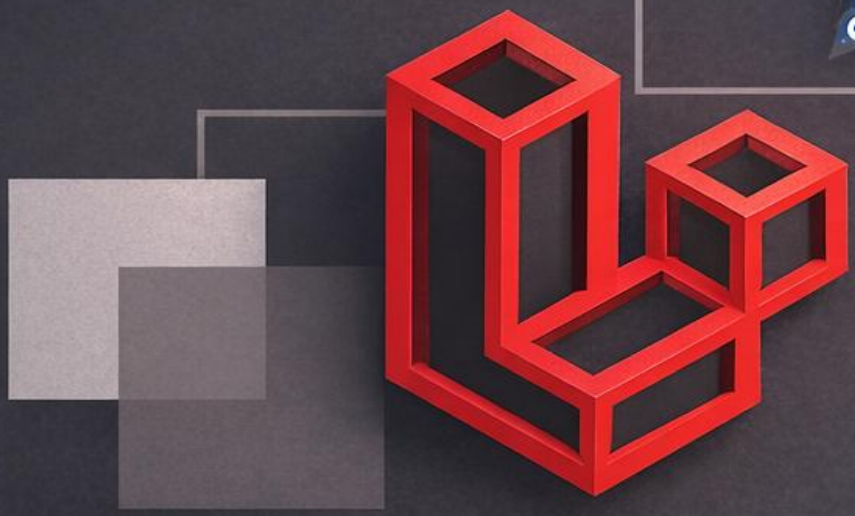
FUNDAMENTOS ARQUITECTÓNICOS DE

Laravel



Análisis estructural y desarrollo de aplicaciones web basadas en el framework Laravel

Wilson Mamani Rodrigo
Kevin Arturo Quispe Apaza
Julio César Llocllli Champi

Wilson Mamani Rodrigo, editor
2026

LARAVEL

Wilson Mamani Rodrigo
Kevin Arturo Quispe Apaza
Julio César Lloclli Champi

HOJA DE CRÉDITOS

Título de la obra:
Fundamentos Arquitectónicos de Laravel: Análisis estructural y desarrollo de aplicaciones web basadas en el framework Laravel

Autores:
Wilson Mamani Rodrigo
Kevin Arturo Quispe Apaza
Julio César Lloclli Champi

Editado por:
Wilson Mamani Rodrigo
Domicilio: Av. Garcilazo del Vega Calle 01 LT. 3, Abancay - Perú
Correo electrónico: rodrigo.peruvian@gmail.com

Edición:
Primera edición digital, abril 2026

Publicación digital disponible en: https://nova.innovationunamba.tech/

Año de publicación:
2026

Lugar de publicación:
Perú

ISBN: _____
Depósito Legal: _____

Primera edición publicada en el Perú, 2026.

ÍNDICE DE CONTENIDO

CAPÍTULO 1

INTRODUCCIÓN A LARAVEL

Laravel es un Framework de PHP, en esta unidad didáctica aprenderemos:

- Desarrollar aplicaciones WEB con Laravel 9
- Instalación de herramientas
- Rutas
- Vistas
- El motor de plantillas de Blade
- Los controladores
- Las migraciones de base de datos
- El uso de Eloquent ORM (CRUD)
- Mensajes de sesión
- Autenticación (Login, registro)

1.1 FRAMEWORK

Es una estructura base para poder elaborar un proyecto de software, algunos suelen tener módulos, programas, bibliotecas, entre otras herramientas. Todo esto para poder simplificar una tarea o proceso de desarrollo de software.

Laravel es un Framework de aplicaciones web con una sintaxis expresiva y elegante. Y éste trabaja con PHP y también trabaja con la arquitectura MVC (Modelo Vista Controlador). Y éste Framework está basado en esta arquitectura para que usted pueda desarrollar desde una aplicación web muy sencilla hasta una aplicación más compleja.

1.2 CONOCIMIENTOS REQUERIDOS

- Lo básico de PHP
- MySql
- POO y MVC
- HTML, CSS, JAVASCRIPT (Básico)

1.3 REQUISITOS DE INSTALACIÓN

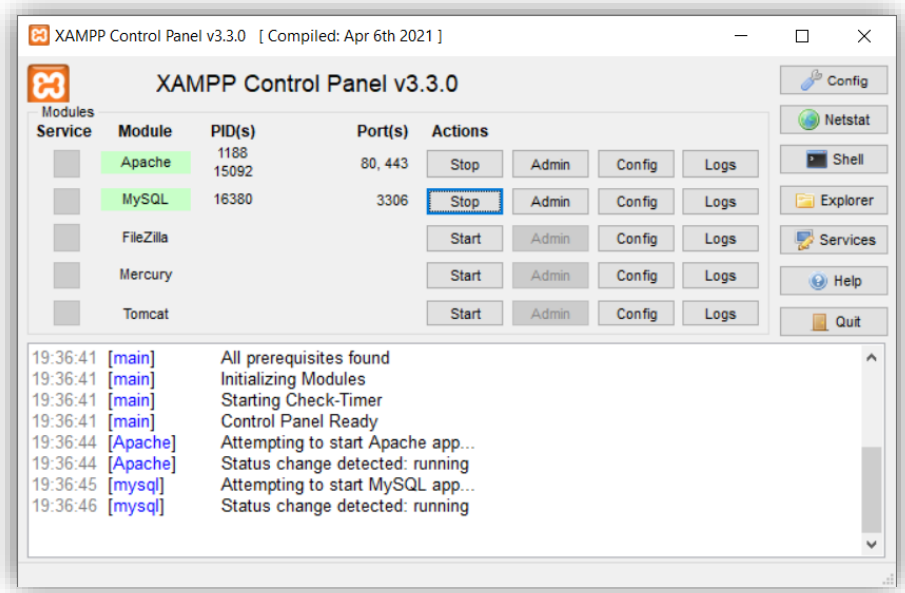
- PHP versión 8 o superior, el 7 ya no es compatible, a menos que usemos Laravel versión 8, pero en este caso utilizaremos Laravel 9
- Necesitaremos APACHE SERVER
- MySql 5.7 o superior
- XAMPP o AppServ (ya trae instalado APACHE, PHP y MySql entre otras herramientas)
- En la configuración de PHP vamos a necesitar algunas extensiones muy específicas, para la cual conforme avancemos con la Unidad Didáctica veremos cual usaremos.

CAPÍTULO 2

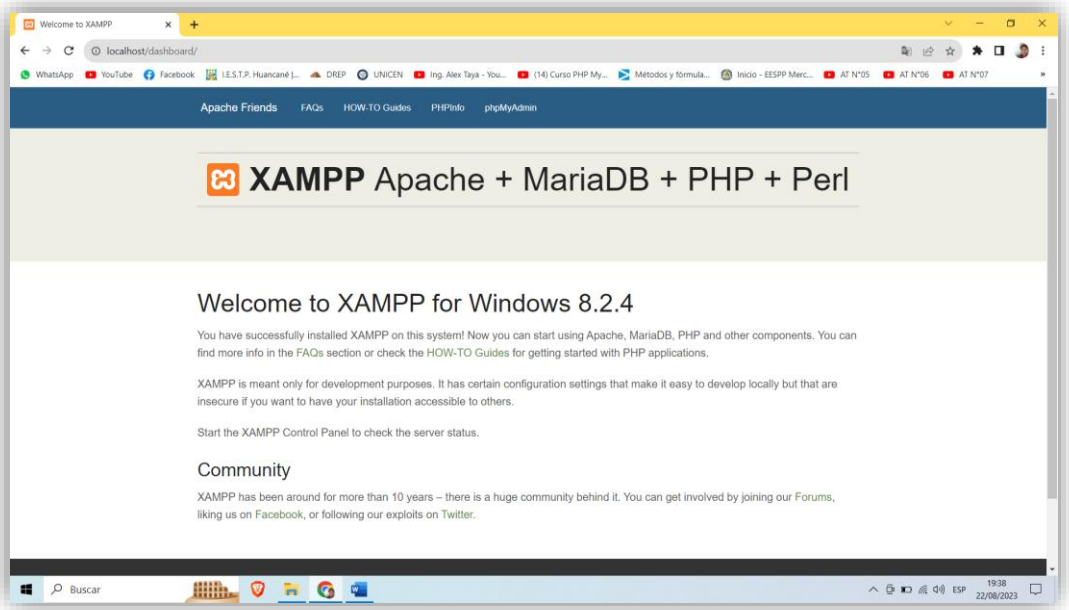
INSTALACIÓN DE LARAVEL

Lo primero que debemos instalar es XAMPP o APPSERV, luego debemos de probar si nuestro servidor funciona. Pero antes debe de fijarse en las versiones de sus programas. En este caso utilizaremos el XAMPP.

Inicializar el apache y el MySql.

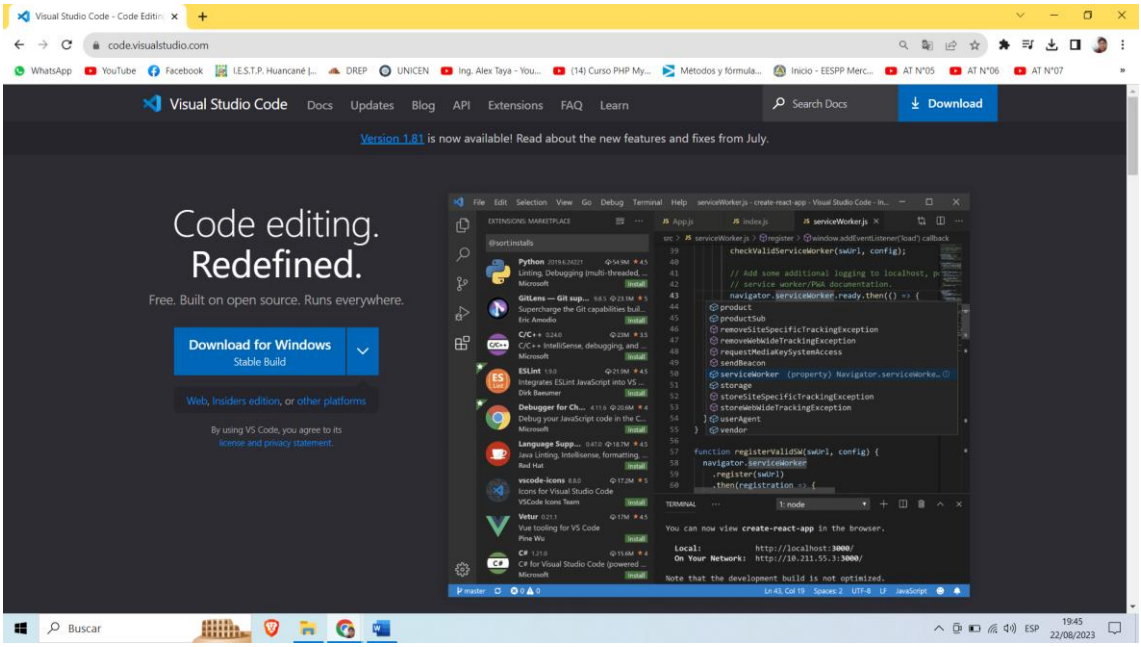


Probar el servidor.

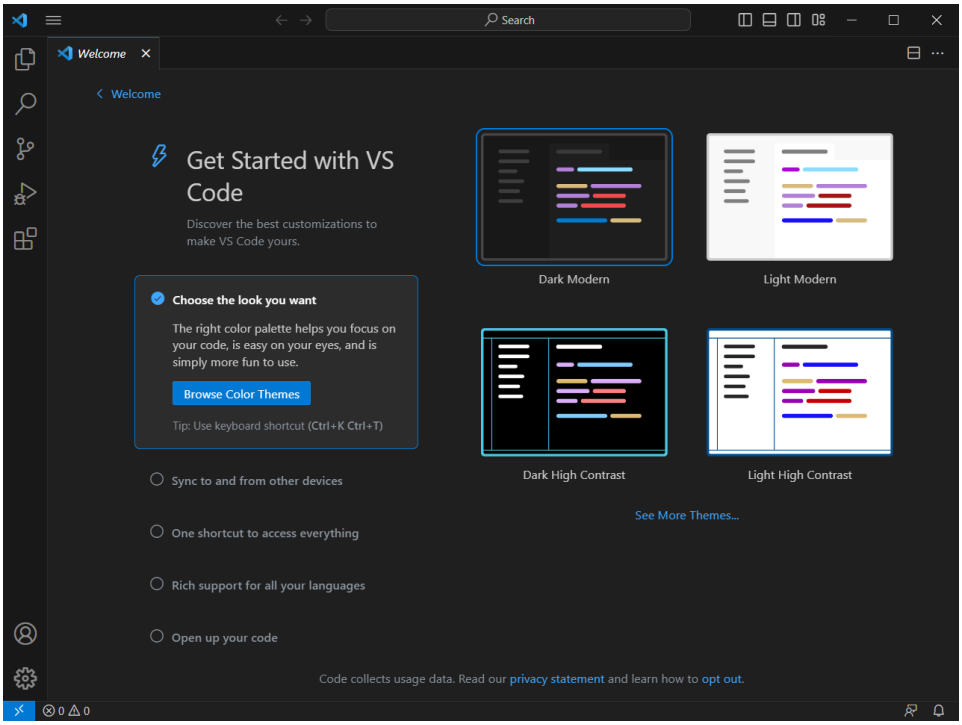


2.1 INSTALACIÓN DE VISUAL ESTUDIO CODE

Luego vamos a necesitar un editor de código, en este caso vamos a trabajar con Visual Studio Code, por su popularidad y por su interfaz amigable para trabajar con Laravel, Usted puede trabajar con cualquier editor de código (Block de notas, Sublime Text, Atom, etc.)

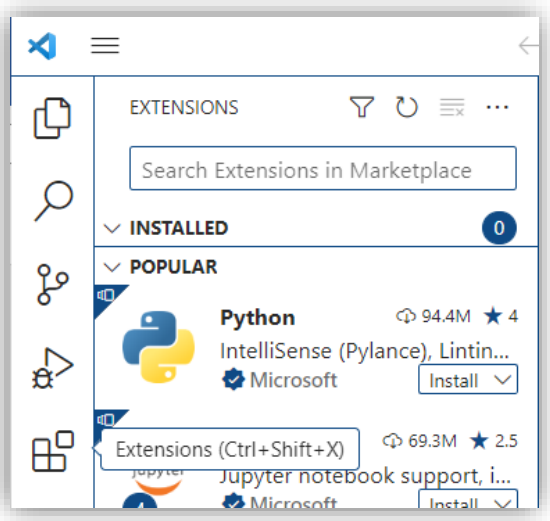


Cuando inicie se verá así: (puede elegir el tema de su preferencia)

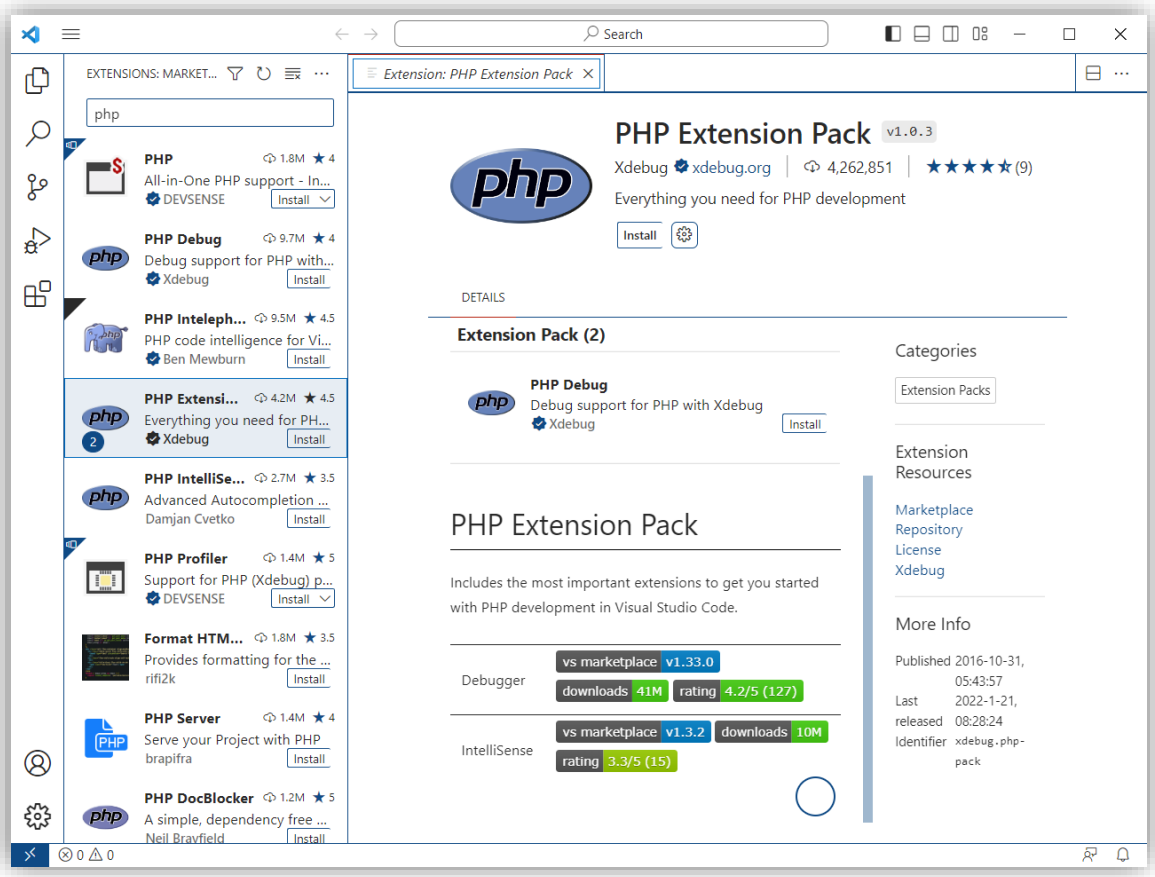


2.2 AGREGAR EXTENSIONES PARA QUE DETECTE EL LENGUAJE PHP

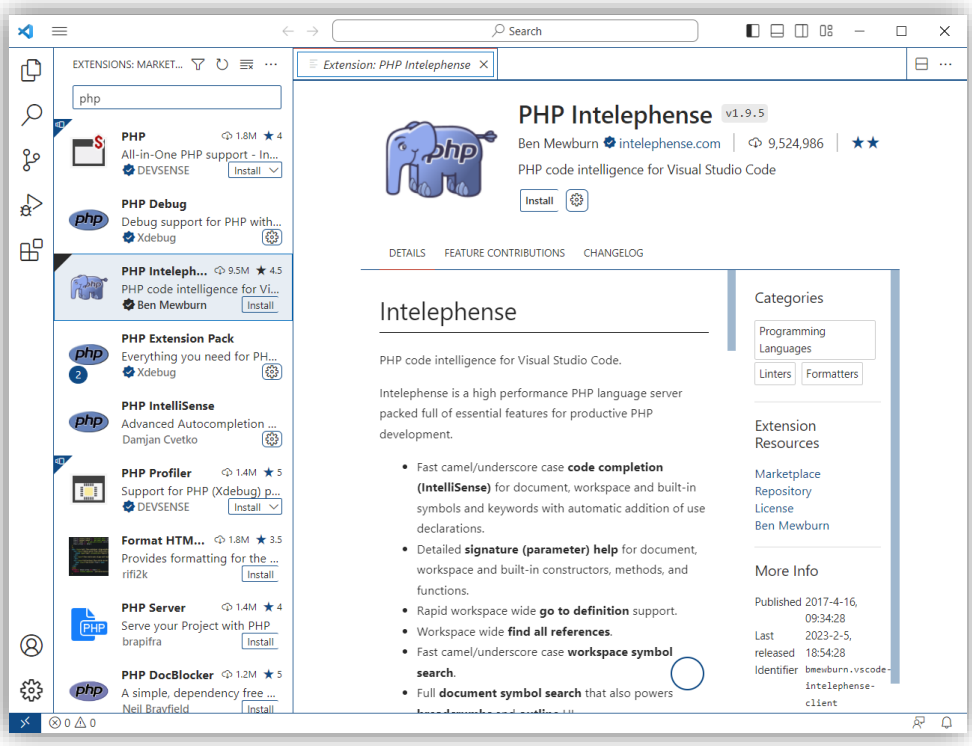
En el Visual Studio Code, ir al apartado de extensiones.



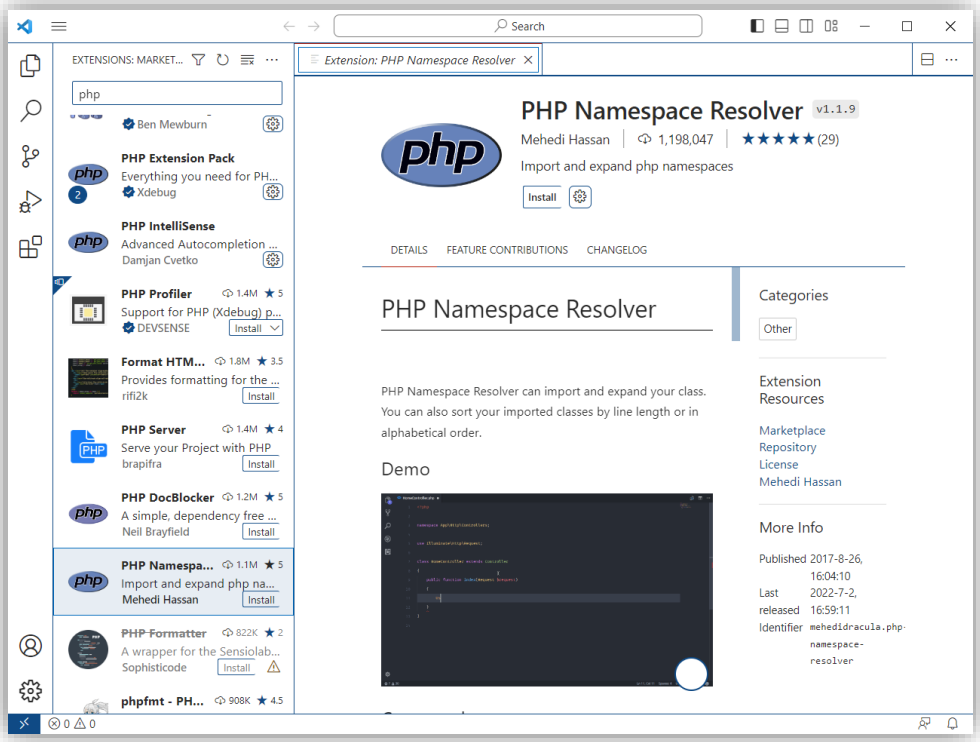
Escribimos PHP y añadiremos las extensiones más importantes en PHP: PHP Extension Pack (luego le damos clic en instalar)



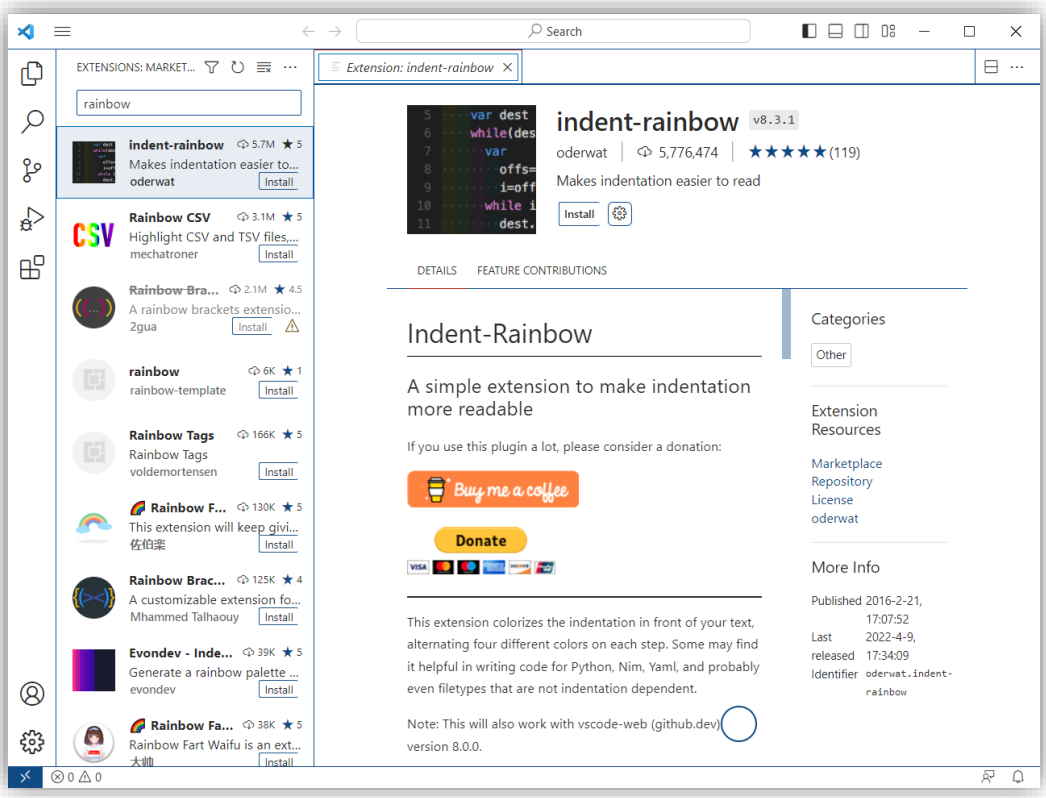
También instalaremos el PHP Intelephense



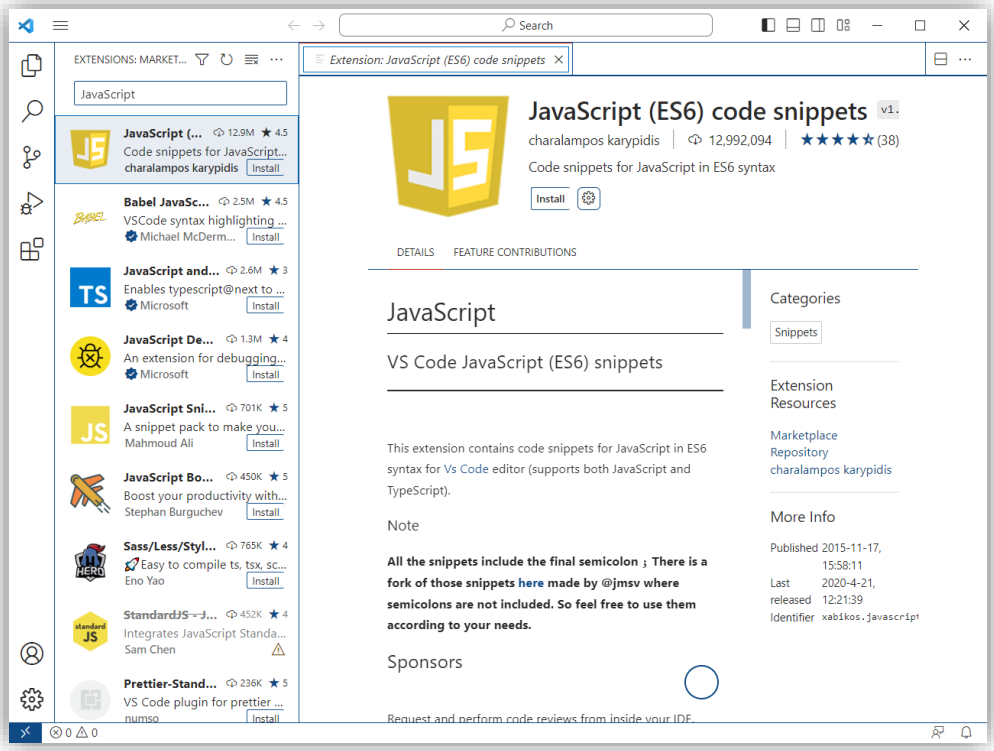
También instalaremos el PHP Namespace Resolver, que es el que nos crea un espacio de nombres e importa de forma automática.



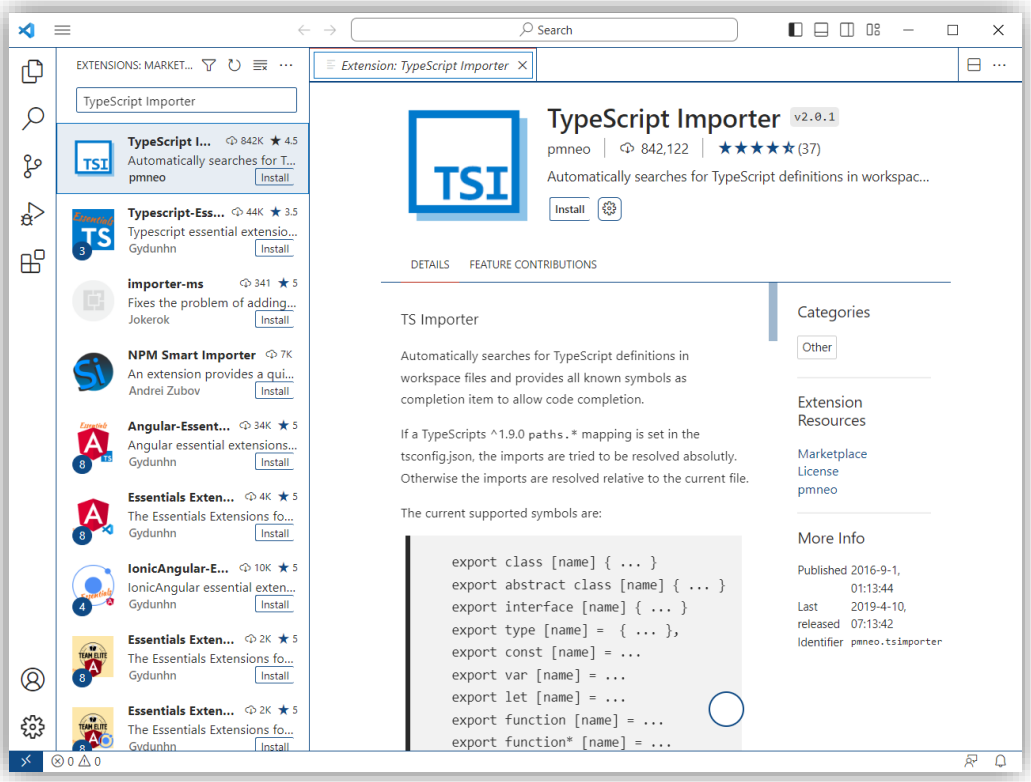
También instalamos indent-rainbow para los colores, o inicios o cierres de cada función.



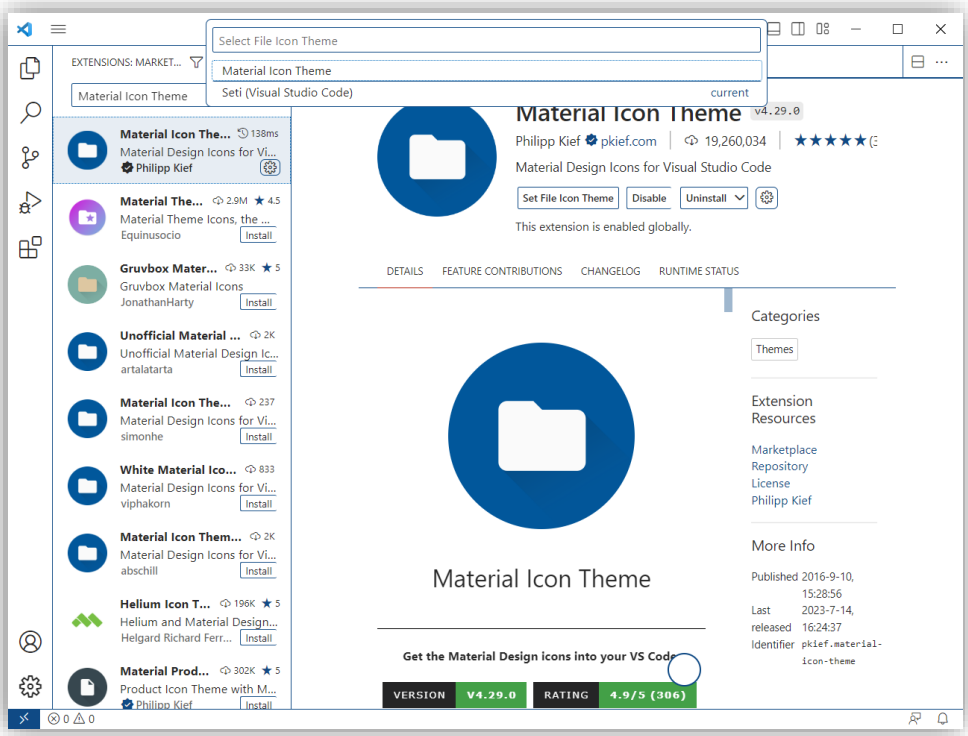
También instalaremos JavaScript (ES6) code snippets para las páginas web.



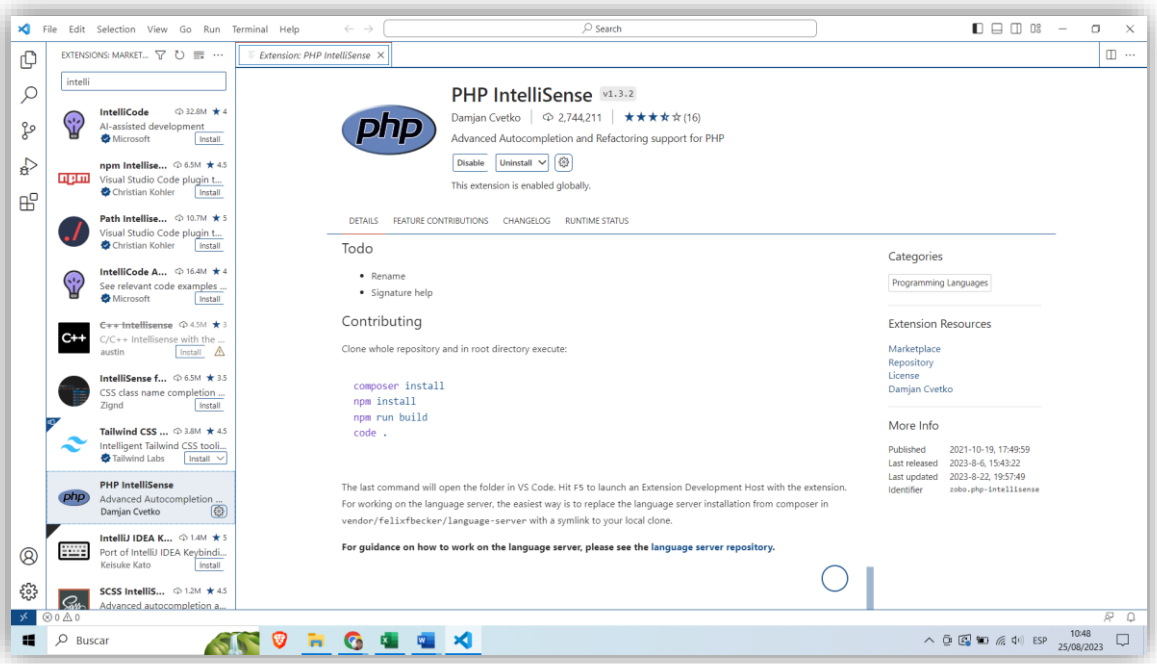
Instalamos el TypeScript Importer



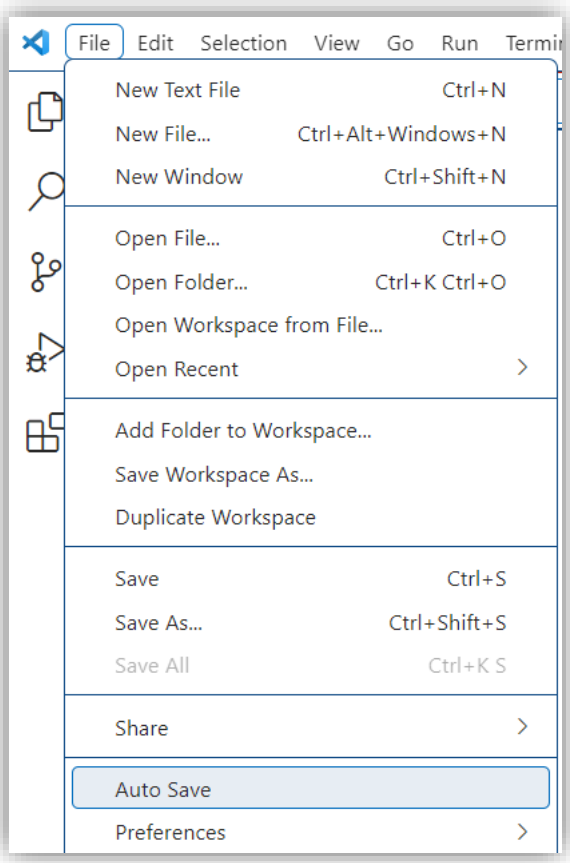
Instalamos Material Icon Theme, para mejorar los iconos



Instalamos PHP IntelliSense

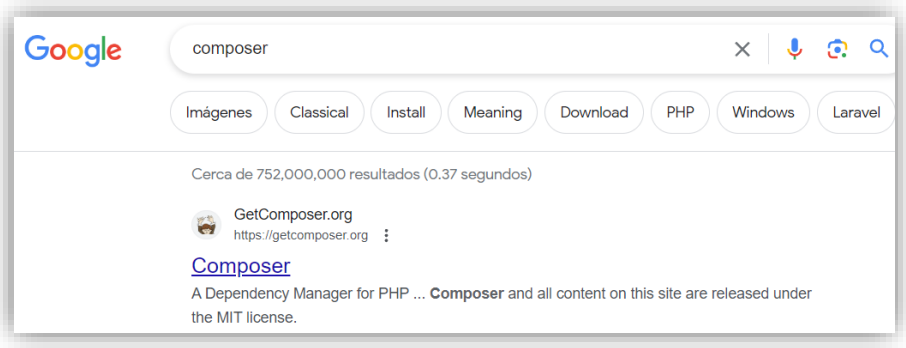


Finalmente activamos el autoguardado

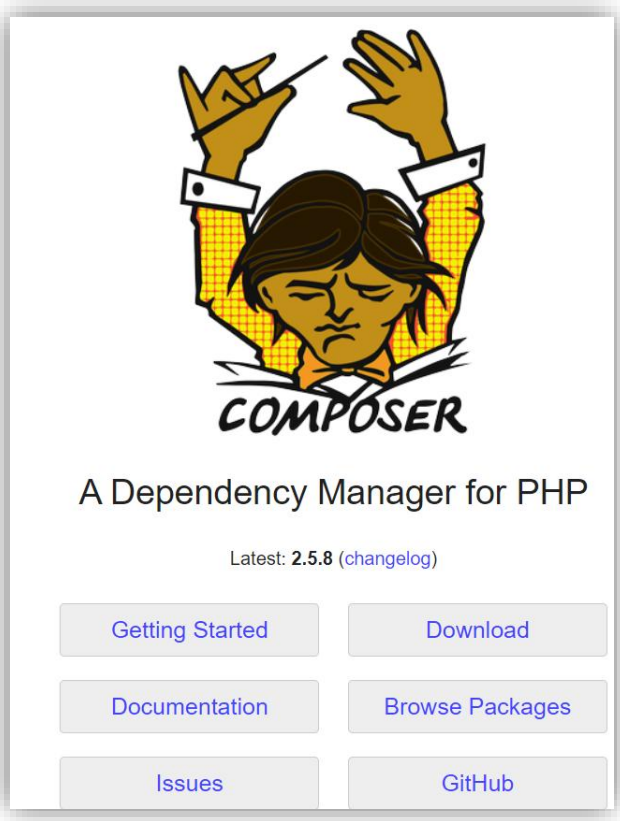


2.3 INSTALACIÓN DEL COMPOSER

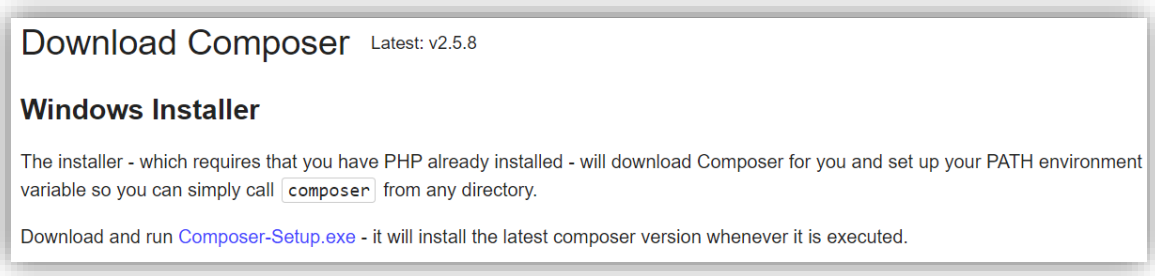
También se necesita otro componente muy necesario para que podamos instalar Laravel 9, en un navegador escribir **Composer,** luego damos clic en el link de getcomposer.org y lo vamos a descargar.



Para descargar hagan clic en **Download**

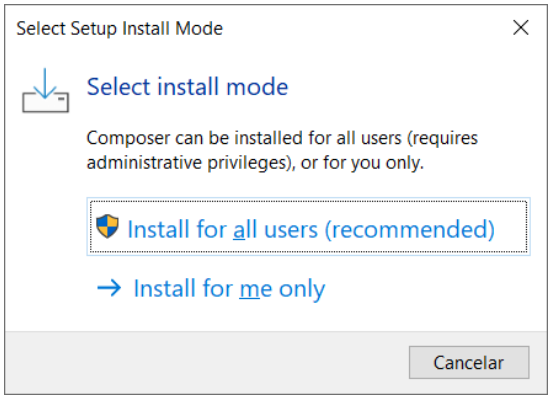


Nos va a dirigir a una página que nos va permitir descargar Laravel 9, debemos hacer clic en **Composer-setup.exe**

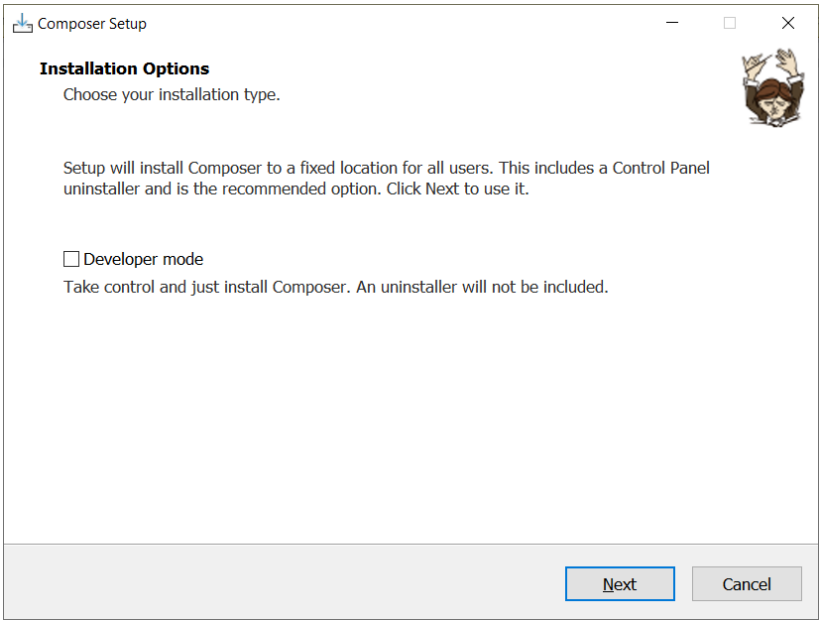


Realizado esta acción, el archivo ya habrá descargado, luego ejecutamos el archivo descargado. Esto nos va permitir descargar o gestionar los paquetes que vamos a trabajar.

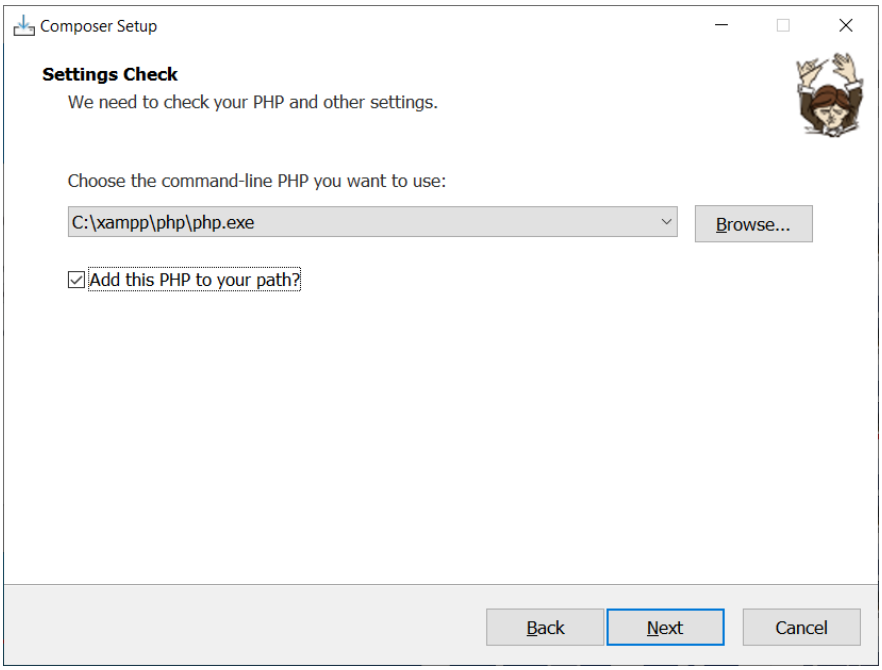
Hacer clic en Install for all users



Luego de otorgar los permisos, hacemos clic en next.

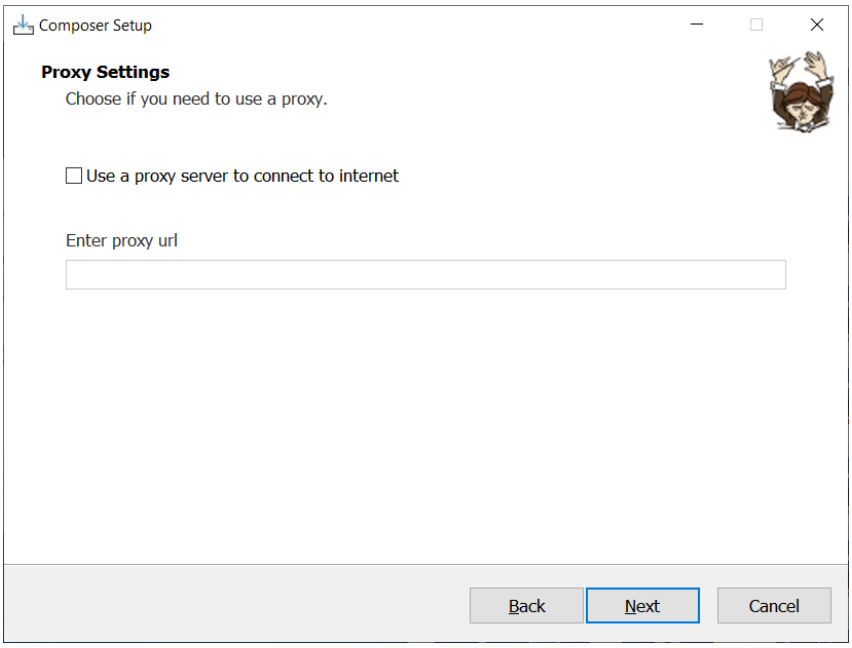


En este paso va a detectar donde se encuentra el ejecutable del PHP

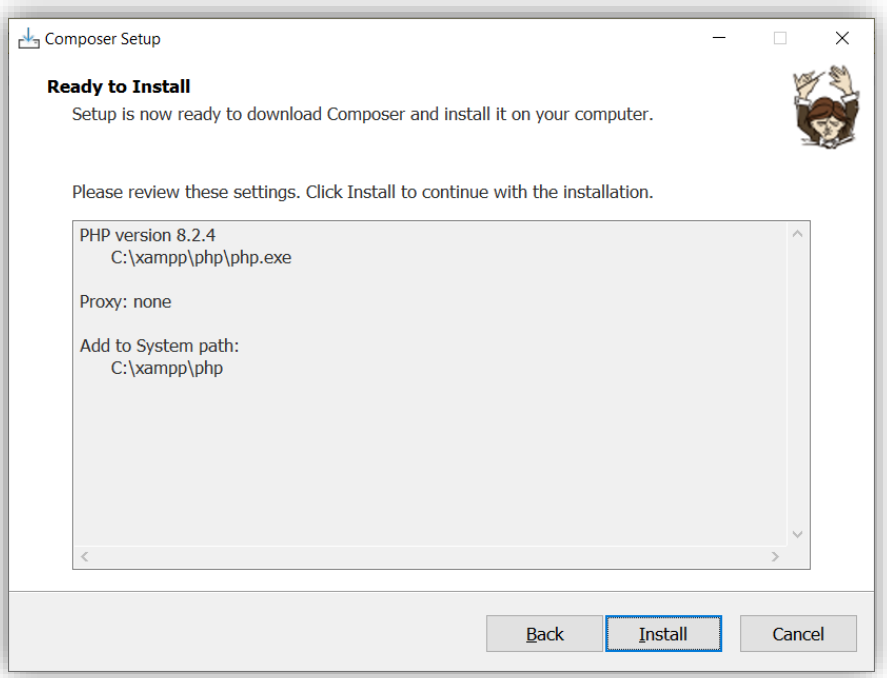


Debemos añadir al path, es decir hacer clic en **Add this PHP your path,** porque más adelante nos va permitir reconocer desde la consola de Windows. Luego damos clic en **Next.**

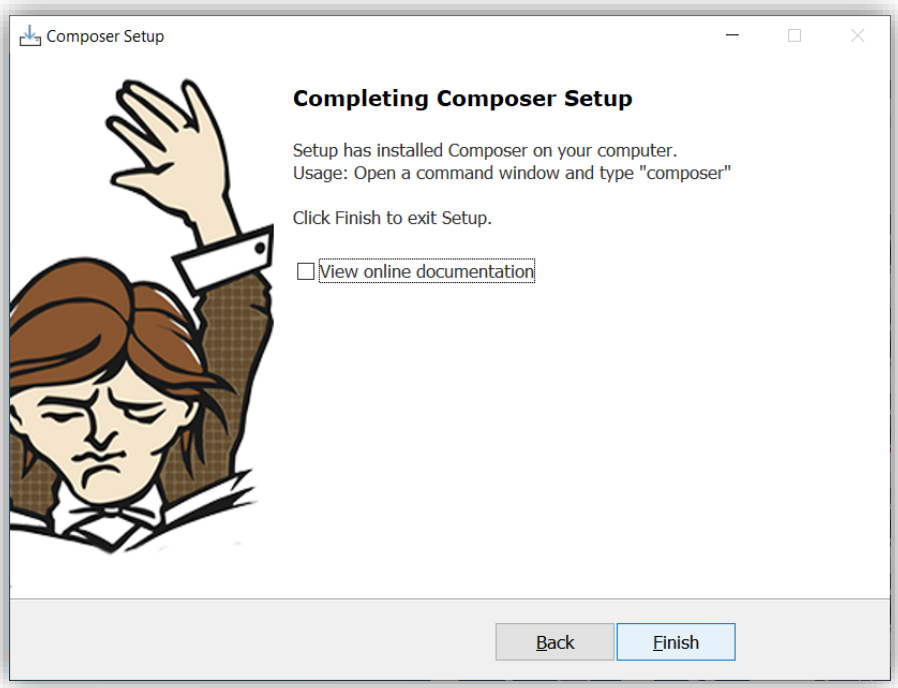
En la nueva ventana, hacer clic en **Next**



Luego clic en **Install**

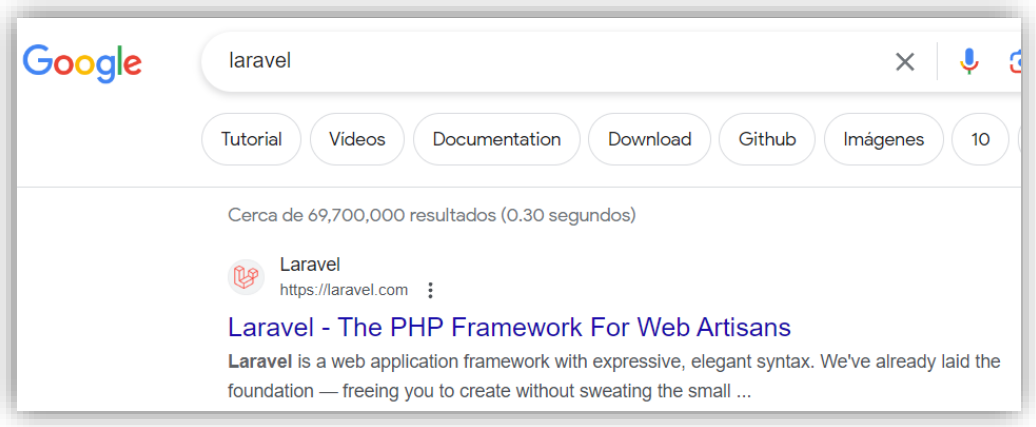


Esto es un gestor de paquetes para que podamos descargar Laravel y hacer las actualizaciones de este Framework, luego finalizamos.

