 9.',
                'excerpt' => 'Esto es uin texto loco escripto na
            ],
            [
                'title' => 'Curso de laravel 9.',
                'excerpt' => 'Esto es uin texto loco escripto na
```

```

        ],
        [
            'title' => 'Manejo basico de Eloquent',
            'excerpt' => 'Esto es uin texto loco escripto na
        ],
    ];

    return view('welcome',[
        'posts' => $posts,
    ]);
}
}

```

Y la ruta nos queda así:

```

#Definimos un array del controlador al que estamos haciendo refe
#como segundo parametro el metodo que estamos llamando
Route::get('/', [HomeController::class,"show"]);

```

## Tinker

Tinker es una herramienta interactiva de línea de comandos que viene incluida con Laravel. Permite interactuar directamente con la aplicación Laravel desde la consola, lo que facilita la prueba de código, la manipulación de datos y la depuración.

Algunas características y usos comunes de Tinker incluyen:

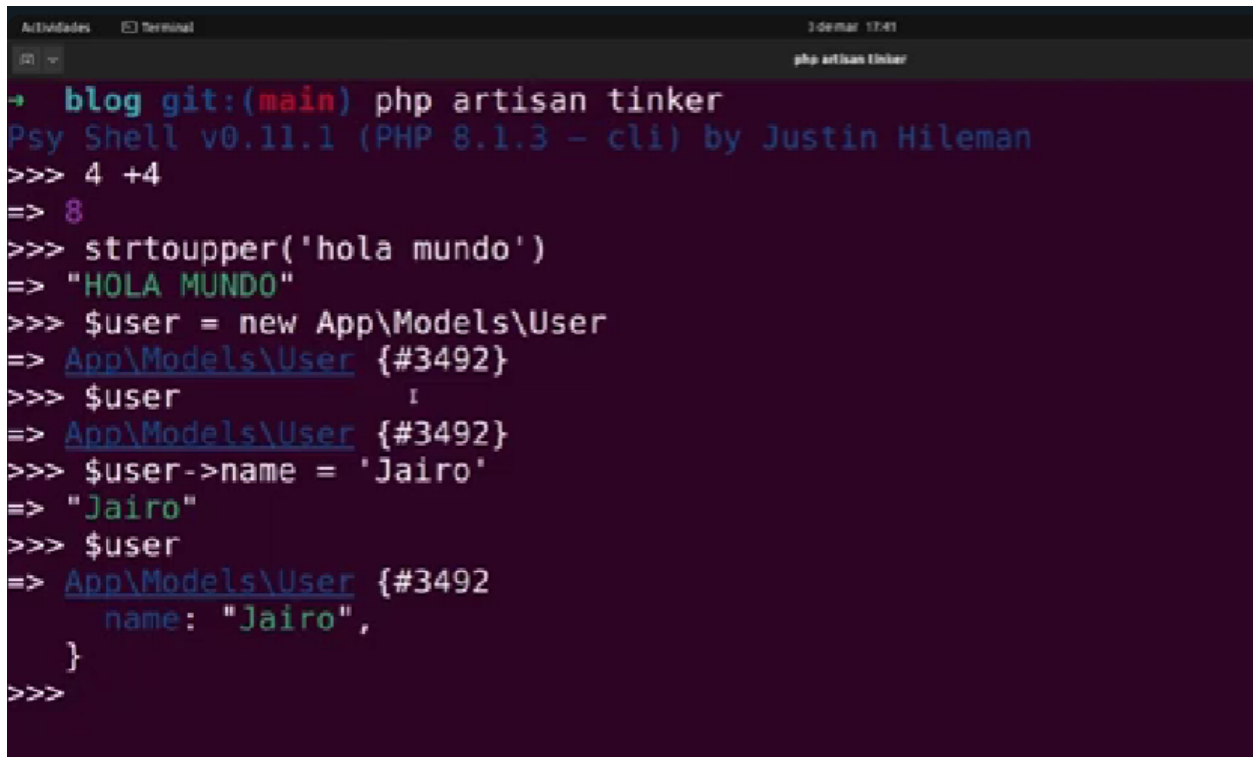
- Interactuar con modelos de Eloquent para crear, leer, actualizar o eliminar registros de la base de datos.
- Probar funciones y métodos de la aplicación sin necesidad de escribir código en los archivos del proyecto.
- Explorar relaciones entre modelos.
- Ejecutar consultas complejas de base de datos.

Para iniciar Tinker, se utiliza el siguiente comando en la terminal:



```
php artisan tinker
```

Una vez dentro de Tinker, puedes escribir y ejecutar código PHP directamente en la consola, lo que lo convierte en una herramienta muy útil para el desarrollo y la depuración de aplicaciones Laravel.



```
→ blog git:(main) php artisan tinker
Psy Shell v0.11.1 (PHP 8.1.3 - cli) by Justin Hileman
>>> 4 +4
=> 8
>>> strtoupper('hola mundo')
=> "HOLA MUNDO"
>>> $user = new App\Models\User
=> App\Models\User {#3492}
>>> $user
=> App\Models\User {#3492}
>>> $user->name = 'Jairo'
=> "Jairo"
>>> $user
=> App\Models\User {#3492
    name: "Jairo",
}
```

para salir de Tinker presionamos Ctrl+c

## Migraciones

Las migraciones en laravel son clases de php que nos permiten recrear y modificar esquemas de bases de datos.

Por defecto laravel ya nos trae algunas de estas migraciones en /database/migrations

Todas las migraciones tienen dos métodos, el método **up** y el método **down**, en el método up definimos la creación o la modificación de una tabla, y en el método Down eliminamos o deshacemos lo que creamos en el método up

Al ejecutar `php artisan migrate` en la terminal va a recorrer todas las migraciones que se encuentran en la carpeta migrations y va ejecutar el método **up** en cada una de ellas

Al ejecutar `php artisan migrate:rollback` en la terminal se van a ejecutar todos los **métodos down** de todas las migraciones del **ultimo lote de migraciones**

El comando `php artisan migrate:fresh` en Laravel realiza las siguientes acciones:

- Elimina todas las tablas de la base de datos.
- Ejecuta todas las migraciones desde cero, creando nuevamente todas las tablas y estructuras de la base de datos.

Este comando es útil cuando quieres reiniciar completamente tu base de datos, eliminando todos los datos existentes y recreando la estructura desde el principio. **ES IMPORTANTE TENER EN CUENTA QUE ESTE COMANDO BORRARÁ TODOS LOS DATOS, POR LO QUE DEBE USARSE CON PRECAUCIÓN, ESPECIALMENTE EN ENTORNOS DE PRODUCCIÓN.**

A diferencia de `php artisan migrate:rollback`, que solo deshace el último lote de migraciones, `migrate:fresh` elimina y recrea toda la estructura de la base de datos.

para hacer un archivo de migración nuevo ejecutamos el comando `php artisan make:migration nombre_de_migracion`. Normalmente por convención se le da el nombre de **create\_nombre\_table**

## Nueva columna en tabla sin perder datos (migration)

Para agregar una nueva columna a una tabla de la base de datos sin perder los datos existentes, puedes utilizar una migración en Laravel. Aquí te explico cómo hacerlo:

1. Crea una nueva migración usando el comando Artisan:

```
php artisan make:migration add_nueva_columna_to_nombre_tabla --table=nombre_tabla
```

2. En el archivo de migración generado, utiliza el método `addColumn` en la función `up` para agregar la nueva columna:

```
public function up()
{
    Schema::table('nombre_tabla', function (Blueprint $table) {
        $table->string('nueva_columna')->nullable(); // Ajusta el tipo de dato
    });
}
```

3. En la función `down`, agrega el código para revertir los cambios si es necesario:

```
public function down()
{
    Schema::table('nombre_tabla', function (Blueprint $table) {
        $table->dropColumn('nueva_columna');
    });
}
```

4. Ejecuta la migración con el comando:

```
php artisan migrate
```

Este proceso agregará la nueva columna a tu tabla sin afectar los datos existentes. Es importante recordar que las migraciones en Laravel nos permiten modificar esquemas de bases de datos de manera segura y controlada.

Si necesitas revertir esta migración en particular, puedes usar el comando:

```
php artisan migrate:rollback --step=1
```

Esto ejecutará el método `down` de la última migración, eliminando la columna que acabas de agregar.

# Eloquent el ORM de laravel (Object-relational mapping)

En laravel tenemos un orm llamado **Eloquent** donde cada tabla de la base de datos tiene un modelo correspondiente que nos permitirá interactuar con ella. Podremos leer, crear, actualizar y eliminar registros de las bases de datos de una forma orientada a objetos

Un modelo de Eloquent es simplemente una clase de php que se extiende de la clase **Illuminate\Database\Eloquent\Model** para heredar todas las funcionalidades

## Modelos Eloquent

Se puede hacer un modelo con el comando `php artisan make:model NombreModelo` (por convención el nombre se escribe **PascalCase**) por convención el nombre va en **singular y mayúscula**

Como todo modelo necesitara una migración tenemos un comando que nos facilita el trabajo con el que podemos hacer el modelo y la migración que es `php artisan make:model NombreModelo -m`

## Convenciones Modelo Eloquent

Modelos generados por el `make:model` el comando se colocará en el directorio `app/Models`. Examinemos una clase modelo básica y discutamos algunas de las convenciones clave de Eloquent:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
```

```
// ...  
}
```

## Nombres de Tabla

Después de mirar el ejemplo anterior, es posible que haya notado que no le dijimos a Eloquent qué tabla de base de datos corresponde a nuestra modelo `Flight`. Por convención, el "snake case", **el nombre plural de la clase** se utilizará como el nombre de la tabla a menos que se especifique explícitamente otro nombre. Entonces, en este caso, Eloquent asumirá que el modelo `Flight` almacena registros en la tabla `flights`, mientras que un modelo `AirTrafficController` almacenaría registros en una tabla `air_traffic_controllers`.

Si la tabla de base de datos correspondiente a su modelo no se ajusta a esta convención, puede especificar manualmente el nombre de la tabla del modelo definiendo una propiedad `table` en el modelo:

```
<?php  
  
namespace App\Models;  
  
use Illuminate\Database\Eloquent\Model;  
  
class Flight extends Model  
{  
    /**  
     * The table associated with the model.  
     *  
     * @var string  
     */  
    protected $table = 'my_flights';  
}
```

## Funcionalidades básicas de Eloquent en Tinker

```
Post::get();  
Post::find($id);  
$post->save();  
$post->delete();
```

Con get obtenemos todos los registros de la base de datos, con find buscamos un registro con el id especificado, con save guardamos los cambios que hallamos hecho (como cambiar el campo a algun registro o guardar un registro nuevo y finalmente con delete borramos un registro de la base de datos.

```
>>> Post::find(1)  
=> App\Models\Post {#4436  
    id: 1,  
    title: "First Post",  
    body: "Content",  
    created_at: "2022-05-17 12:11:21",  
    updated_at: "2022-05-17 12:11:21",  
}
```

```
Terminal: Local x +  
>>> $post->title = "Modified title";
```

```
> $post = new Post;  
= App\Models\Post {#5145}  
  
> $post->title = 'first post'  
= "first post"  
  
> $post->body = 'second post body'  
= "second post body"  
  
> $post->save()  
= true
```

## Buenas Practicas

Es buena idea acceder a las rutas por nombres en lugar de escribir directamente las url.

En las rutas se tiene que agregar por ejemplo:

```
Route::get('/posts/{id}', "show")->name('posts.show');
```

Y en las vistas las buscamos de la siguiente manera:

```
<a href={{route('posts.show',$usuario->id)}}>
```

Si la ruta necesita un id por ejemplo, se lo pasamos como segundo parámetro en la vista (en route)

Es importante que en todos los formularios con el método Post agreguemos la directiva @csrf. Ejemplo:

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

## Validaciones

para ver todos los valores de validaciones disponibles ir a <https://laravel.com/docs>