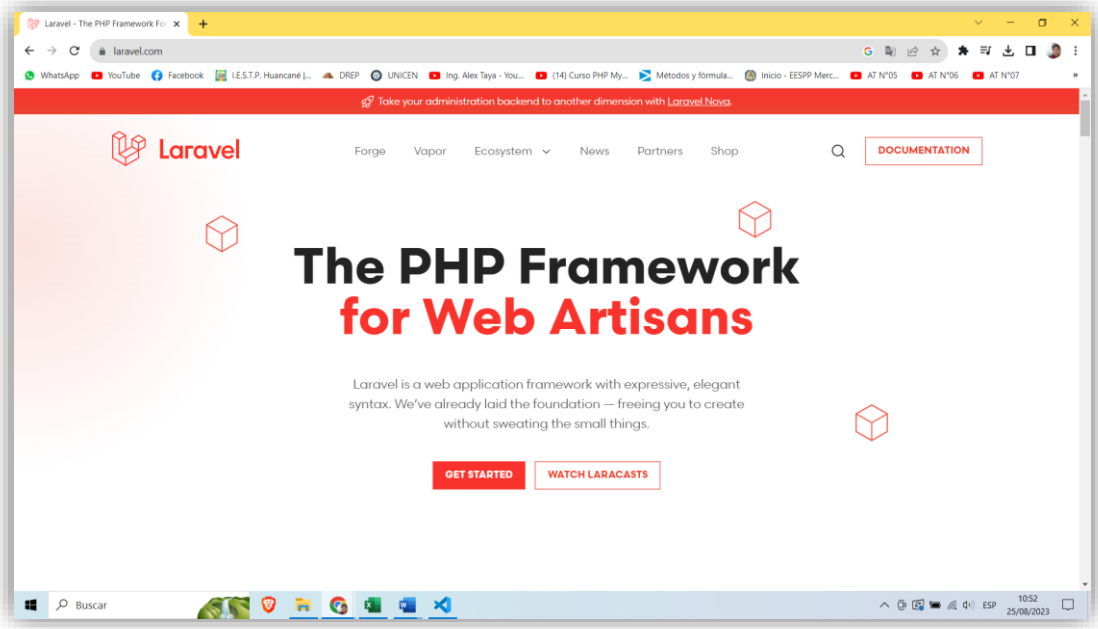


2.4 INSTALACIÓN DE LARAVEL

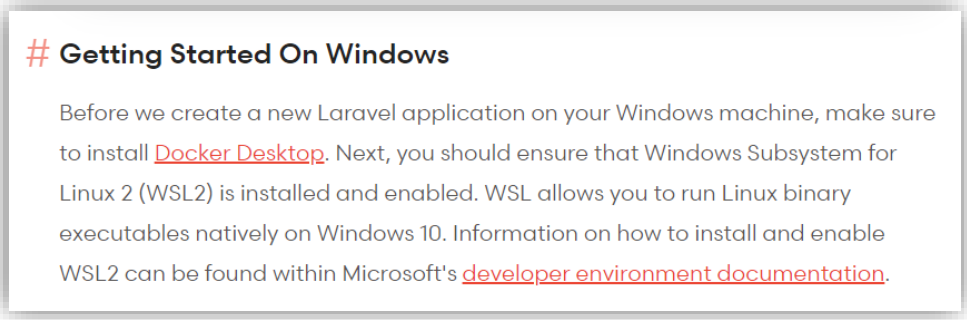
Buscar Laravel en el navegador para ver las instrucciones de descarga.



Haces clic en la página.

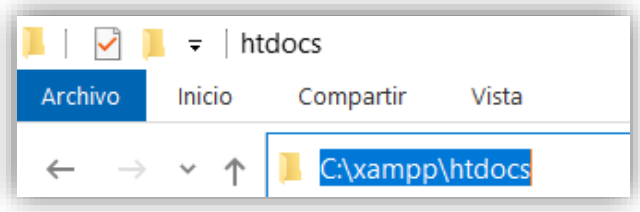


Por el momento, la última versión es la 9, hacer clic en **DOCUMENTACIÓN** y buscar **Getting Started On Windows**

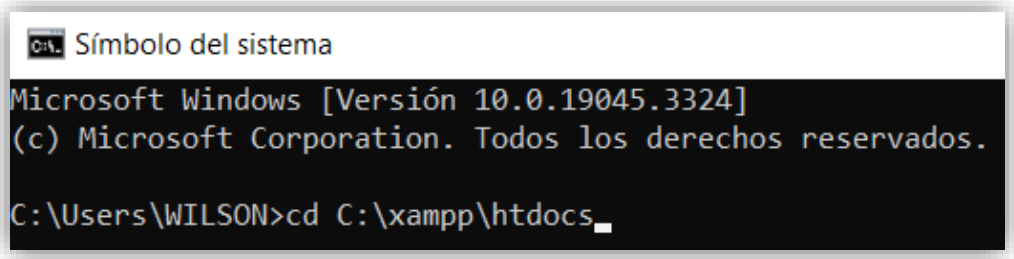


En este apartado nos da diferentes instrucciones, también podemos trabajar con Docker desktop, pero en esta ocasión trabajaremos con **Installation Vía Composer,** Buscaremos **Your First Laravel Project**

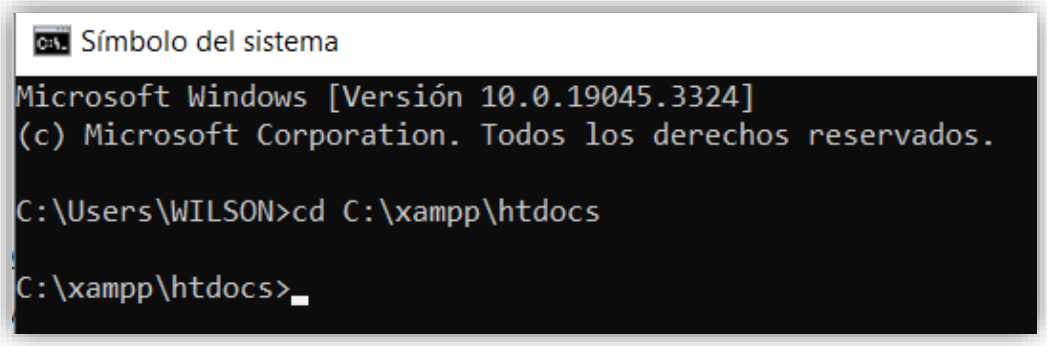
Para ello, vamos a ejecutar la consola CMD de Windows y también vamos a abrir la carpeta donde instalamos XAMPP, y buscar htdocs, es aquí donde vamos a trabajar porque es nuestro servidor web.



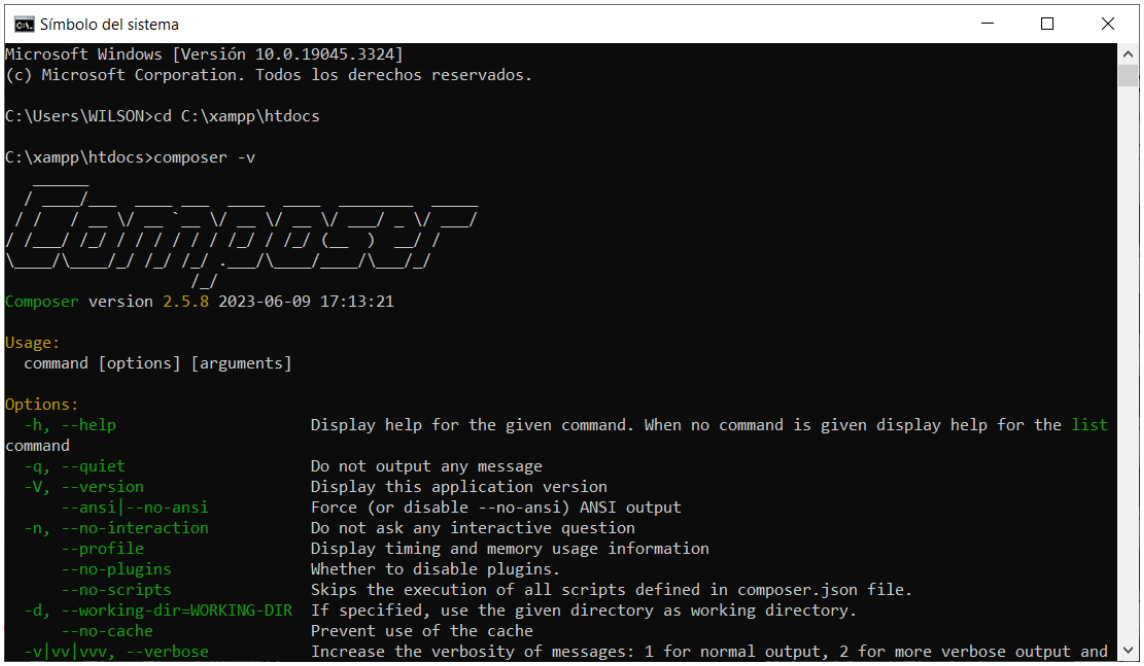
Copiamos la dirección y lo ponemos en la línea de comandos ejemplo: cd y lo copiado.



Presionamos ENTER

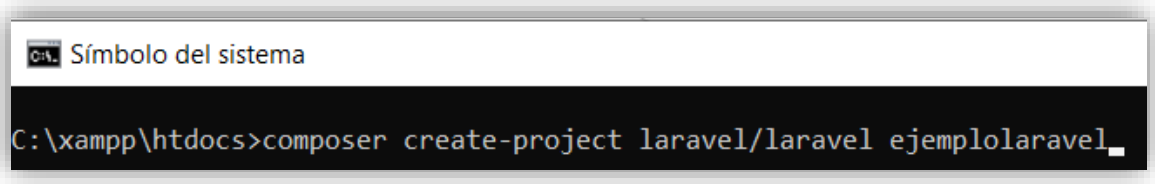


Acá, revisamos si ya está integrado **Composer,** para ello poner composer -v en el cmd. Y si aparece igual que la en la figura de abajo significa que ya está instalado y configurado composer, (si no aparece puede buscar un manual para configurar composer)



Entonces procedemos con la instalación, las instrucciones son las siguientes:

C:\xampp\htdocs>composer create-project laravel/laravel y el nombre del proyecto que va a ser el nombre de nuestra carpeta que le va agregar. Como ejemplo pondremos ejemplolaravel



Presionamos ENTER y luego vamos a esperar porque va descargar paquetes que necesita Laravel, esto lo vamos hacer cada vez que quieran generar un proyecto en laravel.

También es una recomendación que, si quieres integrarlo a tu instalación, puedes hacerlo de la siguiente manera:

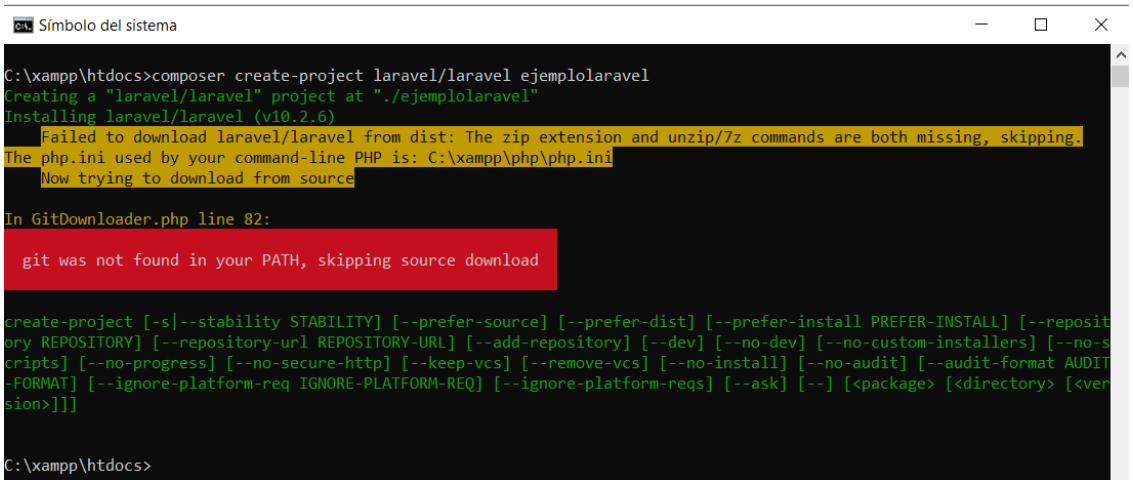
```
composer global require laravel/installer
```

Con lo anterior se integra a composer y ya solo cuando deseamos agregar un proyecto escribimos el siguiente comando

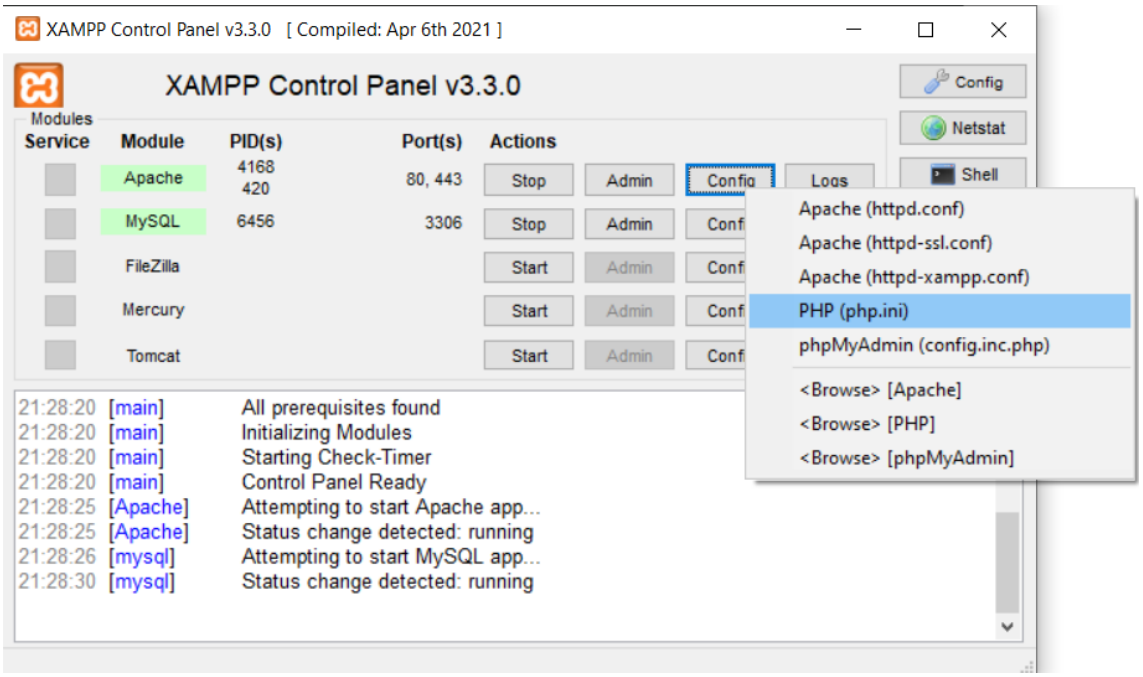
```
laravel new example-app
```

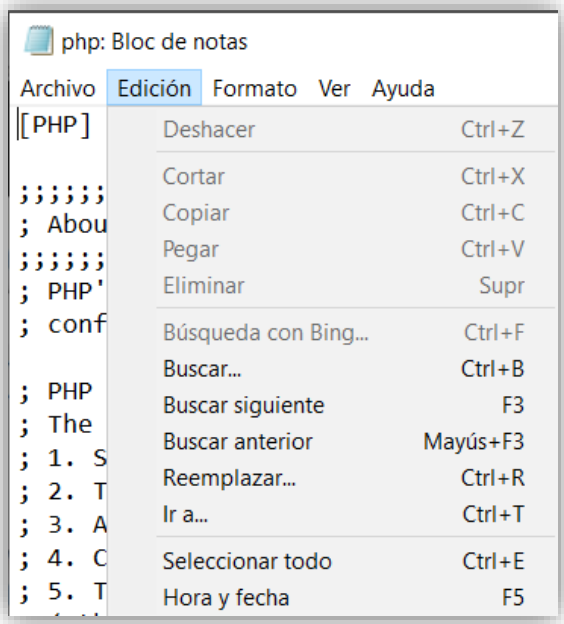
Pero eso dependerá de cómo quieras instalarlo.

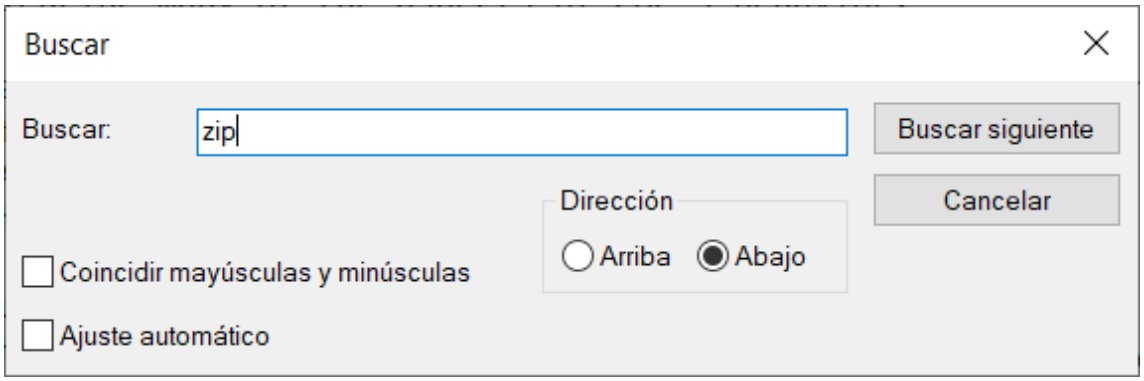
En caso salga la siguiente opción: debemos de arreglarlo.



Como arreglarlo:


XAMPP Control Panel v3.3.0 [Compiled: Apr 6th 2021]
XAMPP Control Panel v3.3.0
Modules
Service
Module
PID(s)
Port(s)
Actions
Apache
4168
420
80, 443
Stop
Admin
Config
Logs
MySQL
6456
3306
Stop
Admin
FileZilla
Start
Admin
Mercury
Start
Admin
Tomcat
Start
Admin
Config
Netstat
Shell
Apache (httpd.conf)
Apache (httpd-ssl.conf)
Apache (httpd-xampp.conf)
PHP (php.ini)
phpMyAdmin (config.inc.php)
<Browse> [Apache]
<Browse> [PHP]
<Browse> [phpMyAdmin]
21:28:20 [main] All prerequisites found
21:28:20 [main] Initializing Modules
21:28:20 [main] Starting Check-Timer
21:28:20 [main] Control Panel Ready
21:28:25 [Apache] Attempting to start Apache app...
21:28:25 [Apache] Status change detected: running
21:28:26 [mysql] Attempting to start MySQL app...
21:28:30 [mysql] Status change detected: running


php: Bloc de notas
Archivo
Edición
Formato
Ver
Ayuda
[PHP]
; Abou
; PHP'
; conf
; PHP
; The
; 1. S
; 2. T
; 3. A
; 4. C
; 5. T
Deshacer Ctrl+Z
Cortar Ctrl+X
Copiar Ctrl+C
Pegar Ctrl+V
Eliminar Supr
Búsqueda con Bing... Ctrl+F
Buscar... Ctrl+B
Buscar siguiente F3
Buscar anterior Mayús+F3
Reemplazar... Ctrl+R
Ir a... Ctrl+T
Seleccionar todo Ctrl+E
Hora y fecha F5


Buscar
Buscar:
zip
Buscar siguiente
Cancelar
Dirección
Arriba
Abajo
Coincidir mayúsculas y minúsculas
Ajuste automático

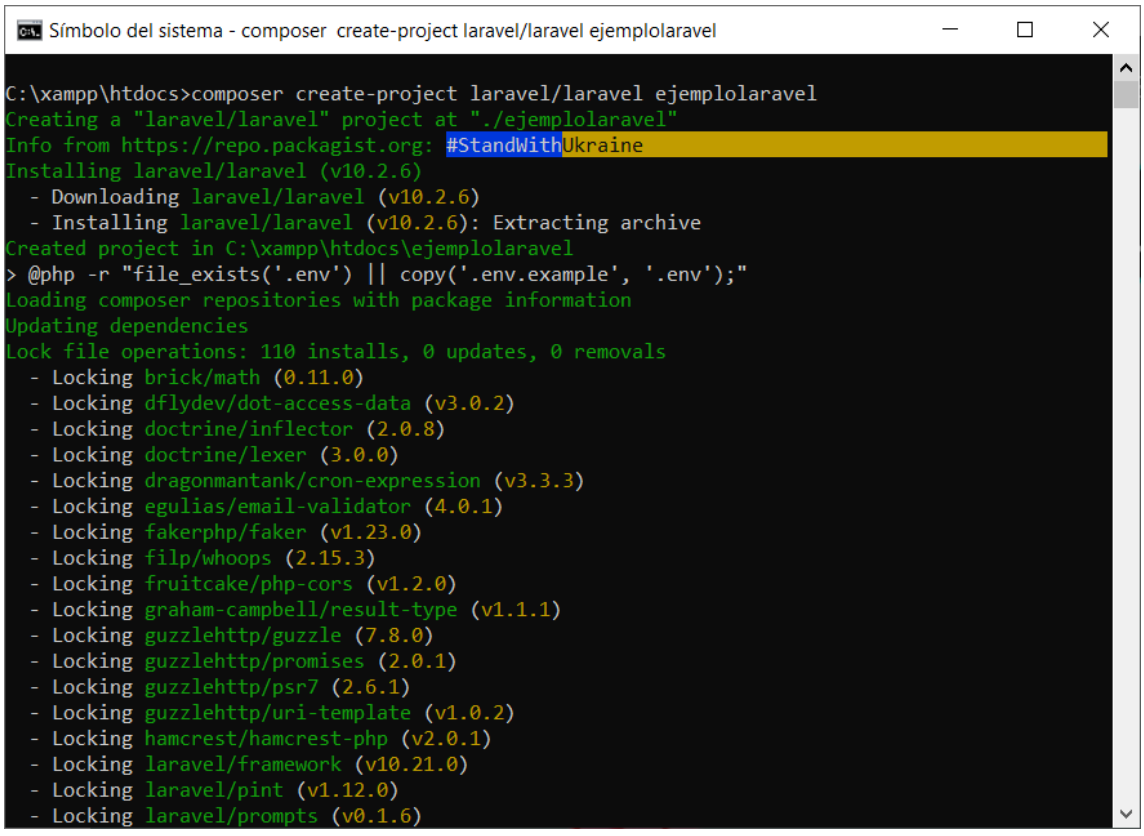
```
;extension=soap
;extension=sockets
;extension=sodium
;extension=sqlite3
;extension=tidy
;extension=xsl
;extension=zip
```

Borrar el punto y coma

```
;extension=soap
;extension=sockets
;extension=sodium
;extension=sqlite3
;extension=tidy
;extension=xsl
extension=zip
```

Luego grabar y cerrar el block de notas.

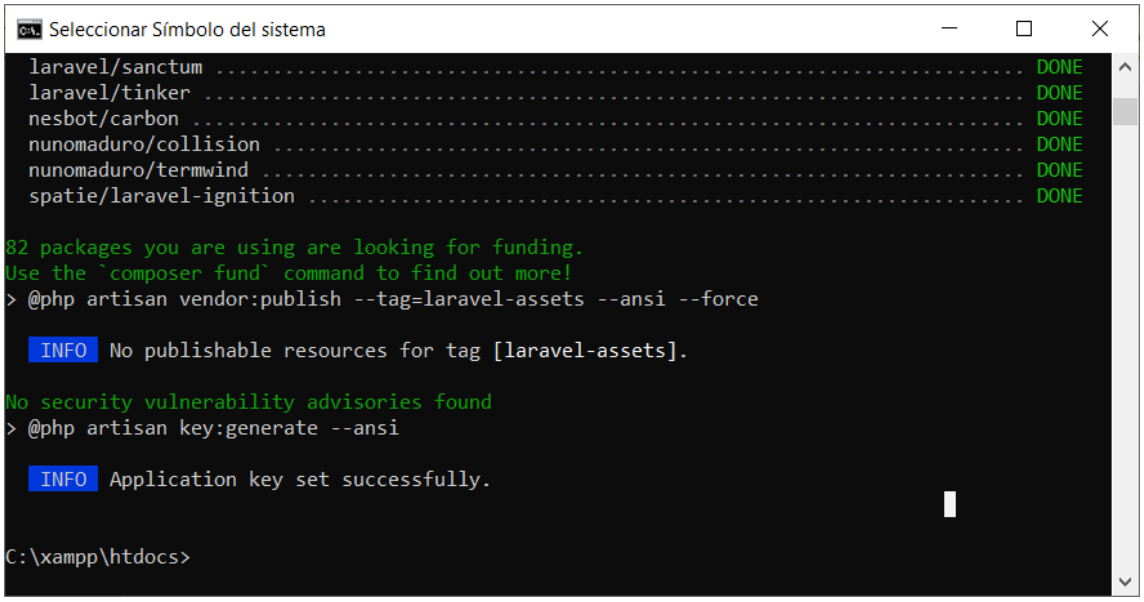
Ahora ejecutar nuevamente la instalación de laravel



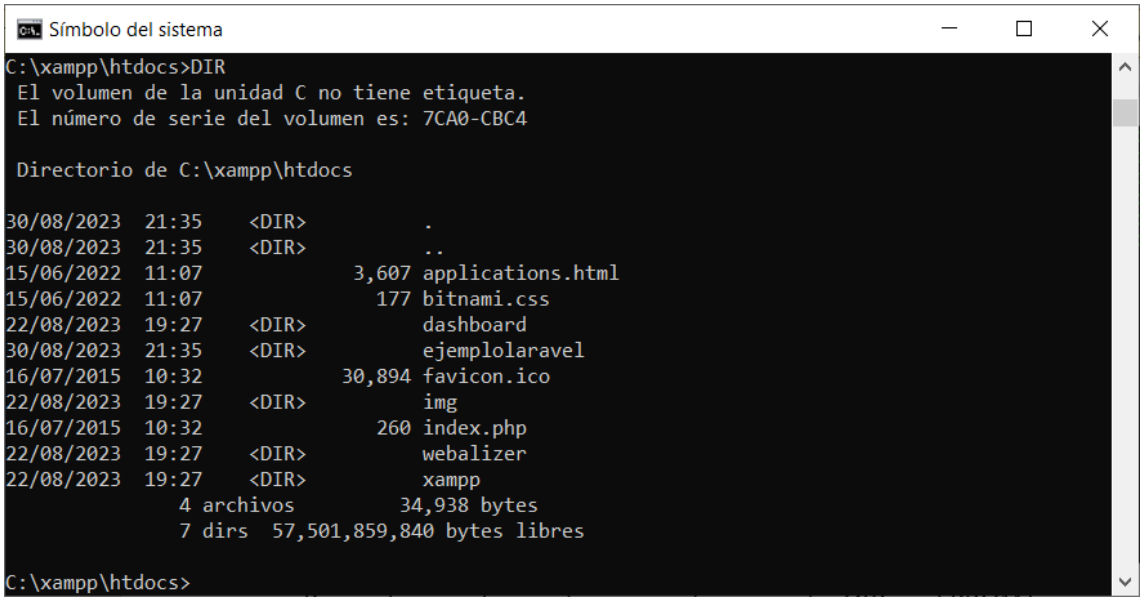
Ahora si funciona y nos daremos cuenta que se está instalando. (ojo, debemos considerar que debemos tener acceso a internet para la instalación, en caso contrario no

va avanzar la instalación)

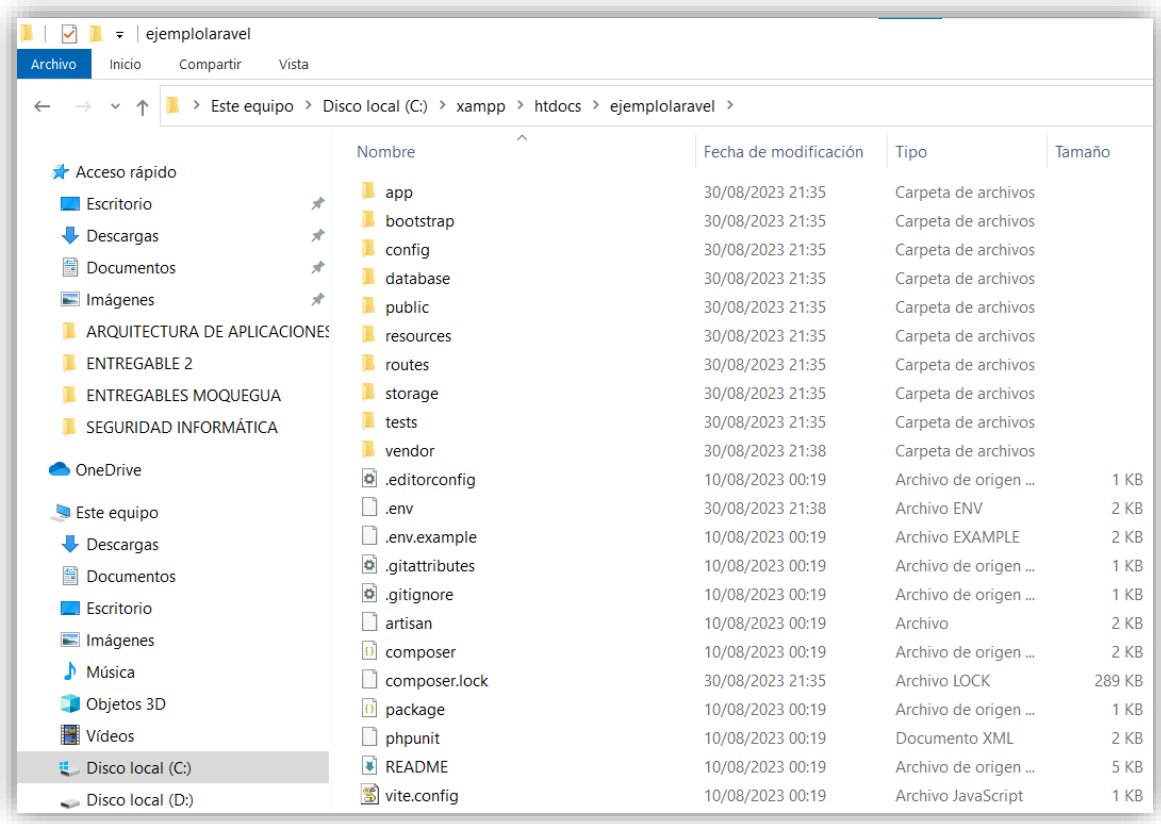
La instalación terminará cuando aparezca el promt del servidor, como se ve:



Para poder encontrar su ubicación podemos escribir DIR en el PROMT



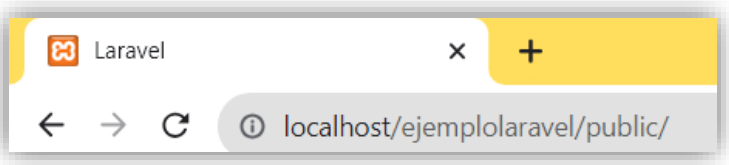
Si abrimos una ventana de Windows, podemos ver los archivos generados por Laravel.

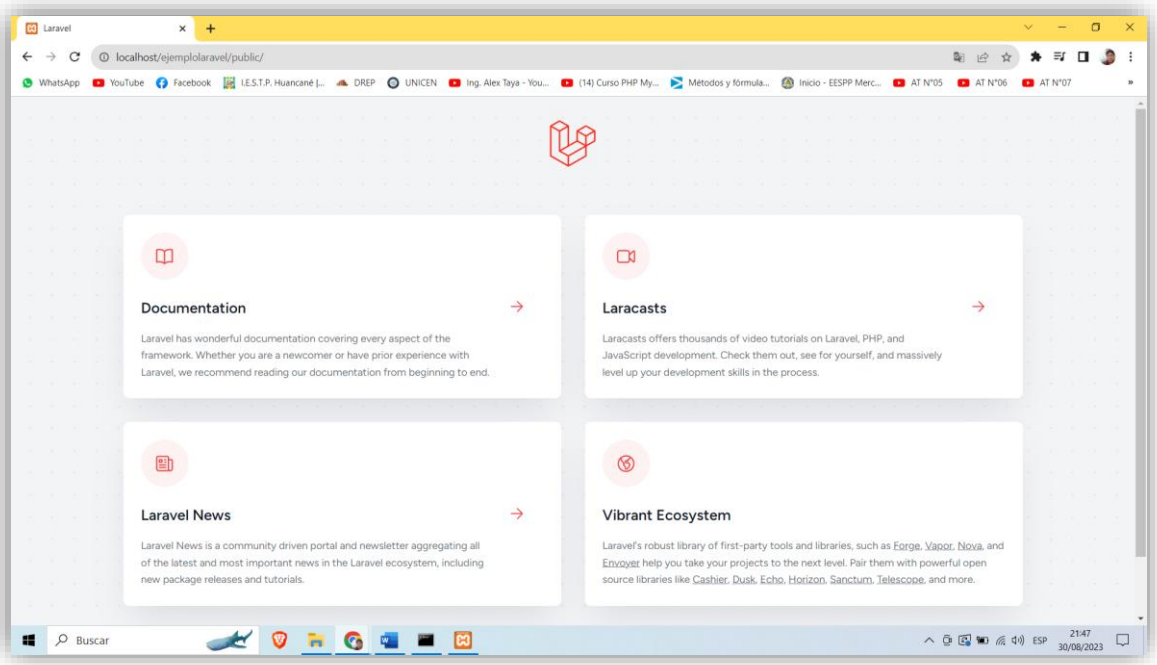


Se ve bastante información, pero poco a poco vamos a ver para que sirve cada carpeta, que necesitamos guardar en ellas, y los archivos de configuración.

2.5 VISUALIZACIÓN DEL FRAMEWORK

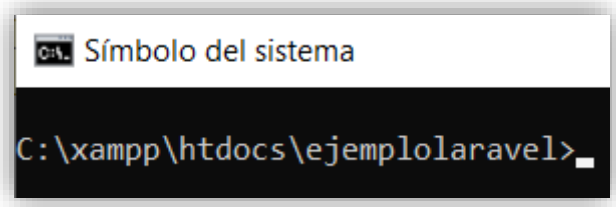
En el navegador escribir la carpeta del proyecto.



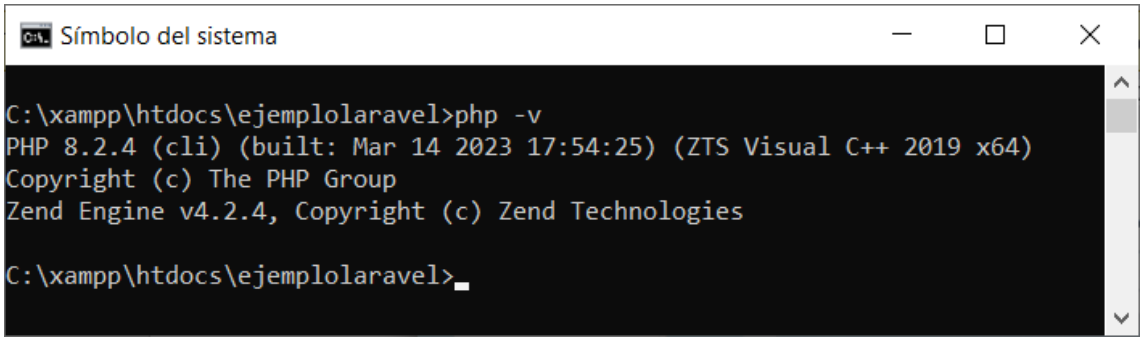


Todo lo anterior lo vamos a modificar para hacer las adecuaciones, pero si te sale esa pantalla, ya hemos instalado correctamente Laravel

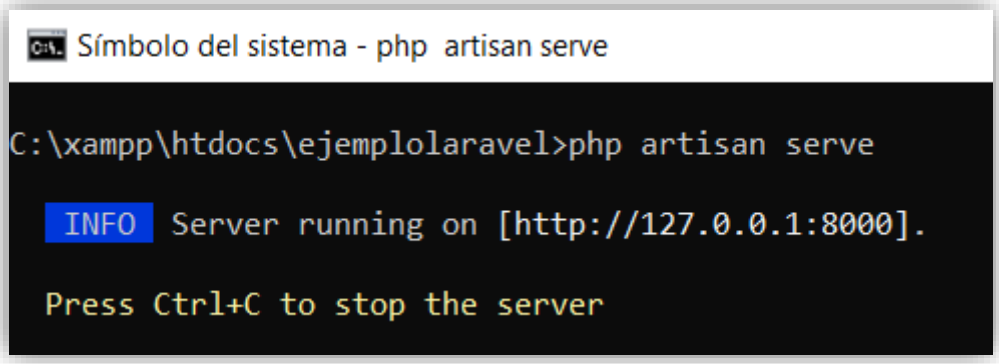
Laravel también tiene una herramienta llamada **ARTISAN,** que también vamos a instalarlo, también se ejecuta desde la consola de comandos, la condición es que debemos de estar dentro de la carpeta en la cual esta nuestro proyecto.



Colocar php -v para verificar que tenemos configurado en la terminal.

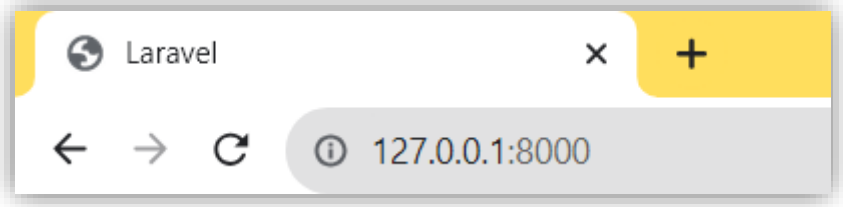


Luego escribir **php artisan serve**



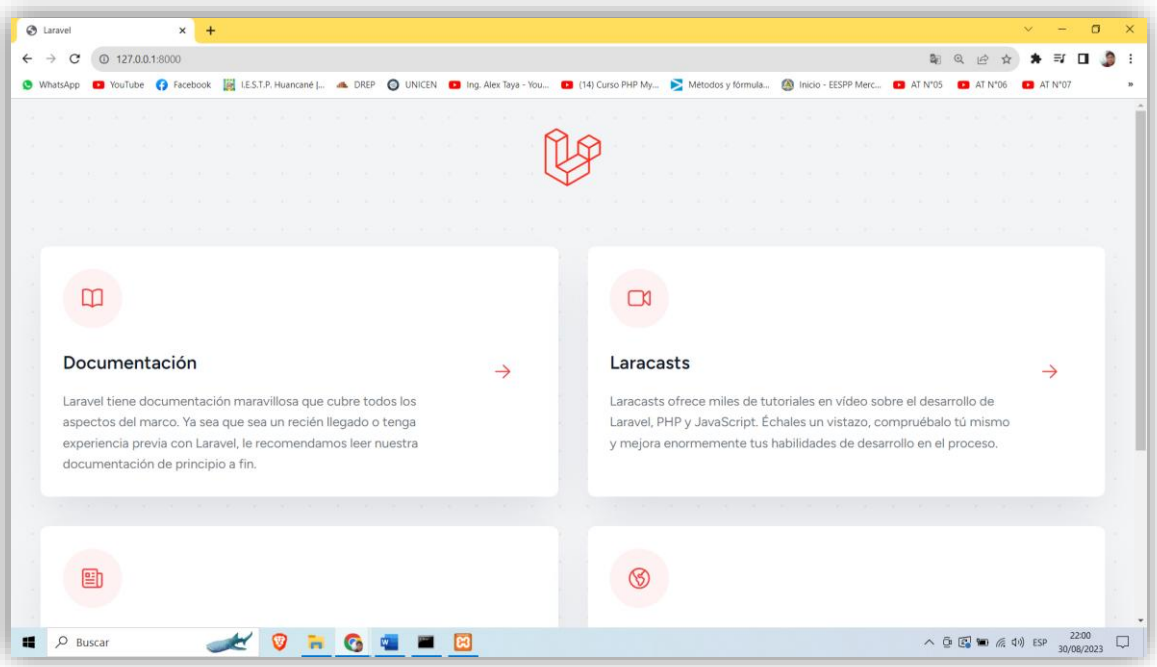
Acá nos va proporcionar una ruta que necesitamos agregar al navegador.

http://127.0.01:8000 el 8000 viene a ser el puerto, ejecutamos y vemos que es lo mismo que la primera ruta ejecutada.



Todo dependerá desde donde lo quiere consultar. El primero es con apache server y esta última configuración es con **ARTISAN.**

Y una vez que ya tenemos nuestro Framework instalado como una aplicación, ya vamos a poder trabajar y a conocer este Framework Laravel 9

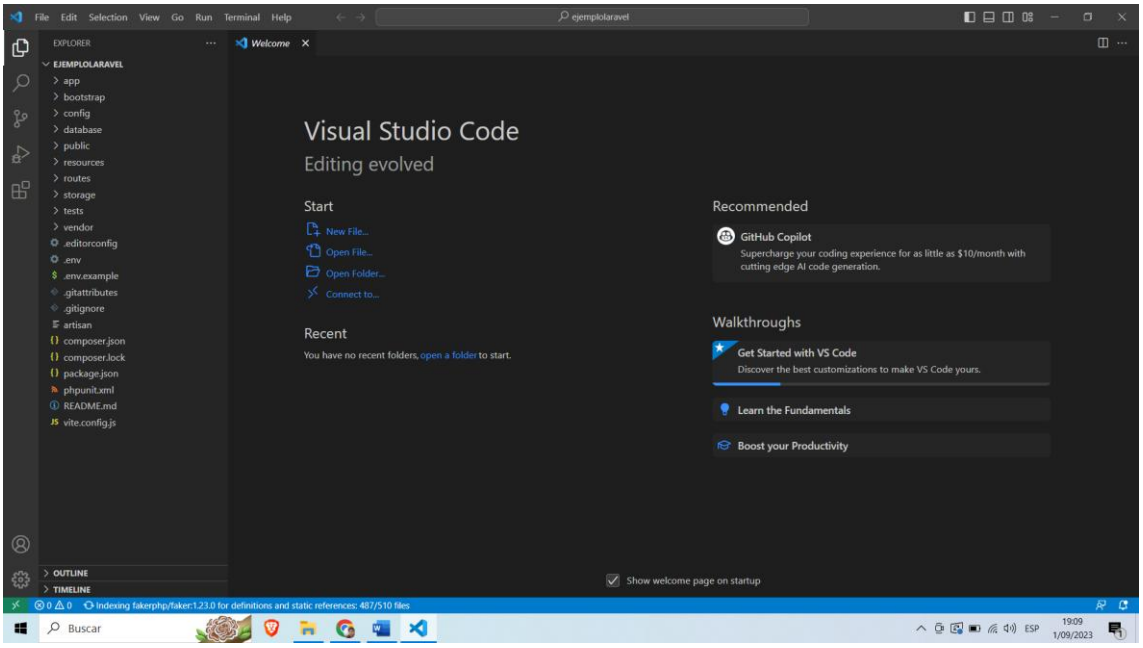

Laravel
127.0.0.1:8000
Documentación
Laravel tiene documentación maravillosa que cubre todos los aspectos del marco. Ya sea que sea un recién llegado o tenga experiencia previa con Laravel, le recomendamos leer nuestra documentación de principio a fin.
Laracasts
Laracasts ofrece miles de tutoriales en vídeo sobre el desarrollo de Laravel, PHP y JavaScript. Échales un vistazo, compruébalo tú mismo y mejora enormemente tus habilidades de desarrollo en el proceso.
Buscar
ESP
22:00
30/08/2023

CAPÍTULO 3

CONFIGURACIÓN

En este tema vamos a aprender a realizar configuraciones para poner en marcha nuestro Framework Laravel 9.

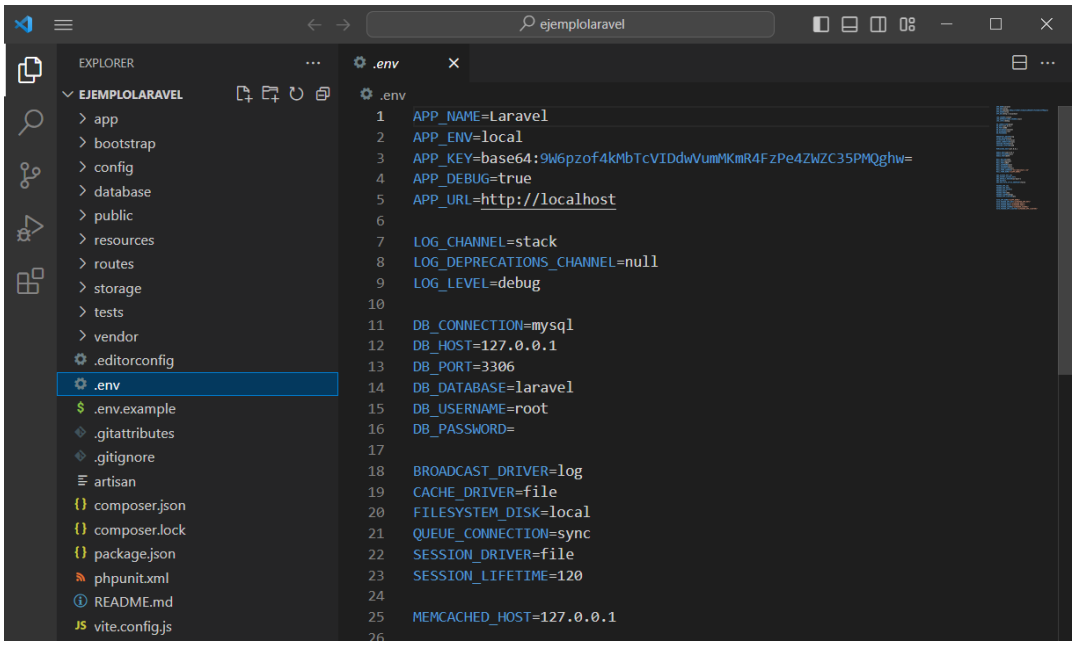
Abriremos la carpeta donde hemos instalado Laravel y lo vamos a cargar con nuestro editor Visual Studio Code. (Solo tenemos que arrastrar la carpeta en el Visual Studio Code)



Vamos a conocer un poco en las carpetas donde vamos a encontrar la configuración y como la necesitamos realizar.

3.1 ARCHIVO .env

El primer archivo que vamos a explicar y que se encuentra en nuestro proyecto se llama **.env** y lo podemos abrir.

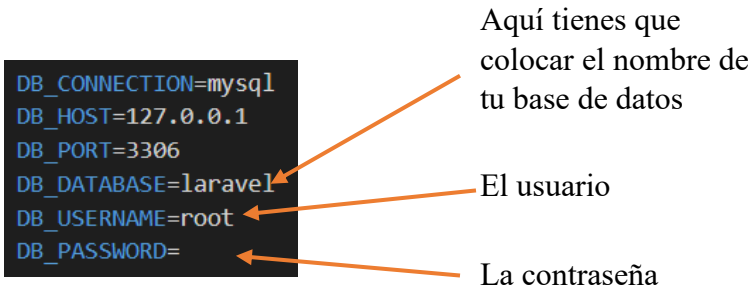


Este es uno de los archivos que necesitamos configurar al inicio, porque aquí vamos a encontrar diferentes apartados para la configuración, como por ejemplo la URL de tu aplicación.

```
APP_NAME=Laravel
APP_ENV=local
APP_KEY=base64:9W6pzof4kMbTcVIDdwVumMKmR4FzPe4ZWZC35PMQghw=
APP_DEBUG=true
APP_URL=http://localhost
```

Esto nos va ayudas más adelante cuando trabajemos los estilos, o algún archivo, darle la ubicación. Con esta URL vamos a poder asignarle mucho mas rápido.

También podemos encontrar los datos de conexión a la base de datos.



Mas abajo se aprecia más configuraciones que a lo largo del curso vamos a trabajar. Acá es muy importante revisar la configuración inicial, en este caso en **APP_ENV** va a ser local, porque por el momento va ha ser de desarrollo. Ya cuando lo pasemos a producción le vamos a colocar production y en **APP_DEBUG** le vamos a poner False, para que ya no nos envíe ningún mensaje de en la pantalla, sino que todo se vaya al archivo que maneja laravel, y ahí nos v mostrar todos los errores, entonces ya no será una pantalla con mucho código. Y en **APP_URL** vamos a colocar la ruta de nuestra aplicación de como ingresamos al navegador, en este caso: **http://localhost/ejemplolaravel/public** pero ya dependerá de como ingresemos a la aplicación.

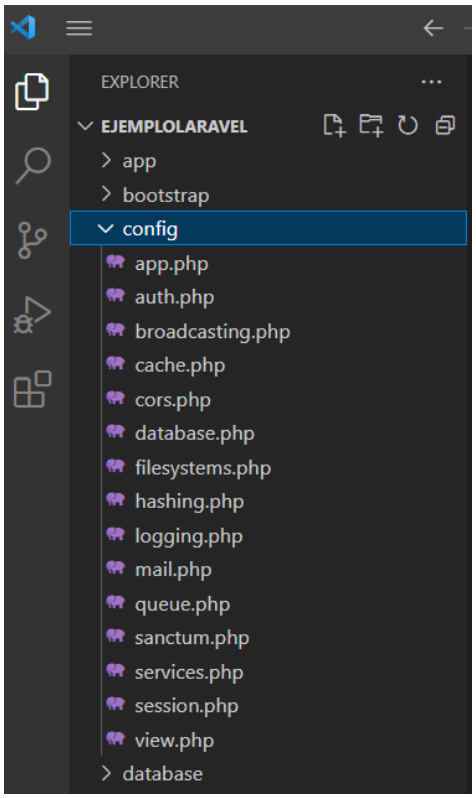
```
APP_NAME=Laravel
APP_ENV=local
APP_KEY=base64:9W6pzof4kMbTcVIDdwVumMKmR4FzPe4ZWZC35PMQghw=
APP_DEBUG=true
APP_URL=http://localhost
```

Cuando pasemos a producción, puedes cambiar el dominio, tal vez ahí ya no tengamos que agregar la carpeta de tu proyecto o tal vez tampoco el public. Todo dependerá de donde lo subas y como lo vayas a trabajar.

```
APP_NAME=Laravel
APP_ENV=local
APP_KEY=base64:9W6pzof4kMbTcVIDdwVumMKmR4FzPe4ZWZC35PMQghw=
APP_DEBUG=true
APP_URL=http://localhost/ejemplolaravel/public
```

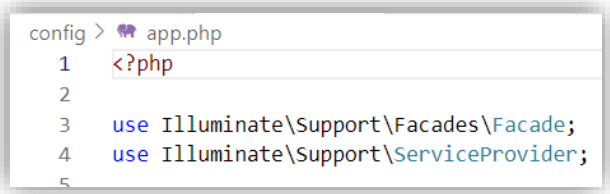
En este archivo es donde realizamos las configuraciones más importantes, pero podemos encontrar estas más detalladas.

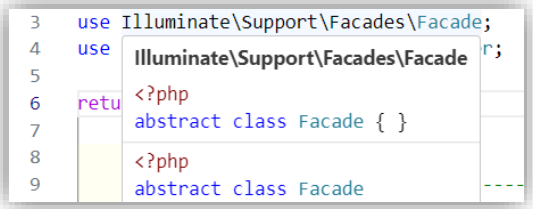
P or ejemplo, si vamos a la carpeta **CONFIG** en donde podemos encontrar un archivo para cada tipo de configuración.

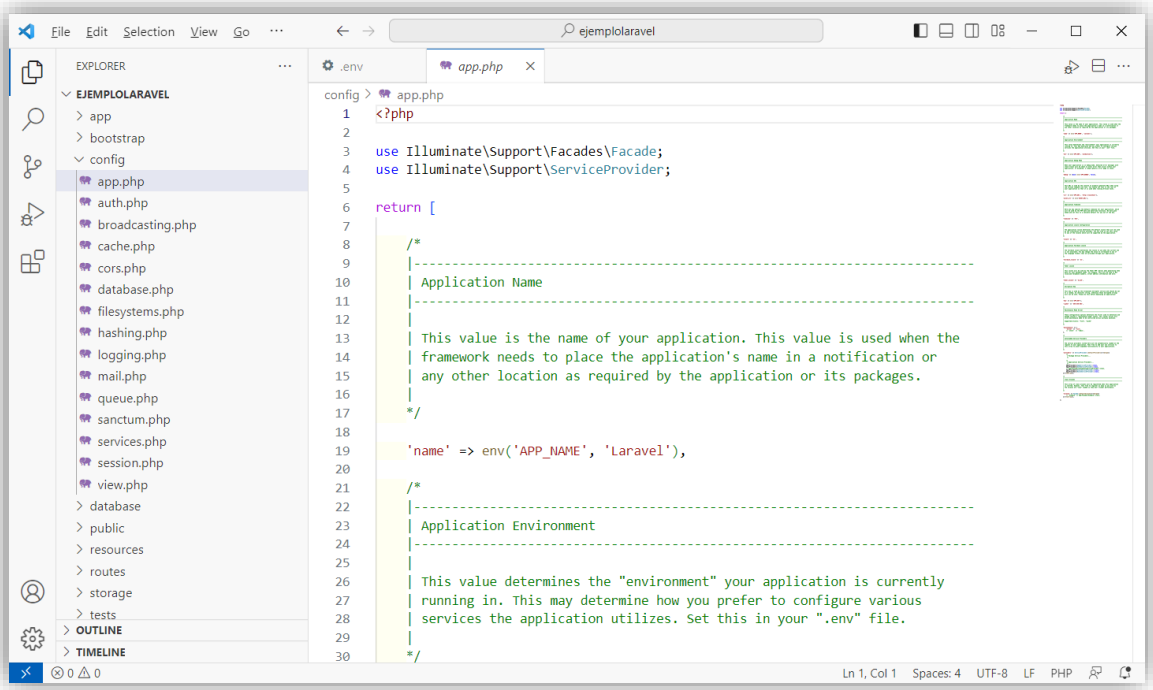


Vamos a abrir el archivo **app.php** Es aquí donde se aprecia la estructura de como se maneja el Framework Laravel. Vamos a trabajar mucho con name space.

En la parte de arriba (Línea 3) estamos indicando que vamos a trabajar con ese archivo y esa ubicación. Puedes buscarlo ahí para saber en donde se encuentra. Y **Illuminate\Support\Facades\Facade;** va a ser el name space de este archivo para hacer esta configuración,







Líneas más abajo encontramos el **APP_NAME** que es el mismo que encontramos al inicio en el archivo **.env**

```
'name' => env('APP_NAME', 'Laravel'),
```

Aquí también podemos hacer toda la configuración si es para producción

```
'env' => env('APP_ENV', 'production'),
```

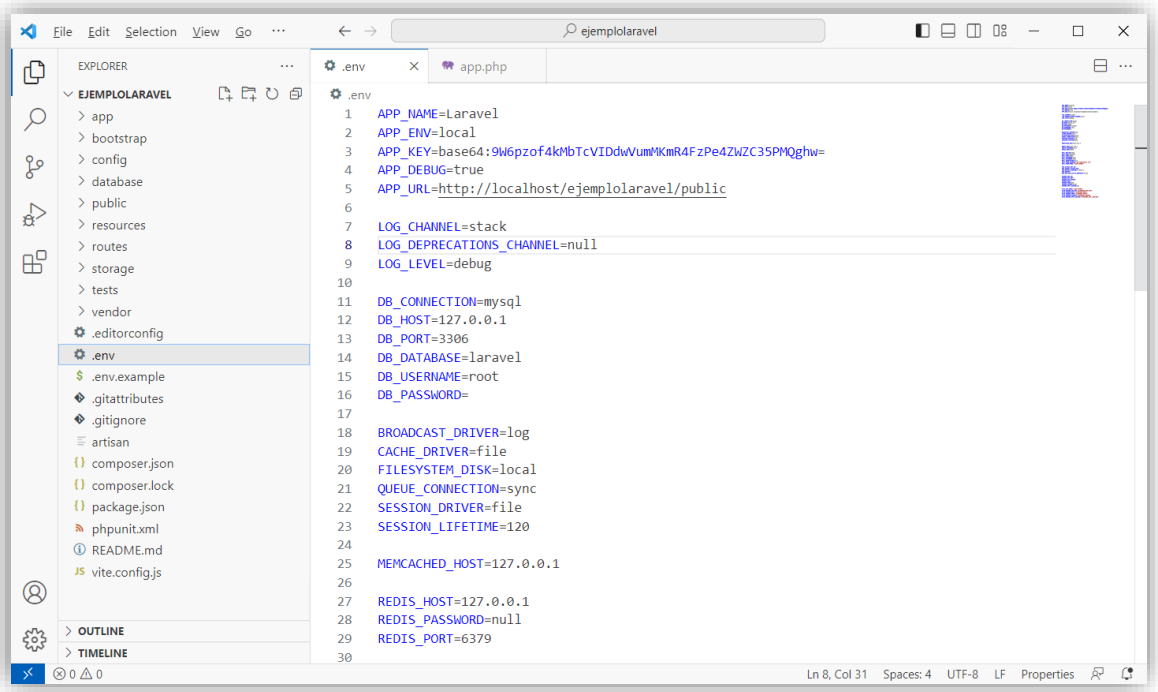
Lo podemos cambiar con **development** (desarrollo) pero no lo haremos aún.

```
'env' => env('APP_ENV', 'development'),
```

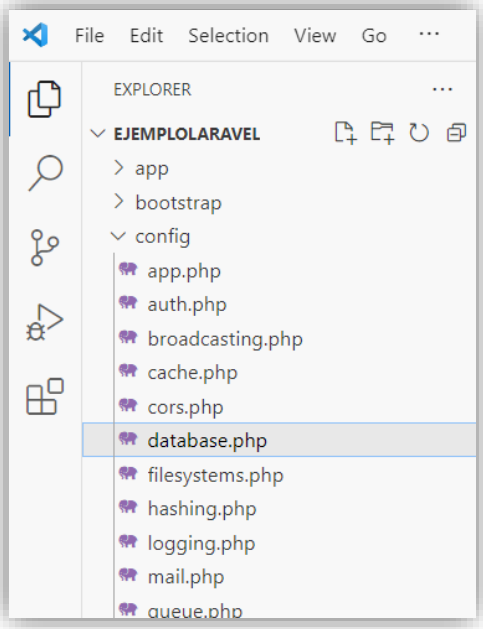
Mas abajo en APP_DEBUG está como falsa

```
'debug' => (bool) env('APP_DEBUG', false),
```

Debo aclarar que la configuración inicial esta para **producción,** por eso es que en ocasiones usamos más el **.env** para poder hacer esta configuración



También hay otro archivo que se llama **database.php**



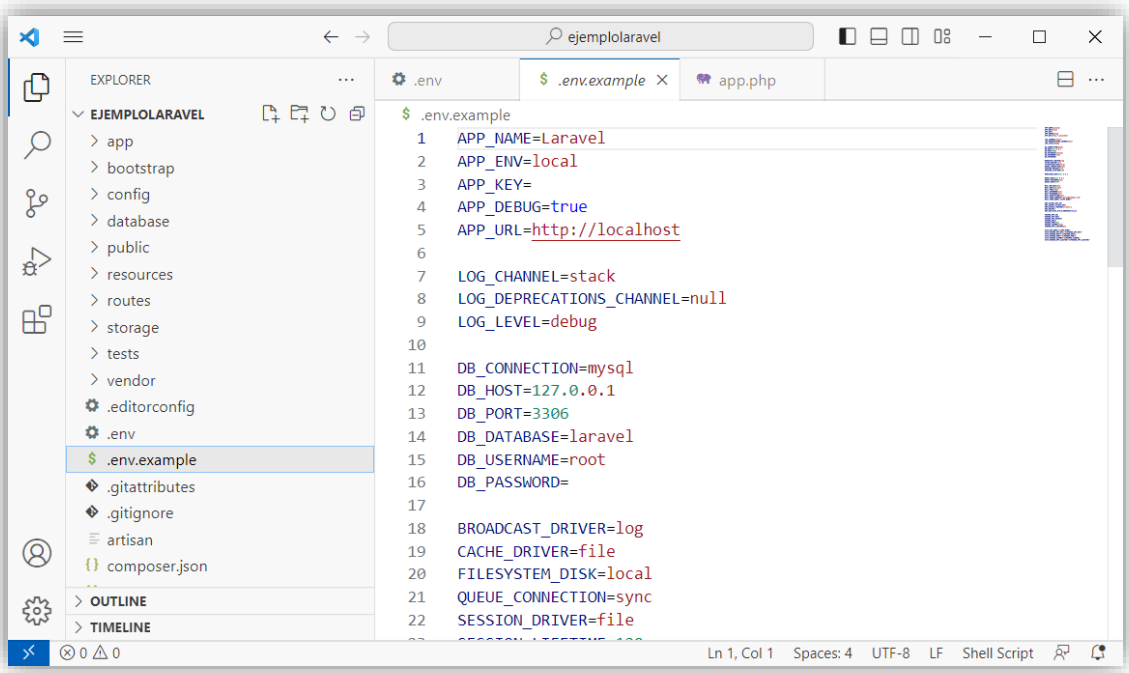
Y aquí vamos a poder hacer el cambio de tus datos a la base de datos. Aquí es muy importante precisar por qué usamos el **.env** y no todos los archivos detallados de esta forma.

```
'mysql' => [
    'driver' => 'mysql',
    'url' => env('DATABASE_URL'),
    'host' => env('DB_HOST', '127.0.0.1'),
    'port' => env('DB_PORT', '3306'),
    'database' => env('DB_DATABASE', 'forge'),
    'username' => env('DB_USERNAME', 'forge'),
    'password' => env('DB_PASSWORD', ''),
    'unix_socket' => env('DB_SOCKET', ''),
    'charset' => 'utf8mb4',
    'collation' => 'utf8mb4_unicode_ci',
    'prefix' => '',
    'prefix_indexes' => true,
    'strict' => true,
    'engine' => null,
    'options' => extension_loaded('pdo_mysql') ? array_filter([
        PDO::MYSQL_ATTR_SSL_CA => env('MYSQL_ATTR_SSL_CA'),
    ]) : [],
```

Es muy común de que cuando trabajemos con el Framework Laravel o con algún otro Framework o un proyecto, compartimos el proyecto con otros colaboradores o programadores que van a implementar algún otro módulo, puede que tengas datos sensibles como la conexión a la base de datos, algún token que generes específicamente para ti.

Y si tú los dejas en estos archivos tal vez puedas olvidarlo, entonces lo que comúnmente se realiza en estos tipos de Framework es dejar esta configuración tal y como está. Solo debemos agregar a toda la configuración en el archivo **.env**

Si vamos nuevamente a la raíz, vamos a encontrar un archivo que se llama **.env.example** y es prácticamente similar al que ya estamos configurando.



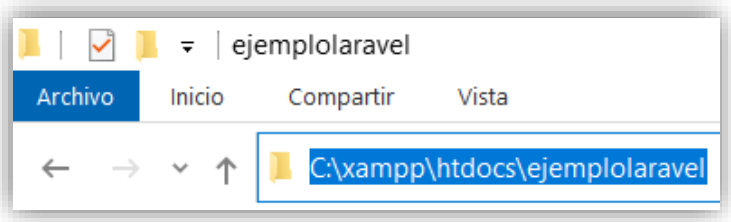
Esto es para que al momento en que compartas el código o si lo subes a algún repositorio elimines el archivo **.env** o no se haya subido, para que no tenga los datos que tal vez son sensibles para que otras personas no lo sepan. Y les dejas el archivo **.env.example** para que ellos realicen su propia configuración a la base de datos, tal vez algunas claves de correo electrónico y algún otro dato mas específico que vaya a usar cada programador.

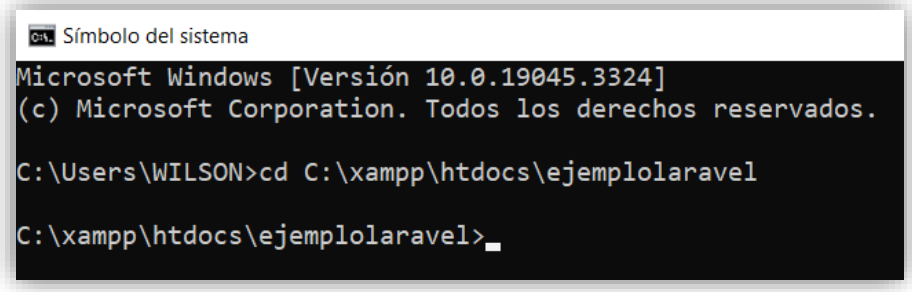
Por eso es muy importante trabajarlo desde este archivo, pero también trabajarlo desde **database.php**

Bien, ya vimos algo de configuración, ahora vamos a nuestro navegador y vamos a poner **localhost/ejemplolaravel/public**

3.2 INICIANDO LA APLICACIÓN USANDO ARTISAN

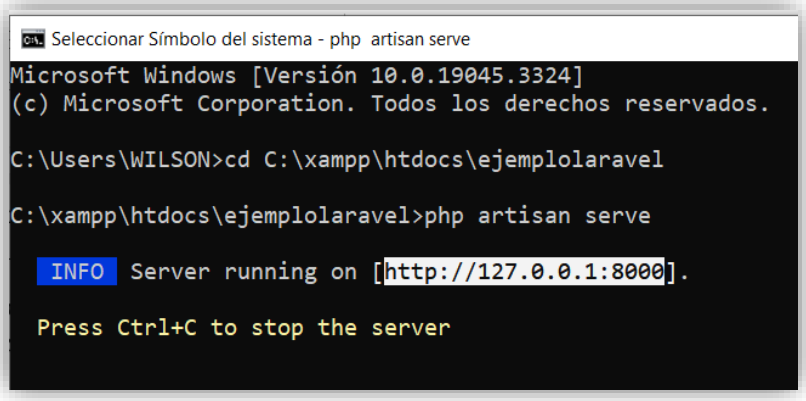
Vamos a iniciar también con CMD, pero ahora vamos a hacerlo con ARTISAN, ubicamos la dirección donde está ubicado nuestro proyecto, luego copiamos la ruta para ejecutarla en CMD

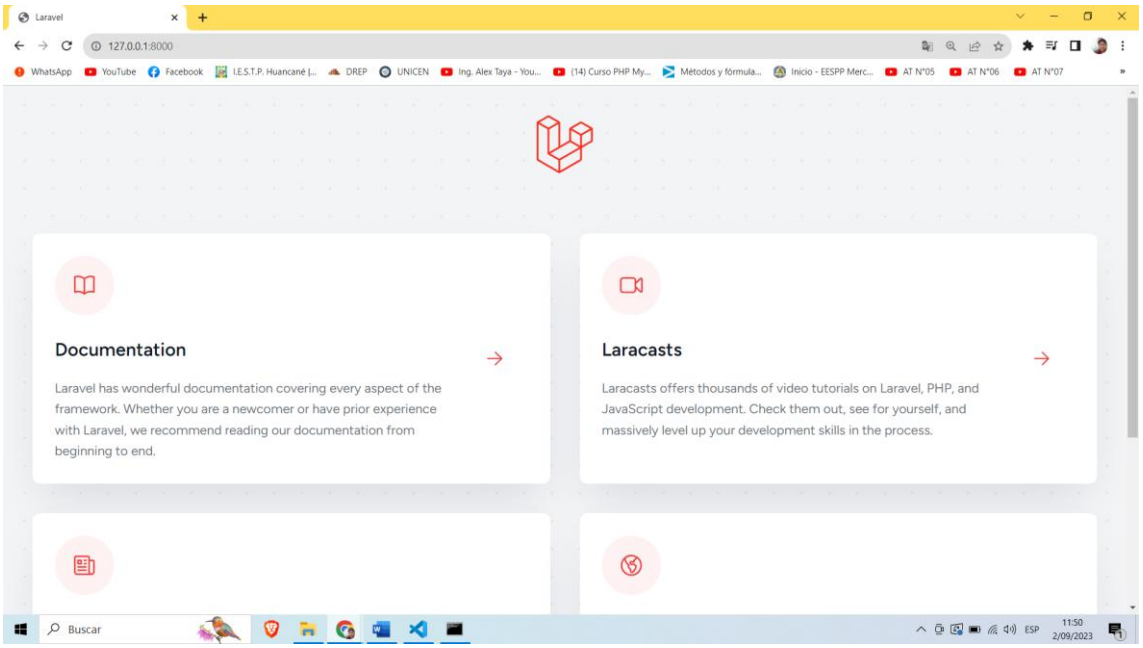




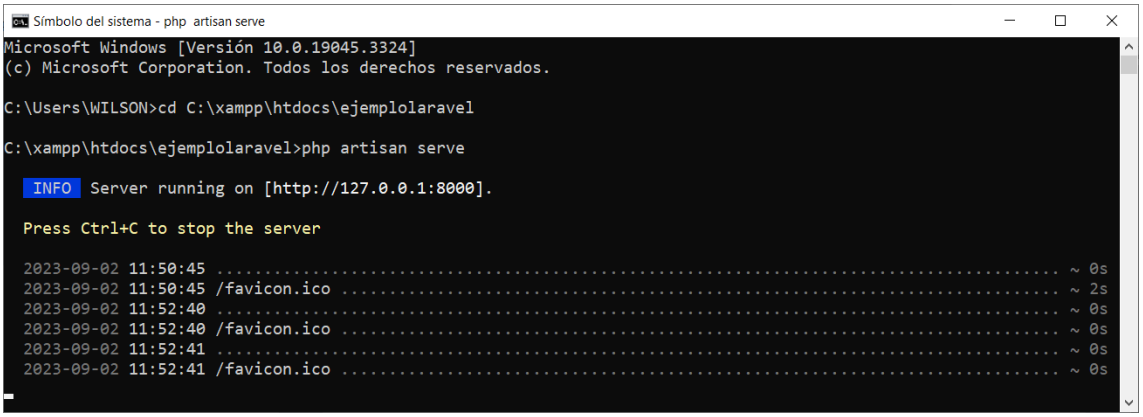
Escribir **php artisan serve** (si recuerdan, nos da una dirección IP que es nuestro localhost, y un puerto que es el 8000)

Ese IP vamos a colocarlo en el navegador

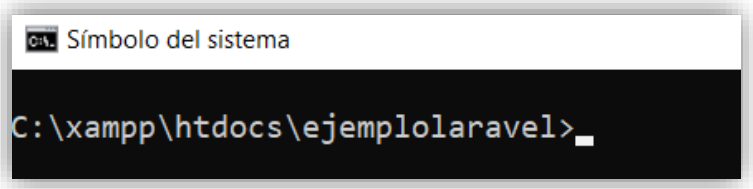




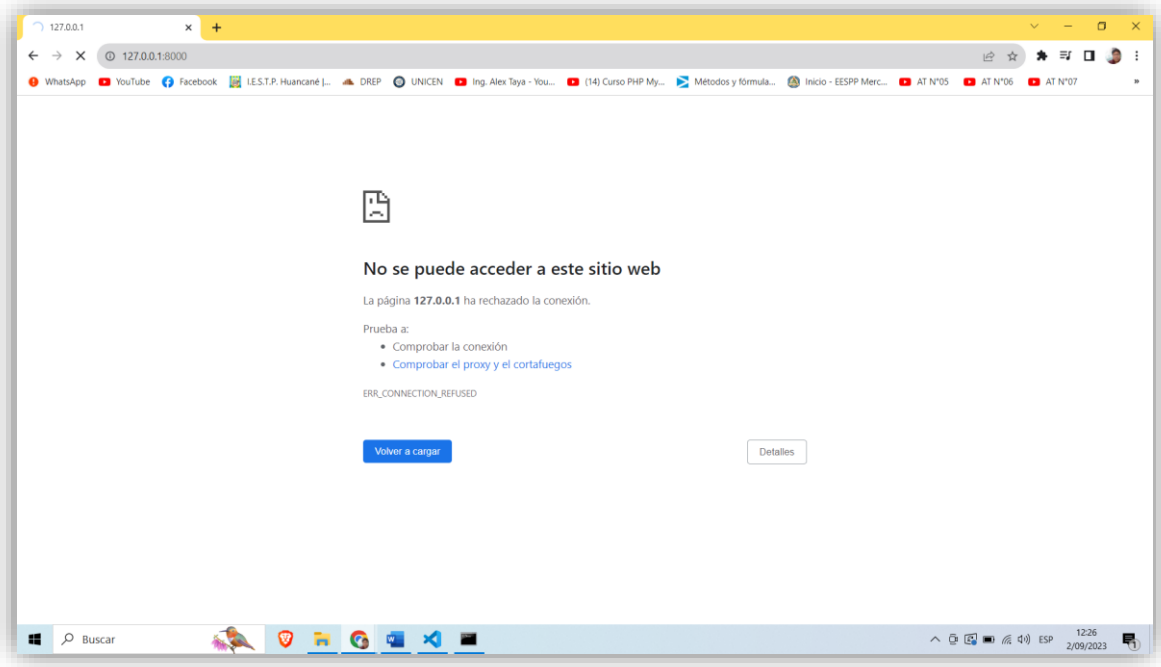
Esta ejecución ya nos carga directamente la pantalla principal del Framework, luego vamos a la consola y veremos todas las peticiones que se van reflejando y si actualizamos la página abierta, veremos que la petición se vuelve a hacer.



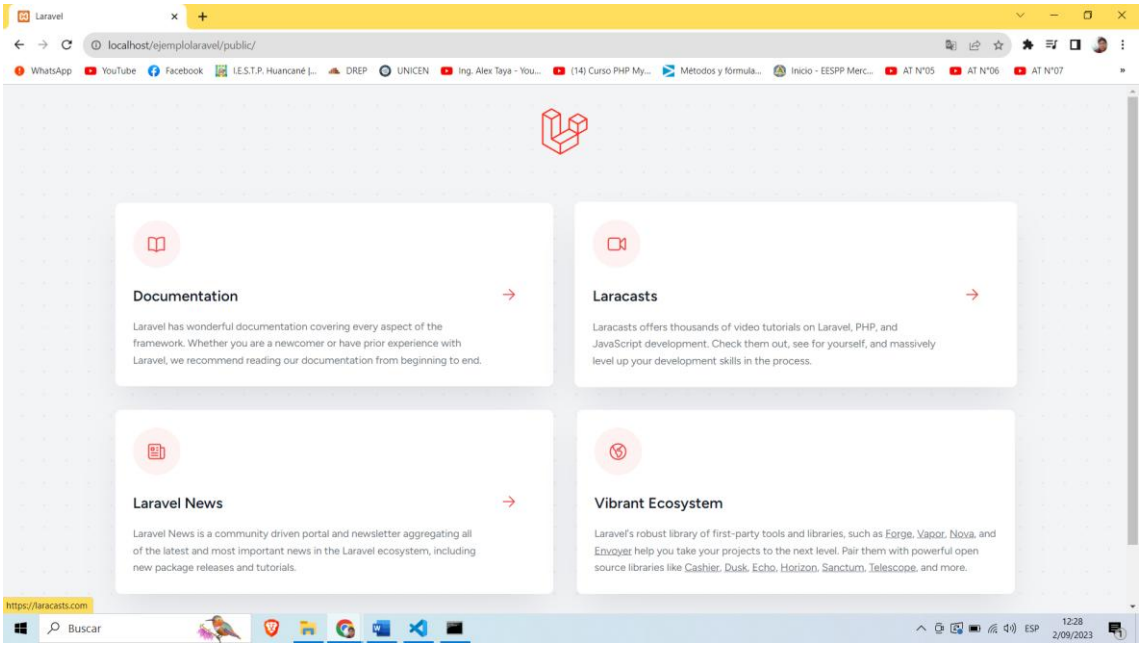
Vamos a cerrar este proceso con las teclas **CTRL + C** luego podemos limpiar pantalla con CLS



Luego podemos regresar al navegador y verificar si sigue cargando la pagina con el uso de ARTISAN. Como puede ver, al actualizar la página vemos que ya no carga el Laravel porque hemos cerrado este servidor.



Pero podemos observar que con Apache sigue funcionando, porque es independiente.



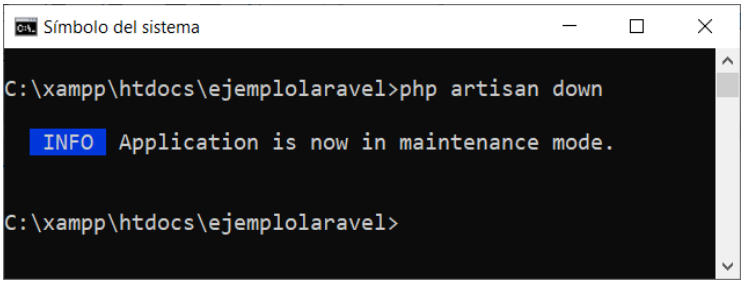
Y qué pasa si le queremos dar mantenimiento al proyecto, nos puede dar la configuración para agregar este mantenimiento y sea vea como una vista predefinida. Si lo hacen con Apache tal vez necesiten desactivar Apache o quitar la carpeta, pero ARTISAN nos ayuda a realizarlo de manera más sencilla.

3.3 COLOCAR ARTISAN EN MODO MANTENIMIENTO

En la ventana de comandos, dentro de nuestro proyecto vamos a colocar:

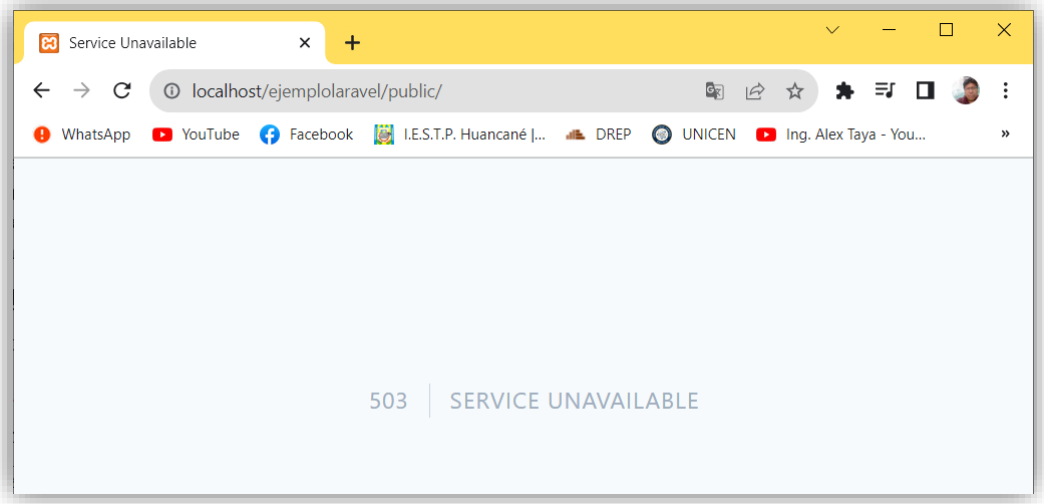
3.3.1 php artisan down

Aquí dice que la aplicación está ahora en modo de mantenimiento.



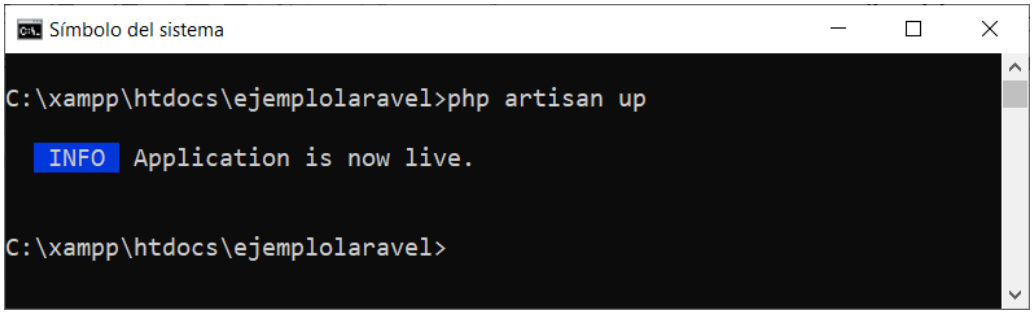
Obviamente no lo vamos a poder visualizar porque aun no hemos ejecutado el servidor ARTISAN. Pero si lo vemos con apache, miren lo que sucede.

Ojo que no hemos detenido apache, ni nada, solamente cambiamos la aplicación al modo mantenimiento mediante ARTISAN, es decir, ya de forma automática detecta la configuración que está en modo mantenimiento, y nos lanza esa pantalla.



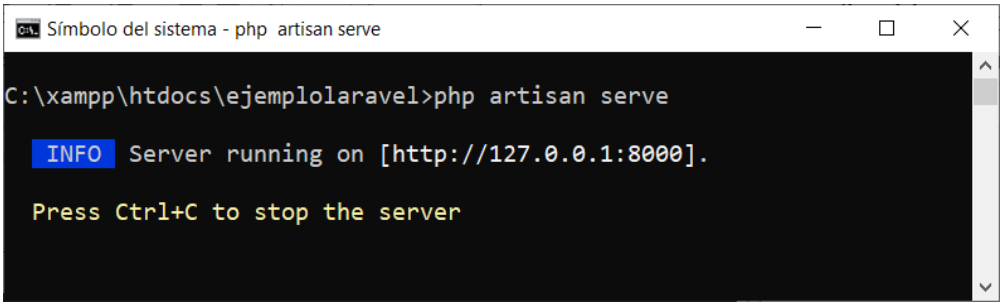
3.4 QUITAR EL MODO MANTENIMIENTO

Colocamos **php artisan up**

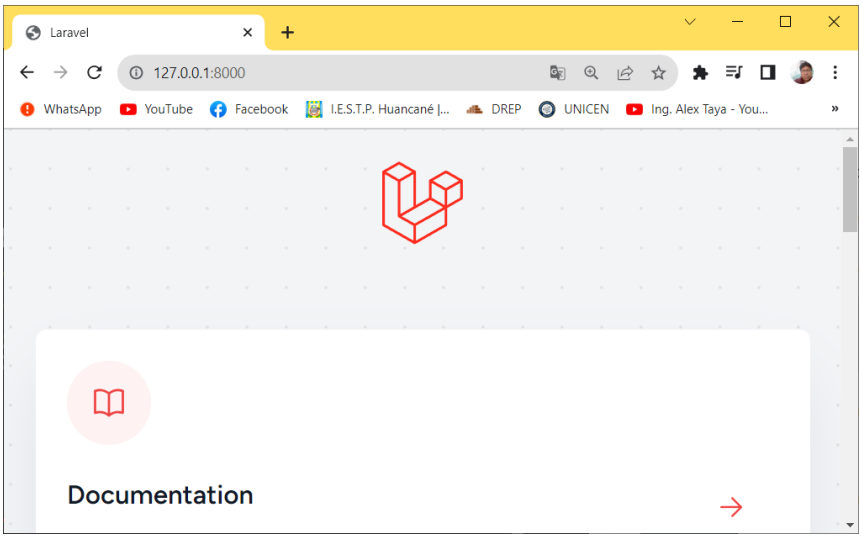


Hagamos una prueba

Activamos artisan



Podemos abrir la aplicación con artisan.

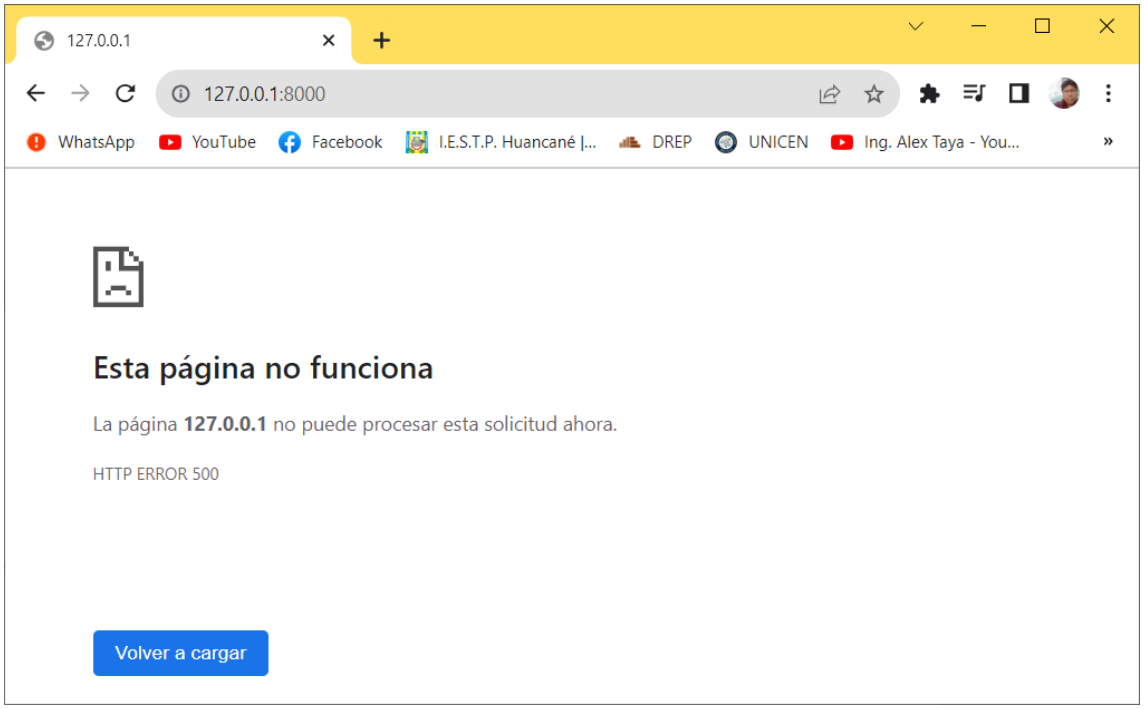


En caso de que quieras agregar algún dato procura que no haya espacios, probablemente no visualice nada, porque hay error de configuración por ejemplo en .env si APP_NAME pones LARAVEL OTRO

```
.env
1   APP_NAME=Laravel OTRO
2   APP_ENV=local
3   APP_KEY=base64:9W6pzof4kMbTcVIDdwVumMKmR4FzPe4ZWZC35PMQghw=
4   APP_DEBUG=true
5   APP_URL=http://localhost/ejemplolaravel/public
```

Luego podemos comprobar:

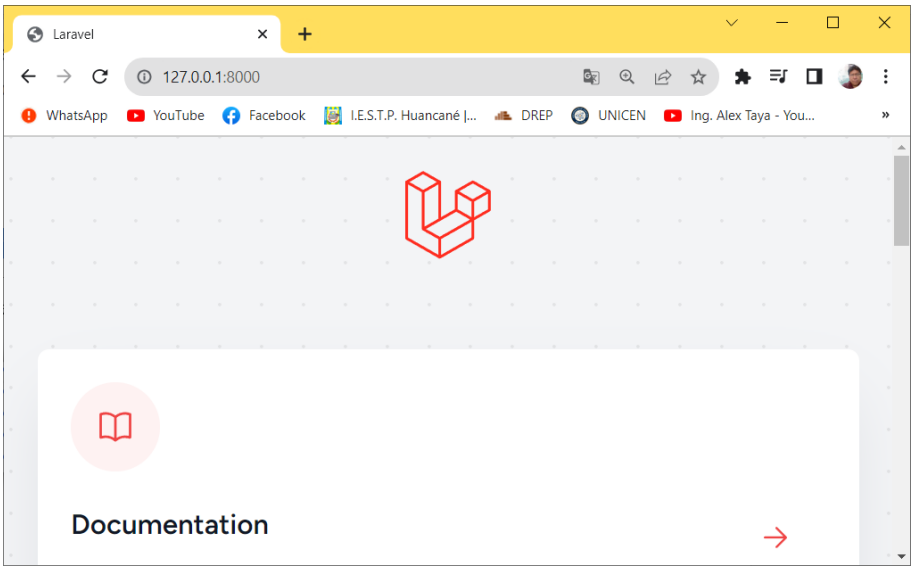
Vemos que no muestra nada, hay que tener en cuenta de hacer bien los cambios.



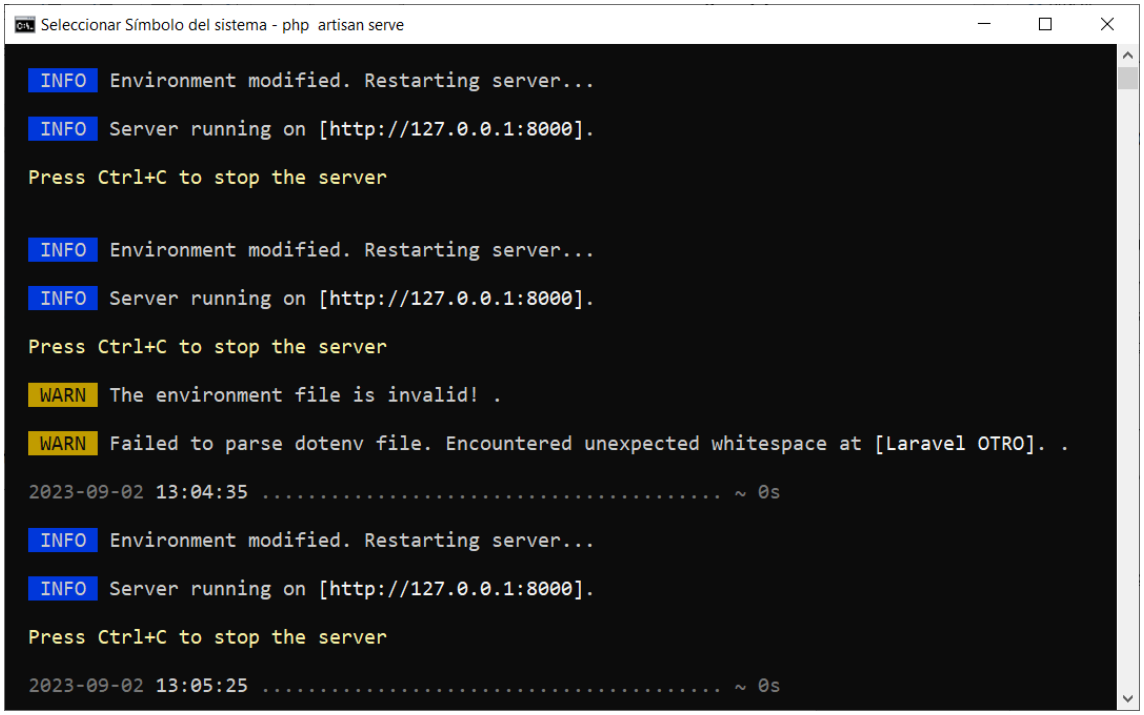
Ahora si vas a hacer cambios con espacios, deberás ponerlo entre comillas dobles.

```
1   APP_NAME=" Laravel Otro"
2   APP_ENV=local
3   APP_KEY=base64:9W6pzof4kMbTcVIDdwVumMKmR4FzPe4ZWZC35PMQghw=
4   APP_DEBUG=true
5   APP_URL=http://localhost/ejemplolaravel/public
```

El resultado sigue intacto.



Si vemos la consola, vemos todas las acciones e incluso los errores.

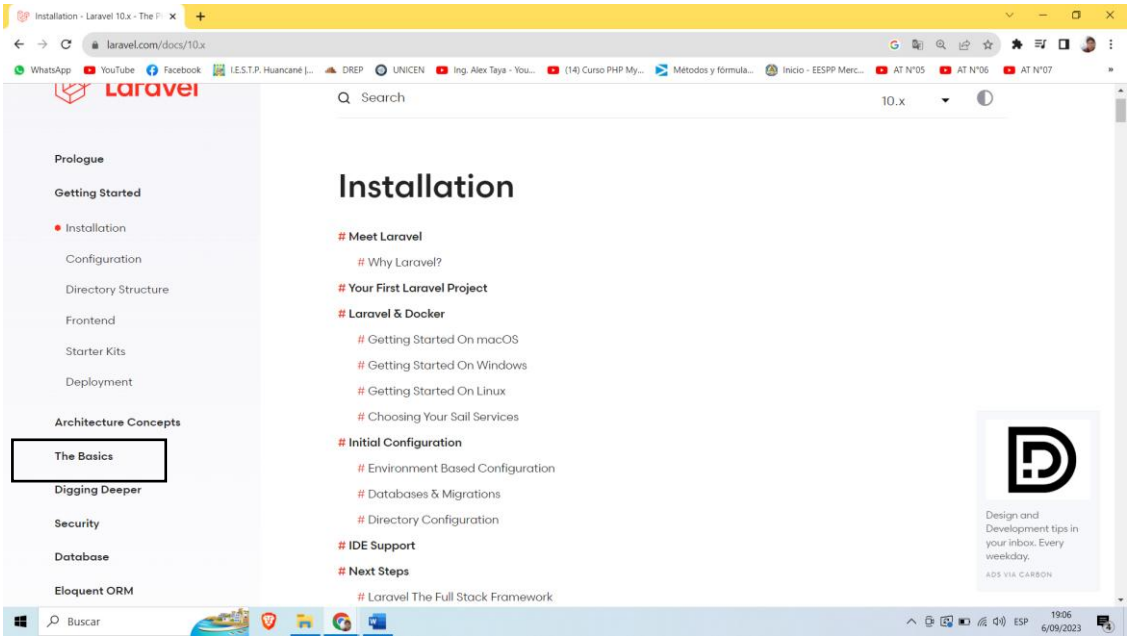


CAPÍTULO 4

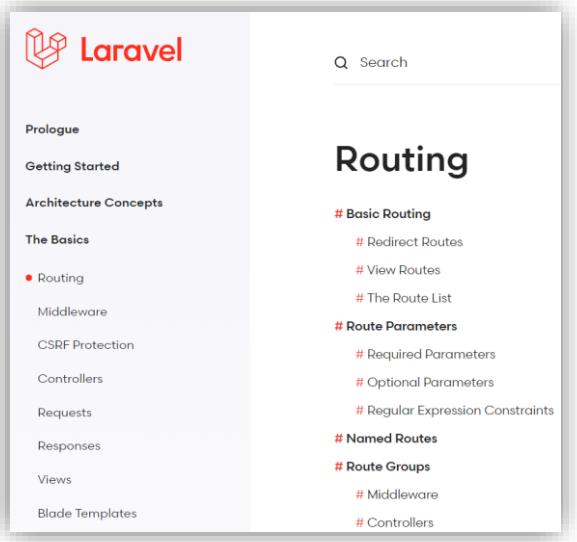
RUTAS EN LARAVEL

En esta actividad vamos a trabajar con rutas, puede obtener una guía de la página de Laravel, donde se encuentran detalles de los comandos sobre lo que se puede hacer con este Framework.

Ir a la documentación de la página de laravel. Ir a **the basics**



Al hacer clic en **the basics**, se va a visualizar temas, usted deberá de estudiar todas esas opciones, por lo pronto hagamos clic en **Routing.**

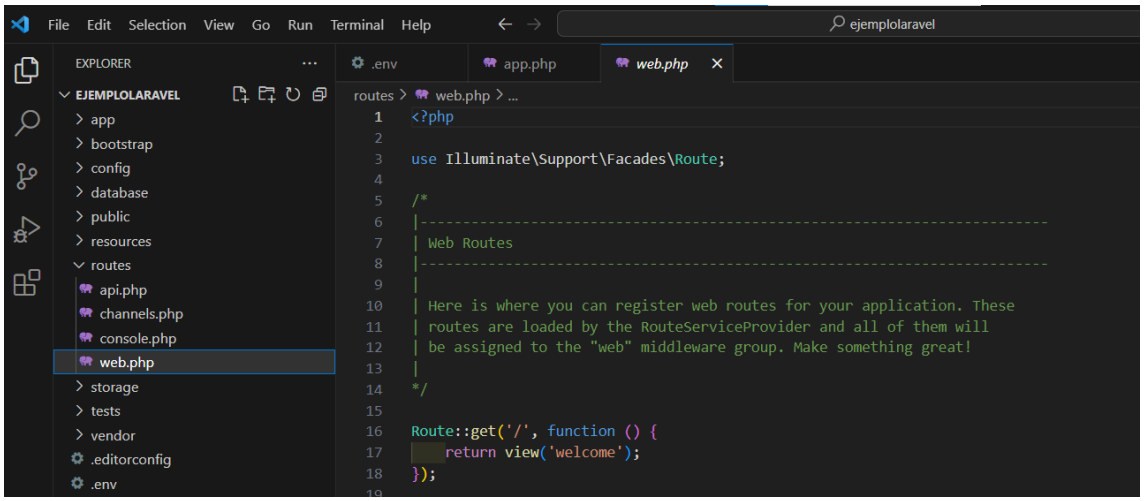


Esas son las rutas que van a utilizar en el Framework, para poder acceder a diferentes apartados.

Abrir Visual Studio Code, también vamos a abrir con localhost nuestro proyecto en el navegador.

4.1 RUTAS

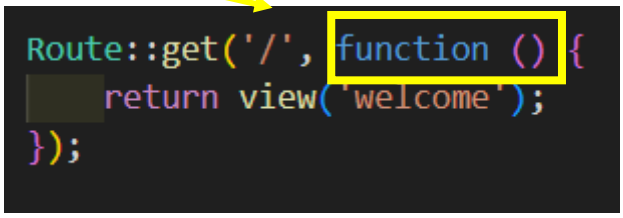
Trata de cómo podemos ingresar a diferentes apartados, para esto vamos al Framework, luego hacemos clic en **Routes,** luego clic en **web.php**



Ahí es donde inicia todo para comenzar con el Framework, en este caso la ruta es el / (Slash) o el inicio de nuestra aplicación es esta función.

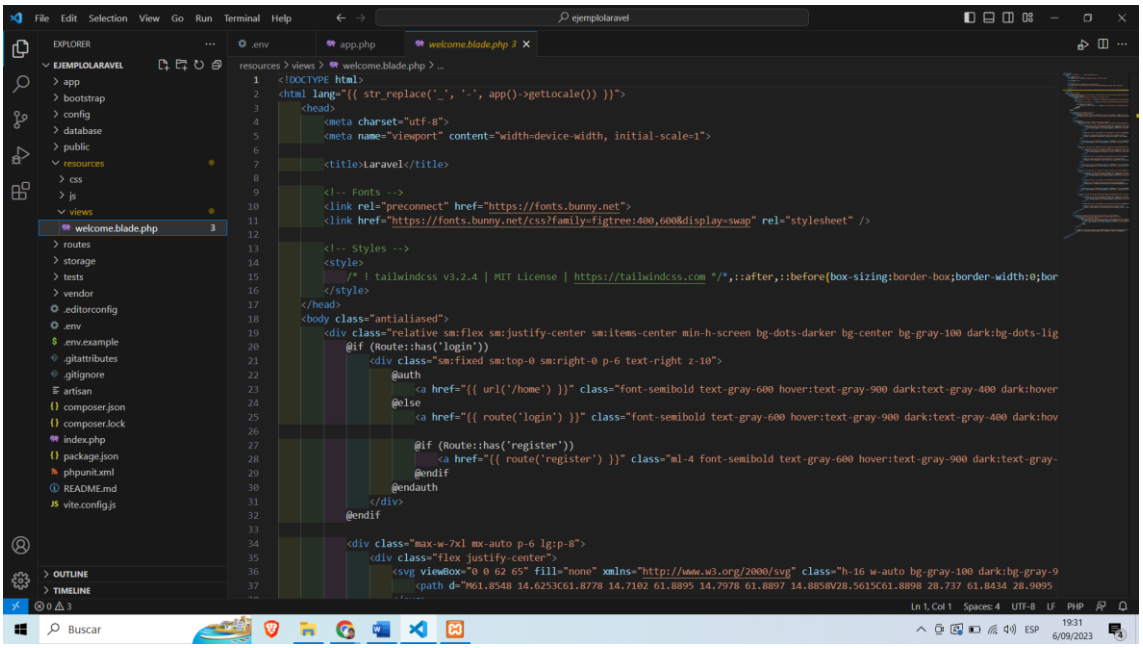
Cuando ingresamos a la página principal http://localhost/ejemplolaravel/public/ se dispara esta función, la cual retorna una vista que se llama **welcome**

Página principal



Vamos a ver dónde redirige la función que nos dice que retorna la función **view**

Welcome es una página que está en la carpeta **resource/views/welcome.blade.php**



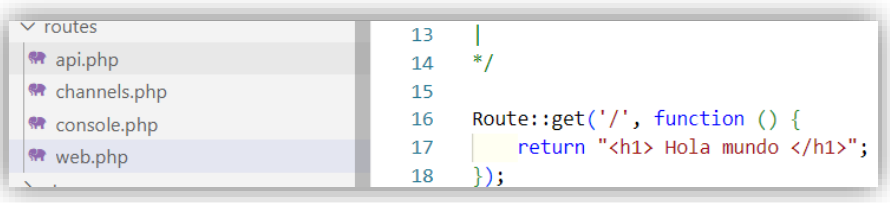
Esta es una página en HTML normal que tiene algunos tags (etiqueta) de laravel, como por ejemplo @if (Route::has('login'))

```
<body class="antialiased">
    <div class="relative sm:flex sm:justify-center sm:item
        @if (Route::has('login'))
            <div class="sm:fixed sm:top-0 sm:right-0 p-6 t
                @auth
                    <a href="{{ url('/home') }}" class="fo
                @else
                    <a href="{{ route('login') }}" class="

                    @if (Route::has('register'))
                        <a href="{{ route('register') }}"
                    @endif
                @endauth
            </div>
        @endif
```

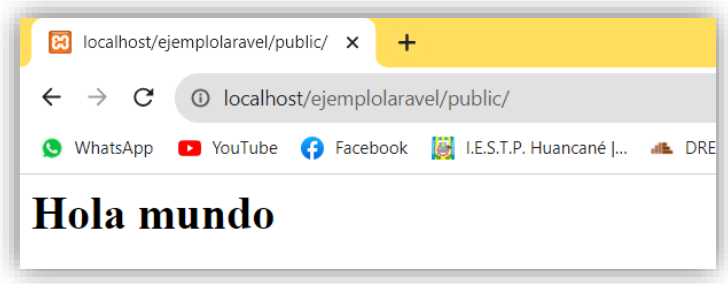
Con esto referimos al uso de plantillas **Blade,** que detallaremos posteriormente, podemos observar que hay código php pero de una forma diferente, y es como así lo trabaja Laravel para mostrar las vistas, es por eso que se llama el Framework con código elegante, porque no estamos abriendo ni cerrando php, sino que ya se va trabajar de un modo diferente.

Abramos nuevamente la ruta **web.php**, recordemos que **welcome** nos permitirá abrir **welcome.blade.php** que se encuentra en la carpeta resources/views.

En el archivo **web.php** vamos a poder definir las diferentes rutas que vamos a estar usando a lo largo del desarrollo de nuestra aplicación, para este ejemplo, haremos un cambio, quitaremos el **view(welcome)** y solo vamos a colocar "Hola mundo", pero lo hacemos con tag de HTML h1 para que sea un encabezado mas grande, luego grabamos y ejecutamos la aplicación.



Resultado:



Como podemos ver ahora tenemos nuestra primera modificación, el famoso **Hola mundo**, pero podemos generar diferentes rutas, así como también es importante saber que estamos trabajando con unos métodos, acá estamos llamando con el método **get.**

4.2 EL MÉTODO GET

El método **get** es cuando hacemos una solicitud mediante la URL, si nosotros quisiéramos guardar información mediante un formulario, comúnmente lo hacemos con el método **post**, y en las rutas también vamos a poder usar ese método, pero lo haremos en clases posteriores, por el momento trabajaremos con el método **get.**

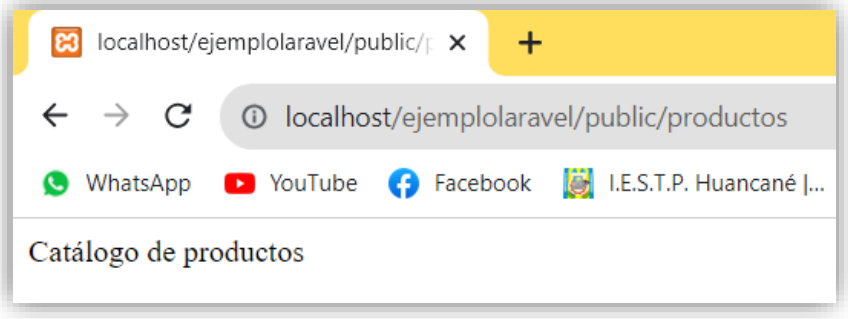
Ejemplo: vamos a crear una nueva ruta, para el módulo de productos a la cual le agregaremos una función anónima, es decir, no le ponemos nombre, luego abrimos la llave y colocamos un return para catálogo de productos, y podemos ver su comportamiento.

```
route::get('productos', function(){
    return "Catálogo de productos";
});
```

Para abrir la ruta creada, al link http://localhost/ejemplolaravel/public/ le agregaremos productos.

El link completo seria: http://localhost/ejemplolaravel/public/productos

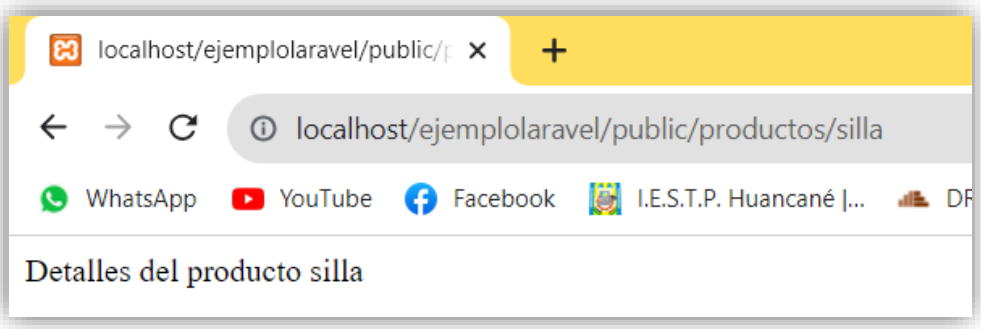
Como podemos ver, al ejecutar la página, ya nos muestra el contenido *catálogo de productos*.



Pero, podemos agregar más detalles, como por ejemplo el detalle de un producto individual, para la cual se adiciona el / como separador, y para aceptar parámetros en nuestras rutas se debe de hacer mediante llaves, y dentro de estas llaves vamos a colocar el nombre de este parámetro. Luego ponemos la función anónima, pero recibiendo un parámetro con una variable que se llame producto y finalmente vamos a agregar un return de detalles del producto, luego agregamos la variable producto.

```php
Route::get('/', function () {
    return "<h1> Hola mundo </h1>";
});

route::get('productos', function(){
    return "Catálogo de productos";
});
route::get('productos/{producto}', function($producto){
    return "Detalles del producto $producto";
});
```

Ahora como entramos al http://localhost/ejemplolaravel/public/productos y agregamos silla: http://localhost/ejemplolaravel/public/productos/silla



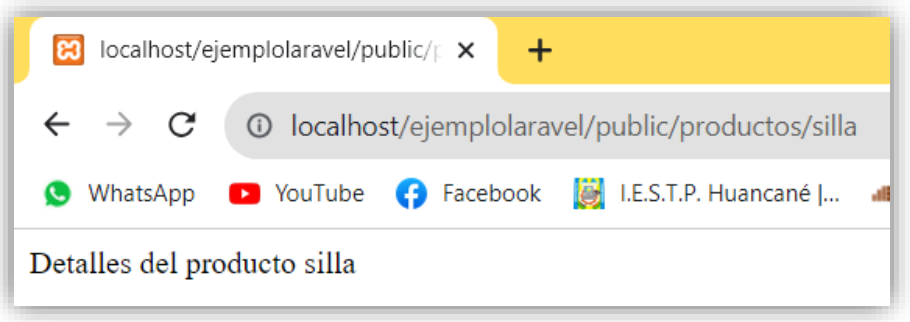
El parámetro silla lo recibe de esta forma (se encuentra entre llaves):

```php
route::get('productos/{producto}', function($producto){
    return "Detalles del producto $producto";
});
```

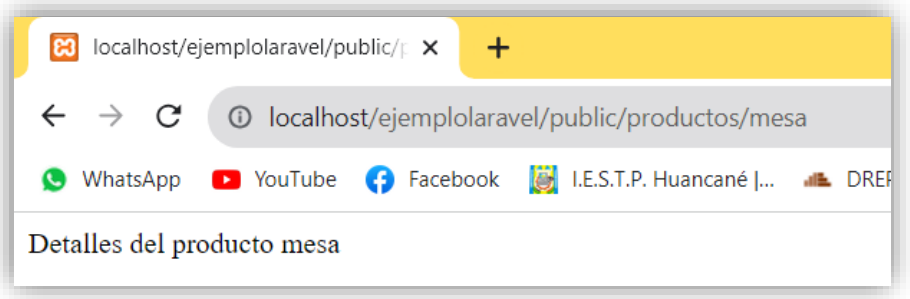
En donde se le indica que delante de la carpeta producto va a tener un nuevo parámetro, y este parámetro se almacena en la variable $producto

```php
route::get('productos/{producto}', function($producto){
    return "Detalles del producto $producto";
});
```

Ya ejecutando: http://localhost/ejemplolaravel/public/productos/silla podemos decir silla se almacena en la variable $producto y por eso se muestra en el resultado.

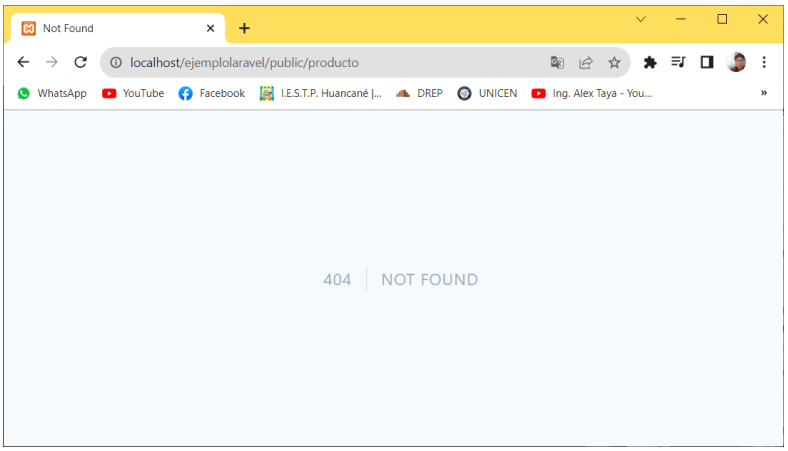


Podemos cambiar a mesa (lo hará de forma dinámica)



Ahora ¿Qué sucede si agregamos una ruta que no existe? Como, por ejemplo:

http://localhost/ejemplolaravel/public/producto (no ejecuta, dice que no existe)

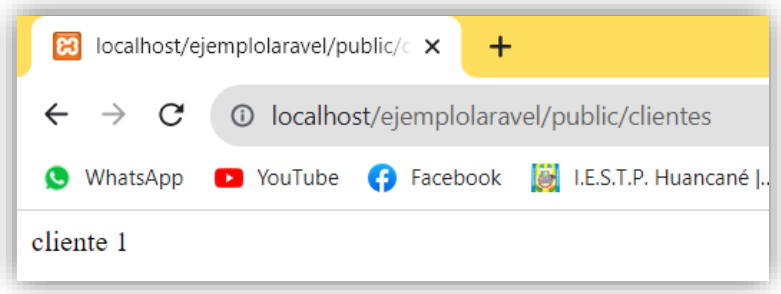


Vamos a hacer una modificación para saber cómo podemos recibir datos nulos y el cómo los podemos procesar. Vamos a agregar otra ruta get, pero ahora vamos a trabajar con clientes, luego /, y vamos a agregar dentro de las llaves el **id** pero colocamos el signo

de interrogación **?** después del nombre **id**, luego la función, luego vamos a agregar un valor predeterminado como por ejemplo **$id = 1** (esto lo vimos en php cuando llamamos a una función y nos solicita un parámetro, y si no le enviamos el parámetro con el valor 1, Laravel lo tomara como un valor predefinido, cosa que asi no podrá generar ningún error y va poder generarse esa función). Luego colocamos un **return** cliente e imprimimos el **id**

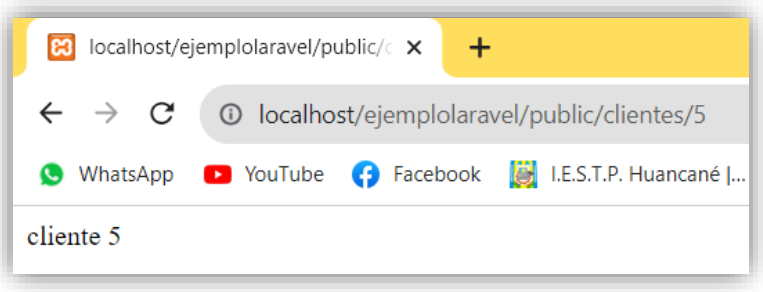
```
Route::get('/', function () {
    return "<h1> Hola mundo </h1>";
});
route::get('productos', function(){
    return "Catálogo de productos";
});
route::get('productos/{producto}', function($producto){
    return "Detalles del producto $producto";
});
Route::get('clientes/{id?}', function($id = 1){
    return "cliente $id";
});
```

Resultado usando la ruta clientes



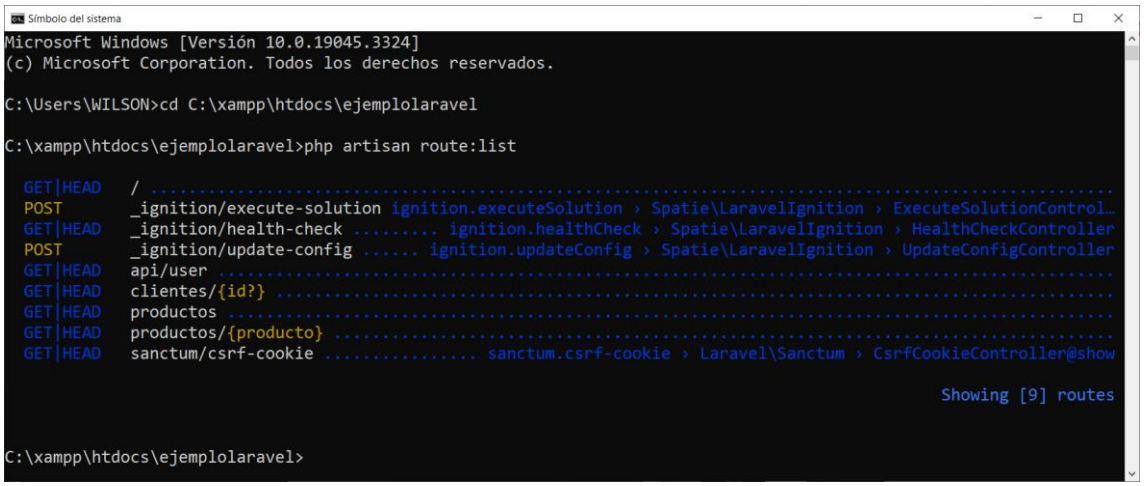
Vemos que ha enviado como resultado cliente 1, es decir acá está tomando un valor predefinido que nosotros le estamos dando en function, así es que el como se recibe esta información, en caso de que por algún motivo no lo esté enviando en la ruta.

Entonces que sucede si en la ruta agregamos el número 5: http://localhost/ejemplolaravel/public/clientes/5 , *veremos que ya lo reconoce*



Laravel ya sabe que le estamos enviando un valor y nos imprime de forma correcta. Aquí (web.php) es muy importante que nosotros establezcamos todo lo que vamos trabajando.

Para ver todas las rutas que hemos generado utilizaremos el cmd, nos posicionamos en la carpeta del proyecto y vamos a colocar **php artisan route:list** luego presionamos Enter.



Luego de ejecutar la línea de comando, nos va a mostrar todas las rutas que hemos generado, así como el tipo que estamos enviando, como por ejemplo del método **get** y algunas de **post** y mas abajo, las que nosotros hemos generado.

Ahora vamos a crear otra **ruta,** pero vamos a combinar dos rutas ya establecidas y dos dinámicas, como por ejemplo con la ruta de cliente vamos a colocar:

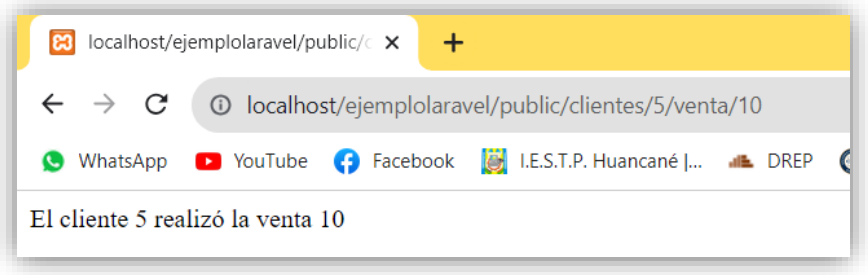
Ojo que acá estamos colocando 2 parámetros dinámicos (`{id} {idVenta}`) y 2 establecidos (`clientes venta`), también agregamos la función anónima para que podamos recibir un dato, en este caso el $id y el $idVenta. Luego vamos a retornar el cliente $id realizo la venta $idVenta.

Si quieres agregar algún otro valor predefinido, lo puedes hacer de manera similar.

```
Route::get('clientes/{id}/venta/{idVenta}', function($id, $idVenta){
    return "El cliente $id realizó la venta $idVenta";
});
```

Si ejecutamos la ruta http://localhost/ejemplolaravel/public/clientes/5/venta no va ejecutar porque le falta el idVenta, pero si colocamos un valor como 10.

http://localhost/ejemplolaravel/public/clientes/5/venta/10

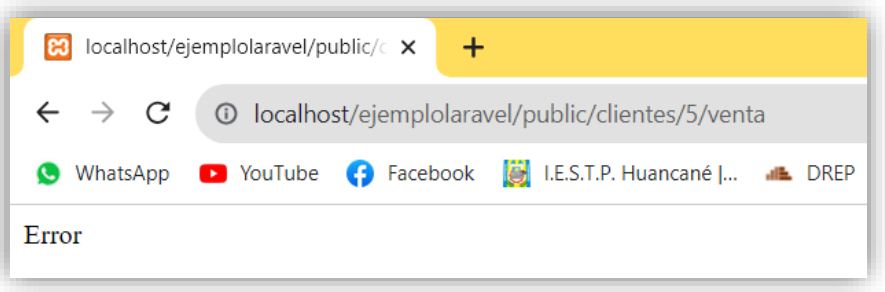


Ahora si queremos que nos muestre algo, aunque no hayamos puesto el idVenta, la solución estaría en colocarle forzosamente el símbolo ? después de idVenta. Además, debemos de colocar un if como vemos abajo:

```
Route::get('clientes/{id}/venta/{idVenta?}', function($id, $idVenta = null){
    if($idVenta == null)
    {
        return "Error "
    }
    return "El cliente $id realizó la venta $idVenta";
});
```

Para probar vamos al navegador y pondremos la ruta sin el valor para idVenta http://localhost/ejemplolaravel/public/clientes/5/venta

Resultado:



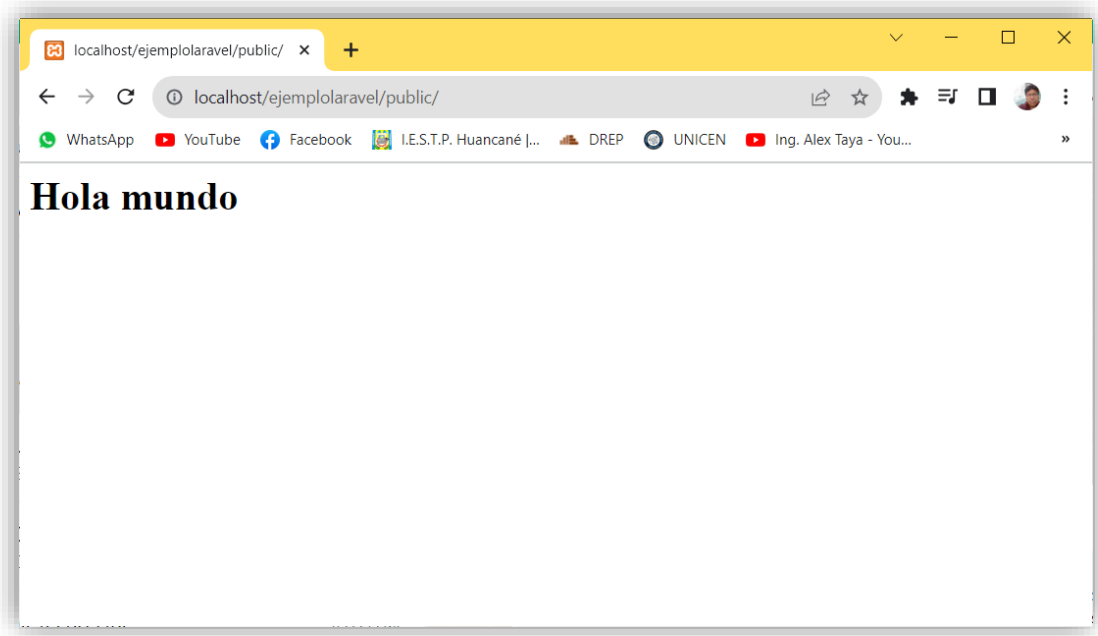
Vemos que nos muestra error, con esta acción estaríamos controlando lo que el usuario no está ingresando.

CAPÍTULO 5

CONTROLADORES

En esta actividad aprenderemos el modelo MVC, es decir, la modelo vista controlador, y vamos a iniciar con los controladores.

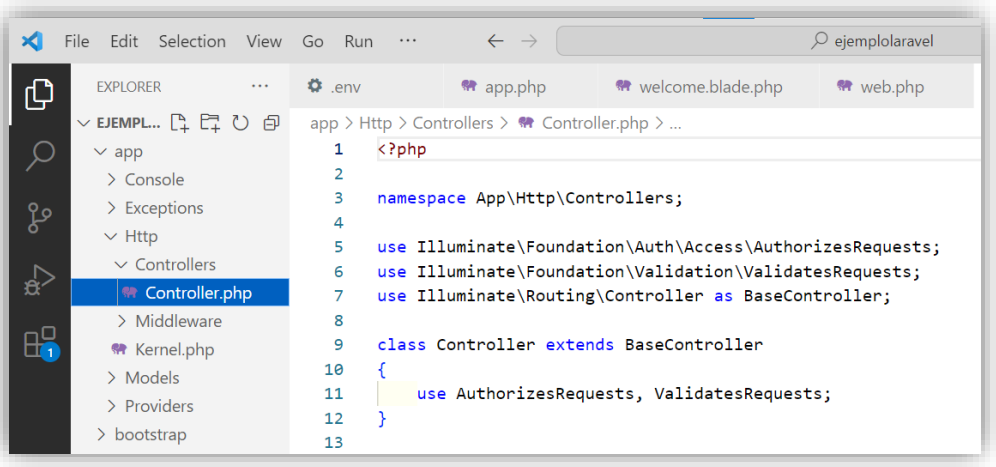
Para esto, vamos a abrir nuestro proyecto: http://localhost/ejemplolaravel/public/



Ahora vamos a nuestro proyecto en **visual studio code,** hasta ahora ya hemos aprendido a generar nuestras rutas para poder redireccionar a cada uno de los elementos que va trabajar el usuario, pero todo ha sido con rutas en el archivo **web.php**, pero ahora vamos a generar un controlador.

5.1 GENERACIÓN DE UN CONTROLADOR

Ir a la carpeta **App** del proyecto, luego la carpeta **Http,** y luego la carpeta **Controller**, ahora abrir el archivo **Controller.php**

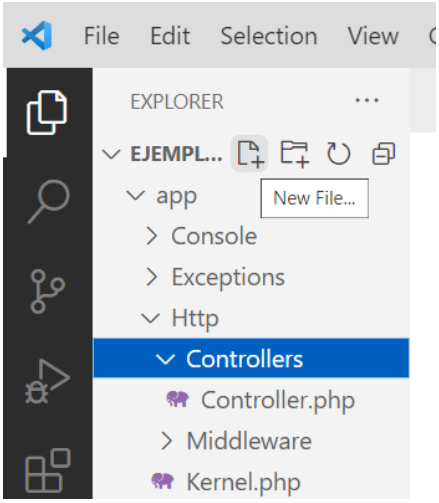


Este archivo es una base para poder generar nuestros controladores, de tal manera que sean muy simples, porque si no, necesitaríamos agregar todo ese código que aparece en ese archivo para que pueda trabajar con él.

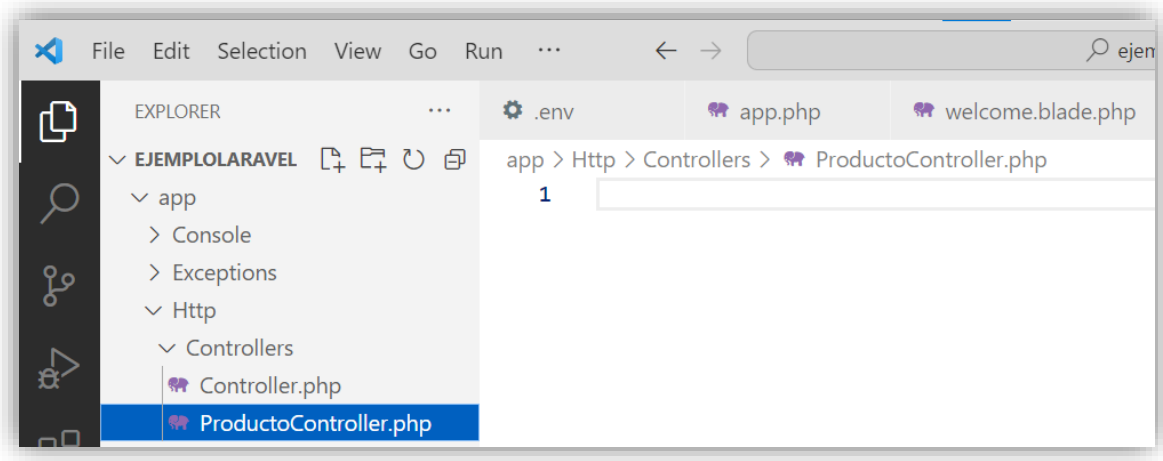
Un controlador es donde vamos a poder agregar toda la lógica de nuestro módulo o de nuestras funciones, como por ejemplo la creación de un formulario, es aquí donde hacemos todas las validaciones, puedo obtener información que el usuario está introduciendo y su posterior procesamiento.

Empecemos, primero vamos a generar un controlador, lo vamos a hacer de dos formas, la forma manual y la forma automática.

Vamos a crear un **Controller**, agregamos un nuevo archivo o módulo que llamaremos **ProductoController.php**



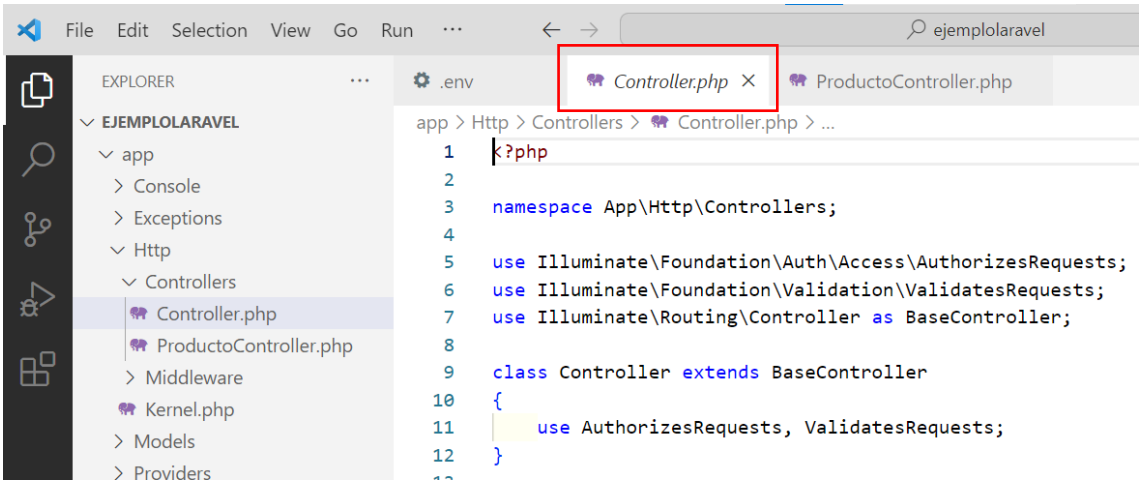
Hacemos clic en **New File** y escribes el nombre del controlador



Es importante colocar la palabra **Controller** al final, y después **punto php,** para que lo puedan reconocer. Se recomienda colocar el nombre del módulo en singular e iniciando con la primera letra en Mayúscula, así como Controller también con la primera letra mayúscula.

Iniciamos php, luego vamos a indicar el **Namespace**, que es el lugar donde estamos trabajando, y vas a colocar toda la ruta que tenemos en el controlador, de la siguiente manera: **namespace App\Http\Controllers ;**

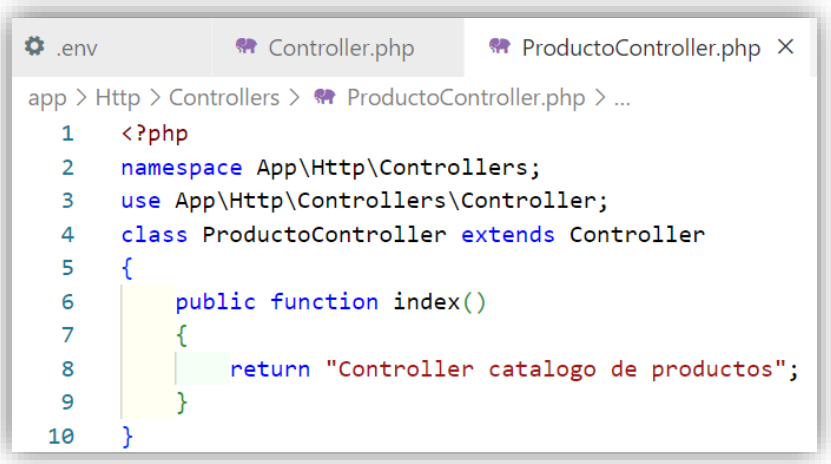
Esta es nuestra ruta donde vamos a estar trabajando este archivo, ósea el **Namespace,** después de esto podemos agregar un **use App\Http\Controllers\Controller;** porque vamos a indicar que necesitamos usar o importar el archivo **Controller.php**



Después de todo, al final un controlador es una **clase**, y vamos a crearla: (Debe ser el mismo nombre del archivo **.php** pero vamos a heredar (**extends**) de la clase **Controller** que está en la parte donde utilizamos **use**)

5.1.1 Class ProductoController

Dentro de esta clase ya podemos generar nuestras propias funciones



Con esto, hemos generado nuestro controlador y tenemos nuestra función index, ahora la pregunta es ¿Cómo podemos acceder a esta función? Recordemos que necesitaremos hacer modificaciones en **web.php,** básicamente en las rutas, vamos a hacer un cambio en una de las rutas que ya habíamos creado, vamos a indicar que va seguir siendo por el método GET, que se llame **productos**, pero ahora ya no vamos a llamar a la función anónima, asi lo vamos a borrar, lo que vamos a indicar es que vamos a agregar una clase, esto se debe hacer entre corchetes, luego el nombre de la clase

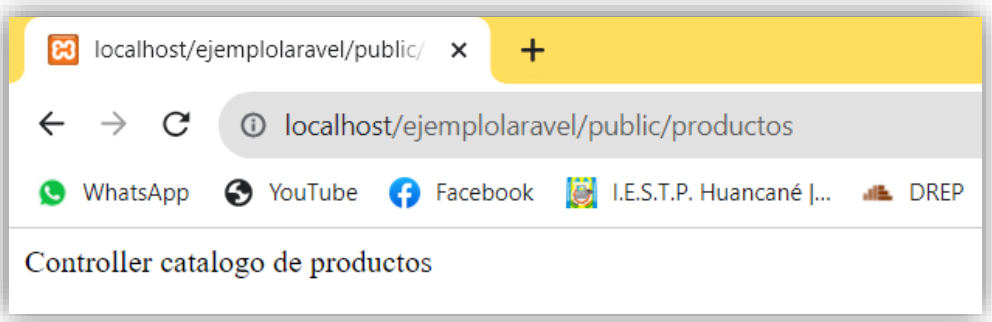
ProductoController::class, y el nombre de la función **index**

Entonces cuando nos redirija a productos tiene que mostrar el index, pero de nuestro controlador que ya estamos trabajando. También va a ser importante que en la parte superior agreguemos el use de la ruta **ProductoController**, acá al parecer visual studio lo agrega automáticamente en la parte superior.

Pero ten cuidado, porque si no lo coloca, no nos va redireccionar correctamente.

```php
<?php

use App\Http\Controllers\ProductoController;
use Illuminate\Support\Facades\Route;

/*
|--------------------------------------------------------------------------
| Web Routes
|--------------------------------------------------------------------------
|
| Here is where you can register web routes for your application. These
| routes are loaded by the RouteServiceProvider and all of them will
| be assigned to the "web" middleware group. Make something great!
|
*/

Route::get('/', function () {
    return "<h1> Hola mundo </h1>";
});

route::get('productos', [ProductoController::class, 'index']);
```

Hagamos la prueba.

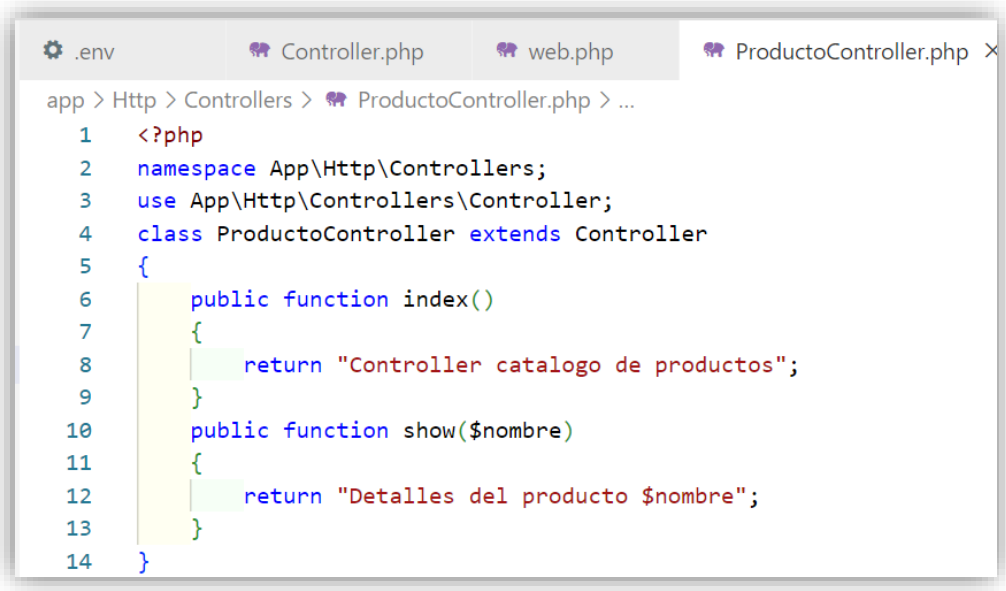


Como vera si está trabajando con el index, y está mostrando lo que retorna index.

```php
class ProductoController extends Controller
{
    public function index()
    {
        return "Controller catalogo de productos";
    }
}
```

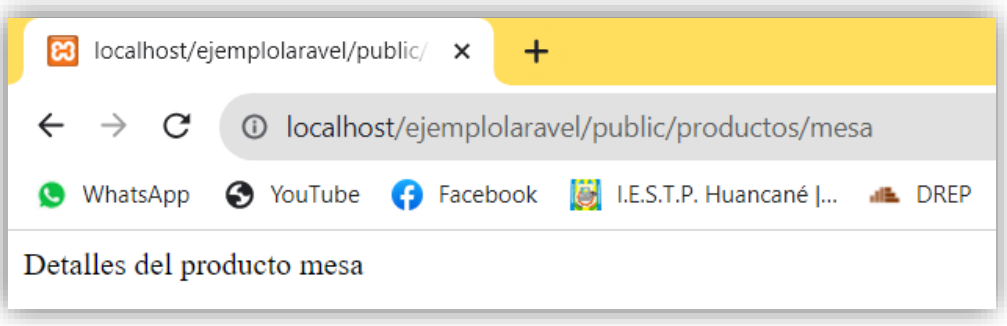
Muy bien, ya estamos ingresando al controlador desde una ruta.

Ahora vamos a hacer otra función **show** para visualizar el producto, escribimos *detalles del producto* ***e imprimimos el nombre.***



Vamos también a las rutas (**web.php**), y vamos a editar la parte en el que trabajamos con variables.

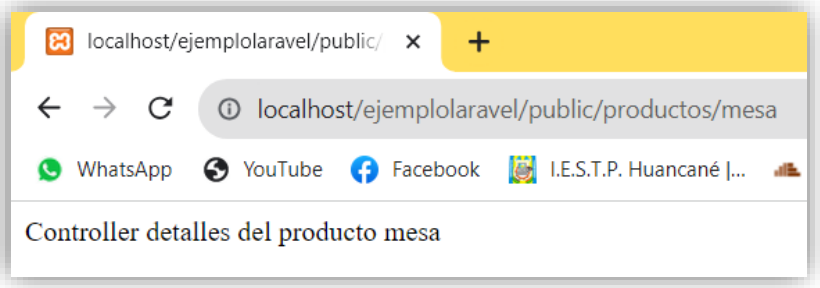
Vamos a hacer la prueba (no se olvide de grabar), agregamos **mesa** para que se asigne a la variable.



Se encuentra bien, pero ahora vamos a cambiar el mensaje para que diga Controller, para que veamos que si lo está redireccionando y no este trabajando con las rutas que anteriormente habíamos colocado.

```php
public function show($nombre)
{
    return "Controller detalles del producto $nombre";
}
```

Resultado.



El manejo del controlador fue de forma manual, Laravel tiene un método para poder generar de forma automática estos Controllers.

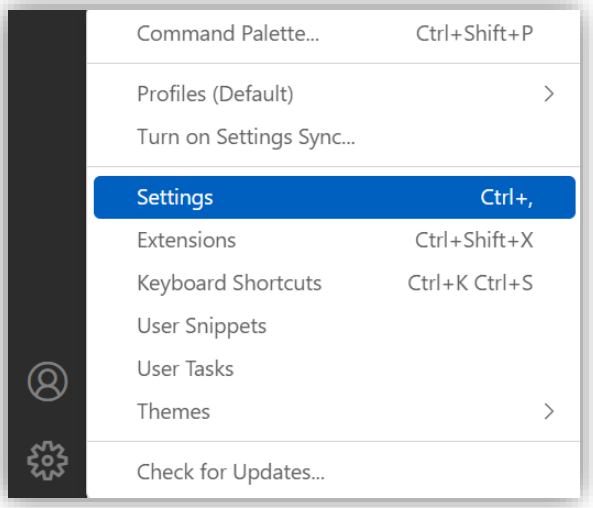
Podemos hacerlo de dos formas:

1. Puedes hacerlo directamente en Visual Studio Code, hacer clic en menú ver/terminal

Es una terminal de Windows o una Power Shell y aquí antes de agregar los comandos necesitamos verificar si tenemos agregado PHP al path y a visual studio code.

Colocamos php -v

Y si nos muestra la versión significa que, si está correcto, en caso de que no nos reconozca el comando, lo que debe hacer es agregarlo, para ello ir al administrador, elegir configuración o settings.

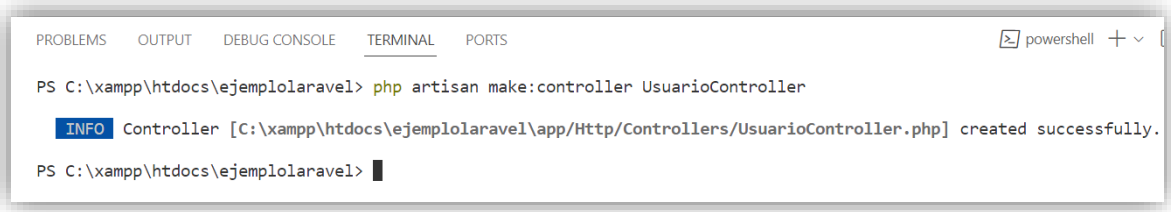


En la parte para buscar poner **php**, buscas, pero antes cierras el Power Shell de Visual Studio Code. Vas a buscar PHP: Executable Path, dar clic en editar en settings.json y ahí nos va colocar el path para ejecutar php

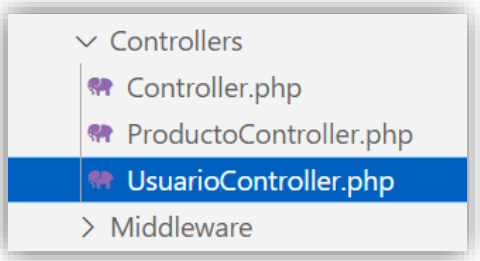
Ahí debes de colocar la ruta de donde se encuentra el php.exe para que visual studio code reconozca este comando.

Ahora vamos a ver el comando que nos va permitir generar nuestros controladores de manera más rápida, para ello debemos abrir el terminal y poner la ruta del proyecto, luego escribir **php artisan make:Controller nombre_del_controlador** luego presionamos Enter.

Y nos dice que ya se creó este controlador



Podemos verificar si en realidad fue creado el controlador, para ello ir a la ruta de los controladores y verificar. Podemos ver que si está.



Si abrimos el archivo nos podemos dar cuenta que ha creado la estructura de un controlador, entonces es más rápido generarlo. Aquí nos agrega una línea del **use** que dice que es de Request

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UsuarioController extends Controller
{
    //
}
```

Este **Request** nos va permitir obtener algunos parámetros que se envíen por el método Post, por el método Get u otros.

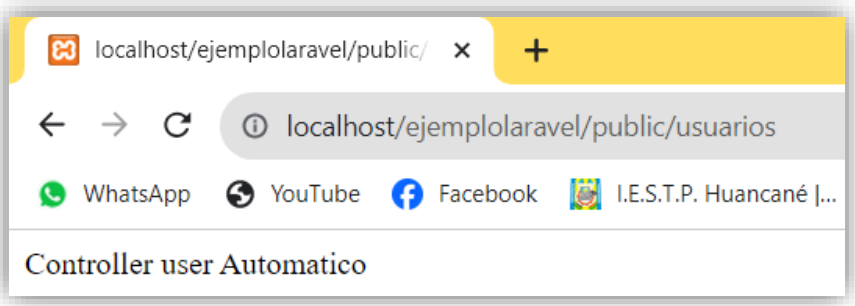
En el **Class** ya podemos generar nuestras funciones como el index. Ejemplo:

```php
class UsuarioController extends Controller
{
    public function index(){
        return "Controller user Automatico";
    }
}
```

Luego debemos de agregarlo en web.php (se encuentra remarcado de azul lo que hemos agregado)

```php
Route::get('/', function () {
    return "<h1> Hola mundo </h1>";
});

route::get('productos', [ProductoController::class, 'index']);
route::get('usuarios', [UsuarioController::class, 'index']);

route::get('productos/{producto}', [ProductoController::class, 'show']);
```

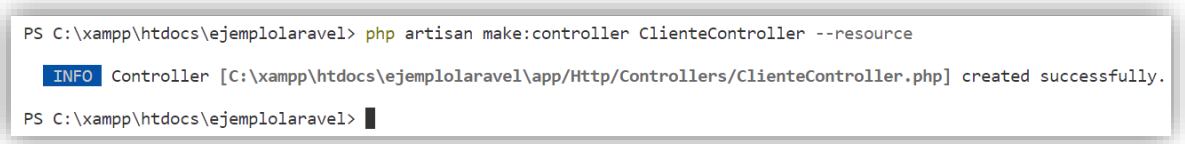
Luego vamos a probarlo.



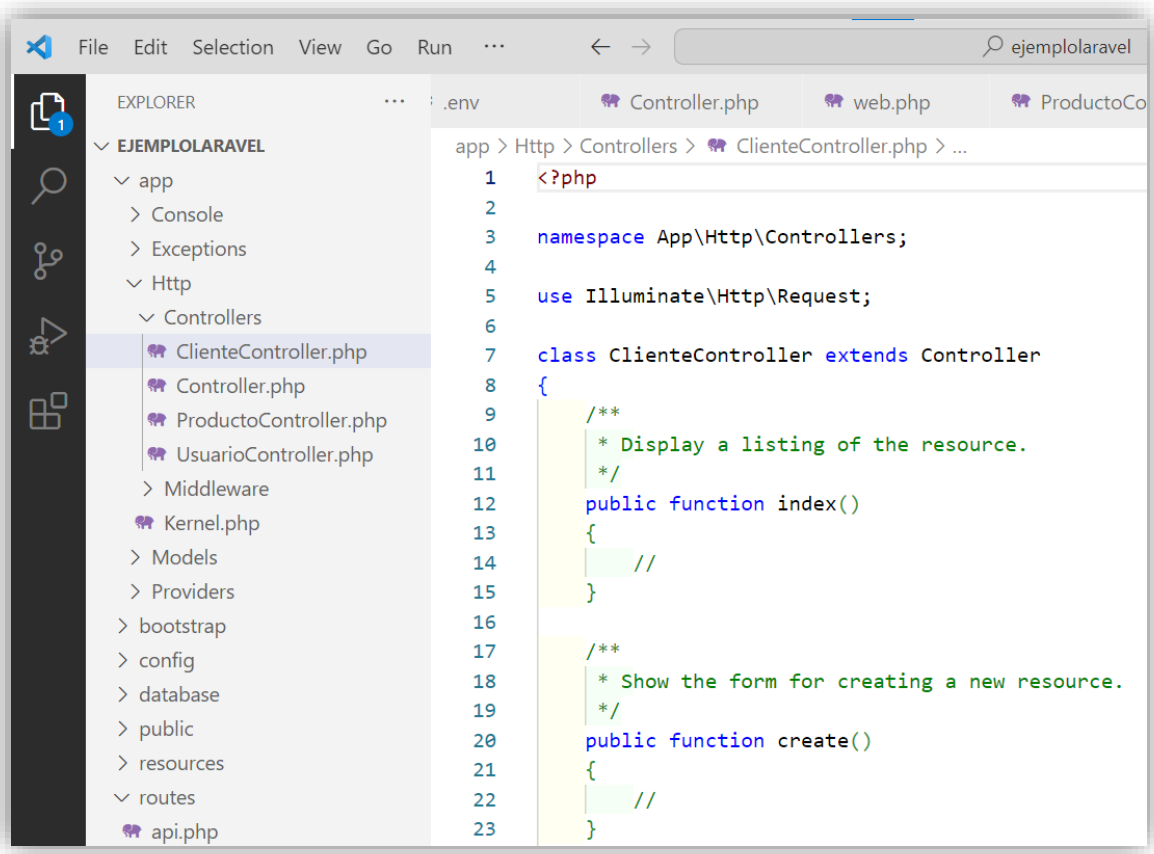
Como verán se hizo más rápido que el método anterior, pero hay un paso aun mas rápido.

Vamos a realizarlo también con la consola, abrimos la Terminal, podemos limpiar pantalla si así lo desean, escribir:

php artisan make:controller nombre_controlador –resource luego presionar Enter



Ahora vamos a revisar lo que ha creado, ir a la carpeta de los controladores, abrimos el controlador creado, pero aquí, ya es un controlador más completo, pues ya nos está generando las funciones principales para hacer un CRUD (crear, leer, actualizar, eliminar de una base de datos)



Tenemos el index, que es para mostrar, pero también ya nos trae un poco de documentación porque nos está indicando que es lo que va a retornar este método.

```php
class ClienteController extends Controller
{
    /**
     * Display a listing of the resource.
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        //
    }
```

La función **create** nos va poder visualizar el formulario para poder crear un registro. El de **store** va a guardar ese registro, el **show** va mostrar los detalles de ese registro, **edit** te va permitir visualizar el formulario para poder editar la información, **update** va actualizar lo que tu hayas modificado del registro, **destroy** va eliminar este registro.

Como puede ver ya genera todo lo necesario. Pero para poder entender como funciona esto, necesitamos ver un poco de documentación, para ello buscar Laravel en Google, ir a documentación, luego the basic, Controller, acá puede encontrar todo lo que ya vimos y los comandos que hemos trabajado, pero veremos una tabla que es muy importante.

Actions Handled By Resource Controller

Verb	URI	Action	Route Name
GET	`/photos`	index	photos.index
GET	`/photos/create`	create	photos.create
POST	`/photos`	store	photos.store
GET	`/photos/{photo}`	show	photos.show
GET	`/photos/{photo}/edit`	edit	photos.edit
PUT/PATCH	`/photos/{photo}`	update	photos.update
DELETE	`/photos/{photo}`	destroy	photos.destroy

Aquí nos va a indicar como es que Laravel interpreta las peticiones, ya vimos la de GET, solo colocamos el nombre de nuestro módulo y nos envía al index directamente, y nos indica la ruta que está generando.

GET	/photos	index	photos.index

Pero si nosotros queremos el formulario para crear un nuevo registro lo vamos a consultar así:

```
/photos/create
```

Eso con el método GET, y nos va redireccionar

```
photos.create
```

Pero qué pasa si necesitamos guardar este formulario, vamos a solicitar la ruta photos, pero mediante la petición POST pero la ruta a la que nos envía a photos.store.

Ahora que pasa si queremos trabajar los detalles de un producto, en donde interviene el ID entre llaves, que es el parámetro que nos va a filtrar. Y también esta la ruta hacia donde irá.

GET	/photos/{photo}	show	photos.show

En resumen, es así como necesitamos enviar nuestros parámetros para que los pueda captar correctamente.

Para poder hacer que funcione el controlador **ClienteController** debemos de agregarlo al web.php para las rutas, pero aquí, necesitamos agregarlo de forma diferente.

Vamos a borrar todo lo que hemos trabajado como clientes.

```php
Route::get('/', function () {
    return "<h1> Hola mundo </h1>";
});

route::get('productos', [ProductoController::class, 'index']);
route::get('usuarios', [UsuarioController::class, 'index']);
route::get('productos/{producto}', [ProductoController::class, 'show']);
```

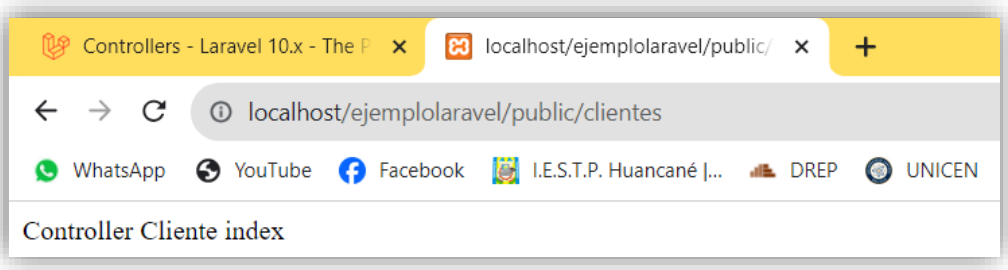
Lo de usuarios lo vamos a bajar y abajo colocamos otra ruta, aquí no vamos a colocar ni el index, ni el show, ahí le indicamos que ya son todos los recursos que va a tener, ya sabe que significa el index, el store, el update, y el método que les corresponde para cada función.

```php
route::get('productos', [ProductoController::class, 'index']);
route::get('productos/{producto}', [ProductoController::class, 'show']);
route::get('usuarios', [UsuarioController::class, 'index']);
route::resource('clientes', ClienteController::class);
```

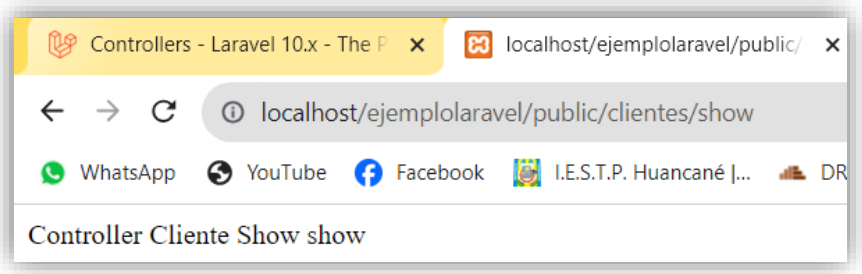
Hagamos la prueba para ver que sucede, pero antes vamos a crear unos mensajes para que nos muestre algo. (en **ClienteController.php**), el store ahora no lo vamos a poder emular, pero podemos continuar con el show, y con esos haremos la prueba.

```php
public function index()
{
    Return "Controller Cliente index";
}
/**
 * Show the form for creating a new resource.
 */
public function create()
{
    Return "Controller Cliente create";
}
/**
 * Store a newly created resource in storage.
 */
public function store(Request $request)
{
    //
}
/**
 * Display the specified resource.
 */
public function show(string $id)
{
    Return "Controller Cliente Show $id";
}
```

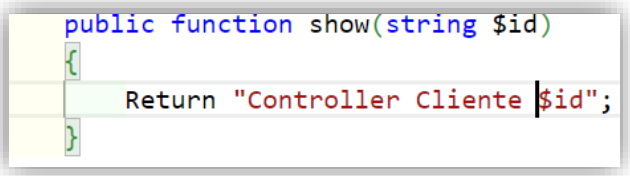
Primero clientes.



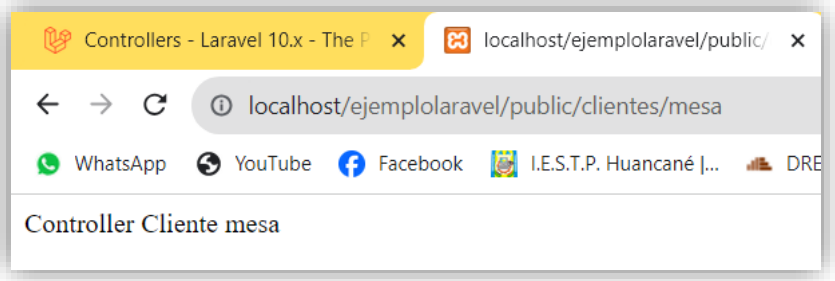
Ahora probemos con show



Pero vemos que se ha repetido la palabra show, porque hemos colocado la palabra Show, pero no es necesario colocarle la ruta show



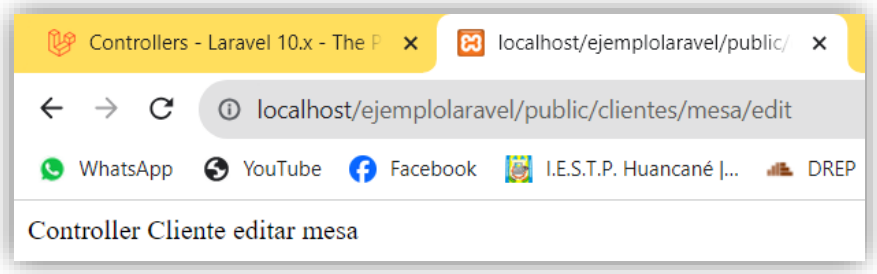
Si cambiamos mesa por show, la variable id asumirá ese valor.



El edit también lo podemos trabajar.

```php
public function edit(string $id)
{
    Return "Controller Cliente editar $id";
}
```

Según la tabla mostrada anteriormente nos dice que vamos a necesitar el ID, o el parámetro. Por ejemplo:



Así es como vamos a ir formando nuestras rutas, nuestros métodos desde nuestro controlador.

CAPÍTULO 6

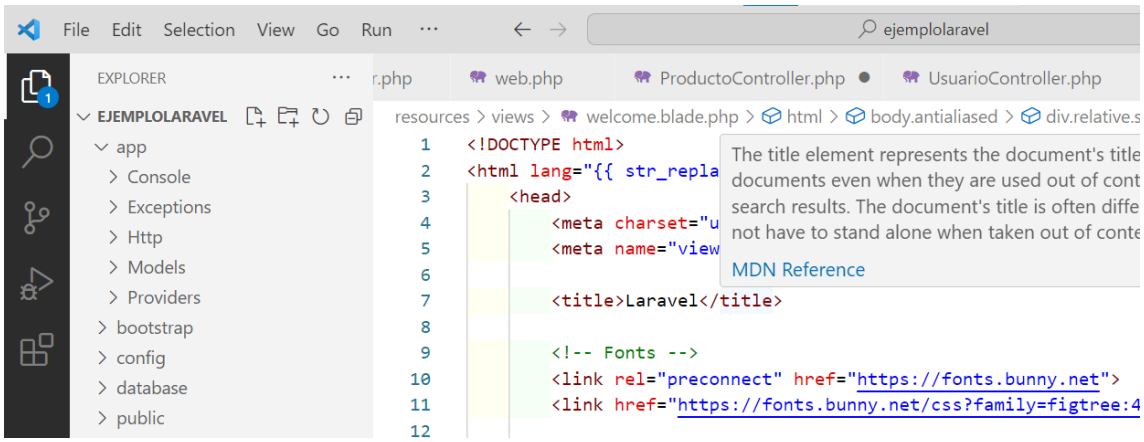
VISTAS Y PLANTILLAS BLADE

6.1 VISTAS

Ahora vamos a trabajar con las vistas para ya presentar la información un poco más visual al usuario. Para ello, vamos a trabajar con algo de HTML

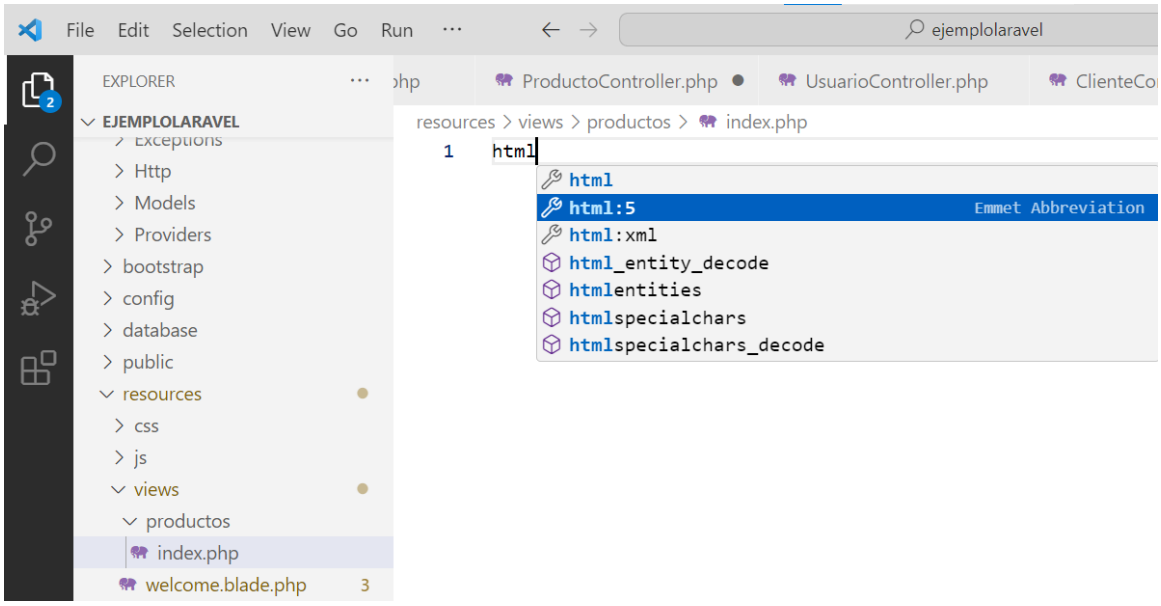
Para que pueda agregar una vista aquí en nuestro proyecto, necesitamos ir a:

Resource/views/welcome.blade.php



Podemos ver la estructura de este archivo.

Dentro de la carpeta **views** vamos a crear una nueva carpeta, esto para el módulo de **productos**, y dentro de este vamos a crear los archivos de las vistas como por ejemplo **index.php** es así como se llamará nuestra vista inicial, y vamos a agregar un HTML 5 muy sencillo.



Podemos colocar el título **Catálogo de productos,** en el cuerpo vamos a agregar un título <h1> **Catálogo de productos**

Es una vista muy simple, pero lo que importa por ahora es como la podemos llamar, si recuerdan en rutas ya teníamos un ejemplo de como se hace la llamada, pero vamos a hacerlo.

resources > views > productos > index.php > ...

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Catálogo de productos</title>
</head>
<body>
    <h1>Catálogo de productos</h1>
</body>
</html>
```

Ir a la carpeta rutas y vamos a modificar **productos** en donde el return va ha ser **view('Indicar el módulo o carpeta')**

La ruta la puedes poner con diagonal o con punto

```php
Route::get('/', function () {
    return "<h1> Hola mundo </h1>";
});

route::get('productos', function(){
    return view('productos.index');
});
route::get('productos/{producto}', [ProductoController::class, 'show']);
route::get('usuarios', [UsuarioController::class, 'index']);
route::resource('clientes', ClienteController::class);
```

Ya con esto estamos diciendo que me traiga a la vista lo que está en la carpeta **productos** y el archivo de nombre **index**.

Vamos a hacer una prueba

Puede apreciar que ya estamos mostrando un HTML básico creado por nosotros.

Aquí en el mismo modulo vamos a mostrar los detalles de un producto, entonces en la misma carpeta **productos** vamos a crear el archivo **show.php** luego también agregaremos un HTML 5, y en title vamos a poder **Detalles del producto** y en el cuerpo de la página poner un título en <h1> que diga **Detalles del producto** y vamos a imprimir el dato que estamos pasando.

resources > views > productos > show.php > ...

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Detalles del producto</title>
</head>
<body>
    <h1>Detalles del producto</h1>
</body>
</html>
```

Recordar que cuando trabajamos con los controladores y también las rutas podemos enviar algunos parámetros, vamos a trabajarlo desde ProductoController.php

Vamos a colocar view, dentro del paréntesis **productos.show** luego es importante que para pasarle la variable que estamos solicitando o que estamos obteniendo, vamos a separarlo con una coma, abrimos llaves [ ] para un arreglo, en donde el índice se va llamar producto, y esto va ser igual a la variable $nombre (con esto le estamos enviando la información) y de paso hacemos el cambio en la función **index** y colocamos view('productos.index');

app > Http > Controllers > ProductoController.php > ...

```
<?php
namespace App\Http\Controllers;
use App\Http\Controllers\Controller;
class ProductoController extends Controller
{
    public function index()
    {
        return view('productos.index');
    }
    public function show($nombre)
    {
        return view('productos.show', ['producto'=> $nombre]);
    }
```

Después nos vamos a las rutas, hay que quitar la función anónima que teníamos previamente definida y hay que colocar nuestro Controller y la función index.

```
[ProductoController::class, 'index']
```

```php
Route::get('/', function () {
    return "<h1> Hola mundo </h1>";
});

route::get('productos', [ProductoController::class, 'index']);

route::get('productos/{producto}', [ProductoController::class, 'show']);
```

Por último, nos vamos a la vista de show para poder recibir el parámetro que le estamos enviando aquí: (ProductoController.php)

```php
    public function show($nombre)
    {
        return view('productos.show', ['producto'=> $nombre]);
    }
```

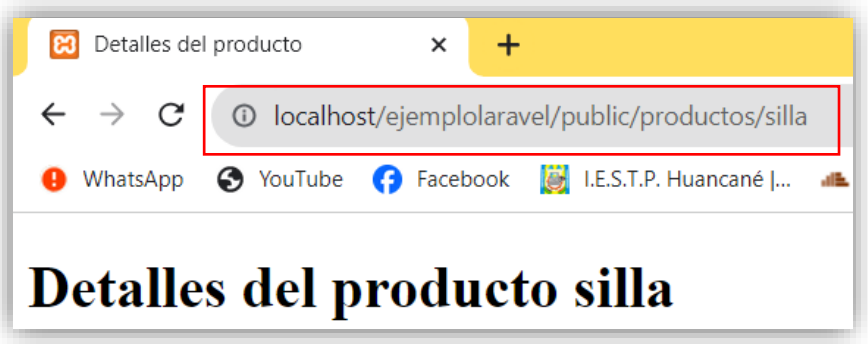
Hacemos la llamada con php y le pedimos que nos muestre lo que contiene la variable producto.

resources > views > productos > show.php > ...

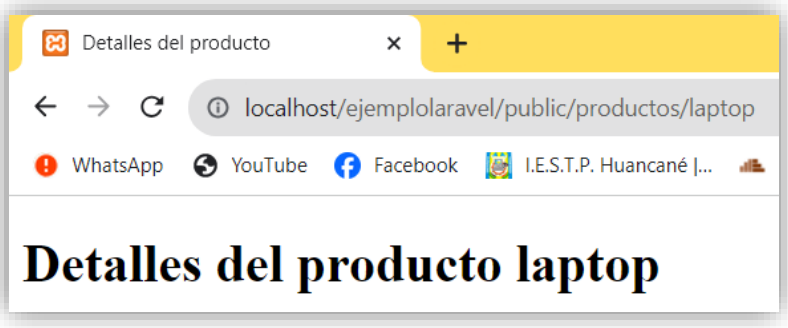
```php
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Detalles del producto</title>
</head>
<body>
    <h1>Detalles del producto <?php echo $producto;?></h1>
</body>
</html>
```

Vamos a ver ahora como en el navegador:

Colocamos el producto silla.



Con eso estamos pasando parámetro, podemos cambiar a laptop:



Así es como regularmente trabajamos con el modelo vista controlador, en este caso estamos trabajando las vistas, el controlador y las rutas.

6.2 PLANTILLAS BLADE

En Laravel también vamos a encontrar unas plantillas que en este caso se van a llamar **blade**

Nos ayuda a trabajar mejor con la información que pasamos entre las rutas o el controlador y que dejemos de utilizar esta forma para poder imprimir variables en nuestro HTML, porque en ocasiones empezamos a meter HTML, PHP y demás aplicaciones, de tal forma que se empieza a saturar nuestra vista:

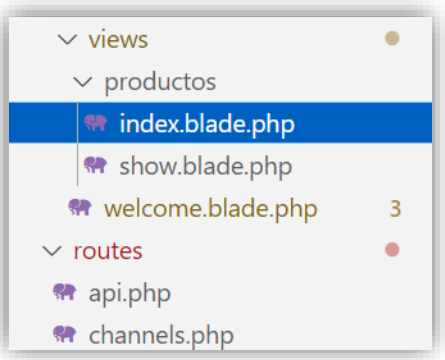
```
<body>
    <h1>Detalles del producto <?php echo $producto;?></h1>
</body>
</html>
```

En lugar de lo anterior vamos a trabajar con **blade,** donde blade es un motor de plantillas que nos ayuda a agregar código PHP, pero de otra forma mas sencilla y menos desordenada.

Para trabajar con esto, necesitamos cambiar el nombre de las vistas como:

Index.blade.php

Show.blade.php



Si ejecutamos la aplicación seguirá trabajando de forma normal, pero este cambio que hicimos nos ayudara en ya no es necesario poner en php, sino con llaves dobles y dentro colocamos el nombre de la variable producto, luego borramos el código php en el que llamábamos a la variable producto.

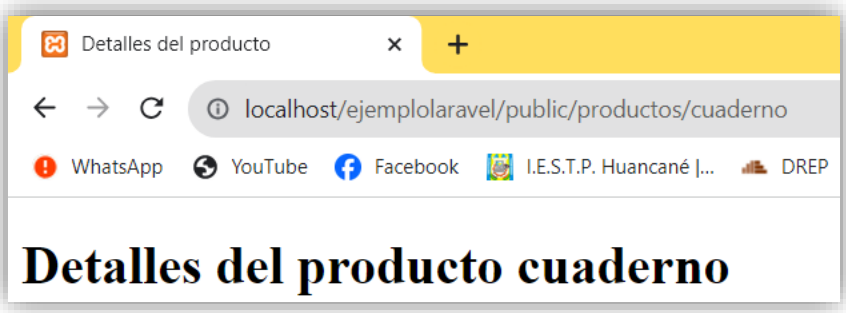
resources > views > productos > show.blade.php > ...

```
1   <!DOCTYPE html>
2   <html lang="en">
3   <head>
4       <meta charset="UTF-8">
5       <meta name="viewport" content="width=device-width, initial-scale=1.0">
6       <title>Detalles del producto</title>
7   </head>
8   <body>
9       <h1>Detalles del producto {{$producto}}</h1>
10  </body>
11  </html>
```

Fíjense la cantidad de código que hemos reducido.

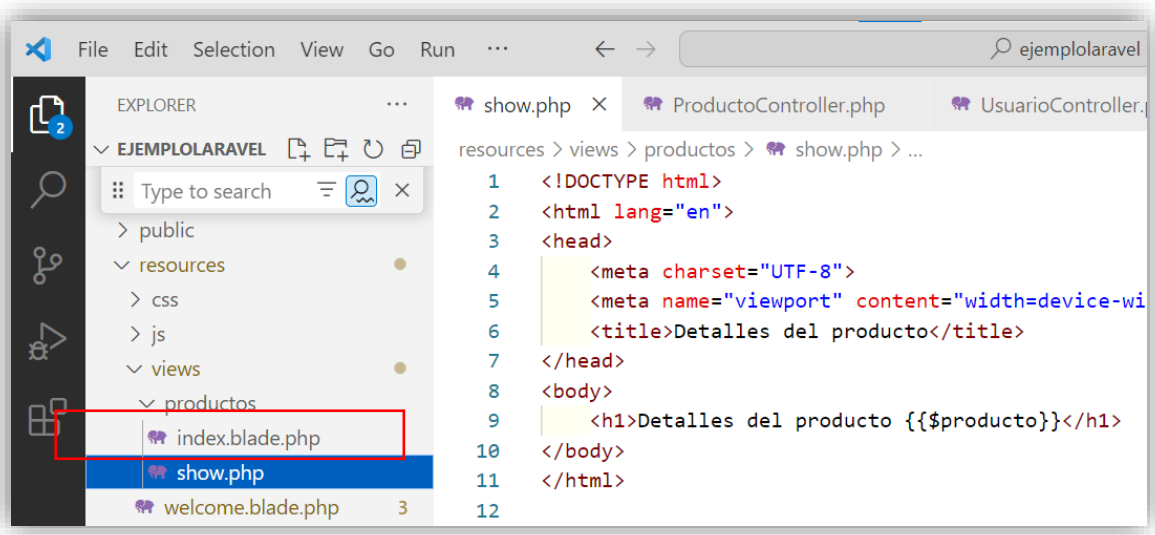
Probemos lo anterior con otro producto

Por ejemplo, con el producto cuaderno

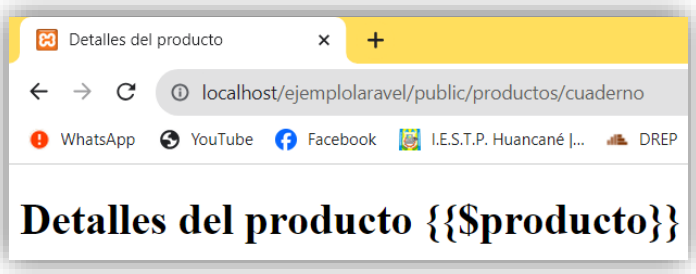


Vemos que imprime de la forma correcta sin el uso del php.

Qué pasa si al nombre del archivo no le pones blade



Probamos con el nuevo cambio:



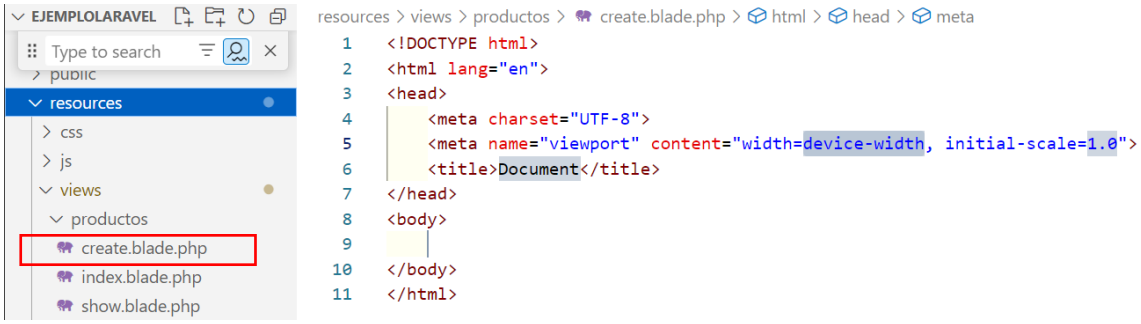
Vemos que si no ponemos al archivo como **show.blade.php** no reconoce esa sintaxis, es por eso que forzosamente debemos colocar blade para trabajar con esa sintaxis.

Con las plantillas blade no solamente vamos a poder imprimir variables de otra forma, sino que podemos generar algunas plantillas, como el generar un encabezado, un pie de página e insertar diferentes páginas HTML.

Por ejemplo, vamos a hacer uno donde crearemos un encabezado, un pie de página y desde otro archivo agreguemos el contenido a nuestra página.

Esto lo vamos a hacer con el create, que es el formulario para crear un producto.

En la carpeta productos crear el archivo **create.blade.php** luego agregamos un HTML 5



En el **title** del HTML generado, vamos a introducirlo de forma dinámica. Blade trabaja con algunos argumentos para agregar algunos atributos o parámetros en la sección que tu estes.

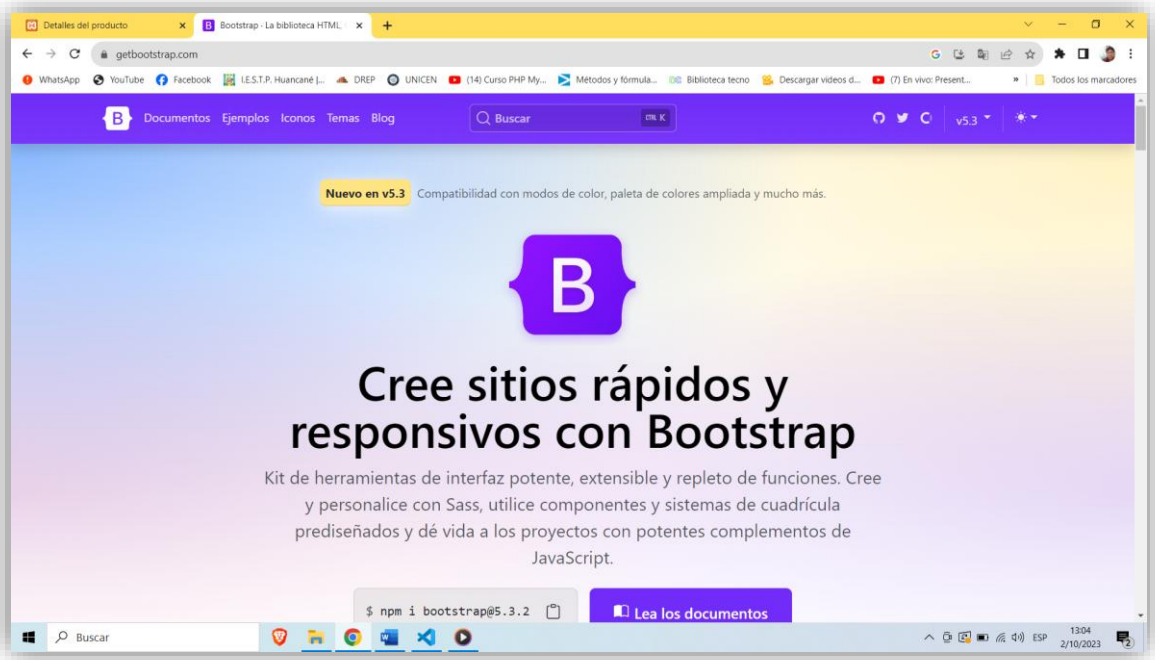
Vamos a colocar @yield('title')

Title es el nombre del parámetro que vamos a pasar a la plantilla para que lo pueda imprimir.

```
resources > views > productos > create.blade.php > html > head > title
1   <!DOCTYPE html>
2   <html lang="en">
3   <head>
4       <meta charset="UTF-8">
5       <meta name="viewport" content="width=device-width, initial-scale=1.0">
6       <title>@yield('title')</title>
7   </head>
8   <body>
9
10  </body>
11  </html>
```

Después de esto vamos a trabajar con Bootstrap, donde tu puedes agregar tu propio estilo o puedes dejarlo, así como está.

Ir a Google y escribir Bootstrap, seleccionar la versión y luego clic en lea los documentos.



Copiar el link de css

Vamos a elegir la versión 5.1.3, vamos a copiar el link del css

Luego lo agregamos después del title en nuestro HTML del archivo create.blade.php

Y también puede agregar el js

Quedaría así:

Aunque el script no va ser tan necesario.

```
resources > views > productos > create.blade.php > ...
1   <!DOCTYPE html>
2   <html lang="en">
3   <head>
4       <meta charset="UTF-8">
5       <meta name="viewport" content="width=device-width, initial-scale=1.0">
6       <title>@yield('title')</title>
7
8       <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="
9       <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/bootstrap.bundle.min.js"
10  </head>
11  <body>
12
13  </body>
14  </html>
```

Dentro del body ponemos:

Como vamos a trabajar con Bootstrap vamos a agregar el **main**, un **div** que va ser un **class container py-4** (agrega un espacio de 4 en arriba)

Luego colocamos un **footer** dentro colocamos *Códigos de programación,* pero dentro del footer le agregamos un class **pt-3 mt-4 text-muted border-top** (esto es para que nos coloque un borde arriba antes de iniciar el footer, esto es código de Bootstrap. Esa a va ser nuestra plantilla

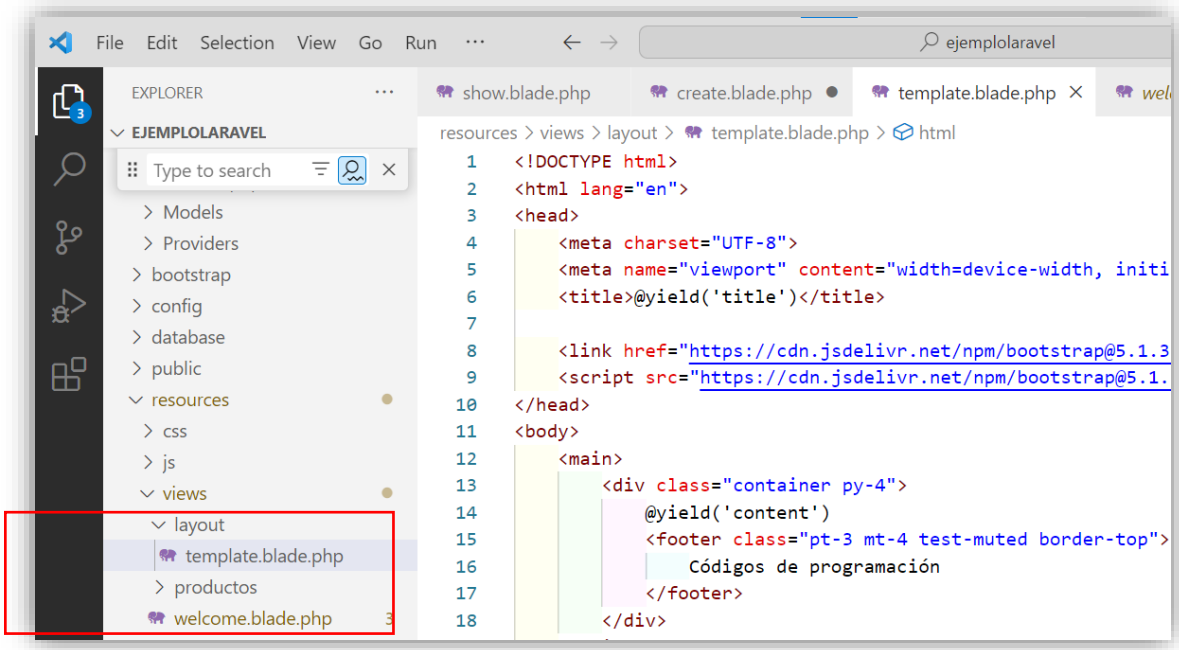
Ahora vamos a agregar un **@yield('content')** *(que va ser todo el contenido dinámico que vamos a estar pasando)*

Todo este código en el archivo **create.blade.php**

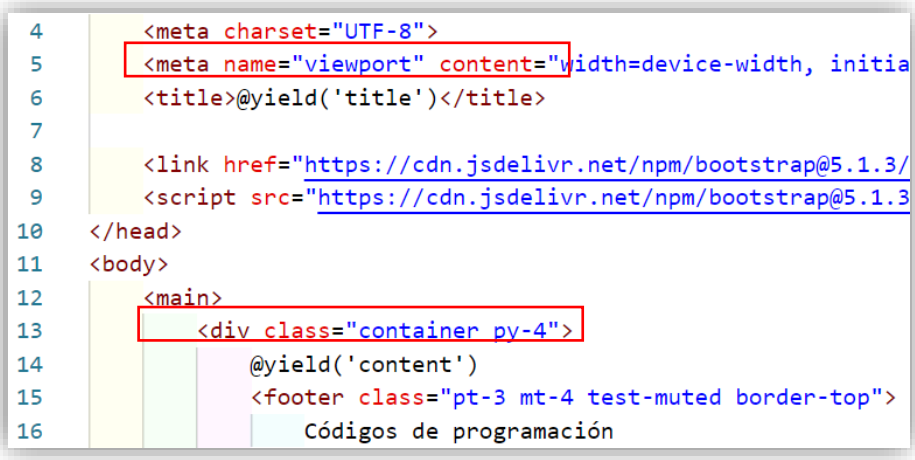
```
12      <main>
13          <div class="container py-4">
14              @yield('content')
15              <footer class="pt-3 mt-4 test-muted border-top">
16                  Códigos de programación
17              </footer>
18          </div>
19      </main>
20  </body>
```

Como dijimos el archivo anterior seria como una plantilla y ésta la vamos a poder reutilizar en todas las vistas que quisiéramos, así que es necesario crear una nueva carpeta en **views** que se llame **layout** y aquí vamos a crear un archivo que se llame **template.blade.php**

Entonces, vamos a seleccionar y **cortar** todo lo que ya hemos realizado y lo vamos a copiar en la plantilla **template.blade.php**



Recordemos que acá tenemos dos parámetros que vamos a pasar de forma dinámica, uno el titulo y otro el contenido.



Uno vez hecho estos cambios nos vamos al archivo **create.blade.php** vamos a indicarle a este archivo que necesitamos usar el template, esto va ser con **@extends** luego entre paréntesis deben colocar la ruta de la carpeta **views** en donde se encuentra el **template,** por ejemplo, sería **@extends('layout/template')** *aquí lo debemos resaltar es*

que cuando trabajamos con laravel no necesitamos colocar el .php y en este caso tampoco el .blade laravel lo va reconocer de forma automática

Después vamos a crear una sección y la vamos a llamar **@section** y dentro de las comillas colocar la sección que vamos a cubrir, en este caso si vemos la plantilla, vamos a cubrir el **yield** entonces ponemos **title** luego una coma, y el valor que le vamos a pasar como por ejemplo **Nuevo producto**

Después creamos otro **@section('content')** el contenido que le vamos a agregar en **@yield('content')** del archivo template.blade.php

Ahora vamos a crear un formulario y lo vamos a trabajar con Bootstrap, declaramos un **div** con un **class mb-3** dentro vamos a colocar un **label** para el nombre, pero le agregamos el dato adicional de Bootstrap **form-label**

Abajo vamos a colocar un **input** de tipo **text** agregamos el **class form-control** (esto es solo para Bootstrap)

Vamos a copiarlo y lo pegamos abajo para modificar algunos parámetros

Luego terminamos con un Button de tipo submit, para guardar, también le colocamos las etiquetas para Bootstrap **btn btn-primary**

Finalmente cerramos el **content** con **@endsection**

Es importante saber dónde se inicia una sección y donde la vamos a terminar, porque necesitas revisar la jerarquía, es decir si la abres primero, la debes cerrar al final, si la abres en medio, también en esa posición la vas a cerrar para que no haya interferencias entre todas las secciones que vayas abriendo.

```
resources > views > productos > create.blade.php > ...
1   @extends('layout/template')
2   @section('title', 'Nuevo producto')
3   @section('content')
4
5   <form action="" method="post">
6       <div class="mb-3">
7           <label for="nombre" class="form-label">Nombre</label>
8           <input type="text" name="nombre" id="nombre" class="form-control">
9       </div>
10
11      <div class="mb-3">
12          <label for="precio" class="form-label">Precio</label>
13          <input type="text" name="precio" id="precio" class="form-control">
14      </div>
15
16      <button type="submit" class="btn btn-primary">Guardar</button>
17  </form>
```

El siguiente paso es crear nuestra ruta y también nuestro controlador, primero abriremos **web.php** para configurar la ruta.

Aquí es importante ver la jerarquía, recuerda que para ingresar al archivo **create.blade.php** debemos colocar **productos/create** y si lo colocamos después de la parte de azul de la imagen, tal vez no vaya a entrar, va a detectarlo como si le estaríamos pasando un parámetro y no a funciona, nos a va enviar a show.

```
route::get('productos', [ProductoController::class, 'index']);

route::get('productos/{producto}', [ProductoController::class, 'show']);

route::get('usuarios', [UsuarioController::class, 'index']);
route::resource('clientes', ClienteController::class);
```

Vamos a hacer la prueba. (agregando ruta)

```
route::get('productos', [ProductoController::class, 'index']);
route::get('productos/{producto}', [ProductoController::class, 'show']);
route::get('productos/create', [ProductoController::class, 'create'])
```

Vamos al Controller **ProductoController.php** para revisar si lo tenemos, y como no lo tenemos, escribiremos código.

app > Http > Controllers > ProductoController.php > ...

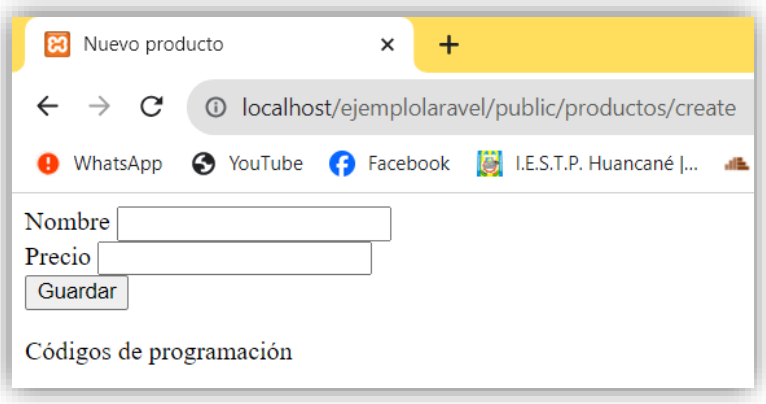
```php
<?php
namespace App\Http\Controllers;
use App\Http\Controllers\Controller;
class ProductoController extends Controller
{
    public function index()
    {
        return view('productos.index');
    }
    public function show($nombre)
    {
        return view('productos.show', ['producto'=> $nombre]);
    }
    public function create()
    {
        return view('productos.create');
    }
}
```

Hagamos la prueba

Lo que está pasando es que se está ejecutando el **show** y lo está tomando al create como un parámetro para mostrar el producto, entonces es por eso que necesitamos cambiar la jerarquía, donde la ruta del create debe de ir primero o más arriba del show

```php
Route::get('/', function () {
    return "<h1> Hola mundo </h1>";
});

route::get('productos', [ProductoController::class, 'index']);
route::get('productos/create', [ProductoController::class, 'create']);
route::get('productos/{producto}', [ProductoController::class, 'show']);

route::get('usuarios', [UsuarioController::class, 'index']);
route::resource('clientes', ClienteController::class);
```

Probamos nuevamente



Al parecer hay un error con el Bootstrap, encontrar el error.

CAPÍTULO 7

PETICIONES HTTP Y PROTECCIÓN CSRF

Ahora vamos a trabajar con peticiones HTTP y protección contra falsificación de solicitudes.

Como ya hemos venido trabajando con los controladores y las rutas, ya hemos visto algunos métodos, como por ejemplo el método GET, que es el método por el cual solicitamos una URL y dentro de esta URL podemos enviar información.

La siguiente tabla es una adaptación de la tabla que hicimos anteriormente.

Método	URL	Acción	Proceso
GET	**/productos**	**index**	**Catálogo**
GET	/productos/create	create	Formulario nuevo
POST	**/productos**	**store**	**Guardar**
GET	/productos/{id}	show	Mostrar
GET	/productos/{id}/edit	edit	Formulario editar
PUT/PATCH	/productos/{id}	update	Actualizar
DELETE	/productos/{id}	destroy	Eliminar

Que sucede si queremos **guardar** el formulario, necesitaríamos enviarlo por el método POST. Por ejemplo:

Para el acction del formulario, le diremos que vaya al URL.

resources > views > productos > create.blade.php > ...

```
@extends('layout/template')
@section('title', 'Nuevo producto')
@section('content')

<form action="{{url('/productos')}}" method="post">
    <div class="mb-3">
        <label for="nombre" class="form-label">Nombre</label>
        <input type="text" name="nombre" id="nombre" class="form-control">
    </div>

    <div class="mb-3">
        <label for="precio" class="form-label">Precio</label>
        <input type="text" name="precio" id="precio" class="form-control">
    </div>

    <button type="submit" class="btn btn-primary">Guardar</button>
</form>
```

Esta función de URL lo que va a realizar es colocar la ruta de:

localhost/ejemplolaravel/public

```
<form action="{{url('/productos')}}" method="post">
```

En muchos casos se va repetir la misma URL como por ejemplo cuando consultas el catálogo y cuando guardas el formulario, es la misma URL, pero se va diferenciar por el método sobre el cual lo estas enviando.

Método	URL	Acción	Proceso
GET	**/productos**	**index**	**Catálogo**
POST	**/productos**	**store**	**Guardar**

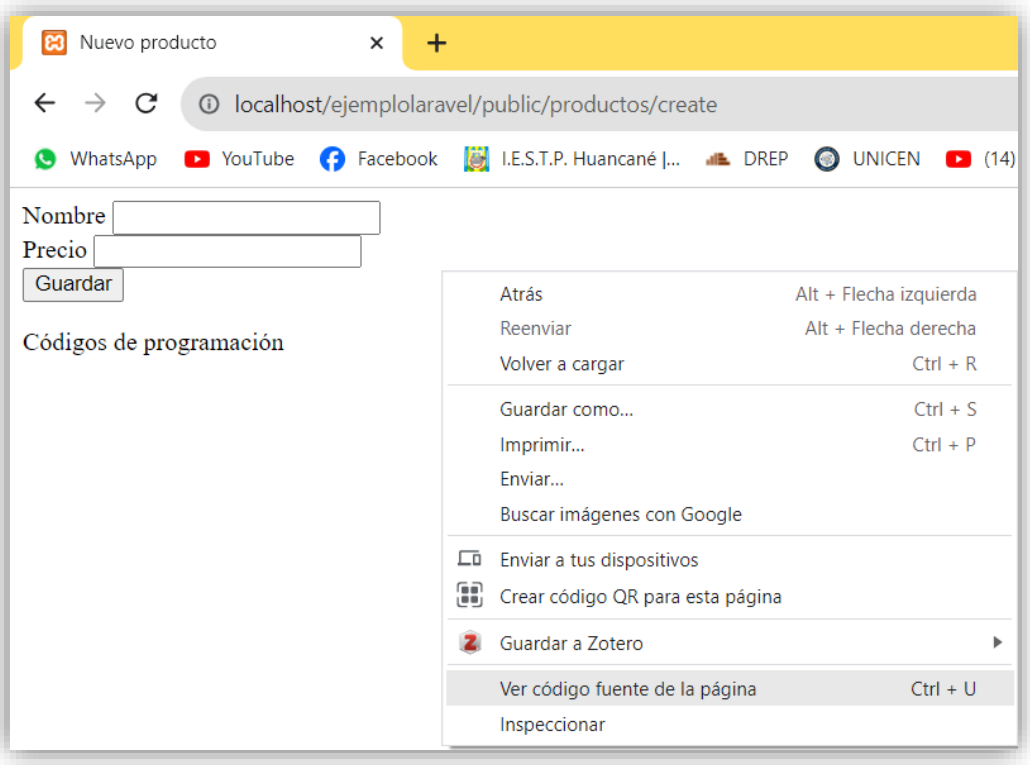
Mas abajo tenemos otros métodos que vamos a ir revisando, pero por ahora haremos la parte de guardar el formulario.

7.1 GUARDAR EL FORMULARIO

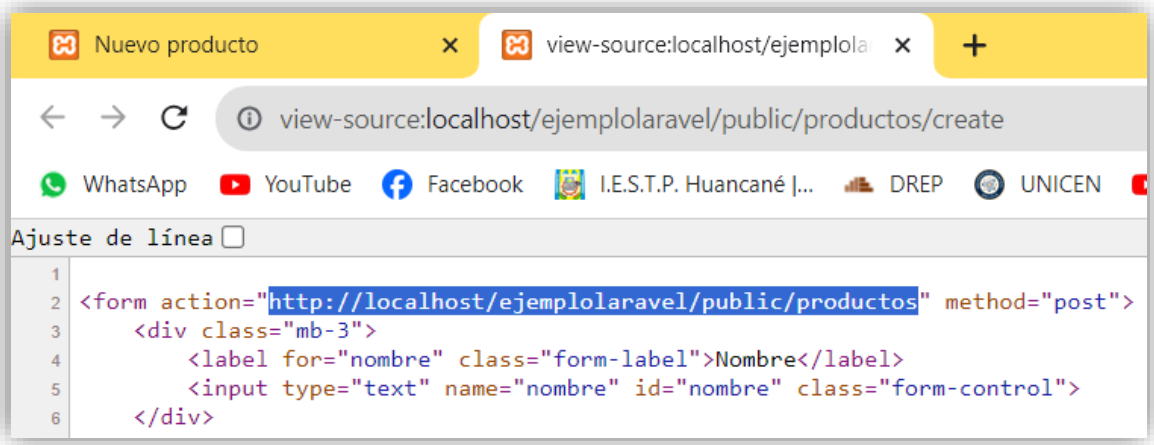
Estamos diciendo que nos vamos a productos y por el método post

```
<form action="{{url('/productos')}}" method="post">
```

Podemos revisar el formulario, clic derecho y seleccionar **ver código fuente de la página.**



Aparece el código fuente de la página, y qui pueden ver la URL



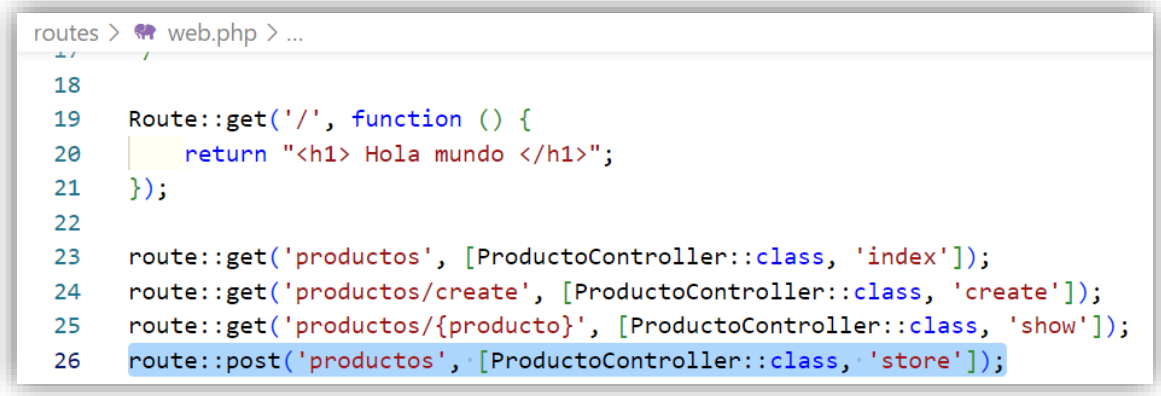
Si copias la URL y lo pegas en el navegador, te va enviar al catalogo del producto, porque es una petición GET

Pero si envías el formulario se convierte en una petición POST, porque asi le estamos indicando en **method**.

```
<form action="{{url('/productos')}}" method="post">
```

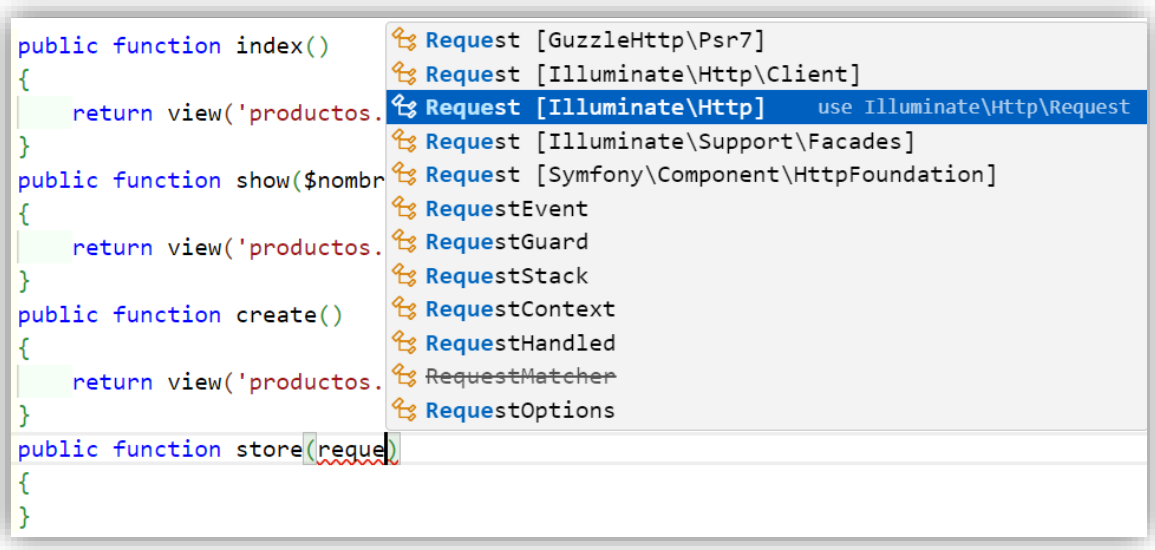
Falta agregar el **Route** en web.php

Ahí en lugar de GET colocamos POST, y la función se va llamar **store,** con esto ya estaríamos creando nuestra ruta.

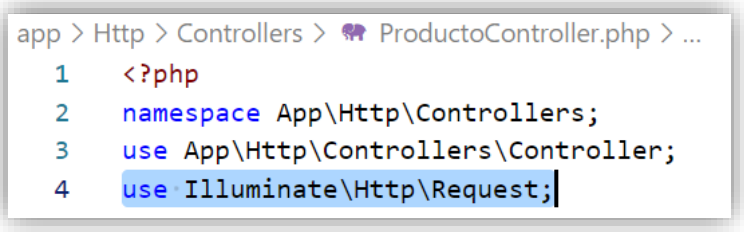


Después nos vamos a **ProductoController.php** en este archivo ya hemos trabajado algunas funciones, pero vamos a agregar otra, va ha ser **Public function store**

Como esta función va a recibir información mediante el formulario con el método POST vamos a usar **REQUEST** y seleccionaremos el que dice **Iluminate/http**



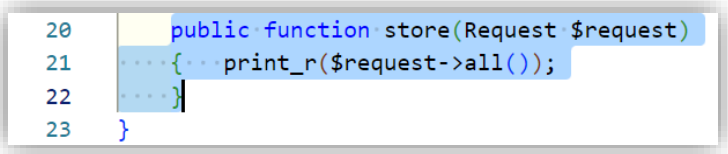
Cuando lo seleccionas en la lista que apareció al escribir **Request** se va generar la siguiente línea:



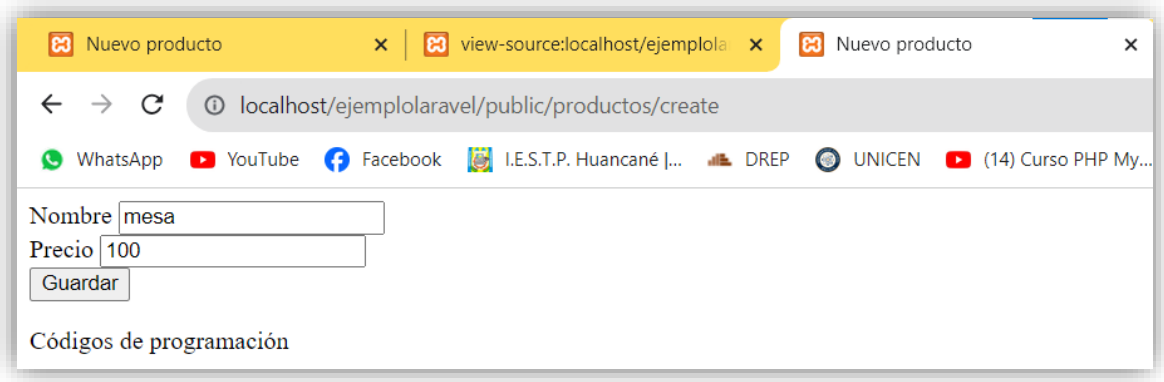
En el caso de que no lo genere, deberás de escribirlo para poder utilizar este **Request** y poder trabajar. Ese es el tipo que vamos a recibir, pero la variable también se va llamar **$request,** luego solo haremos una impresión:

Print_r($request->all());

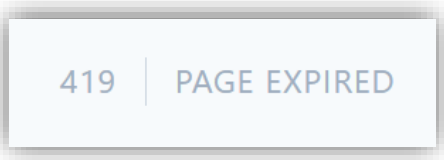
Para que nos envíe toda la información de esta solicitud.



Vamos a hacer la prueba enviando el formulario a create, ingresamos datos:



Vamos a darnos cuenta que nos va a generar un error, para ello hacer clic en guardar.



Nos da un error de que la página a expirado, lo que pasa, es que se esta protegiendo de solicitudes que no sean auténticas, esta es una opción de seguridad que nos da Laravel, esto nos va servir para que no cualquier aplicación pueda usar nuestro programa.

Esto lo podemos comparar cuando en PHP hemos utilizado sesiones, es decir, si hemos ingresado mediante un Login y este Login lo hemos programado con sesiones, quiere decir que podremos hacer modificaciones solo si hemos ingresado a la plataforma mediante sesiones, es decir, cualquier otra persona o aplicación no podrá hacer modificaciones.

Podemos agregar un token para mas seguridad, es decir si tiene la contraseña y además el token, entonces lo van a poder utilizar.

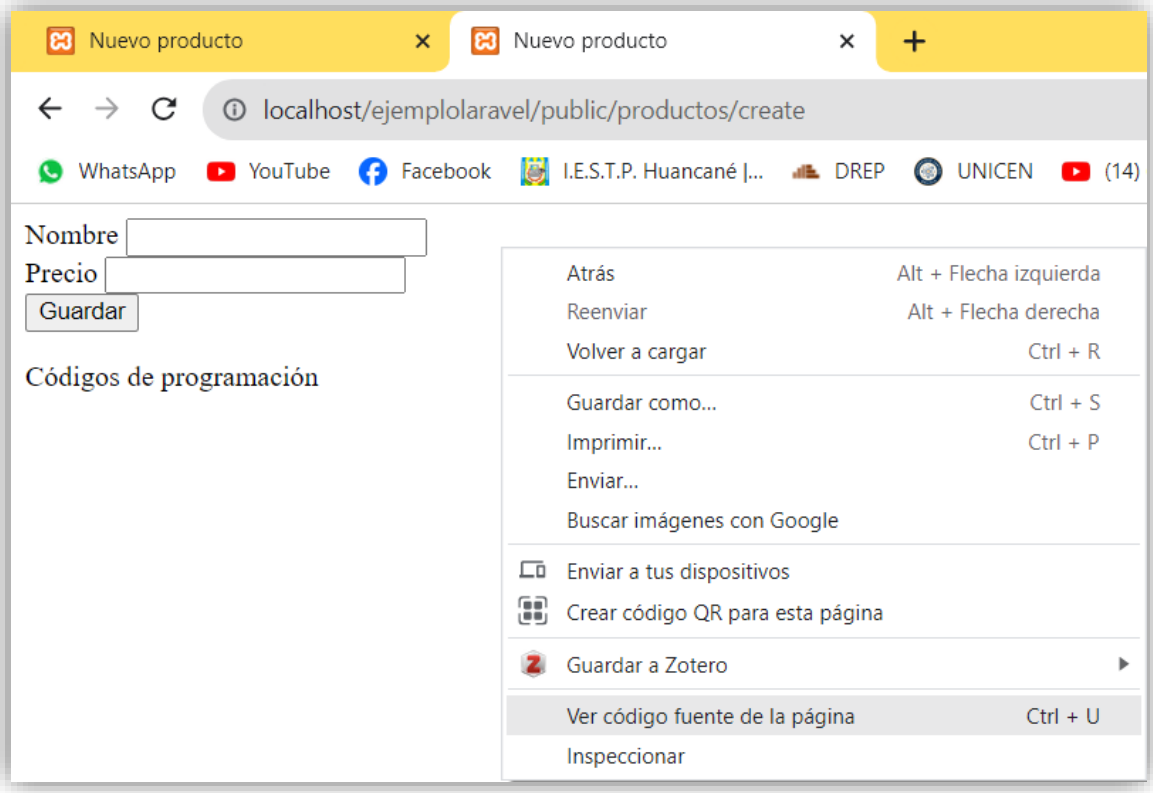
El funcionamiento va a ser de la siguiente manera:

En el formulario **create.blade.php** vamos a crear una función **@csrf** con esto nos va imprimir un input oculto con el token

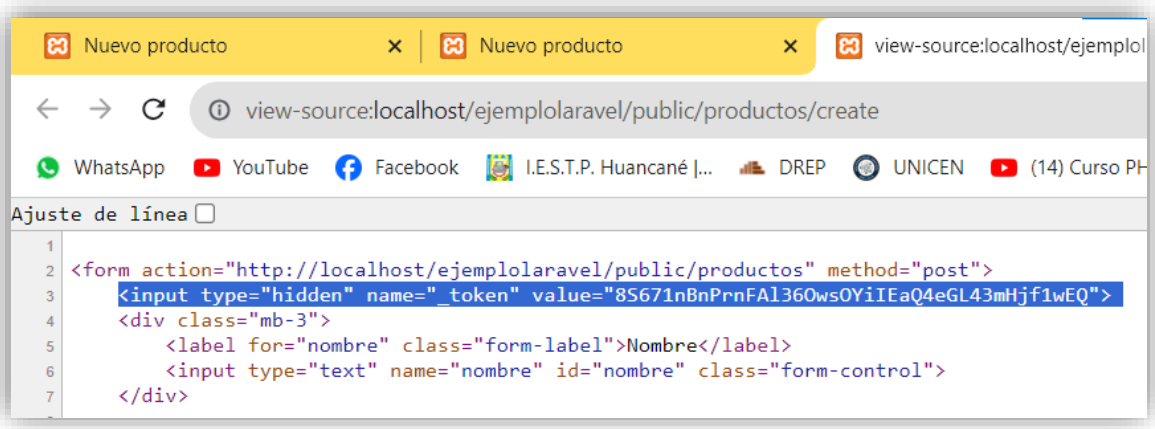
resources > views > productos > create.blade.php > form

```
1   @extends('layout/template')
2   @section('title', 'Nuevo producto')
3   @section('content')
4
5   <form action="{{url('/productos')}}" method="post">
6       @csrf
7       <div class="mb-3">
8           <label for="nombre" class="form-label">Nombre</label>
9           <input type="text" name="nombre" id="nombre" class="form-control">
10      </div>
```

Vamos a la página de create

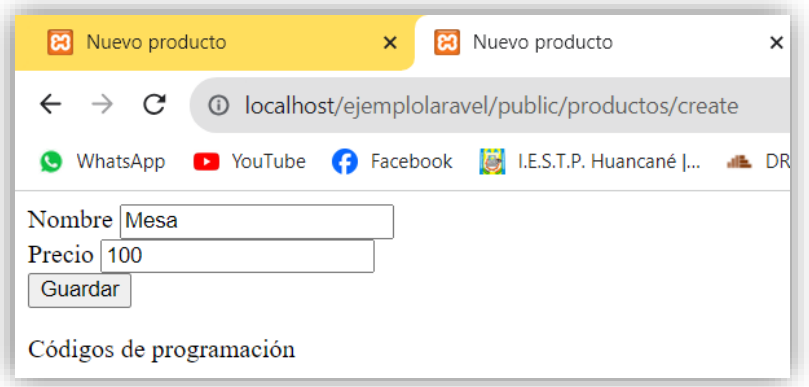


Ver código fuente de la página, y vemos que ya está nuestro token.



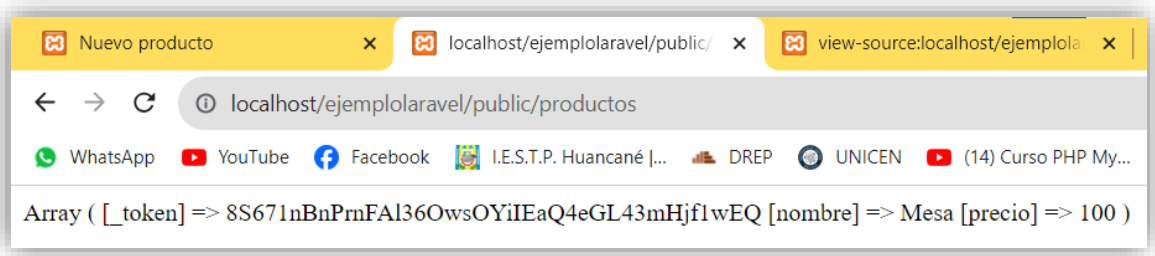
La parte en el que dice value es el token de nuestra sesión, si usted abre el sistema con otro navegador, el sistema va a generar otro token, nunca va a ser el mismo, precisamente por seguridad lo va a estar cambiando, y si no envías este token, no va a ser una solicitud autentica.

Hagamos la prueba nuevamente.



Clic en guardar.

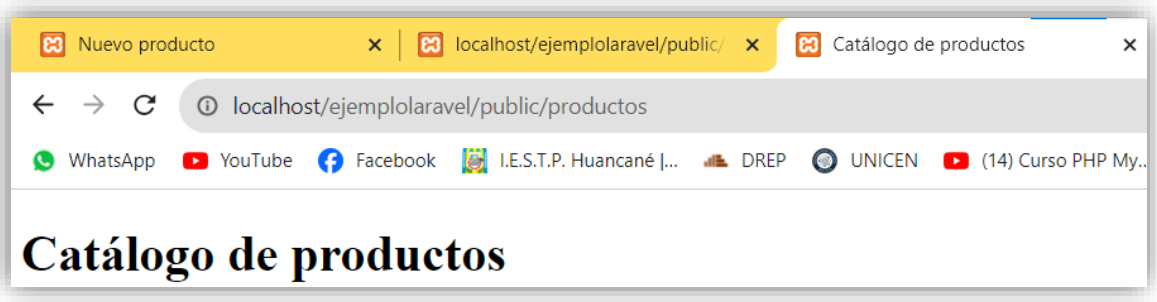
Vemos que ya se está ejecutando correctamente



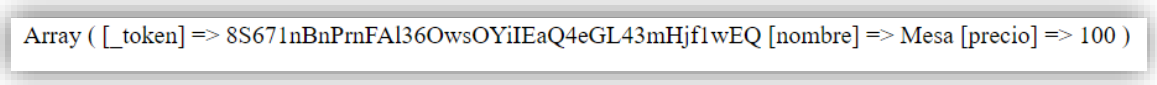
Primero mira la ruta.

http://localhost/ejemplolaravel/public/productos

Si copiamos esa ruta y la ejecutamos en otra ventana, nos va enviar a otro lado, lo que va a hacer la diferencia es el método por el cual lo estamos enviando y ya ingresaría a la función **store** y nos imprime todo el request.



En el **request** vamos a encontrar el token que estamos enviando, que lo valida, y si es el mismo que el lo generó, dirá que si es autentico y te dejara pasar. En la cual te muestra el nombre y el precio.



Ya puedes procesar toda la información.

Ahora veamos que más nos puede proporcionar el **request**

Colocamos un echo y vamos a imprimir con un HTML

Para saber el método con el cual se está enviando esta solicitud.

```
echo $request->method();
```

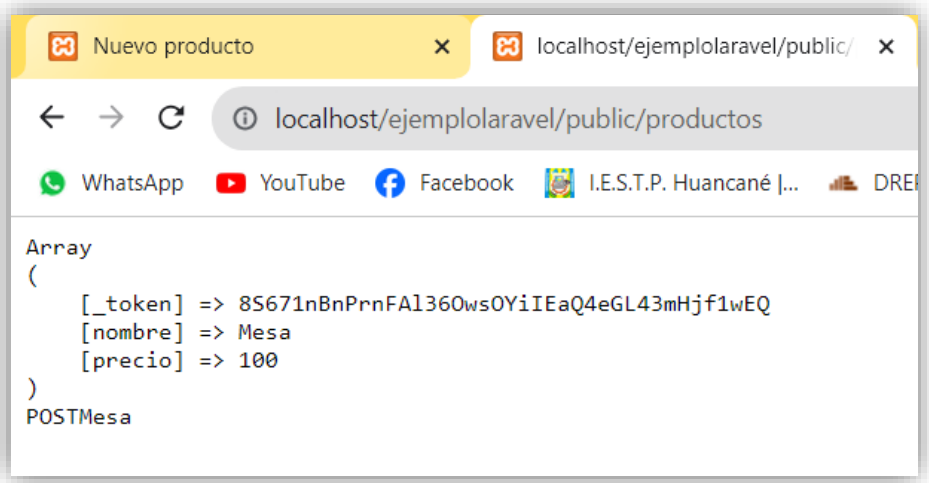
Y en caso de que deseemos recibir el valor del campo **nombre**

```
echo $request->input('nombre');
```

app > Http > Controllers > ProductoController.php > PHP Intelephense > ProductoController > store

```php
namespace App\Http\Controllers;
use App\Http\Controllers\Controller;
use Illuminate\Http\Request;

class ProductoController extends Controller
{
    public function index()
    {
        return view('productos.index');
    }
    public function show($nombre)
    {
        return view('productos.show', ['producto'=> $nombre]);
    }
    public function create()
    {
        return view('productos.create');
    }
    public function store(Request $request)
    {
        echo "<pre>";
        print_r($request->all());
        echo $request->method();
        echo $request->input('nombre');
        echo "</pre>";
    }
}
```

Luego guardamos y actualizamos la página anterior. OJO DEL FORMULARIO, NO DEL LINK COPIADO.



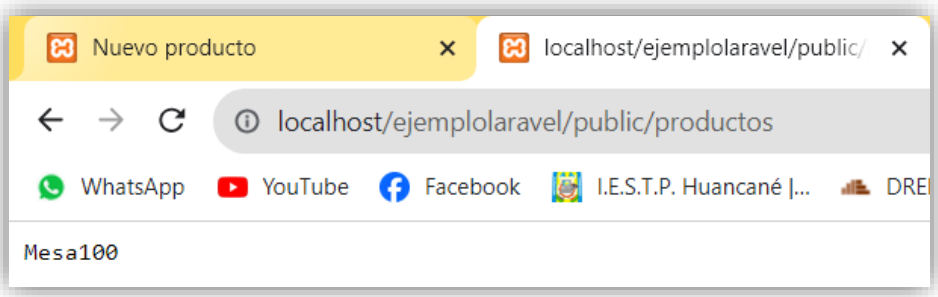
En la parte de abajo nos muestra POST por ser el tipo de solicitud que se está

realizando. Y Mesa es el nombre que estamos enviando.

Ahora podemos recibir todos los datos.

```php
public function store(Request $request)
{
    echo "<pre>";
    echo $request->input('nombre');
    echo $request->input('precio');
    echo "</pre>";
}
```

Resultado



Vemos nuestra tabla.

Método	URL	Acción	Proceso
GET	/productos	index	Catálogo
GET	/productos/create	create	Formulario nuevo
POST	/productos	store	Guardar
GET	/productos/{id}	show	Mostrar
GET	/productos/{id}/edit	edit	Formulario editar
PUT/PATCH	/productos/{id}	update	Actualizar
DELETE	/productos/{id}	destroy	Eliminar

Ya vimos la parte de **Guardar.**

Ahora vamos a hacer las siguientes rutas. Como pueden darse cuenta, son rutas muy similares.

7.2 FORMULARIO EDITAR

Configuremos la nueva ruta

Vamos a hacerlo por el método GET, donde el módulo es **productos,** después colocamos / y el parámetro que va ser el id y otro parámetro que va ser el formulario de editar. Y este se tiene que ir a la función **edit**

7.3 FORMULARIO ACTUALIZAR

Luego vamos a hacer lo de **actualizar**, y es diferente a guardar, porque acá trabajaremos con update. Con el método PUT, entonces no importa que tengamos la misma URL si el método está cambiando

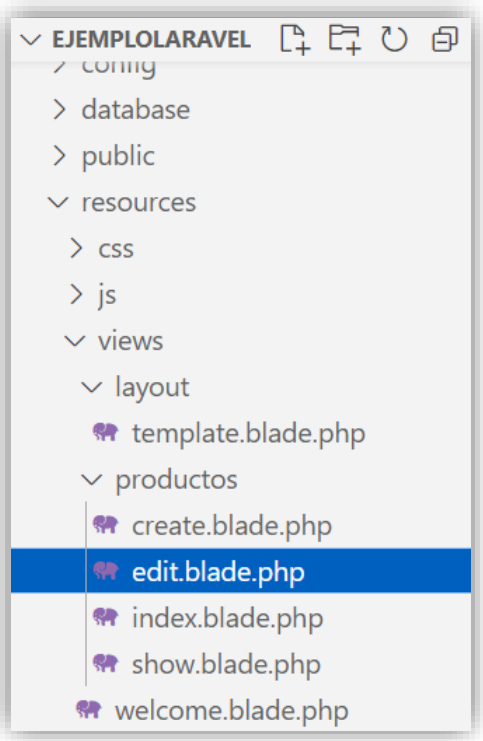
7.4 FORMULARIO ELIMINAR

Donde también es la misma ruta, pero ahora el método va ser DELETE

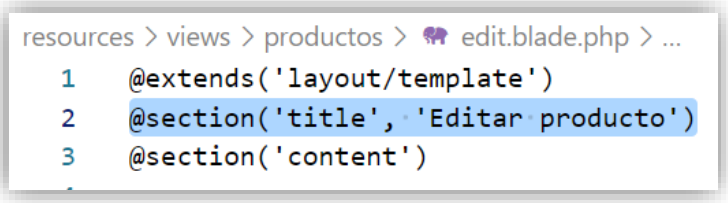
routes > web.php > ...

```
18
19  Route::get('/', function () {
20      return "<h1> Hola mundo </h1>";
21  });
22
23  route::get('productos', [ProductoController::class, 'index']);
24  route::get('productos/create', [ProductoController::class, 'create']);
25  route::get('productos/{producto}', [ProductoController::class, 'show']);
26  route::post('productos', [ProductoController::class, 'store']);
27  route::get('productos/{id}/edit', [ProductoController::class, 'edit']);
28  route::put('productos/{id}', [ProductoController::class, 'update']);
29  route::delete('productos/{id}', [ProductoController::class, 'destroy']);
```

Ya hemos agregado nuestras rutas, ahora, vamos por las vistas, y para poder crear la vista de **editar** vamos a copiar el de create.blade.php con un ctrl + c y ctrl + v ahí mismo y lo cambiamos a edit.blade.php.



Y podemos cambiar el título por **Editar producto**



Este se va ir a **productos / y** necesitamos enviar el id que vamos a editar, aquí vamos a trabajar con PHP concatenando la variable id y también va ser por el método **post,** ahora si vamos a la tabla no es necesario el método post, sino el método PUT o PATCH

En method no se puede poner solo PUT porque no lo va a reconocer, lo vamos a hacer con el método POST, pero vamos a agregar un dato adicional @method (“PUT”), también puede se PATCH.

```
resources > views > productos > edit.blade.php > form
1   @extends('layout/template')
2   @section('title', 'Editar producto')
3   @section('content')
4
5   <form action="{{url('/productos/.$id')}}" method="post">
6       @method("PUT")
7       @csrf
```

Luego guardamos y enviamos al **ProductoController.php**

Aquí creamos la función edit

```
app > Http > Controllers > ProductoController.php > PHP Intelephense > ProductoController > edit
18          return view('productos.create');
19      }
20      public function store(Request $request)
21      {
22          echo "<pre>";
23          echo $request->input('nombre');
24          echo $request->input('precio');
25          echo "</pre>";
26      }
27      public function edit($id){
28          return view('productos.edit', ['id'=>$id]);
29      }
30  }
```

Este id, cuando llamemos a la vista, la va recibir e imprimir en ese apartado

```
resources > views > productos > edit.blade.php > form
1   @extends('layout/template')
2   @section('title', 'Editar producto')
3   @section('content')
4
5   <form action="{{url('/productos/.$id')}}" method="post">
6       @method("PUT")
7       @csrf
```

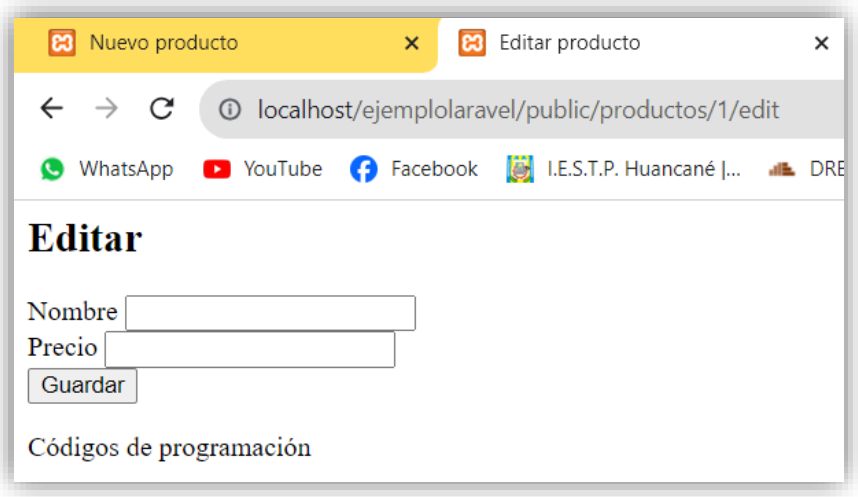
Vamos a hacer la prueba y buscamos el id 1/edit

Podemos ver en el titulo **Editar producto,** aunque también le podemos agregar

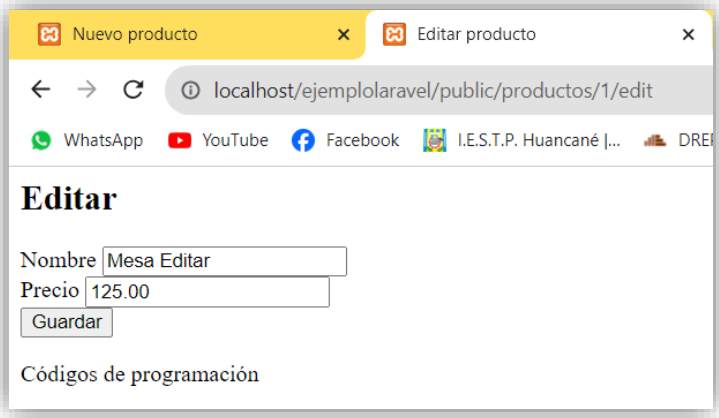
un título **Editar**

```
resources > views > productos > edit.blade.php > ...
1   @extends('layout/template')
2   @section('title', 'Editar producto')
3   @section('content')
4   <h2>Editar</h2>
```

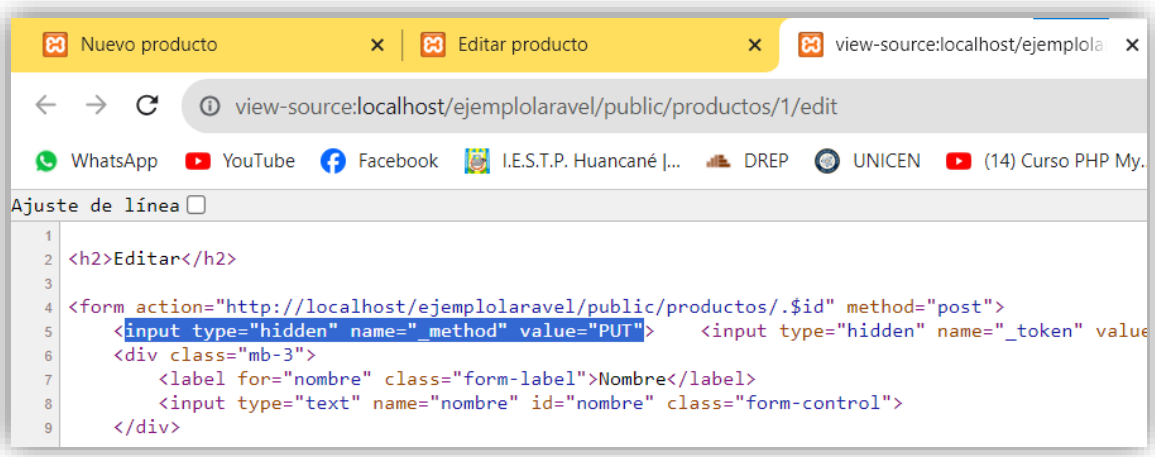
Grabamos y vemos



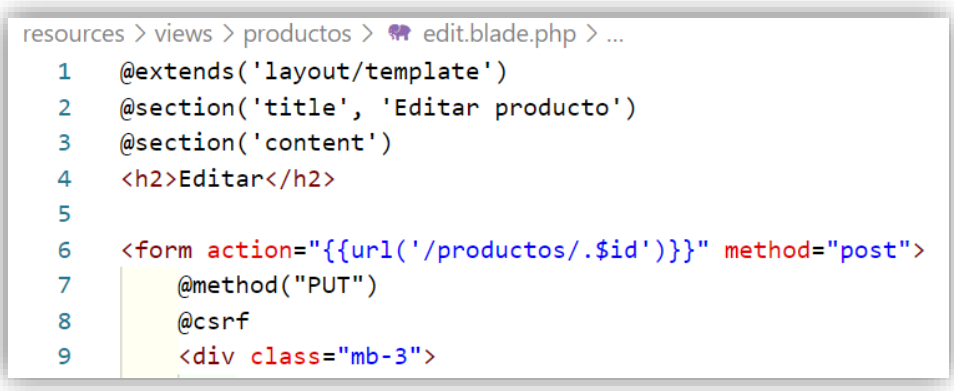
Como vemos ya estamos en el formulario de **editar** y como aún no hemos conectado con la base de natos, no va traer información, pero hagamos la prueba, por ejemplo, en nombre poner Mesa Editar y en precio 125.00

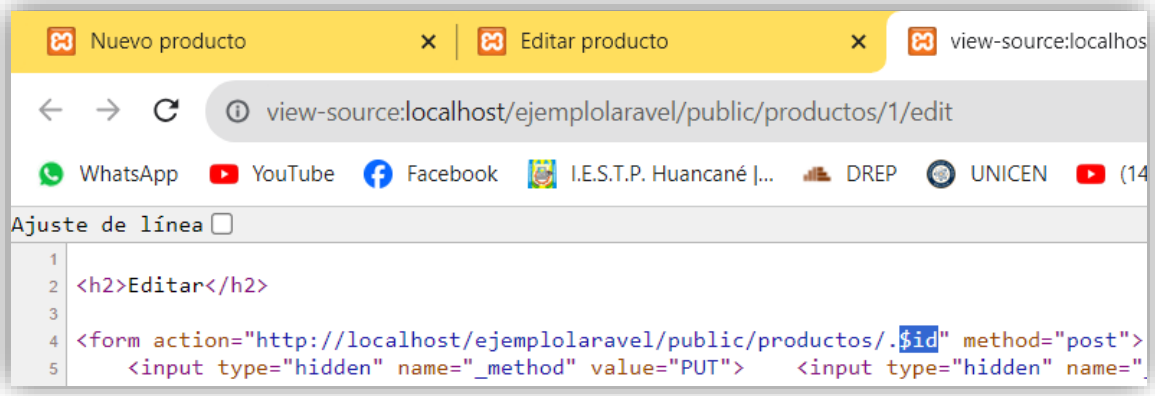


Luego podemos dar clic derecho, ver código fuente, y mira lo que cambia aquí, en donde nos esta enviando otro campo oculto que se llama _method y esta pasando el valor PUT



Acá tenemos un detalle, si vemos el código de la página, no nos esta imprimiendo el código del id.

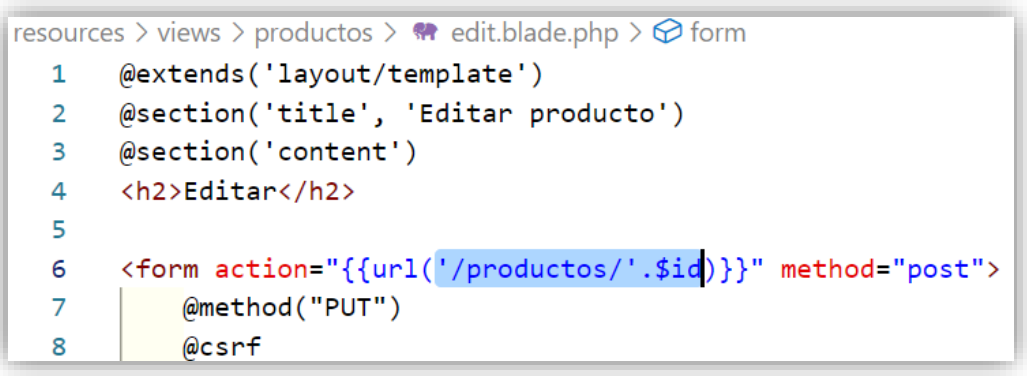




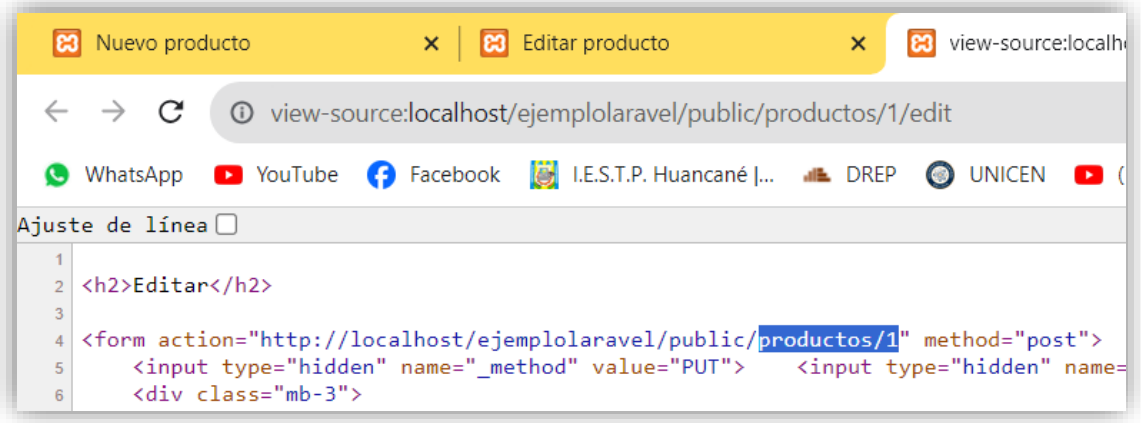
Esto pasa porque hicimos mal la concatenación en **edit.blade.php**

La comilla simple colocarla antes de la concatenación,

Corrigiendo:



Actualizamos la página editar, y luego clic derecho y ver código fuente.



Como puede ver, ahora si está pasando el parámetro id correctamente. Ahora ya solo enviamos los parámetros Mesa editar y el precio 125.

Pero antes de enviarla, necesitamos crear la ruta de abajo

```
resources > views > productos > edit.blade.php > form
1   @extends('layout/template')
2   @section('title', 'Editar producto')
3   @section('content')
4   <h2>Editar</h2>
5
6   <form action="{{url('/productos/'.$id)}}" method="post">
7       @method("PUT")
8       @csrf
```

La ruta ya lo hablamos creado ahí, solo nos falta la función update

```
route::get('productos', [ProductoController::class, 'index']);
route::get('productos/create', [ProductoController::class, 'create']);
route::get('productos/{producto}', [ProductoController::class, 'show']);
route::post('productos', [ProductoController::class, 'store']);
route::get('productos/{id}/edit', [ProductoController::class, 'edit']);
route::put('productos/{id}', [ProductoController::class, 'update']);
route::delete('productos/{id}', [ProductoController::class, 'destroy']);
```

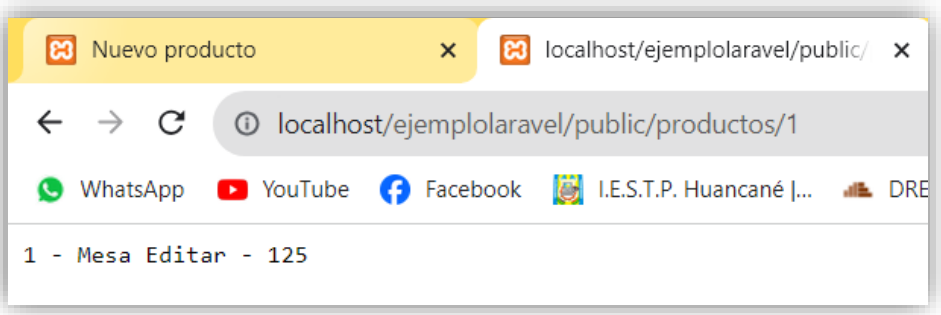
Para ello, vamos a generar una función update en ProductoController.php

En donde la función va a recibir dos datos, el primer dato puede ser de tipo request y también necesita recibir un id, el id lo estamos enviando mediante la URL.

```
app > Http > Controllers > ProductoController.php > ...
30      public function update(Request $request, $id){
31          echo "<pre>";
32          echo $id . ' - ';
33          echo $request->input('nombre') . ' - ';
34          echo $request->input('precio');
35          echo "</pre>";
36      }
37  }
```

Luego guardamos y probamos el formulario editar, ingresamos los datos y damos clic en guardar.

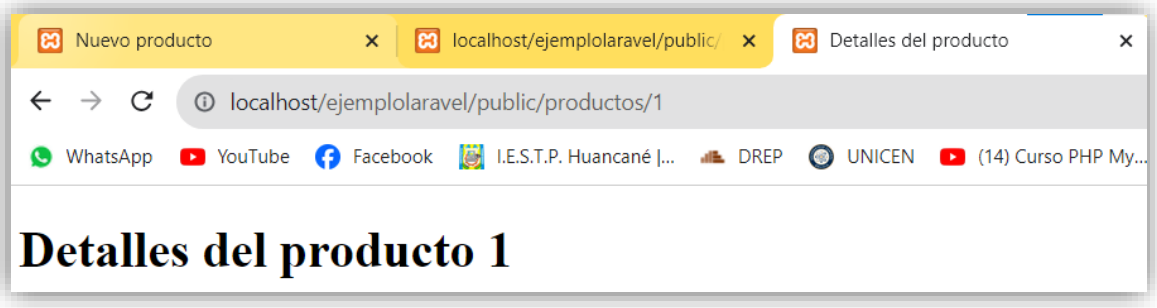
Resultado.



Y vemos que ahora se esta enviando al siguiente URL:

http://localhost/ejemplolaravel/public/productos/1

Ahora, si copiamos esa URL y la pegamos en otra ventana, lo estaríamos solicitando por GET, entonces nos estaría enviando a los detalles del producto



Ahí podemos ver la diferencia entre mismas rutas, pero son métodos de envío de datos diferentes

Ahora vamos eliminar, que es con el método DELETE

Esto lo vamos a trabajar con la vista del Index, porque regularmente ahí es donde registramos los registros.

En esta parte vamos a generar un formulario también de tipo POST, pero ahora este se va ir a un URL de productos y con el id. Esta ruta es la misma que la anterior, solo que ahora va a cambiar el método.

Agregamos nuestro método DELETE y también agregamos la protección csrf

resources > views > productos > index.blade.php > ...

```
8   <body>
9       <h1>Catálogo de productos vista</h1>
10      <form action="{{url('/productos/'.$id)}}" method="post">
11          @method("DELETE")
12          @csrf
13          <button type="submit">Eliminar</button>
14      </form>
15  </body>
16  </html>
```

Es lo único que vamos a hacer, pues la ruta ya lo tenemos creada. Pues se va ir a la función destroy, así que debemos implementar la función destroy.

Claro en ProductoController.php

app > Http > Controllers > ProductoController.php > ...

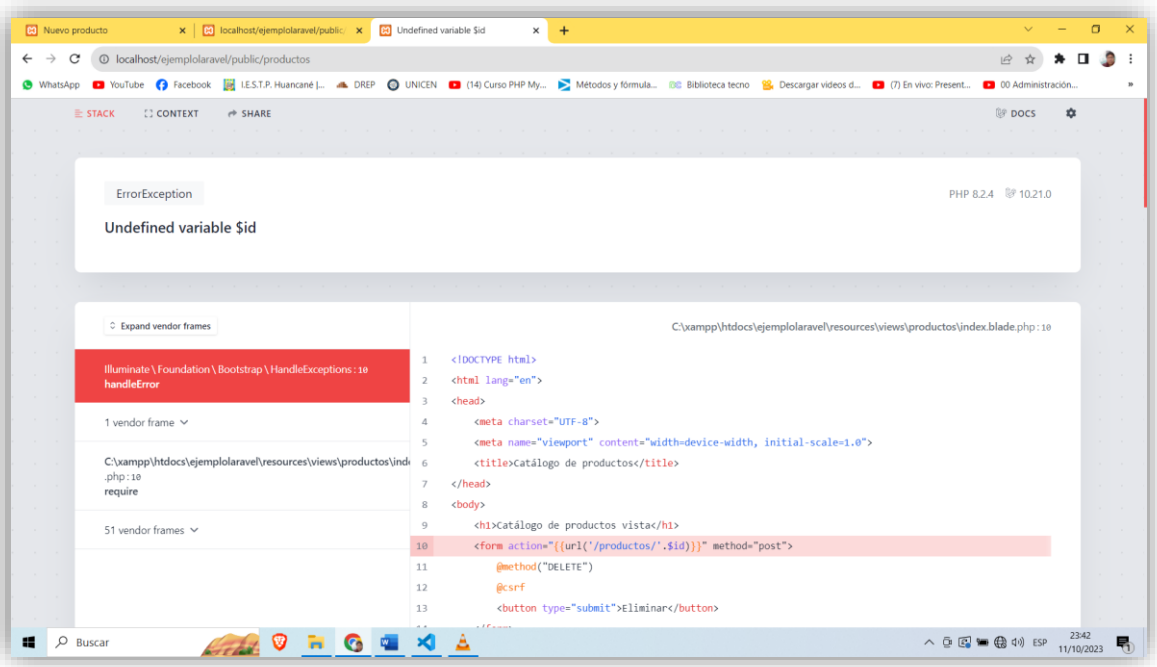
```
37      public function destroy($id){
38          echo "Registro $id eliminado";
39      }
40  }
```

Guardamos y ahora hagamos la prueba.

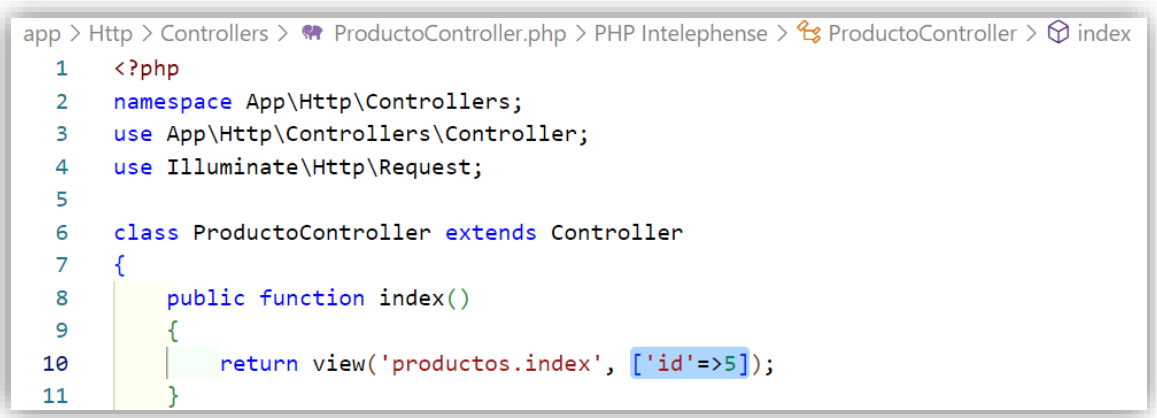
Vemos que tenemos un error, este error nos esta mostrando porque no detecta el id en la vista, en index.blade.php en esa parte le estamos declarando el id, pero no lo detecta porque no lo estamos pasando al Controller.

resources > views > productos > index.blade.php > ...

```
8   <body>
9       <h1>Catálogo de productos vista</h1>
10      <form action="{{url('/productos/'.$id)}}" method="post">
```

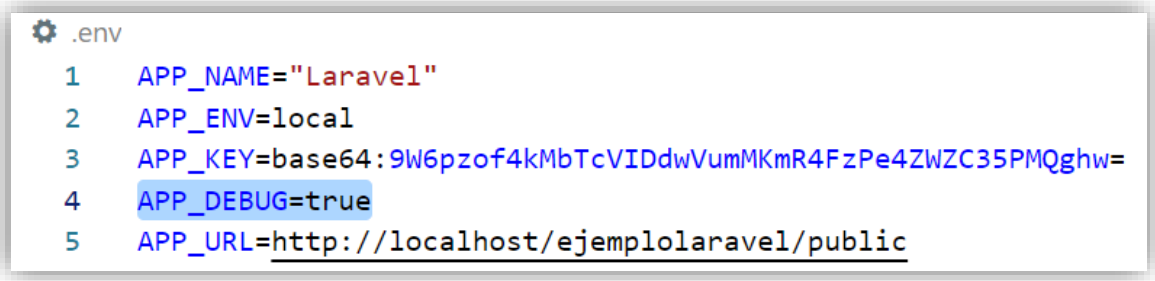


Así que nos vamos a la función index del archivo ProductoController.php, que es el que muestra esta vista, y le vamos a agregar un parámetro que se llame id y lo igualamos a 1 o a 5, lo guardamos, actualizamos la vista y probamos.



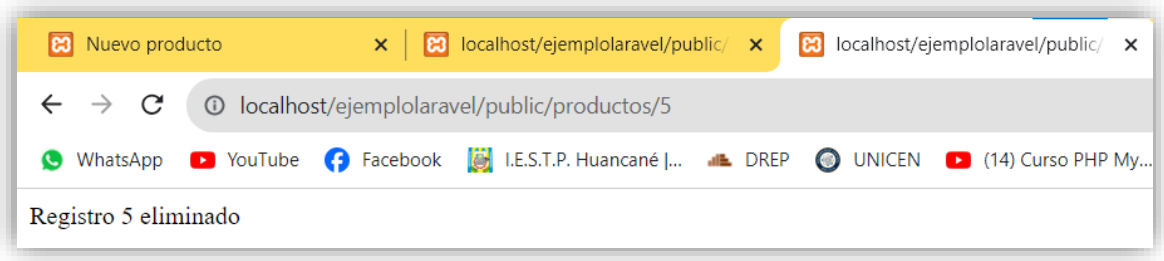
Probando.

En caso no te aparezcan los errores, puedes ir a .env y donde dice APP_DEBUG colocas TRUE para que te muestre los errores.



Luego en

Hagamos clic en eliminar

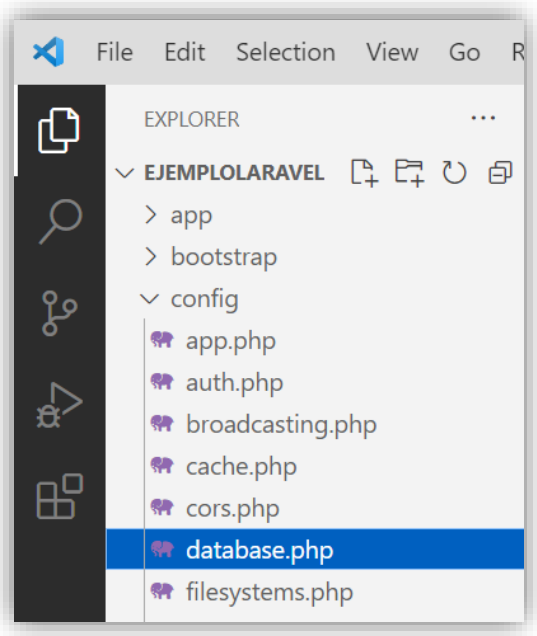

Nuevo producto
localhost/ejemplolaravel/public/
localhost/ejemplolaravel/public/
localhost/ejemplolaravel/public/productos/5
WhatsApp
YouTube
Facebook
I.E.S.T.P. Huancané |...
DREP
UNICEN
(14) Curso PHP My...
Registro 5 eliminado

CAPÍTULO 8

CONEXIÓN A BASE DE DATOS

Ahora vamos a aprender a trabajar con base de datos en MySql. Con Laravel vamos a poder trabajar normalmente para traer información y mostrarlo en nuestras vistas.

Para poder hacer una configuración, vamos a abrir Visual Studio Code y vamos a buscar el archivo en la carpeta de **config/database.php**



Aquí encontrarás la información para hacer la conexión a la base de datos, la predefinida es la MySql, pero también podemos conectarnos a otras bases de datos como SQL Server, PostgreSQL, SQLite, entre otras.

En este caso vamos a trabajar con MySql, en la parte de abajo podemos encontrar las configuraciones para los motores con las cuales se puede conectar Laravel

```
'sqlite' => [
    'driver' => 'sqlite',
    'url' => env('DATABASE_URL'),
    'database' => env('DB_DATABASE', database_path('database.sqlite')),
    'prefix' => '',
    'foreign_key_constraints' => env('DB_FOREIGN_KEYS', true),
],

'mysql' => [
    'driver' => 'mysql',
    'url' => env('DATABASE_URL'),
    'host' => env('DB_HOST', '127.0.0.1'),
    'port' => env('DB_PORT', '3306'),
    'database' => env('DB_DATABASE', 'forge'),
    'username' => env('DB_USERNAME', 'forge'),
```

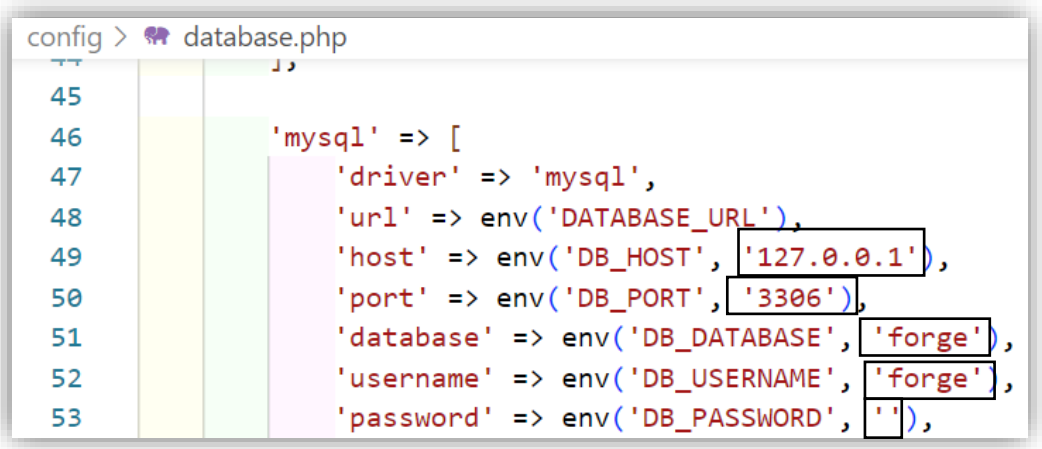
Y la que nos interesa es la de MySql, entonces en la parte de arriba debe estar colocado el mysql.

```
'default' => env('DB_CONNECTION', 'mysql'),
```

8.1 CONFIGURACIÓN PARA CONECTARNOS A LA BASE DE DATOS

Luego debemos de revisar la configuración del mysql, y ahí vamos a colocar la configuración para conectarnos a la base de datos, como, por ejemplo:

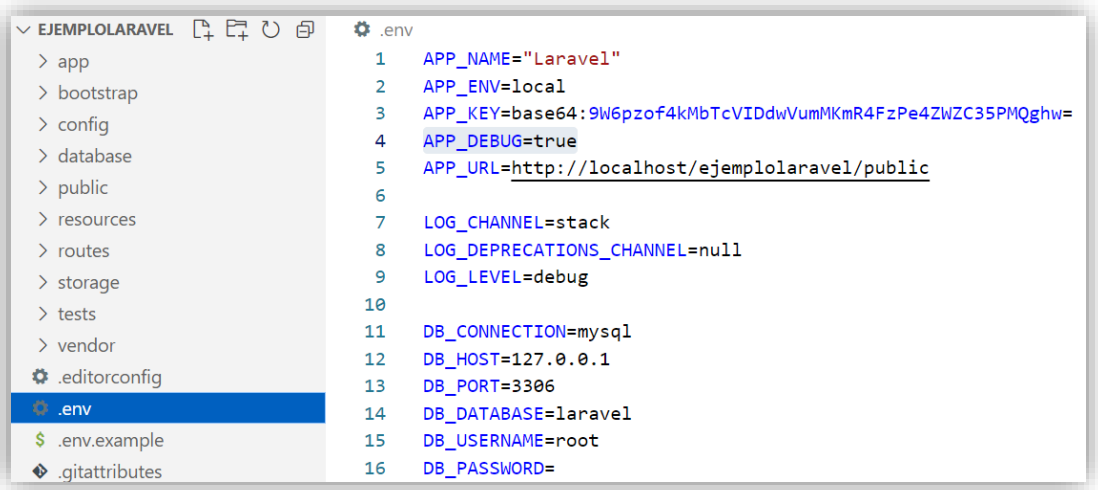
host	Ubicación de nuestro servidor	localhost
port	Puerto predefinido de MySQL	3306
database	Nombre de la base de datos	
username	El usuario para entrar a la Gestor MySQL	root
password	La contraseña del gestor de base de datos	



También debemos de ver el charset que es muy importante, sobre la forma de trabajar con la codificación de caracteres. Pero de preferencia no debemos cambiar algunos parámetros por su sensibilidad.

También debemos de recordar de anteriores sesiones, que debemos de agregar las configuraciones en el archivo **.env** que está en la raíz.

Abrimos el archivo **.env**



Podemos ver que ya tenemos parte de la configuración, ahora si bajamos mas abajo, podemos encontrar la configuración de MySQL. Que es la conexión a la base de datos.

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

La recomendación es que hagamos la configuración ahí.

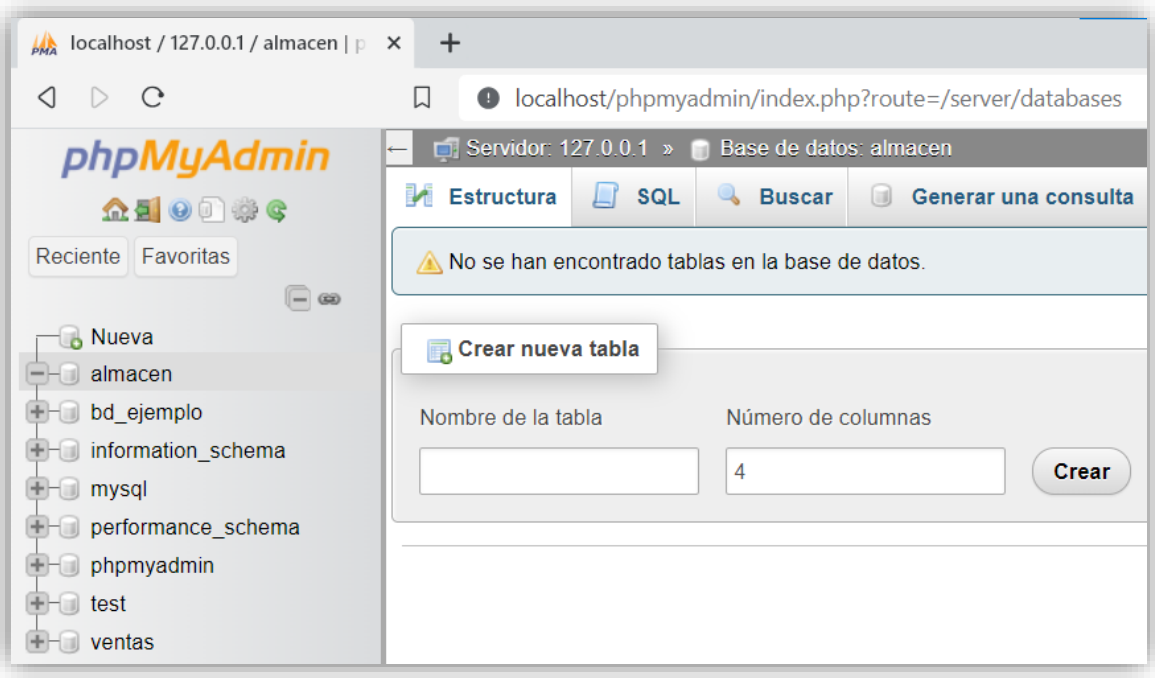
La base de datos lo vamos a cambiar por **almacen** lo demás dejémoslo, así como está.

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

Con eso ya tenemos configurado mi conexión a la base de datos.

El siguiente paso es crear la base de datos, en donde en laravel vamos a tener dos formas de poder hacerlo. En esta sesión vamos a crear la base de datos de manera común, como siempre lo hacemos, posteriormente lo haremos como lo hace Laravel.

Para ello, ingresamos al PhpMyAdmin y creamos la base de datos **almacen**.



Ahora crearemos la tabla **productos** con los siguientes detalles:

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra	Acción
1	id	int(11)			No	*Ninguna*		AUTO_INCREMENT	Cambiar Eliminar Más
2	codigo	varchar(20)	utf8_spanish2_ci		No	*Ninguna*			Cambiar Eliminar Más
3	descripcion	varchar(200)	utf8_spanish2_ci		No	*Ninguna*			Cambiar Eliminar Más
4	precio	decimal(10,0)			No	*Ninguna*			Cambiar Eliminar Más
5	existencia	int(11)			No	*Ninguna*			Cambiar Eliminar Más
6	activo	tinyint(4)			No	1			Cambiar Eliminar Más

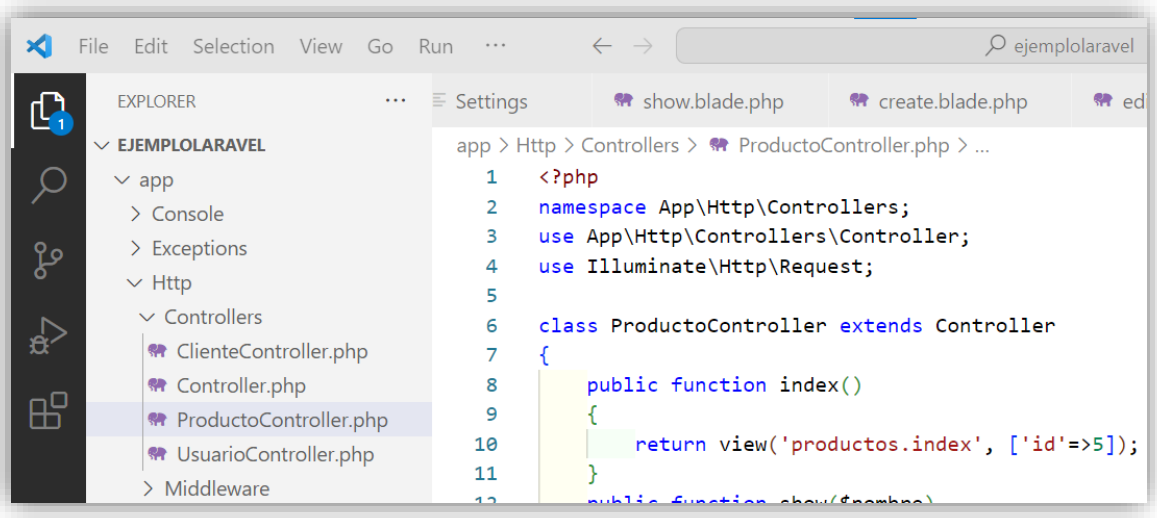
Ahora ingresaremos algunos datos:

	id	codigo	descripcion	precio	existencia	activo
Editar Copiar Borrar	1	1000	Laptop HP	2951	20	1
Editar Copiar Borrar	2	1001	Impresora Epson LX300	600	15	1
Editar Copiar Borrar	3	1002	Lector DVD	100	22	1
Editar Copiar Borrar	4	1003	Disco SSD	200	23	1
Editar Copiar Borrar	5	1004	Memoria USB	35	50	0

Bien, ahora vamos a trabajar con esta tabla y veamos como podemos utilizar esta información, para ello, nos vamos al **Controller** que es el que va interactuar con la base de datos.

Vamos a la carpeta **app/Http/Controller**

Vamos a trabajar con el archivo ProductoController.php



En este archivo ya hemos trabajado y mostrado diferente información, con la aclaración de que únicamente lo trabajamos de forma estática. Por lo que, ahora trabajaremos de forma dinámica, y esto lo haremos en el index de productos. Vamos a visualizarlo en el navegador, **localhost/ejemplolaravel/public/productos**

Ese el index que vamos a trabajar:

8.2 PRIMERO DESDE EL CONTROLLER

Aquí para poder hacer una consulta vamos a colocar una variable productos que es donde vamos a almacenar toda la consulta, luego realizamos una consulta simple.

Aquí falta que agreguemos lo que necesitamos de DB para poder trabajar, lo vamos a agregar en la parte de arriba:

Use Illuminate\Support\Facades\DB

```
app > Http > Controllers > ProductoController.php > ...
1   <?php
2   namespace App\Http\Controllers;
3   use App\Http\Controllers\Controller;
4   use Illuminate\Http\Request;
5   use Illuminate\Support\Facades\DB;
```

Una vez que ejecutamos la consulta, ya se traerá todas las filas a la variable $productos, entonces estas filas las vamos a pasar a nuestra lista, entonces en el **id** lo vamos a cambiar por **lista** y le enviamos la variable **$productos**

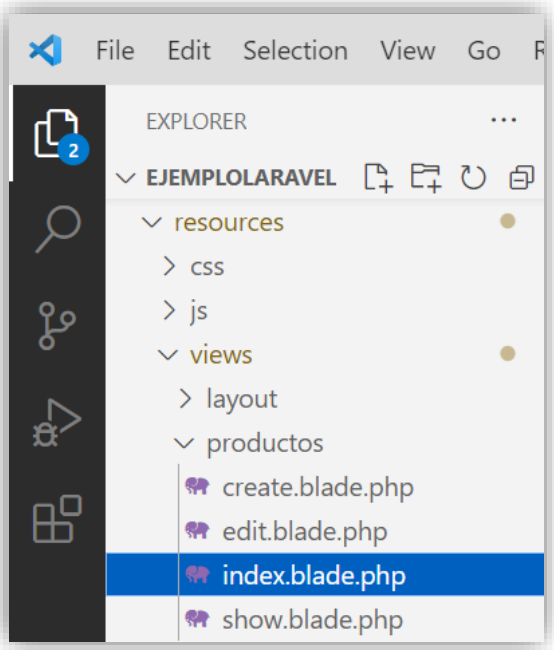
```
app > Http > Controllers > ProductoController.php > PHP Intelephense > ProductoController > store
1   <?php
2   namespace App\Http\Controllers;
3   use App\Http\Controllers\Controller;
4   use Illuminate\Http\Request;
5   use Illuminate\Support\Facades\DB;
6
7   class ProductoController extends Controller
8   {
9       public function index()
10      {
11          $productos = DB::select('SELECT * FROM productos WHERE activo = 1');
12          return view('productos.index', ['lista' => $productos]);
13      }
```

En la parte donde pusimos **productos**, podemos llamarlo de cualquier manera, puede ser listado, o lista, o como lo quieras llamar.

```
['lista' => $productos]);
```

8.3 VAMOS AHORA A LA VISTA

Ir a la carpeta **resources/views/productos/index.blade.php**

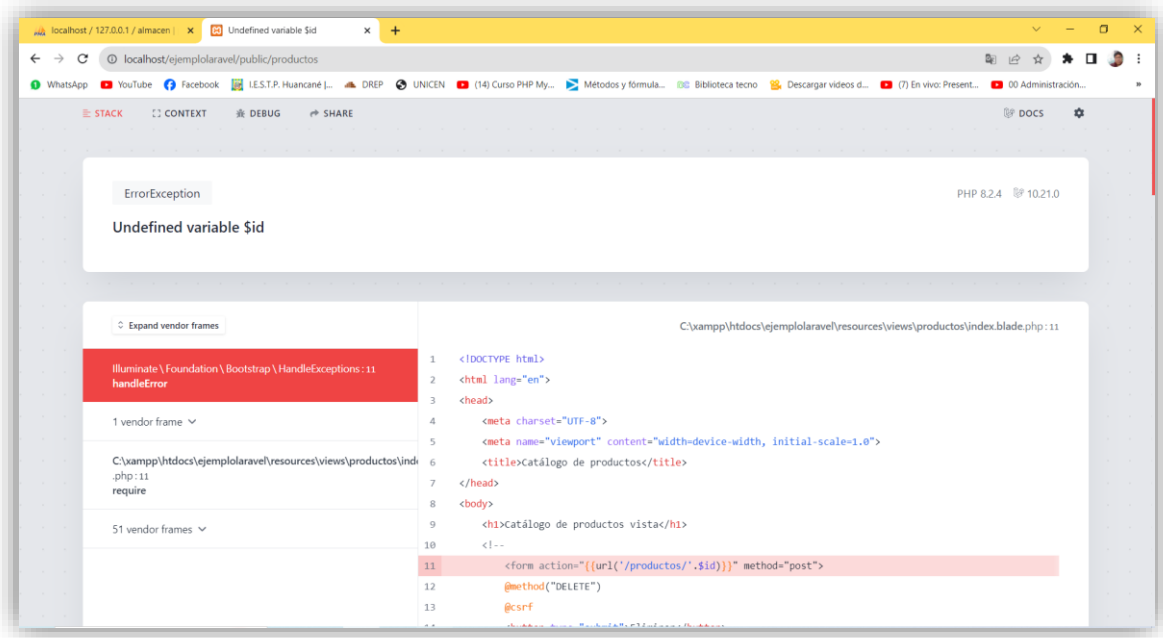


Aquí vamos a quitar el formulario creado anteriormente, claro que mas adelante lo vamos a trabajar, o en su defecto también lo podemos poner como comentario.

```
10      <!--
11          <form action="{{url('/productos/'.$id)}}" method="post">
12          @method("DELETE")
13          @csrf
14          <button type="submit">Eliminar</button>
15          </form>
16      -->
```

Primero actualicemos nuestra vista, y veremos que nos envía un error, porque esta buscando el $id que le enviábamos.

```
11          <form action="{{url('/productos/'.$id)}}" method="post">
```



Para solucionar ese problema, eliminaremos esa parte para que ya no tenga problema

```
<body>
    <h1>Catálogo de productos vista</h1>
    <!--
        <form action="" method="post">
        @method("DELETE")
        @csrf
        <button type="submit">Eliminar</button>
        </form>
    -->
</body>
```

No olvide de guardar cambios, y probamos nuevamente:

Y si hasta acá no nos muestra ningún error, significa que la conexión con la base de datos esta correcta. Pero ahora vamos a ver como mostramos la información en una tabla, de HTML 5

Para el caso de existencia, estamos poniendo Stock

Y agregamos dos más para agregar y eliminar

Esto solo es el encabezado de la tabla

Es así como mostraría el encabezado

Ahora vamos a implementar el contenido con tbody, y dentro vamos a agregar los td que vendría a ser el contenido de la consulta que hemos traído.

Pero acá lo vamos a trabajar con un **foreach**, es decir vamos a agregar código PHP pero de otra forma. Dentro del paréntesis del foreach ponemos el dato que le estamos pasando, que si recordamos lo llamamos lista as $Item

```php
class ProductoController extends Controller
{
    public function index()
    {
        $productos = DB::select('SELECT * FROM productos WHERE activo = 1');
        return view('productos.index', ['lista' => $productos]);
    }
```

En el foreach generado no es necesario agregar una llave, sino que todo lo de abajo lo va tomar como el contenido de foreach.

Vamos a empezar a imprimir con doble lave {{ }}

Agregamos dos partes vacías para editar y eliminar.

resources > views > productos > index.blade.php > html > body

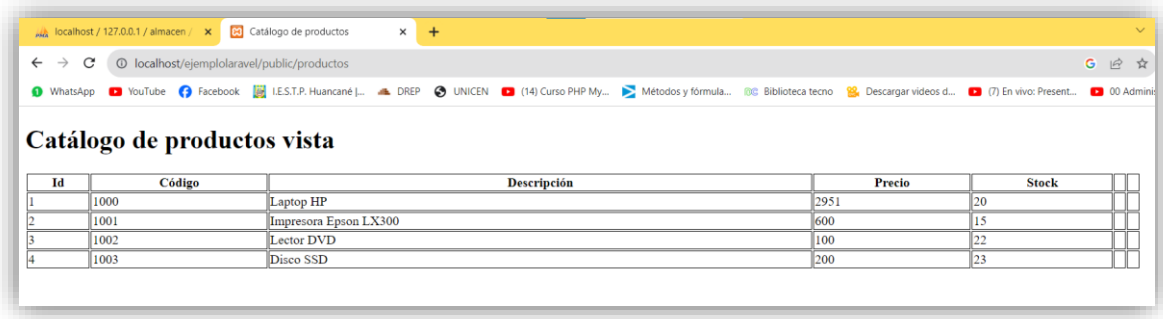
```blade
<body>
    <h1>Catálogo de productos vista</h1>
    <table>
        <thead>
            <tr>
                <th>Id</th>
                <th>Código</th>
                <th>Descripción</th>
                <th>Precio</th>
                <th>Stock</th>
                <th></th>
                <th></th>
            </tr>
        </thead>
        <tbody>
        @foreach($lista AS $Item)
        <tr>
            <td>{{$Item ->id}}</td>
            <td>{{$Item ->codigo}}</td>
            <td>{{$Item ->descripcion}}</td>
            <td>{{$Item ->precio}}</td>
            <td>{{$Item ->existencia}}</td>
            <td></td>
            <td></td>
        </tr>
        @endforeach
    </tbody>
    </table>
```

Ahora podemos hacer unos cambios para mostrarlo mejor. Dentro del <head> vamos a agregar estilos.

resources > views > productos > index.blade.php > html > head

```
1   <!DOCTYPE html>
2   <html lang="en">
3   <head>
4       <meta charset="UTF-8">
5       <meta name="viewport" content="width=device-width, initial-scale=1.0">
6       <title>Catálogo de productos</title>
7       <style>
8           table td, th {
9               border: 1px solid;
10          }
11          table{
12              width: 90%;
13          }
14      </style>
15  </head>
16  <body>
17      <h1>Catálogo de productos vista</h1>
```

Resultado: (y es así como obtenemos los datos de una base de datos)


localhost / 127.0.0.1 / almacen /
Catálogo de productos
localhost/ejemplolaravel/public/productos
WhatsApp
YouTube
Facebook
I.E.S.T.P. Huancané |...
DREP
UNICEN
(14) Curso PHP My...
Métodos y fórmula...
Biblioteca tecno
Descargar videos d...
(7) En vivo: Present...
Catálogo de productos vista
Id
Código
Descripción
Precio
Stock
1
1000
Laptop HP
2951
20
2
1001
Impresora Epson LX300
600
15
3
1002
Lector DVD
100
22
4
1003
Disco SSD
200
23

CAPÍTULO 09

MIGRACIONES

En la clase anterior hemos aprendido a conectarnos con una base de datos en MySql, creamos una BD y sus tablas, pero lo hicimos de la forma tradicional, es decir, con el PhpMyAdmin.

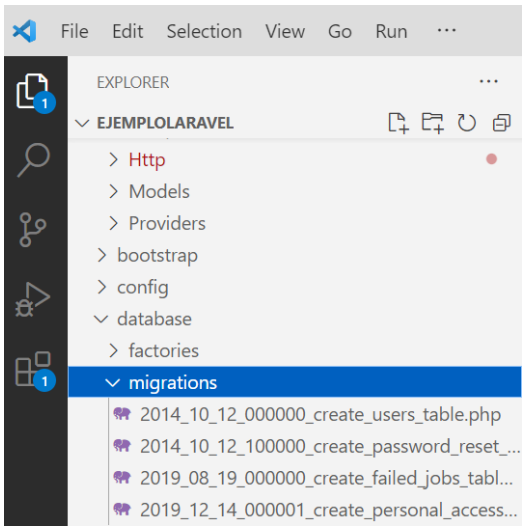
Pero, Laravel nos proporciona una herramienta que se llama migraciones, con las cuales podemos crear nuestras propias tablas, pero directamente desde Laravel, y también podemos llevar un control de las modificaciones de la estructura de la Base de datos.

Con las migraciones podemos generar un archivo donde creamos una tabla, y mas adelante se puede hacer una modificación. Esto nos va ayudar bastante cuando actualicemos nuestra aplicación.

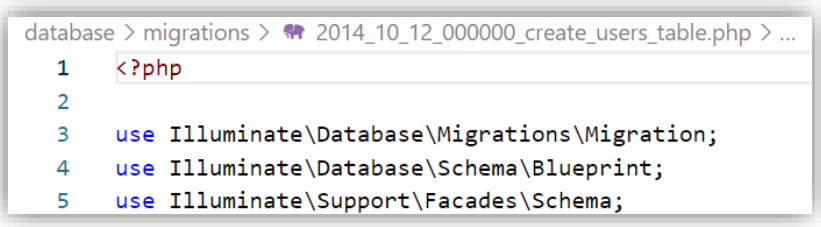
En la página de Laravel podemos ver la documentación de migraciones.

Vamos a abrir nuestro proyecto en Visual Studio Code y vamos a abrir la carpeta **database/migrations**

Aquí, vamos a encontrar algunos archivos, podemos ver la estructura.



En create_users_table.php vamos a encontrar unos **use** que nos van a permitir hacer nuestras migraciones.



En estos archivos vamos a encontrar dos funciones: cuando la función **up( )** se ejecute nos va crear una tabla de create, la tabla se va llamar **users,** y lo que esta como variable van a ser los campos.

Así como trabajamos con PhpMyAdmin en Laravel podemos indicarle el nombre de la tabla, los nombres de los campos y los tipos de datos que tienen, así como otras propiedades.

```php
public function up(): void
{
    Schema::create('users', function (Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->string('email')->unique();
        $table->timestamp('email_verified_at')->nullable();
        $table->string('password');
        $table->rememberToken();
        $table->timestamps();
    });
}
```

Mas abajo tenemos la función **down( )** el cual consiste en hacer cambios a la anterior tabla, así como eliminar la tabla.

9.1 CREACIÓN DE UNA TABLA

Ejercicio. Vamos a crear una tabla con migraciones, con el nombre **productos** pero con la misma estructura o mismos campos que la anterior tabla, pero antes debemos de renombrar la **productos** en productos_1 para que no haya errores (hacerlo en operaciones de PhpMyAdmin)

Solución

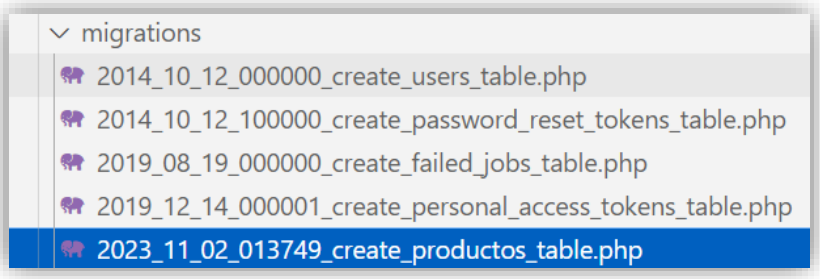
Vamos a abrir la terminal, recuerden que ya lo agregamos al path. Colocaremos

php artisan make:migration create_productos_table para que sepa que vamos a crear una tabla guión bajo el nombre de la tabla, guion bajo y el prefijo table.

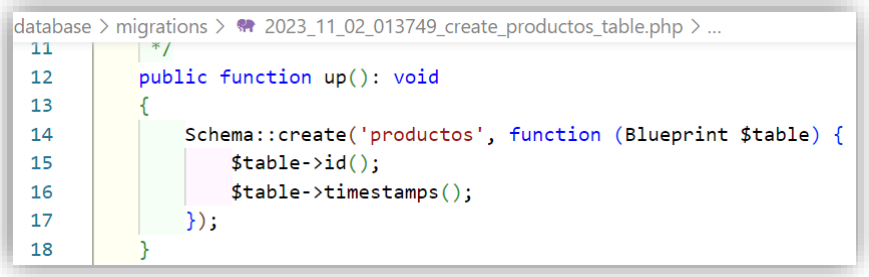
Presionamos ENTER, y con esto el Framework sabe que vamos a crear una tabla con el nombre productos.

```
PROBLEMS 53   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

PS C:\xampp\htdocs\ejemplolaravel> php artisan make:migration create_productos_table

  INFO  Migration [C:\xampp\htdocs\ejemplolaravel\database\migrations/2023_11_02_013749_creat
e_productos_table.php] created successfully.

PS C:\xampp\htdocs\ejemplolaravel>
```

Para esto vamos a buscar el archivo, que va estar en **migrations** y va estar al final. Este nombre, va a ser el nombre que le estamos dando, pero antes le va a poner la fecha de creación de la tabla. Esta fecha es para llevar un control de las modificaciones que se hizo y para saber cuál fue primero o después.



Abramos el archivo creado, y vamos a poder encontrar la estructura de la migración, ya nos trae la sintaxis para crear una tabla que se va llamar productos.

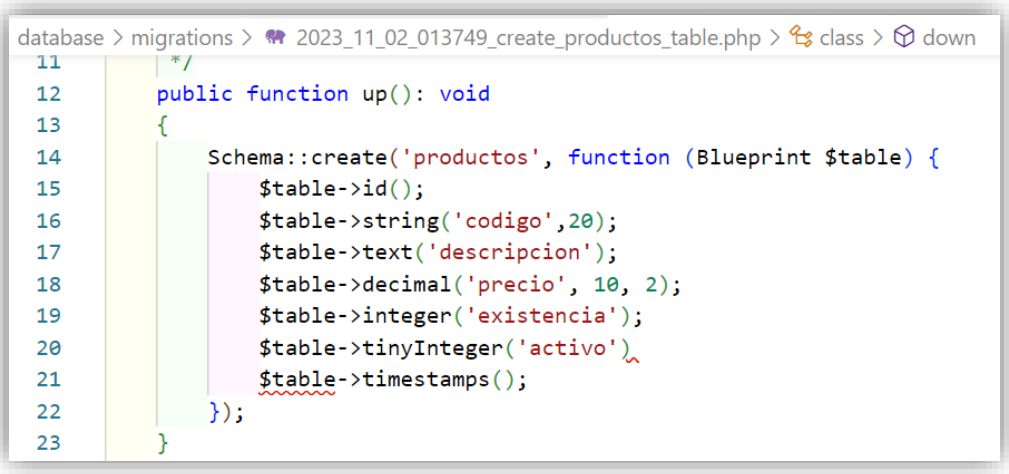


Vamos a crear la misma tabla anterior.

	#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra
☐	1	**id** 🔑	int(11)			No	*Ninguna*		AUTO_INCREMENT
☐	2	**codigo**	varchar(20)	utf8_spanish2_ci		No	*Ninguna*		
☐	3	**descripcion**	varchar(200)	utf8_spanish2_ci		No	*Ninguna*		
☐	4	**precio**	decimal(10,0)			No	*Ninguna*		
☐	5	**existencia**	int(11)			No	*Ninguna*		
☐	6	**activo**	tinyint(4)			No	1		

En documentación de laravel puede encontrar los tipos de datos para los campos de las tablas. Entonces vamos a iniciar con la variable que se llama:

- $table le vamos a asignar un String que se va llamar codigo y va tener 20 de longitud.
- Continuamos con un text que va ser la descripción
- Cuando el tipo de dato es decimal, primero se indica el nombre del campo, luego la cantidad de dígitos y finalmente la cantidad de números decimales.
- Mostramos a continuación:



Podemos ver que tenemos dos datos adicionales, el primero va ser el id( ), este id es el predefinido y el nombre del campo se va llamar **id** va ser **autoincremental** y va ser **primary key** pero también vamos a tener un campo que se va llamar **timestamps,** nos va ha llenar dos campos **created_at y update_at** estas son fechas para cuando creemos el registro (**created_at**) y para cuando lo actualicemos (**update_at**) para llevar un control de las inserciones y modificaciones de los registros.

También es muy importante el tipo de cotejamiento o de caracteres que vamos a usar, usamos el **utf8_spanish2_ci** porque nos permite usa ñ, tilde u otro carácter especial.

Pero como MySql se va actualizando tenemos uno mejor que hasta nos permite guardar emojis, asi es que vamos a colocar al inicio:

$table->charset = ‘utf8mb4’;

Esta es la predefinida que nos maneja laravel y es la que recomienda.

Después agregamos:

$table->collation = ‘utf8mb4_unicode_ci’;

Y con esto ya tenemos nuestra migración para crear una tabla que se llame productos.

```php
public function up(): void
{
    Schema::create('productos', function (Blueprint $table) {
        $table->charset = 'utf8mb4';
        $table->collation = 'utf8mb4_unicode_ci';
        $table->id();
        $table->string('codigo',20);
        $table->text('descripcion');
        $table->decimal('precio', 10, 2);
        $table->integer('existencia');
        $table->tinyInteger('activo');
        $table->timestamps();
    });
}
```

9.2 ELIMINAR UNA TABLA

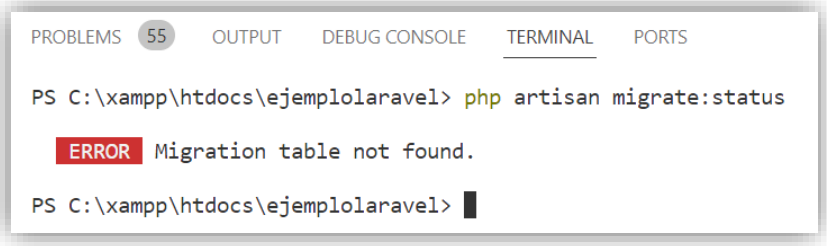
Si queremos deshacer lo anterior, solo debemos eliminar la tabla.

```php
public function down(): void
{
    Schema::dropIfExists('productos');
}
```

Para ejecutar, debemos abrir la **terminal** que estábamos trabajando, escribimos:

Php artisan migrate:status

Esto es para que primero podamos ver el estado de nuestras migraciones.



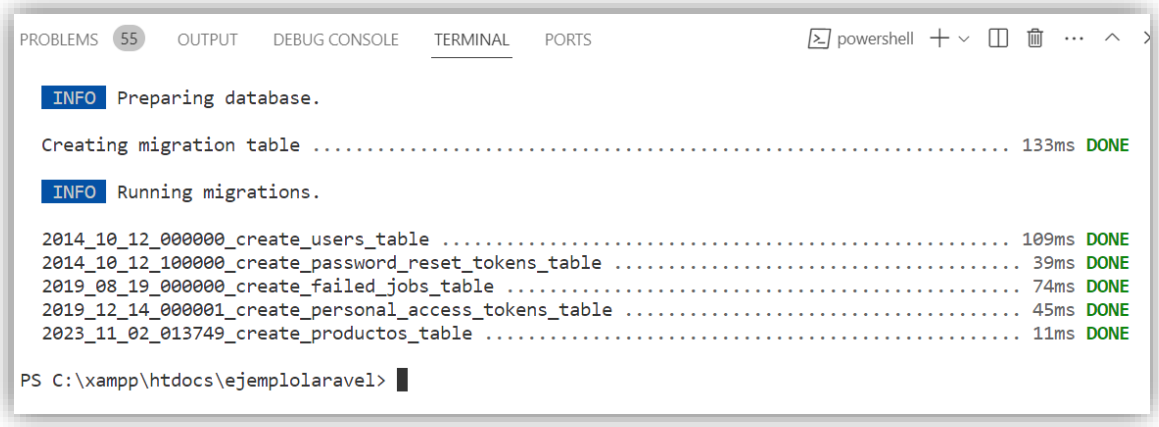
En la pantalla anterior podemos ver un mensaje de error en el cual nos indica que no existe la tabla migraciones. Es decir, no hemos ejecutado lo anterior y no tenemos ninguna tabla.

Así que debemos ejecutarla, para ello colocamos:

Php artisan migrate

Presionamos ENTER

Y nos va generar nuestras migraciones.



Vamos a nuestra base de datos ALMACEN, y vemos que nos ha creado algunas tablas que el Framework ya trae predefinidas.

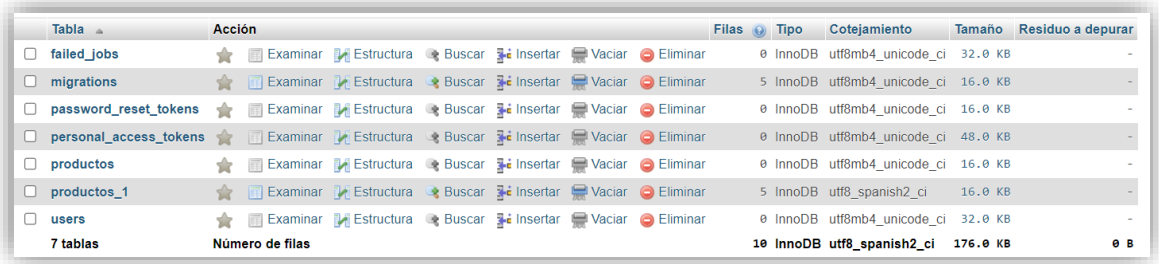
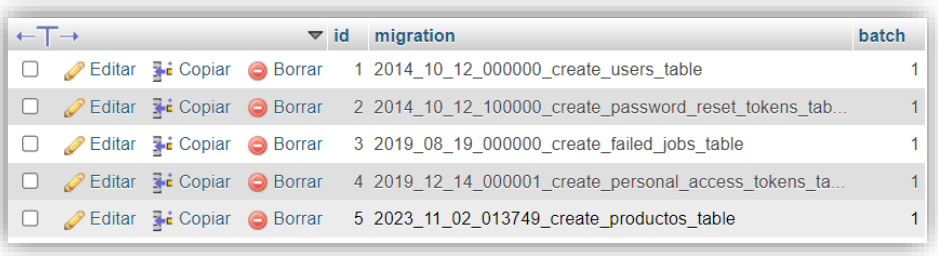


Tabla	Acción	Filas	Tipo	Cotejamiento	Tamaño	Residuo a depurar
failed_jobs	Examinar Estructura Buscar Insertar Vaciar Eliminar	0	InnoDB	utf8mb4_unicode_ci	32.0 KB	-
migrations	Examinar Estructura Buscar Insertar Vaciar Eliminar	5	InnoDB	utf8mb4_unicode_ci	16.0 KB	-
password_reset_tokens	Examinar Estructura Buscar Insertar Vaciar Eliminar	0	InnoDB	utf8mb4_unicode_ci	16.0 KB	-
personal_access_tokens	Examinar Estructura Buscar Insertar Vaciar Eliminar	0	InnoDB	utf8mb4_unicode_ci	48.0 KB	-
productos	Examinar Estructura Buscar Insertar Vaciar Eliminar	0	InnoDB	utf8mb4_unicode_ci	16.0 KB	-
productos_1	Examinar Estructura Buscar Insertar Vaciar Eliminar	5	InnoDB	utf8_spanish2_ci	16.0 KB	-
users	Examinar Estructura Buscar Insertar Vaciar Eliminar	0	InnoDB	utf8mb4_unicode_ci	32.0 KB	-
7 tablas	Número de filas	10	InnoDB	utf8_spanish2_ci	176.0 KB	0 B

Lo que importa acá es la tabla de **migrations**, lo abrimos.



	id	migration	batch
Editar Copiar Borrar	1	2014_10_12_000000_create_users_table	1
Editar Copiar Borrar	2	2014_10_12_100000_create_password_reset_tokens_tab...	1
Editar Copiar Borrar	3	2019_08_19_000000_create_failed_jobs_table	1
Editar Copiar Borrar	4	2019_12_14_000001_create_personal_access_tokens_ta...	1
Editar Copiar Borrar	5	2023_11_02_013749_create_productos_table	1

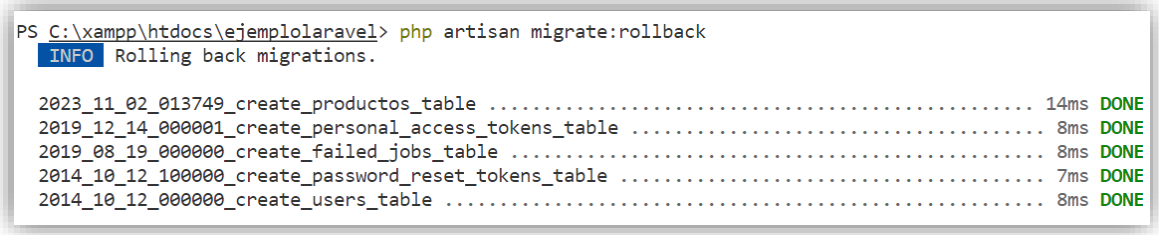
Es ahí donde lleva el control de las migraciones que hacemos.

Veamos ahora la tabla **productos.** Luego hacemos clic en estructura y vemos que ya nos ha generado la estructura

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra	Acción
1	id	bigint(20)		UNSIGNED	No	*Ninguna*		AUTO_INCREMENT	Cambiar Eliminar Más
2	codigo	varchar(20)	utf8mb4_unicode_ci		No	*Ninguna*			Cambiar Eliminar Más
3	descripcion	text	utf8mb4_unicode_ci		No	*Ninguna*			Cambiar Eliminar Más
4	precio	decimal(10,2)			No	*Ninguna*			Cambiar Eliminar Más
5	existencia	int(11)			No	*Ninguna*			Cambiar Eliminar Más
6	activo	tinyint(4)			No	*Ninguna*			Cambiar Eliminar Más
7	created_at	timestamp			Sí	*NULL*			Cambiar Eliminar Más
8	updated_at	timestamp			Sí	*NULL*			Cambiar Eliminar Más

Si hubo algún error en la generación de la tabla, lo puedes revertir con:

Php artisan migrate:rollback



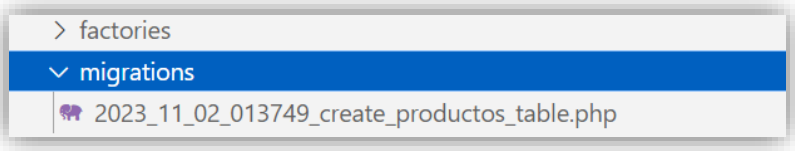
```
PS C:\xampp\htdocs\ejemplolaravel> php artisan migrate:rollback
  INFO  Rolling back migrations.

  2023_11_02_013749_create_productos_table ..... 14ms DONE
  2019_12_14_000001_create_personal_access_tokens_table ..... 8ms DONE
  2019_08_19_000000_create_failed_jobs_table ..... 8ms DONE
  2014_10_12_100000_create_password_reset_tokens_table ..... 7ms DONE
  2014_10_12_000000_create_users_table ..... 8ms DONE
```

Podemos ver nuevamente el estado: php artisan migrate:status y nos dice que si hay migraciones, pero que no están ejecutadas

```
PS C:\xampp\htdocs\ejemplolaravel> php artisan migrate:status

  Migration name ..................................................... Batch / Status
  2014_10_12_000000_create_users_table ..................................... Pending
  2014_10_12_100000_create_password_reset_tokens_table ..................... Pending
  2019_08_19_000000_create_failed_jobs_table ............................... Pending
  2019_12_14_000001_create_personal_access_tokens_table .................... Pending
  2023_11_02_013749_create_productos_table ................................. Pending
```

Se recomienda no eliminar la tabla migraciones, en caso que se quiera regresar al paso 1.

Ahora, de tu proyecto, de **migrations** vamos a eliminar los archivos adicionales para quedarnos solo con la tabla **producto.**

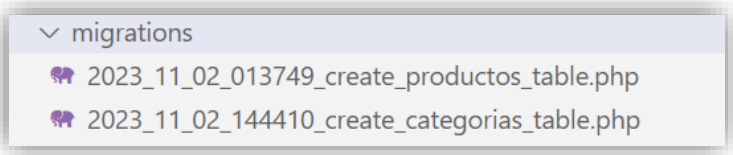


Ahora vamos a crear la migración para las categorías.

php artisan make:migration create_categorias_table

```
PS C:\xampp\htdocs\ejemplolaravel> php artisan make:migration create_categorias_table

  INFO  Migration [C:\xampp\htdocs\ejemplolaravel\database\migrations/2023_11_02_144410_create_categoria
s_table.php] created successfully.

PS C:\xampp\htdocs\ejemplolaravel>
```

Lo que hizo lo anterior fue:



Abrimos el archivo categorías, y vemos que ha creado una estructura muy similar al de los productos, solo que ahora cambia el nombre de la tabla por **categorías**

Implementemos la tabla **categorías**

database > migrations > 2023_11_02_144410_create_categorias_table.php > class > up

```php
    public function up(): void
    {
        Schema::create('categorias', function (Blueprint $table) {
            $table->id();
            $table->string('nombre', 50);
            $table->tinyText('descripcion')->nullable();

            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down(): void
    {
        Schema::dropIfExists('categorias');
    }
};
```

Nota: *nullabel( )* que quiere decir que puede contener texto o la podemos dejar vacía.

Y como dijimos anteriormente en el down solo eliminamos la tabla.

Luego abrimos la terminal y ejecutamos la creación de la tabla.

```
PS C:\xampp\htdocs\ejemplolaravel> php artisan migrate

  INFO  Running migrations.

  2023_11_02_144410_create_categorias_table ..................................................... 20ms DONE

PS C:\xampp\htdocs\ejemplolaravel>
```

Con la anterior ejecución se va ejecutar la nueva migración, vamos a la base de datos y vemos la tabla categorías.

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra	Acción
1	id	bigint(20)		UNSIGNED	No	*Ninguna*		AUTO_INCREMENT	Cambiar Eliminar Más
2	nombre	varchar(50)	utf8mb4_unicode_ci		No	*Ninguna*			Cambiar Eliminar Más
3	descripcion	tinytext	utf8mb4_unicode_ci		Sí	*NULL*			Cambiar Eliminar Más
4	created_at	timestamp			Sí	*NULL*			Cambiar Eliminar Más
5	updated_at	timestamp			Sí	*NULL*			Cambiar Eliminar Más

Haber veamos el estado en visual studio code.

```
PS C:\xampp\htdocs\ejemplolaravel> php artisan migrate:status

  Migration name ..................................................................... Batch / Status
  2019_12_14_000001_create_personal_access_tokens_table ..................................... [2] Ran
  2023_11_02_013749_create_productos_table ................................................... [2] Ran
  2023_11_02_144410_create_categorias_table ................................................. [3] Ran
```

Aquí nos va a registrar o mostrar todas las migraciones que tenemos justamente para saber cuál va primero y cual va después.

Ahora vamos a hacer un rollback

Así: **php artisan migrate:rollback** pero si ejecuto esta instrucción va eliminar todas las migraciones y tal solo queremos eliminar una anterior, es decir, la de categorías. Entonces debemos colocar una instrucción adicional - - step=1 es decir, le decimos que solo regrese a la ultima migración. Presionamos ENTER.

```
PS C:\xampp\htdocs\ejemplolaravel> php artisan migrate:rollback --step=1

  INFO  Rolling back migrations.

  2023_11_02_144410_create_categorias_table ................................................. 13ms DONE

PS C:\xampp\htdocs\ejemplolaravel>
```

Veamos la base de datos y comprobaremos que ya no tenemos la tabla de categorías.

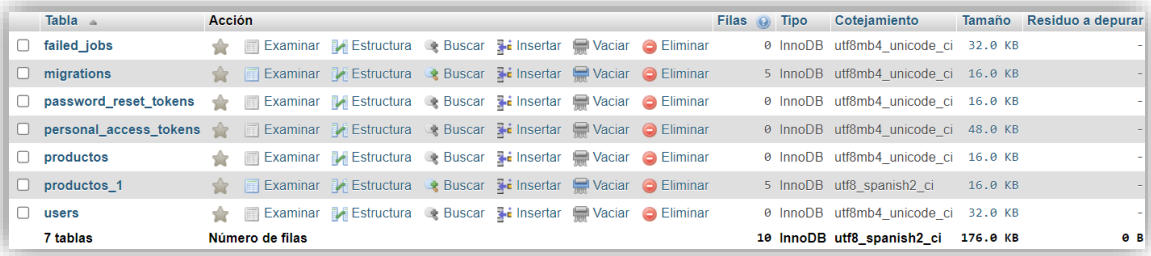
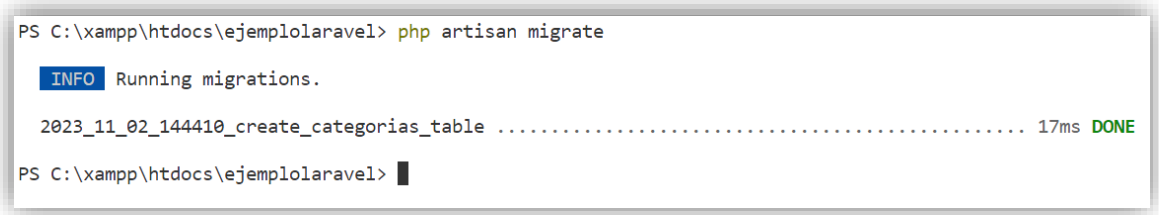


	Tabla	Acción	Filas	Tipo	Cotejamiento	Tamaño	Residuo a depurar
☐	failed_jobs	Examinar Estructura Buscar Insertar Vaciar Eliminar	0	InnoDB	utf8mb4_unicode_ci	32.0 KB	-
☐	migrations	Examinar Estructura Buscar Insertar Vaciar Eliminar	5	InnoDB	utf8mb4_unicode_ci	16.0 KB	-
☐	password_reset_tokens	Examinar Estructura Buscar Insertar Vaciar Eliminar	0	InnoDB	utf8mb4_unicode_ci	16.0 KB	-
☐	personal_access_tokens	Examinar Estructura Buscar Insertar Vaciar Eliminar	0	InnoDB	utf8mb4_unicode_ci	48.0 KB	-
☐	productos	Examinar Estructura Buscar Insertar Vaciar Eliminar	0	InnoDB	utf8mb4_unicode_ci	16.0 KB	-
☐	productos_1	Examinar Estructura Buscar Insertar Vaciar Eliminar	5	InnoDB	utf8_spanish2_ci	16.0 KB	-
☐	users	Examinar Estructura Buscar Insertar Vaciar Eliminar	0	InnoDB	utf8mb4_unicode_ci	32.0 KB	-
	7 tablas	Número de filas	10	InnoDB	utf8_spanish2_ci	176.0 KB	0 B

Podemos ver las migraciones y veremos que tampoco está registrada.

Si la queremos ejecutar de nuevo, solo ejecutaremos la instrucción:

php artisan migrate



Puede ver el estado, y vera que ya está el de categorías. Y es así como haremos las migraciones.

Pero también tenemos otra opción, y esta es hacer modificaciones a un campo o a la estructura de una tabla.

Por ejemplo: quiero hacer una relación de la tabla categorías con la tabla productos. Y a mi tabla de productos le quiero cambiar para que el **código** sea *unique* y no se pueda repetir.

Solución

Para hacer modificaciones a la estructura de un campo o de una tabla necesitamos instalar un componente.

Desde la terminal vamos a instalar:

Composer require doctrine/dbal

Necesitamos este componente para que nos permita generar o hacer las modificaciones a las tablas porque si no, nos va mandar un error y que no podemos hacer esas modificaciones.

Aquí vemos que ya lo instaló correctamente.

```
PS C:\xampp\htdocs\ejemplolaravel> composer require doctrine/dbal
./composer.json has been updated
Running composer update doctrine/dbal
Loading composer repositories with package information
Updating dependencies
Lock file operations: 5 installs, 0 updates, 0 removals
  - Locking doctrine/cache (2.2.0)
  - Locking doctrine/dbal (3.7.1)
  - Locking doctrine/deprecations (1.1.2)
  - Locking doctrine/event-manager (2.0.0)
  - Locking psr/cache (3.0.0)
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 5 installs, 0 updates, 0 removals
  - Downloading psr/cache (3.0.0)
  - Downloading doctrine/event-manager (2.0.0)
  - Downloading doctrine/deprecations (1.1.2)
  - Downloading doctrine/cache (2.2.0)
  - Downloading doctrine/dbal (3.7.1)
  - Installing psr/cache (3.0.0): Extracting archive
  - Installing doctrine/event-manager (2.0.0): Extracting archive
  - Installing doctrine/deprecations (1.1.2): Extracting archive
  - Installing doctrine/cache (2.2.0): Extracting archive
  - Installing doctrine/dbal (3.7.1): Extracting archive
Generating optimized autoload files
```

```
> Illuminate\Foundation\ComposerScripts::postAutoloadDump
> @php artisan package:discover --ansi

  INFO  Discovering packages.

  laravel/sail ..................................................... DONE
  laravel/sanctum .................................................. DONE
  laravel/tinker ................................................... DONE
  nesbot/carbon .................................................... DONE
  nunomaduro/collision ............................................. DONE
  nunomaduro/termwind .............................................. DONE
  spatie/laravel-ignition .......................................... DONE

85 packages you are using are looking for funding.
Use the `composer fund` command to find out more!
> @php artisan vendor:publish --tag=laravel-assets --ansi --force

  INFO  No publishable resources for tag [laravel-assets].

No security vulnerability advisories found
Using version ^3.7 for doctrine/dbal
PS C:\xampp\htdocs\ejemplolaravel>
```

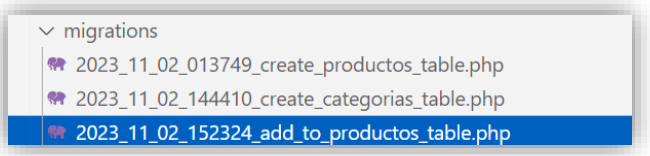
Ahora ya podemos iniciar con nuestras modificaciones. Para la cual necesitamos crear otro archivo de migración. Vamos al terminal y lo vamos a trabajar con otro nombre diferente:

Php artisan make:migration add_to_productos_table

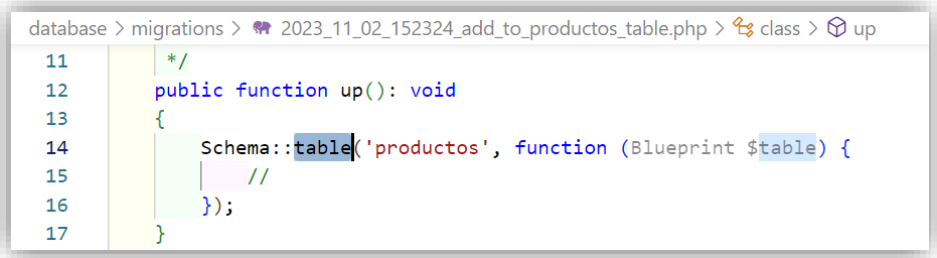
Acá le indicamos que con este archivo vamos a modificar o agregar algún campo a la tabla productos, presionamos Enter.

```
PS C:\xampp\htdocs\ejemplolaravel> php artisan make:migration add_to_productos_table

  INFO  Migration [C:\xampp\htdocs\ejemplolaravel\database\migrations/2023_11_02_152324_add_to_productos
_table.php] created successfully.

PS C:\xampp\htdocs\ejemplolaravel>
```

A generado:



Abrimos el archivo y vemos la diferencia de que ya no dice **create** ya solo dice **table** porque lo que vamos a hacer es modificar la tabla, ya no la vamos a crear, solo vamos a modificarla, y la sintaxis va ser muy similar.



Cambiemos un campo

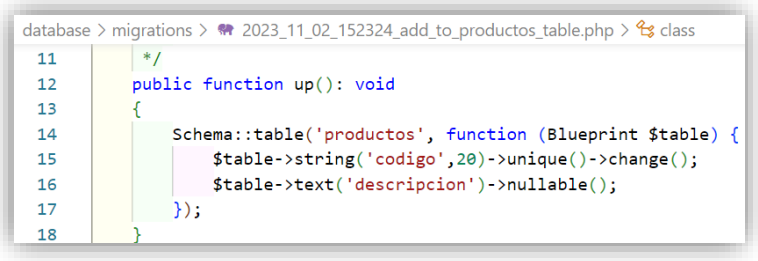
Se escribe el campo creado tal y como se había definido y luego le vamos a agregar la opción **unique( )** y como éste va ser un cambio, necesitamos agregarle al final la función **change( )** porque si no, sería como si estaría creando otro campo, pero me enviaría error porque ya existe un campo con el nombre **codigo**, es por eso que agregamos **change( )** al final para saber que ese campo lo vamos a modificar.

No solo tenemos la opción de modificar o agregarle algún dato, al campo **descripcion** le vamos a decir que sea nulo, recuerda que al final siempre debes agregar **change( )** porque nos va crear otro campo.

*Estos son dos campos que estamos modificando, el primero con un índice **unique** y el segundo para que sea **nulo**.*

Nota: vea que descripción no tiene el change( ), por lo tanto, mostrará un error.

***Corrija ese error**.*



Ahora vamos a crear un nuevo campo que va a ser la clave foránea que va servir para conectarnos con la tabla de categorías. Y aquí también podemos indicar después de que campo lo colocamos el nuevo campo.

Para agregar un nuevo campo usamos la función que se va llamar **after** y le indicamos que lo acomode después del campo **existencia** luego una función anónima, dentro del paréntesis de la función le agregamos la variable $table para que pase valores. Y dentro de la función declaramos el nuevo campo y de que va ser la clave foránea.

Este campo es el que se va crear en la tabla de productos, podemos agregarle mas funciones, el -> normal lo podemos bajar (solo para que se vea el código, normalmente lo podemos escribir a continuación del código) (estas instrucciones las podemos encontrar en la documentación de Laravel)

Podemos escribir constrained y ponemos la tabla con la cual nos vamos a conectar, en este caso con la tabla **categorías** más abajo vamos a agregar dos instrucciones para cuando se actualice o elimine de la tabla padre, en este caso de la tabla de categorías y le vamos a dejar como **cascade** para ambas.

Al finalizar colocamos el ; recordando que porque es como si fuera una sola línea. Y también al finalizar la función colocamos el ;

database > migrations > 2023_11_02_152324_add_to_productos_table.php > class > up > Closure

```php
public function up(): void
{
    Schema::table('productos', function (Blueprint $table) {
        $table->string('codigo',20)->unique()->change();
        $table->text('descripcion')->nullable()->change();

        $table->after('existencia', function($table){
            $table->foreignId('categoria_id')
            ->constrained('categorias')
            ->onUpdate('cascade')
            ->onDelete('cascade');
        });

        //$table->foreign('categoria_id')->references('id')->on('categorias');

    });


}
```

Lo anterior se puede simplificar de la siguiente manera:

Colocamos foreign que va ser categoría_id, luego references que es para indicar con que campo se va conectar, luego en on debemos indicar el nombre de la tabla con la que se va conectar.

La diferencia con la anterior estructura es de que el nuevo campo lo va agregar al final.

```php
$table->foreign('categoria_id')->references('id')->on('categorias');
```

Por lo tanto, solo ejecutaremos con la primera forma.

Ahora, que ya creamos la migración para actualizar la tabla, pero también recuerda que hay una función **down** en la cual necesitamos revertir esos cambios porque sino cuando realicemos el **rollback** lo va dejar tal cual como se hizo con esta actual migración.

Cambiaremos

```php
$table->string('codigo',20)->unique()->change();
```

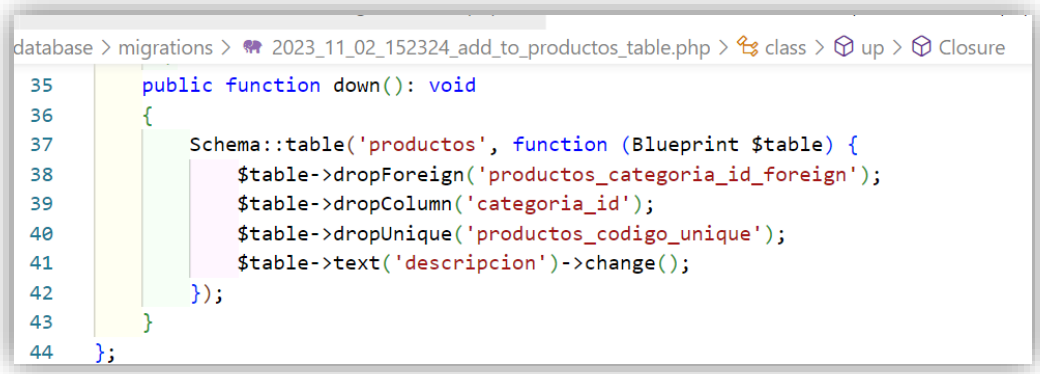
Así es que lo que primero que debemos hacer es eliminar la foreign key, porque si tu quieres eliminar el campo como tal, no te lo va permitir, porque tiene esta relación que hemos creado, entonces tenemos una opción que se llama **dropForeign('nombre del foreign key')**

En aquí no es el nombre del campo, cuando tu creas una relación, también se le da un nombre a esa relación, regularmente va ser el nombre de tu tabla principal **productos_categorias_id_foreign** *(nombre de la tabla inicial y la tabla con la que se conecta)*

Cuando revises la estructura de tu tabla vas a encontrar el nombre especifico ya que eliminas esta relación o este índice.

Luego vamos a eliminar con **dropColumn** y el nombre de la columna que se llama **categoría_id** ya después de eliminar esa columna también le puedes eliminar el unique al campo **codigo**. Ahí también pones el índice pues no puedes poner codigo, seria: **productos_codigo_unique**

Esto va depender de cómo nos vaya a crear este índice, y con esto ya estaríamos revirtiendo los cambios.



Prácticamente solo hay que quitarle el nullabe, pero antes vamos a guardar y ejecutar esa migración en el terminal.

Revisamos los cambios en la tabla de productos.

```
PROBLEMS 53   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

PS C:\xampp\htdocs\ejemplolaravel> php artisan migrate

  INFO  Running migrations.

  2023_11_02_152324_add_to_productos_table ..................... 380ms DONE

PS C:\xampp\htdocs\ejemplolaravel>
```

No debe haber errores, vea y compare los dos cambios.

Antes:

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra	Acción
1	id	bigint(20)		UNSIGNED	No	Ninguna		AUTO_INCREMENT	Cambiar Eliminar Más
2	codigo	varchar(20)	utf8mb4_unicode_ci		No	Ninguna			Cambiar Eliminar Más
3	descripcion	text	utf8mb4_unicode_ci		No	Ninguna			Cambiar Eliminar Más
4	precio	decimal(10,2)			No	Ninguna			Cambiar Eliminar Más
5	existencia	int(11)			No	Ninguna			Cambiar Eliminar Más
6	activo	tinyint(4)			No	Ninguna			Cambiar Eliminar Más
7	created_at	timestamp			Sí	NULL			Cambiar Eliminar Más
8	updated_at	timestamp			Sí	NULL			Cambiar Eliminar Más

Después:

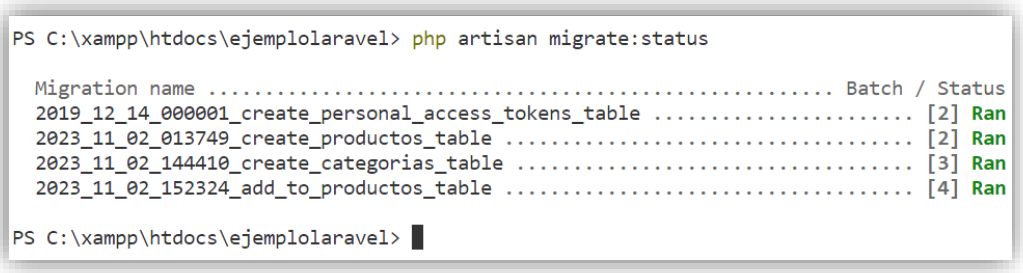
#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra	Acción
1	id	bigint(20)		UNSIGNED	No	Ninguna		AUTO_INCREMENT	Cambiar Eliminar Más
2	codigo	varchar(20)	utf8mb4_unicode_ci		No	Ninguna			Cambiar Eliminar Más
3	descripcion	text	utf8mb4_unicode_ci		Sí	NULL			Cambiar Eliminar Más
4	precio	decimal(10,2)			No	Ninguna			Cambiar Eliminar Más
5	existencia	int(11)			No	Ninguna			Cambiar Eliminar Más
6	categoria_id	bigint(20)		UNSIGNED	No	Ninguna			Cambiar Eliminar Más
7	activo	tinyint(4)			No	Ninguna			Cambiar Eliminar Más
8	created_at	timestamp			Sí	NULL			Cambiar Eliminar Más
9	updated_at	timestamp			Sí	NULL			Cambiar Eliminar Más

También podemos ver los índices, con todo y sus nombres:

Acción	Nombre de la clave	Tipo	Único	Empaquetado	Columna	Cardinalidad	Cotejamiento	Nulo	Comentario
Editar Renombrar Eliminar	PRIMARY	BTREE	Sí	No	id	0	A	No	
Editar Renombrar Eliminar	productos_codigo_unique	BTREE	Sí	No	codigo	0	A	No	
Editar Renombrar Eliminar	productos_categoria_id_foreign	BTREE	No	No	categoria_id	0	A	No	

Puede cambiar el nombre de la clave en caso no esté correcto.

Ahora podemos ver el estado:



Vemos que tenemos varias migraciones y diferentes posiciones de cómo se están ejecutando.

Ahora vamos a revertir con rollback el ultimo estado con:

Php artisan migrate:rollback –step=1

Vemos que se ha eliminado los cambios.

	#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra	Acción
☐	1	**id**	bigint(20)		UNSIGNED	No	*Ninguna*		AUTO_INCREMENT	Cambiar Eliminar Más
☐	2	**codigo**	varchar(20)	utf8mb4_unicode_ci		No	*Ninguna*			Cambiar Eliminar Más
☐	3	**descripcion**	text	utf8mb4_unicode_ci		Sí	*NULL*			Cambiar Eliminar Más
☐	4	**precio**	decimal(10,2)			No	*Ninguna*			Cambiar Eliminar Más
☐	5	**existencia**	int(11)			No	*Ninguna*			Cambiar Eliminar Más
☐	6	**activo**	tinyint(4)			No	*Ninguna*			Cambiar Eliminar Más
☐	7	**created_at**	timestamp			Sí	*NULL*			Cambiar Eliminar Más
☐	8	**updated_at**	timestamp			Sí	*NULL*			Cambiar Eliminar Más

CAPÍTULO 10

SEEDERS

Hasta ahora aprendimos a crear nuestras tablas, alterarlas, a crear toda la estructura, desde el Framework Laravel, pero que sucede cuando reseteamos o hacemos **rollback** a nuestras tablas.

En caso de que ya hayamos agregado información, esa información, esta se va a perder, porque lo estamos eliminando y volviendo a crear.

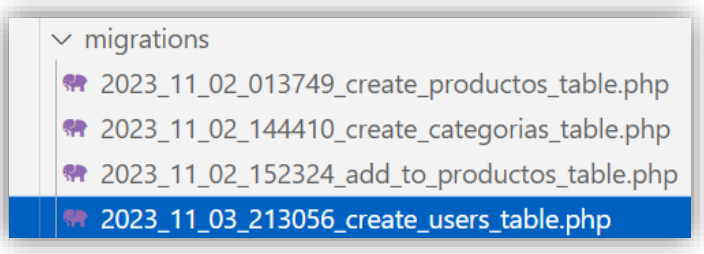
Para esto vamos a trabajar con los Seeders, en donde puede obtener mayor información es en la documentación de Laravel en **Base de datos: Siembra**

Esto nos funciona para insertar información, de preferencia se trabaja cuando estamos en modo **desarrollo**, pero también se puede trabajar en modo **producción**.

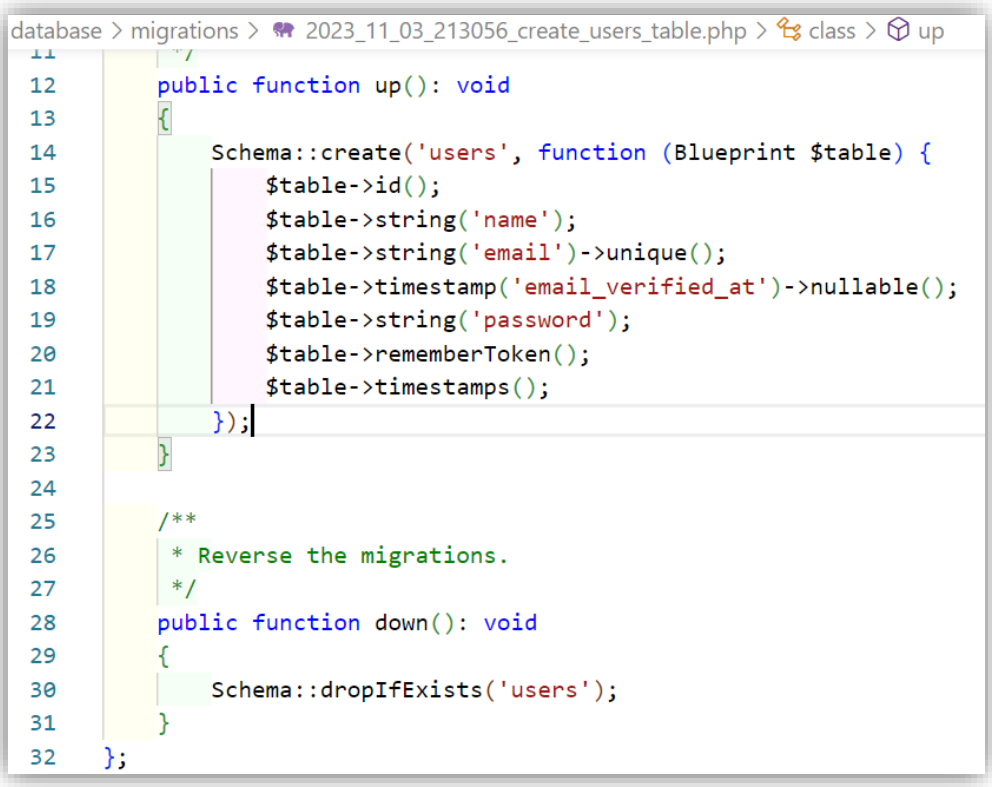
Por **ejemplo**: puedes insertar un usuario que sea el administrador para que pueda ingresar al sistema. También puede agregar información de la tabla productos, lo podemos generar de una clase para después solo ejecutarla y no estar insertando directamente a la base de datos. (como se hace tradicionalmente con insert)

Por ahora vamos a usar la migración para la tabla usuarios (esta es una migración predefinida que ya trae Laravel), si no lo tienes, o lo has borrado lo podemos crear nuevamente. Para ejemplificar, como hemos eliminado el archivo **create_user_table.php** lo que vamos a hacer primeramente es ir a nuestro PhpMyAdmin y vamos a eliminar la tabla **users**, luego de eso en el terminal ejecutar:

php artisan make:migration create_users_table



Luego, en el archivo nuevo de **user** debemos de agregar los campos según estaba predefinido.



Luego de copiar y grabar el código anterior, vamos a ejecutar desde el terminal con: **php artisan migrate**

```
PS C:\xampp\htdocs\ejemplolaravel> php artisan migrate

  INFO  Running migrations.

  2023_11_03_213056_create_users_table ..................................................... 73ms DONE

PS C:\xampp\htdocs\ejemplolaravel>
```

Ahora ya tenemos la tabla **users**.

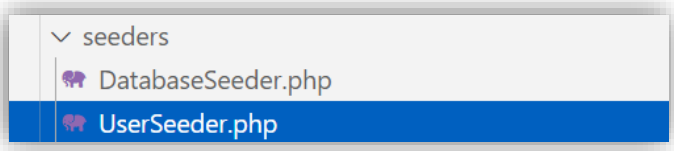
10.1 SIEMBRA DE LA INFORMACIÓN

Vamos a iniciar con la siembre de alguna información, en este caso vamos a crear un registro para un usuario.

En el terminal vamos a colocar **php artisan make:seeder UserSeeder**

```
PS C:\xampp\htdocs\ejemplolaravel> php artisan make:seeder UserSeeder

  INFO  Seeder [C:\xampp\htdocs\ejemplolaravel\database/seeders/UserSeeder.php] created successfully.

PS C:\xampp\htdocs\ejemplolaravel>
```

Luego presionamos Enter, y vamos a ver que nos genera un archivo en **database/Seeders**



Primero abramos el archivo **DatabaseSeeder.php** este es el archivo principal en donde vamos a poder llamar los Seeders, pero esto lo veremos más adelante.

Ahora abramos el archivo **UserSeeder.php** que es el archivo creado por nosotros, esta es la estructura donde contiene los namespace, los use que va ha trabajar, el nombre de la clase UserSeeder, en este archivo solo habrá una función que se va llamar **run** y dentro de esta función vamos a trabajar la siembra de los datos.

Ejemplo: insertar un registro a la tabla users

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra	Acción
1	**id**	bigint(20)		UNSIGNED	No	*Ninguna*		AUTO_INCREMENT	Cambiar Eliminar Más
2	**name**	varchar(255)	utf8mb4_unicode_ci		No	*Ninguna*			Cambiar Eliminar Más
3	**email**	varchar(255)	utf8mb4_unicode_ci		No	*Ninguna*			Cambiar Eliminar Más
4	**email_verified_at**	timestamp			Sí	*NULL*			Cambiar Eliminar Más
5	**password**	varchar(255)	utf8mb4_unicode_ci		No	*Ninguna*			Cambiar Eliminar Más
6	**remember_token**	varchar(100)	utf8mb4_unicode_ci		Sí	*NULL*			Cambiar Eliminar Más
7	**created_at**	timestamp			Sí	*NULL*			Cambiar Eliminar Más
8	**updated_at**	timestamp			Sí	*NULL*			Cambiar Eliminar Más

Vamos a utilizar algunos comandos que nos van a ayudar a agregar estos datos, vamos a llamarlo con DB (esta clase viene de use Illuminate\Support\Facades\DB), tomar en cuenta que los registros a insertar será de acuerdo a los campos que tiene la tabla **users.**

```
use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\DB
```

Es importante que éste use este agregado en la línea de comandos, sino, no vamos a poder trabajar correctamente, después colocamos :: luego **table('nombre de la tabla')** es donde vamos a insertar la información.

Luego usamos la función **insert ([** Aquí agregamos los registros **])**

Para ingresar los registros, primero debemos de poner los nombres de los campos (entre comillas) igual a **Str::**

Str es una clase que previamente la debemos de llamar con la siguiente instrucción:

Use Illuminate\Support\Str

```php
namespace Database\Seeders;

use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Str;
```

Esta clase **Str** nos va servir para generar cadenas aleatorias, así es que llamamos a **random(**tamaño de la cadena**)**

Lo que va hacer es generar una cadena random de longitud 8, para mi nombre.

Lo separamos con una coma para seguir agregando los demás registros.

Para el email vamos a concatenarlo con **@gmail** para que sea el formato de un correo electrónico.

Para el password vamos a utilizar la clase **Hash** (es importante que esté agregado el use Illuminate\Support\Facades\Hash; aunque al escribir la clase, se va generar automáticamente en la línea de comandos, pero si no lo hace, debe de escribir el use)

Hash::make('password') *esto ya nos va generar password cifrado*

Lo vamos a dejar hasta acá, finalizamos con un ;

Listo, ya tenemos nuestra primera siembra, la cual va ser el ingreso de un registro a nuestra base de datos.

Los campos nulos no son necesarios colocarlos. Y el **id** lo coloca de forma automática.

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra	Acción
1	**id**	bigint(20)		UNSIGNED	No	*Ninguna*		AUTO_INCREMENT	Cambiar Eliminar Más
2	**name**	varchar(255)	utf8mb4_unicode_ci		No	*Ninguna*			Cambiar Eliminar Más
3	**email**	varchar(255)	utf8mb4_unicode_ci		No	*Ninguna*			Cambiar Eliminar Más
4	**email_verified_at**	timestamp			Sí	*NULL*			Cambiar Eliminar Más
5	**password**	varchar(255)	utf8mb4_unicode_ci		No	*Ninguna*			Cambiar Eliminar Más
6	**remember_token**	varchar(100)	utf8mb4_unicode_ci		Sí	*NULL*			Cambiar Eliminar Más
7	**created_at**	timestamp			Sí	*NULL*			Cambiar Eliminar Más
8	**updated_at**	timestamp			Sí	*NULL*			Cambiar Eliminar Más

database > seeders > UserSeeder.php > ...

```php
namespace Database\Seeders;

use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Str;

class UserSeeder extends Seeder
{
    public function run(): void
    {
    DB::table('users')->insert([
    'name'=>Str::random(8),
    'email'=>Str::random(8).'@gmail',
    'password'=>Hash::make('password')
    ]);
    }
}
```

Vamos a ejecutarlo para insertar la información, para ello vamos a la terminal, esta ejecución podemos hacerlo de dos formas:

De forma general:

De forma general sería agregarle en el archivo **DatabaseSeeder.php** necesitamos registrarlo acá, porque si no lo registramos, cuando lo ejecutemos con

php artisan db:seed presionamos Enter

```
PS C:\xampp\htdocs\ejemplolaravel> php artisan db:seed

  INFO  Seeding database.

PS C:\xampp\htdocs\ejemplolaravel>
```

Cuando hayamos ejecutado los Seeders veremos que en la tabla usuarios no nos insertado ninguna información, porque **php artisan db:seed** es una ejecución general, y esta ejecución general va hacer todo lo que contenga la función **run** del archivo DatabaseSeeder.php

Necesitaríamos registrarlo, pero antes de registrarlo también podemos ejecutarlo de forma individual mediante el **UserSeeder**

La ejecutamos de la siguiente manera:

Php artisan db:seed --class=UserSeeder

Presionamos Enter, y con esto ya estamos ejecutando nuestro Seeder.

```
PS C:\xampp\htdocs\ejemplolaravel> php artisan db:seed --class=UserSeeder

  INFO  Seeding database.

PS C:\xampp\htdocs\ejemplolaravel>
```

Veamos los registros de nuestra tabla user

	id	name	email	email_verified_at	password	remember_token	created_at	updated_at
Borrar	1	iTxBJykv	2uJEqolP@gmail	NULL	$2y$10$n6nD26Dfs/GWEYaG490wXufcmMQkzf0dbcKMkdewOji...	NULL	NULL	NULL

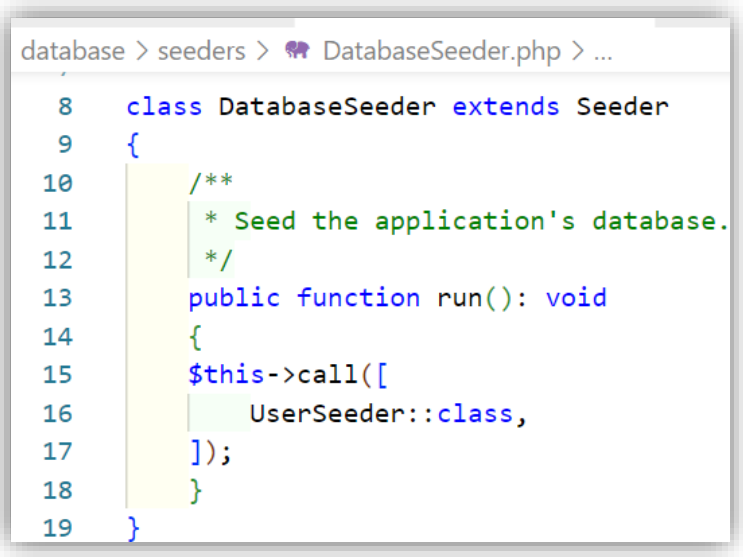
Y vemos que ya nos ha generado un registro con las características que le dimos, pero recordemos que esta ejecución fue de forma individual.

Ahora que pasaría si tenemos otras siembras como para categorías, productos o para otras tablas. En ese caso podemos ejecutarlas todas con un solo comando, entonces para hacer eso necesitamos registrar nuestros Seeders en la función **run** del archivo DatabaseSeeder.php

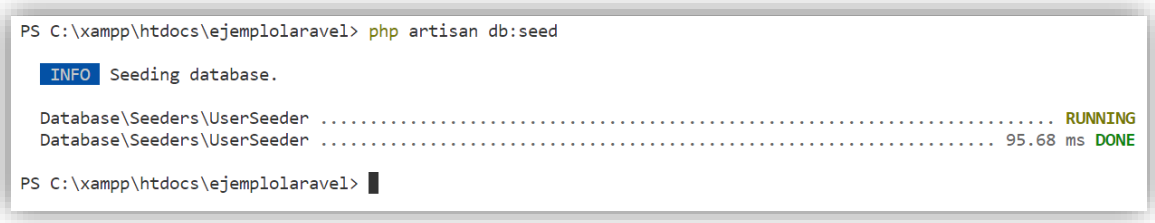
Colocamos:

$this->call([UserSeeder::class])

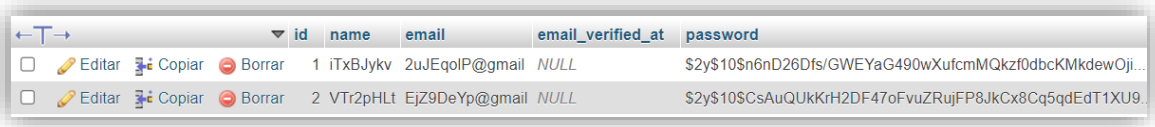
Puedes agregar mas siembras, pero debe separarlo mediante comas.



Grabamos y ahora vamos a ejecutar del terminal, en donde nos va indicando que ya se está ejecutando el UserSeeder.



Vamos a fijarnos a la tabla y vemos que tenemos dos registros.

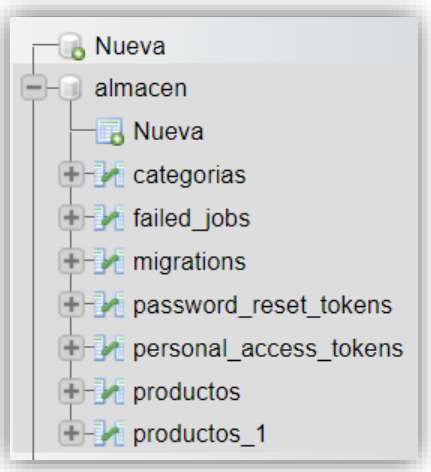


Esto porque hicimos doble inserción, uno de manera particular y el otro de manera general.

Ahora vamos a hacer un ejemplo sobre **migraciones,** por ejemplo, hagamos un rollback con artisan indicándole que solo se regrese un paso.

```
PS C:\xampp\htdocs\ejemplolaravel> php artisan migrate:rollback --step=1

  INFO  Rolling back migrations.

  2023_11_03_213056_create_users_table ..................................

PS C:\xampp\htdocs\ejemplolaravel>
```

Lo que está haciendo es eliminar mi tabla de usuarios, y si nos fijamos, vemos que ya no está la tabla **users**



Ahora nuevamente ejecuto migrate para crear la tabla users

```
PS C:\xampp\htdocs\ejemplolaravel> php artisan migrate

  INFO  Running migrations.

  2023_11_03_213056_create_users_table ..............................

PS C:\xampp\htdocs\ejemplolaravel>
```

Vemos que nuevamente se ha creado la tabla **users** pero, esta tabla se encuentra sin información. Entonces lo que haremos ahora será ejecutar la siembra de datos del modo general porque ya lo tenemos implementado.

```
PS C:\xampp\htdocs\ejemplolaravel> php artisan db:seed

  INFO  Seeding database.

  Database\Seeders\UserSeeder ..................................
  Database\Seeders\UserSeeder ..................................

PS C:\xampp\htdocs\ejemplolaravel>
```

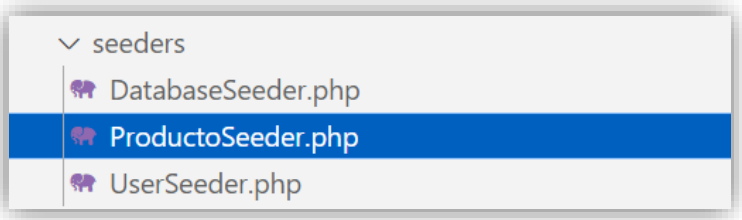
Ya con esto le estamos insertando la información en la tabla **users**

Vamos a hacer otro ejemplo, pero ahora con la tabla **productos**

Solución

Primero vamos a crear el archivo para la siembra:

```
PS C:\xampp\htdocs\ejemplolaravel> php artisan make:seeder ProductoSeeder

  INFO  Seeder [C:\xampp\htdocs\ejemplolaravel\database/seeders/ProductoSeeder.php] created successfully.

PS C:\xampp\htdocs\ejemplolaravel>
```



Antes de trabajar lo de producto vamos a insertar un registro en la tabla categorías.

INSERT INTO `categorias` (`id`, `nombre`, `descripcion`, `created_at`, `updated_at`) VALUES (NULL, **'zapatos'**, NULL, NULL, NULL);

Puede agregarle el registro directamente del PhpMyAdmin, llenar únicamente el campo nombre con zapatos.

Abramos el archivo **ProductoSeeder.php** e insertamos: (No olvide los **use**), colocar únicamente para que se inserte en los campos que pide datos, los demás pueden

estar como Null o vacíos, también puedes colocarle el **create_at** con la función **now( )** para que nos de la fecha y la hora del momento en el que ejecutamos esta instrucción.

database > seeders > ProductoSeeder.php > ...

```php
namespace Database\Seeders;

use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\DB;

class ProductoSeeder extends Seeder
{
    public function run(): void
    {
        DB::table('productos')->insert([
        'codigo'=>'1234567890',
        'precio'=>10.00,
        'existencia'=>50,
        'categoria_id'=>1,
        'activo'=>1,
        'created_at'=>now()
        ]);
    }
}
```

El siguiente paso sería ejecutar este Seeder. (podemos hacerlo de forma individual o de forma general para la cual debemos registrarlo)

database > seeders > DatabaseSeeder.php > ...

```php
class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     */
    public function run(): void
    {
    $this->call([
        UserSeeder::class,
        ProductoSeeder::class
    ]);
    }
}
```

Ahora ejecutamos en la terminal.

Lo que podemos ver es que se ha insertado registro para **users** y para **productos**.

Puede revisar en la tabla y comprobar las inserciones, puede ver que incluso se ha insertado en categorías.

Esto lo trabajamos para datos predefinidos o para datos de prueba.

CAPÍTULO 11

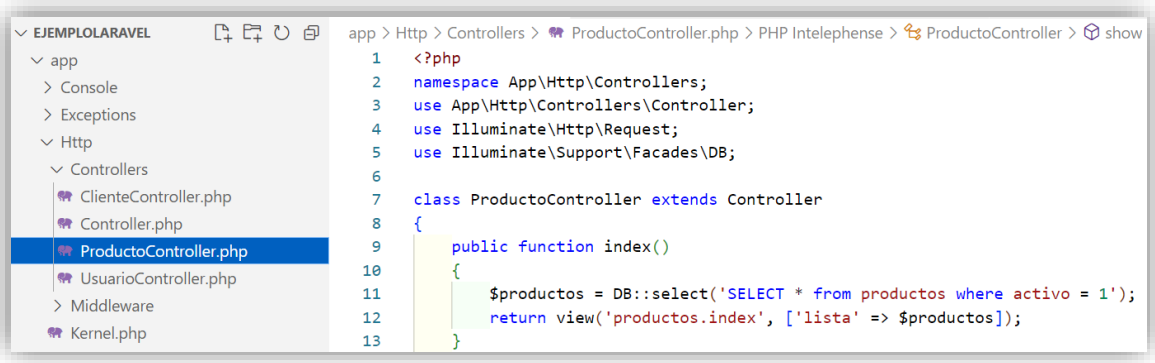
ORM Eloquent

Laravel nos proporciona un ORM (Mapeo Objeto - Relacional) que se llama Eloquent, donde un ORM mapea la estructura de una tabla o de una base de datos y lo pasa a objetos o con clases, para que así lo podamos trabajar directamente en la programación, sin tener que estar haciendo Query o insert o algún otro comando con la base de datos.

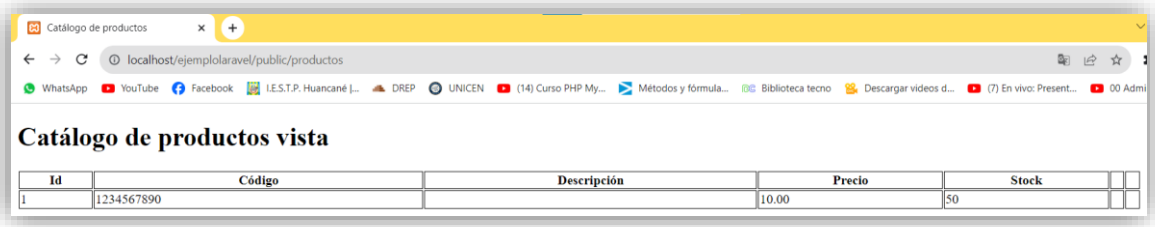
Cuando trabajamos con Laravel tenemos en la documentación Eloquent, y vamos a encontrar gran parte de lo que vamos a hacer y otros comandos adicionales.

Abramos nuestro proyecto y veamos la **forma tradicional**:

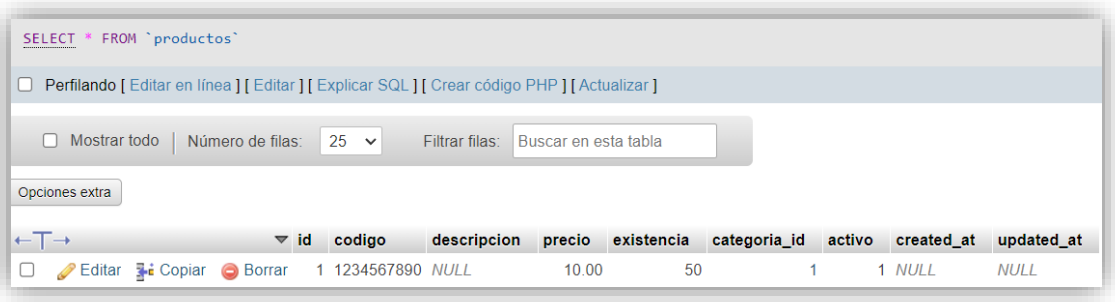
Recordemos cuando trabajamos con los controladores, vamos a *app/http/Controllers* en ProductoController.php hicimos un *select* de la tabla productos y lo mostramos directamente a una vista.



Y nos mostraba lo siguiente:



Y en nuestra base de datos vemos nuestra tabla productos, y hacíamos la consulta directamente desde la función *index* de *ProductoController.php* con la instrucción select.

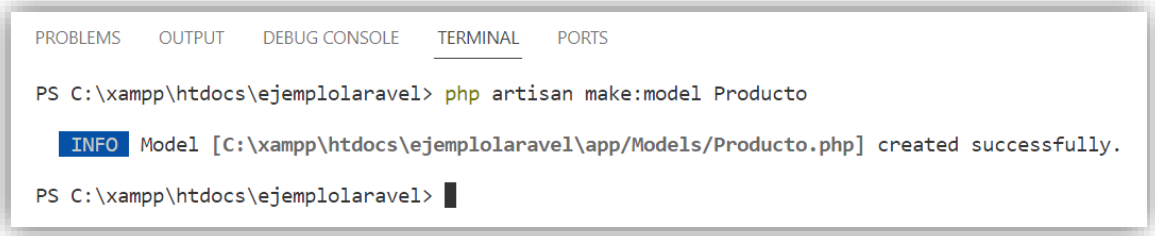


La consulta anterior mostrada es de la forma tradicional, la cual se puede trabajar de esa manera, pero por ahora lo vamos a trabajar con **Eloquent,** nos fijaremos que no lo vamos a trabajar directamente con un Query, pero si vamos a utilizar los comandos como select, where, order by, etc. Pero de una forma diferente, vale decir con el uso de objetos, variables, de tal manera que será más estructurado para su entendimiento.

11.1 GENERACIÓN DE MODELOS

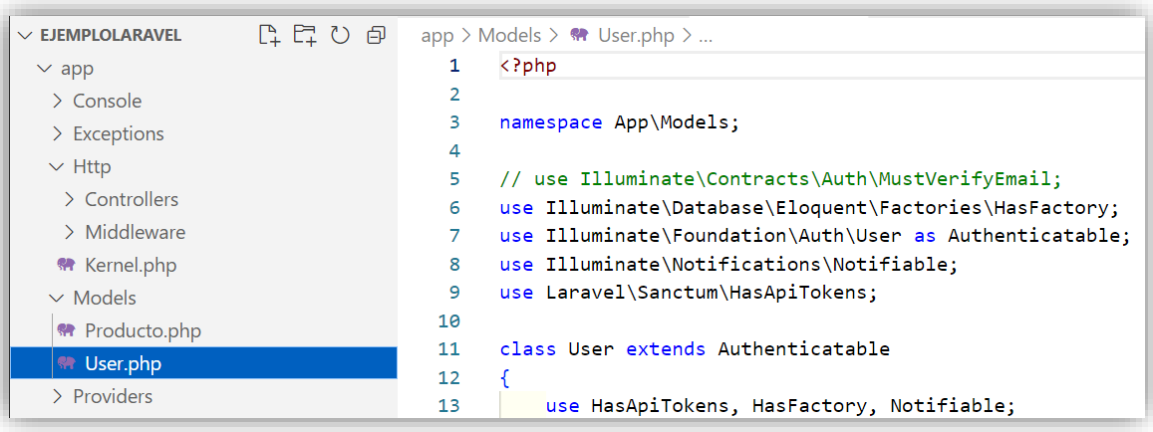
Para esto necesitamos generar unos modelos (ya trabajamos con las vistas, con los controladores, con las rutas, entre otras, pero ahora vamos a pasar al modelo que es aquí donde se comunica directamente con la base de datos para hacer las transacciones)

Vamos a abrir una terminal y vamos a escribir *php artisan make:model nombre_modelo*



Ya se ha creado el modelo, y vamos a encontrarlo en la dirección *app/models/Producto.php*

ya teníamos un *user* que fue creado por defecto en donde podemos ver la composición de este modelo, la cual la dejaremos tal y cual esté.

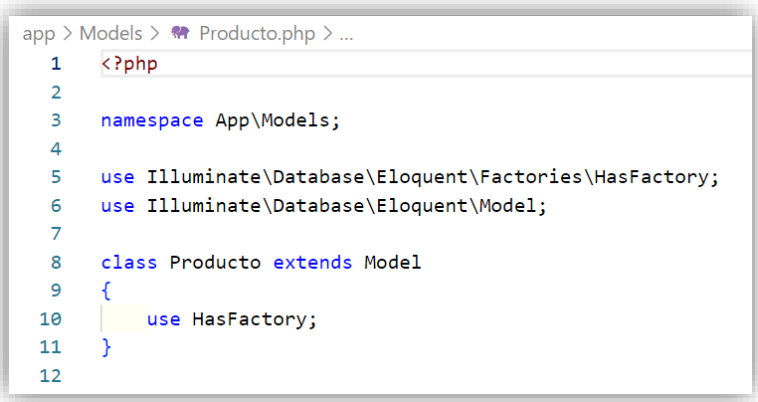


Vamos a trabajar con Producto.php en donde podemos ver un namespace la clase models, va trabajar con

```
use Illuminate\Database\Eloquent\Factories\HasFactory;

use Illuminate\Database\Eloquent\Model;
```

Esto lo vamos a usar para poder extender la clase *modelo* a nuestra clase *Producto*

```
class Producto extends Model
```



Veamos las características que vamos a necesitar para crear nuestro modelo.

Agregamos **Producto** porque vamos a hacer referencia a la tabla de **productos**

(Cuando trabajamos con los modelos, regularmente necesitamos colocar el nombre en singular Class Producto, el ORM que es Eloquent lo que va hacer es agregarle

una s al final para que detecte la tabla que se llama **productos** y es a esa a la que se va conectar**)**

Para este ejemplo nos vamos a conectar a la tabla productos.

Pero que pasa si queremos especificarle una tabla que tenga un nombre diferente, entonces necesitaremos colocar la propiedad **table** y ésta va ser de tipo **Protected,** va tener una variable $table y colocamos el nombre de la tabla, en nuestro caso se llama productos (aquí tal vez es algo redundante porque se va conectar a la tabla que ya tenemos predefinido por el nombre de la clase, pero en caso de que lo necesitemos, lo asignaremos de esa forma).

app > Models > Producto.php > ...

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Producto extends Model
{
    use HasFactory;
    protected $table = "productos";
}
```

Si tenemos el nombre de la tabla como un conjunto de dos nombres o más, en este caso debemos abrir la terminal y escribimos *php artisan make:model DetalleProducto (colocar en mayúscula)*

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\xampp\htdocs\ejemplolaravel> php artisan make:model DetalleProducto

  INFO  Model [C:\xampp\htdocs\ejemplolaravel\app/Models/DetalleProducto.php] created successfully.

PS C:\xampp\htdocs\ejemplolaravel>
```

Y aquí va ser un diferente sobre la llamada a la tabla, lo colocaría como *detalle_productos* los guiones son generados al detectar una mayúscula y al final le agrega una s.

Puedes agregarle de esa manera o de la manera anterior indicando directamente la tabla. Pero, se recomienda trabajarlo así:

```
8   class Producto extends Model
9   {
10      use HasFactory;
11      protected $table = "productos";
12  }
```

Ya que tenemos la tabla productos, también le podemos indicar un primary key, el ORM lo que hace es tomar de forma predefinida que el campo se va llamar id, que es la primary key auto incremental. (Nosotros la tenemos así, pero en caso no estuviese así, la podemos definir con un Protected que se va llamar $primaryKey = nombre_del_id)

Por ejemplo:

```
class Producto extends Model
{
    use HasFactory;
    protected $table = "productos";
    protected $primaryKey = "producto_id";
}
```

En caso de que cumpla con las características del ORM que la tabla se llame productos y que tu primary key se llame id, entonces no es necesario colocar nada de los Protected,

```
class Producto extends Model
{
    use HasFactory;
    protected $table = "productos";
    protected $primaryKey = "producto_id";
}
```

Se pone solo en caso de que no corresponda.

En ciertas ocasiones también lo usamos cuando el primary key no sea entero o no sea auto increment, entonces podemos desactivar esa regla.

Seria de la siguiente manera:

```php
class Producto extends Model
{
    use HasFactory;
    protected $table = "productos";
    protected $primaryKey = "producto_id";
    public $incrementing = false; //para que no sea autoincremental
    protected $keyType = 'string';  //en caso el primary key sea de otro tipo
}
```

Podemos encontrar más comandos en la documentación de laravel.

Recuerden que cuando creamos nuestra migración se ha agregado dos columnas: *created_at* y *updated_at* son las marcas de tiempo de cuando se crea y cuando se modifica el registro. También se puede hacer una modificación sobre estos campos, el ORM directamente va colocar la fecha de creación y de modificación.

Que pasa si colocamos estos campos personalizados, es decir, con otros nombres, la respuesta es que también podemos agregarlo, es decir, si los campos están con otro nombre que nos predefine laravel. Seria:

```php
class Producto extends Model
{
    use HasFactory;
    protected $table = "productos";
    protected $primaryKey = "producto_id";
    public $incrementing = false; //para que no sea autoincremental
    protected $keyType = 'string';  //en caso el primary key sea de otro tipo

    //public $timestamps = false;

    const CREATED_AT = 'fecha_alta';
    const UPDATED_AT = 'fecha_modifica';
}
```

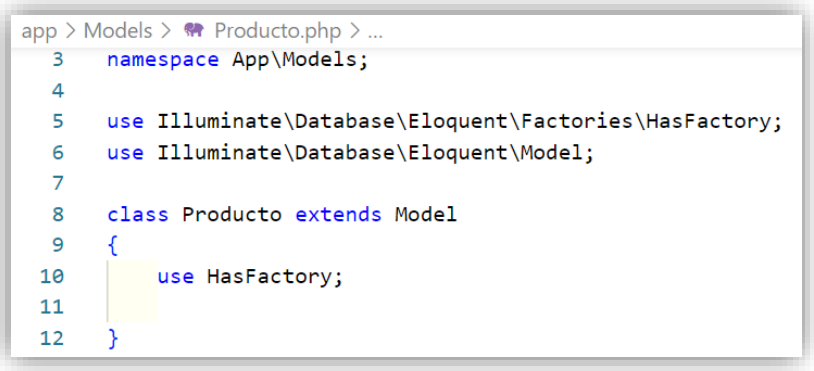
Podemos usar:

```php
public $timestamps = false; O
```

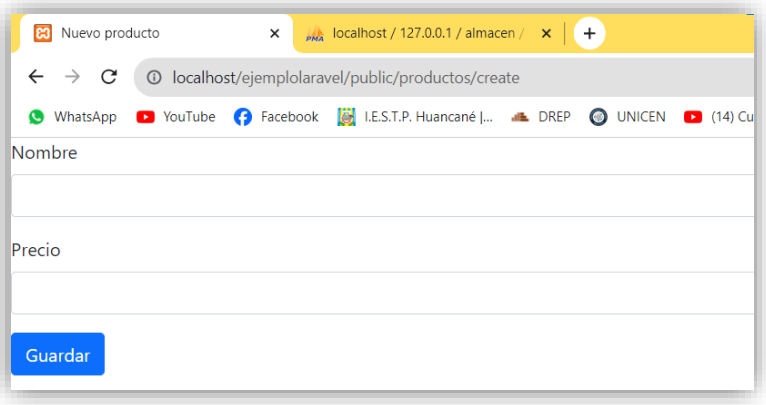
```
const CREATED_AT = 'fecha_alta';

const UPDATED_AT = 'fecha_modifica';

Pero no ambas soluciones
```

11.2 TRANSACCIONES A NUESTRAS BASES DE DATOS

Vamos a iniciar de la manera predefinida. (borremos o pongamos como comentario lo avanzado anteriormente, pues solo fue para explicar)



Recordemos que ya creamos un semi CRUD donde mostramos el catálogo de los productos, en la ruta http://localhost/ejemplolaravel/public/productos/create tenemos un formulario sencillo para agregar un producto.



Entonces, vamos a utilizar estas vistas para trabajar con ellas, entonces vamos al Controller de productos, y vamos a hacer algunos cambios, primero nos vamos a la función **store** que es el que guarda nuestro formulario, en este archivo vemos que solo recibimos los parámetros que enviamos por el método post.

```
app > Http > Controllers > ProductoController.php > PHP Intelephense > ProductoController > show
21      }
22      public function store(Request $request)
23      {
24          echo "<pre>";
25          echo $request->input('nombre');
26          echo $request->input('precio');
27          echo "</pre>";
28      }
```

Ahora ya teniendo estos parámetros, los vamos a insertar nuestra base de datos utilizando nuestro modelo, para esto vamos a crear una instancia y la vamos a llamar $producto = new de la clase Producto (visual studio nos indica que va hacer referencia a ese modelo)

```
public function store(Request $request)
{
    $producto = new produ
    echo "<pre>";
    echo $request->input(
    echo $request->input(
    echo "</pre>";
}
```

```
Producto                                    use App\Models\Producto
ProductoSeeder
ProductoController
DetalleProducto
RunningLaravelDuskInProductionProvider
```

```
public function store(Request $request)
{
    $producto = new Producto();
    echo "<pre>";
    echo $request->input('nombre');
    echo $request->input('precio');
    echo "</pre>";
}
```

Vamos a la parte de arriba y vemos que se ha colocado el use del modelo para hacer referencia al modelo que hemos creado, en caso no se coloque tendremos problemas.

app > Http > Controllers > ProductoController.php > ...

```php
<?php
namespace App\Http\Controllers;
use App\Http\Controllers\Controller;
use App\Models\Producto;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\DB;
```

Ahora con esta instancia que se llama producto, vamos a poder acceder a los campos de nuestra tabla y esto para poder agregar un registro, es decir, vamos a mapear nuestra tabla que se llama productos, que tiene la siguiente estructura:

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra	Acción
1	id	bigint(20)		UNSIGNED	No	*Ninguna*		AUTO_INCREMENT	Cambiar Eliminar Más
2	codigo	varchar(20)	utf8mb4_unicode_ci		No	*Ninguna*			Cambiar Eliminar Más
3	descripcion	text	utf8mb4_unicode_ci		Sí	*NULL*			Cambiar Eliminar Más
4	precio	decimal(10,2)			No	*Ninguna*			Cambiar Eliminar Más
5	existencia	int(11)			No	*Ninguna*			Cambiar Eliminar Más
6	categoria_id	bigint(20)		UNSIGNED	No	*Ninguna*			Cambiar Eliminar Más
7	activo	tinyint(4)			No	*Ninguna*			Cambiar Eliminar Más
8	created_at	timestamp			Sí	*NULL*			Cambiar Eliminar Más
9	updated_at	timestamp			Sí	*NULL*			Cambiar Eliminar Más

Y vamos a poder trabajar cada uno de los campos como si fuera un objeto.

Por ejemplo, vamos a agregar el código 1234567890, recordemos que es único, no se debe repetir, luego nos vamos a la descripción, que también va a ser producto igual a descripción y ahí vamos a trabajar el request que traemos de nuestro formulario, que sería el campo nombre.

app > Http > Controllers > ProductoController.php > PHP Intelephense > ProductoController > store

```php
23    public function store(Request $request)
24    {
25        $producto = new Producto();
26        $producto -> codigo = '1234567890';
27        $producto -> descripcion = $request->input('nombre');
28        $producto -> precio = $request->input('precio');
29        $producto -> existencia = 0;
30        $producto -> categoria_id = 1;
31        $producto -> activo = 1;
32        $producto -> save(); // indica que vamos a guardar todo el registro
33        return ("Registro guardado");
34
35    }
```

Como ven no estamos colocando ningún insert, ni VALUES ni nada, únicamente estamos trabajando con objetos y asignándole los valores, al final agregamos un mensaje: return que puede ser registro guardado.

Pero antes debemos revisar si tenemos una categoría en la tabla categorías, y vemos que tenemos un registro que se llama zapatos. Vamos a cambiarlo por bebidas, desde PhpMyAdmin.

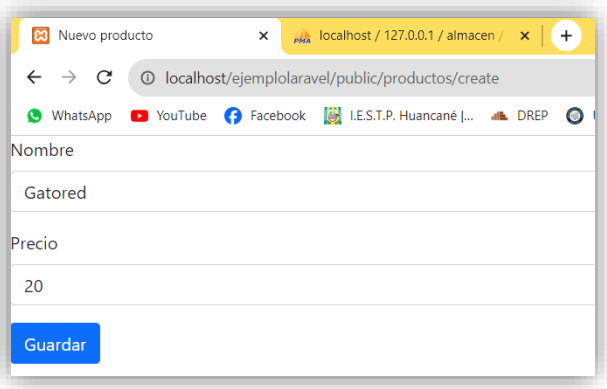
		id	nombre	descripcion	created_at	updated_at
☐	Editar Copiar Borrar	1	bebidas	NULL	NULL	NULL

Veamos que nuestras rutas ya fueron generadas anteriormente, en Routes web.php

routes > web.php > ...

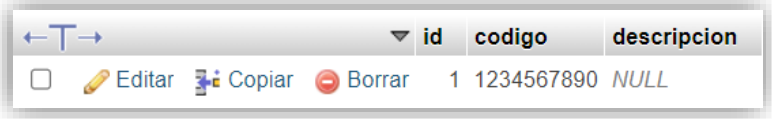
```php
23  route::get('productos', [ProductoController::class, 'index']);
24  route::get('productos/create', [ProductoController::class, 'create']);
25  route::get('productos/{producto}', [ProductoController::class, 'show']);
26  route::post('productos', [ProductoController::class, 'store']);
27  route::get('productos/{id}/edit', [ProductoController::class, 'edit']);
28  route::put('productos/{id}', [ProductoController::class, 'update']);
29  route::delete('productos/{id}', [ProductoController::class, 'destroy']);
30
31  route::get('usuarios', [UsuarioController::class, 'index']);
32  route::resource('clientes', ClienteController::class);
```

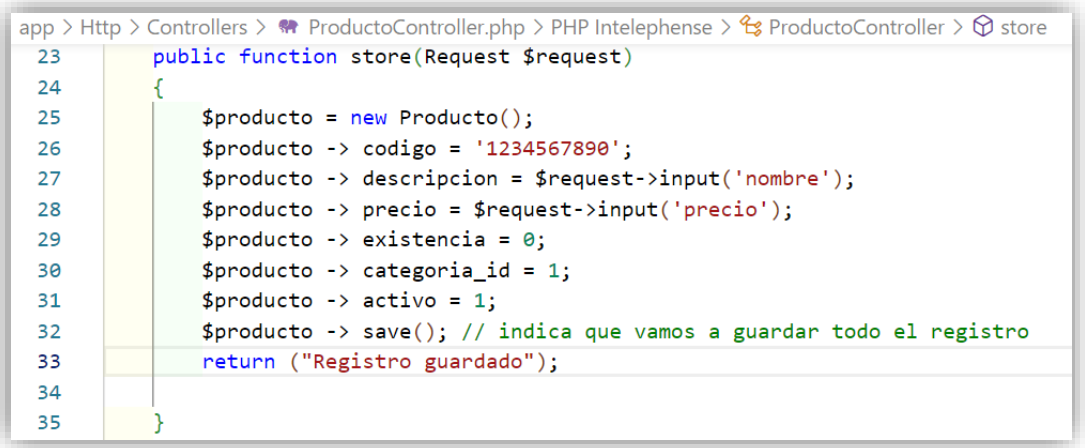
Así es que ingresemos: Gatored y precio 20, luego clic en guardar



Nos aparece un error, en donde nos dice que se esta duplicando una entrada

En este caso es el *código* que debió ser único, y como ya tenemos en productos ese código insertado en la tabla productos y en nuestro store estamos intentando insertar el mismo código:



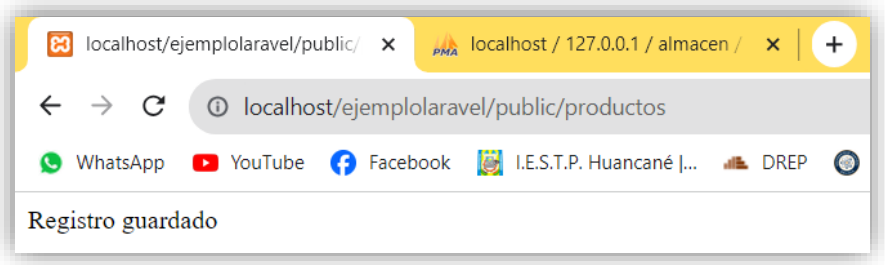


Podemos hacer el cambio en el código, en lugar de 1 pondremos 2, para que haya un cambio:

```
app > Http > Controllers > ProductoController.php > PHP Intelephense > ProductoController > store
22      }
23      public function store(Request $request)
24      {
25          $producto = new Producto();
26          $producto -> codigo = '2234567890';
27          $producto -> descripcion = $request->input('nombre');
28          $producto -> precio = $request->input('precio');
29          $producto -> existencia = 0;
30          $producto -> categoria_id = 1;
31          $producto -> activo = 1;
32          $producto -> save(); // indica que vamos a guardar todo el registro
33          return ("Registro guardado");
```

Regresamos al navegador y probamos:

Ahora vemos que ya nos muestra el mensaje de registro guardado.



Vamos a la base de datos y vemos que ya lo ha insertado.

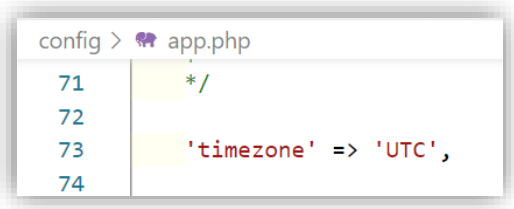
	id	codigo	descripcion	precio	existencia	categoria_id	activo	created_at	updated_at
Editar Copiar Borrar	1	1234567890	NULL	10.00	50	1	1	NULL	NULL
Editar Copiar Borrar	5	2234567890	Gatored	20.00	0	1	1	2023-11-16 03:51:12	2023-11-16 03:51:12

Aquí es muy importante revisar las fechas, cuando hemos creado y cuando lo hemos actualizado, en este momento nos genera la misma fecha para ambos campos, pero cuando hagamos un *update* en el campo *update_at,* la fecha de este campo se va actualizar.

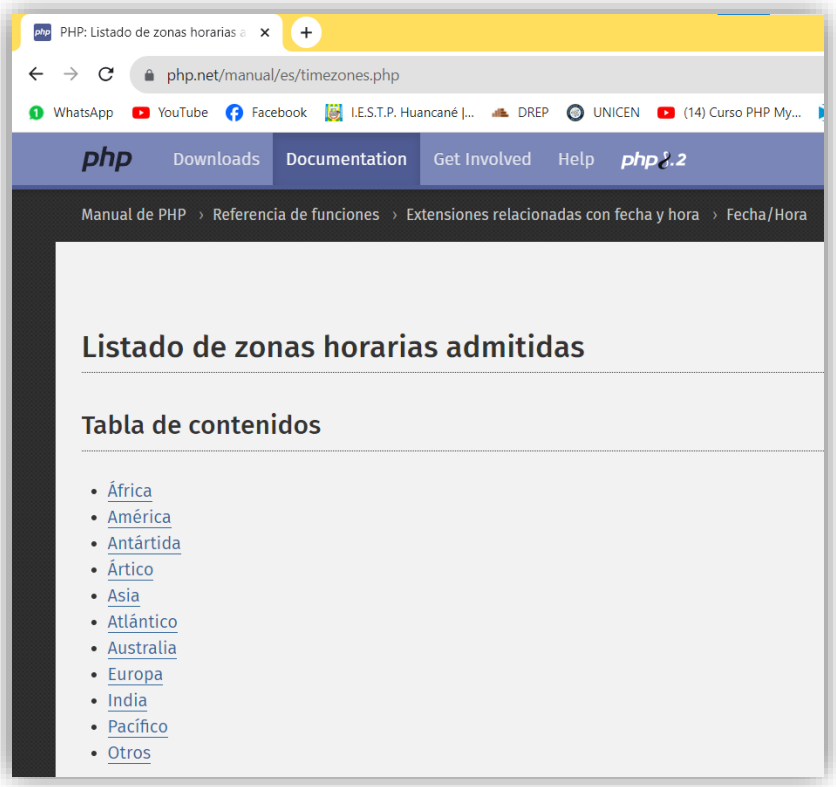
También es importante revisar la fecha y la hora que nos está mostrando, en ahí podemos ver que muestra el 16 de noviembre a las 3:51, pero ese no corresponde al horario en el que estoy trabando, ni mucho menos el día, todo este error es porque tiene otra zona horaria.

11.3 CONFIGURACIÓN DE LA ZONA HORARIA

Lo vamos a encontrar en **app/config/app.php** en ahí encontramos **timezone** y para saber que dato debemos de colocar.



Pero antes buscaremos en Google como: **timezone php** y nos va mostrar un listado de todas las zonas que podemos configurar.



Haces clic en nuestro continente **América,** luego buscamos nuestro país o ciudad, lo seleccionamos y lo cambiamos por lo anterior.

config > app.php

```
71      */
72
73      'timezone' => 'America/Bogota',
74
```

php Downloads Documentation Get Involved Help php8.2

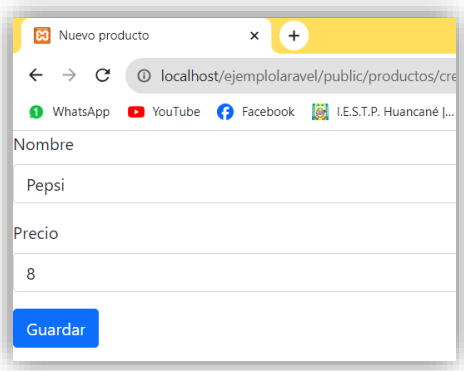
America/Managua	America/Manaus	America/Marigot	America/Martinique
America/Matamoros	America/Mazatlan	America/Menominee	America/Merida
America/Metlakatla	America/Mexico_City	America/Miquelon	America/Moncton
America/Monterrey	America/Montevideo	America/Montserrat	America/Nassau
America/New_York	America/Nipigon	America/Nome	America/Noronha
America/North_Dakota/Beulah	America/North_Dakota/Center	America/North_Dakota/New_Salem	America/Ojinaga
America/Panama	America/Pangnirtung	America/Paramaribo	America/Phoenix
America/Port-au-Prince	America/Port_of_Spain	America/Porto_Velho	America/Puerto_Rico
America/Punta_Arenas	America/Rainy_River	America/Rankin_Inlet	America/Recife
America/Regina	America/Resolute	America/Rio_Branco	America/Santarem
America/Santiago	America/Santo_Domingo	America/Sao_Paulo	America/Scoresbysund
America/Sitka	America/St_Barthelemy	America/St_Johns	America/St_Kitts
America/St_Lucia	America/St_Thomas	America/St_Vincent	America/Swift_Current
America/Tegucigalpa	America/Thule	America/Thunder_Bay	America/Tijuana
America/Toronto	America/Tortola	America/Vancouver	America/Whitehorse
America/Winnipeg	America/Yakutat	America/Yellowknife	

Ahora vamos al **código** de barras y le cambiamos un dato para que cuando insertemos no nos envíe error por duplicidad de datos.

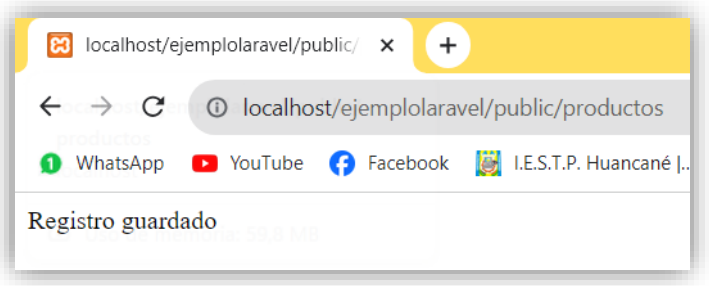
app > Http > Controllers > ProductoController.php > PHP Intelephense > ProductoController > store

```
19      public function create()
20      {
21          return view('productos.create');
22      }
23      public function store(Request $request)
24      {
25          $producto = new Producto();
26          $producto -> codigo = '3234567890';
27          $producto -> descripcion = $request->input('nombre');
28          $producto -> precio = $request->input('precio');
29          $producto -> existencia = 0;
30          $producto -> categoria_id = 1;
31          $producto -> activo = 1;
32          $producto -> save(); // indica que vamos a guardar todo el registro
33          return ("Registro guardado");
```

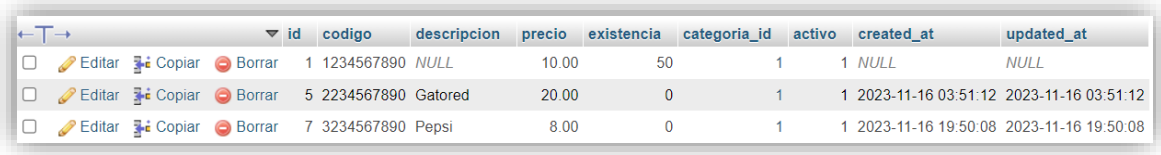
Regresamos al formulario e insertamos datos:



Clic en Guardar.



Ahora regresamos a la base de datos y vemos que la fecha y la hora ya son reales.



	id	codigo	descripcion	precio	existencia	categoria_id	activo	created_at	updated_at
Editar Copiar Borrar	1	1234567890	NULL	10.00	50	1	1	NULL	NULL
Editar Copiar Borrar	5	2234567890	Gatored	20.00	0	1	1	2023-11-16 03:51:12	2023-11-16 03:51:12
Editar Copiar Borrar	7	3234567890	Pepsi	8.00	0	1	1	2023-11-16 19:50:08	2023-11-16 19:50:08

Es así de sencillo insertar registros con Eloquent, no hemos con ningún Query, nada de insert ni de into, es decir, nada de MySql, todo lo estamos trabajando con objetos y valores que traemos de un formulario.

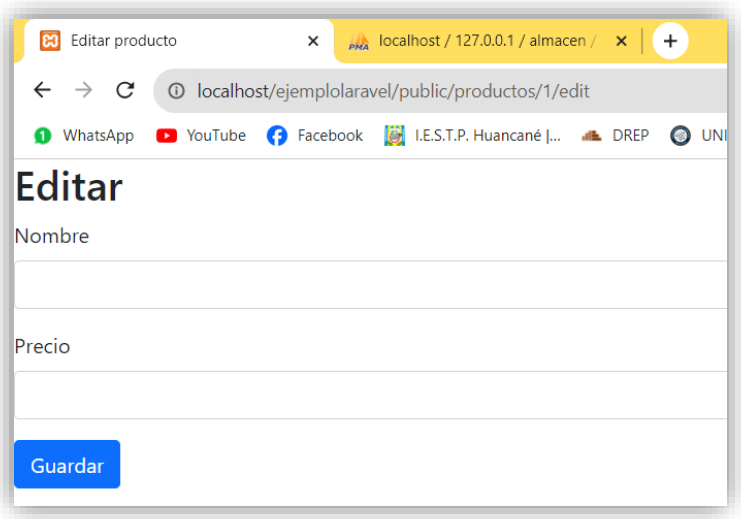
Vamos a trabajar ahora con el **update** para que vean que también es muy sencillo y no vamos a trabajar mucho con la sintaxis de Sql.

11.4 MODIFICAR UN REGISTRO

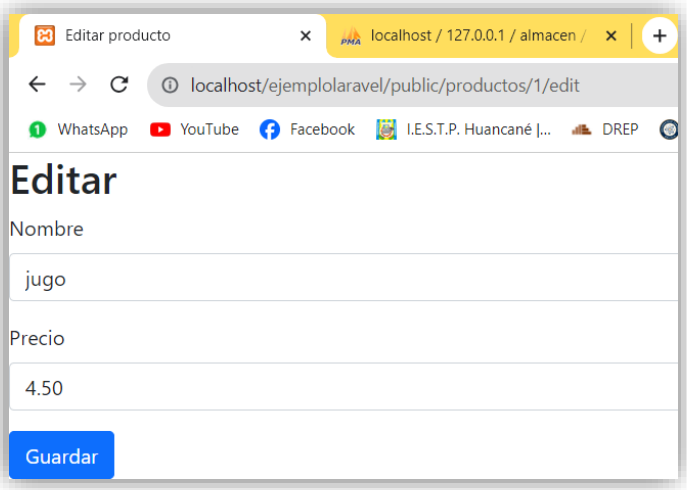
De la tabla vamos a modificar el registro 1 que en el campo **descripcion** esta como NULL, es decir no tiene nombre el producto.

Para esto necesitamos ir a la ruta:

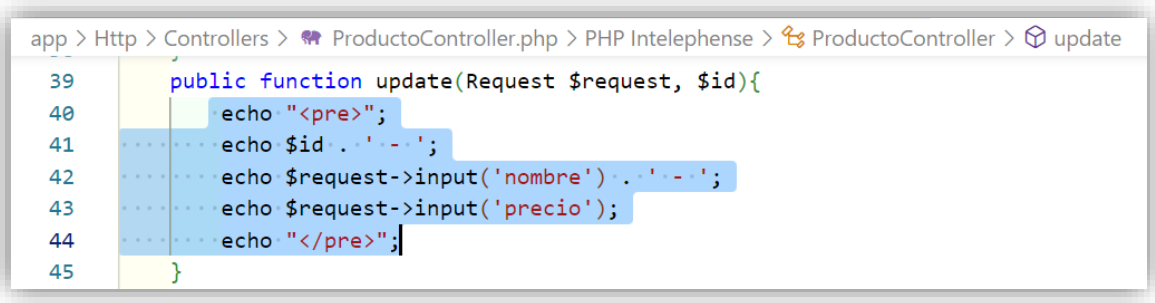
http://localhost/ejemplolaravel/public/productos/1/edit



Recordemos que ya lo trabajamos anteriormente con las rutas, las vistas y los controladores, y como no tiene nombre vamos a colocarlo como Jugo y en precio le ponemos 4.50

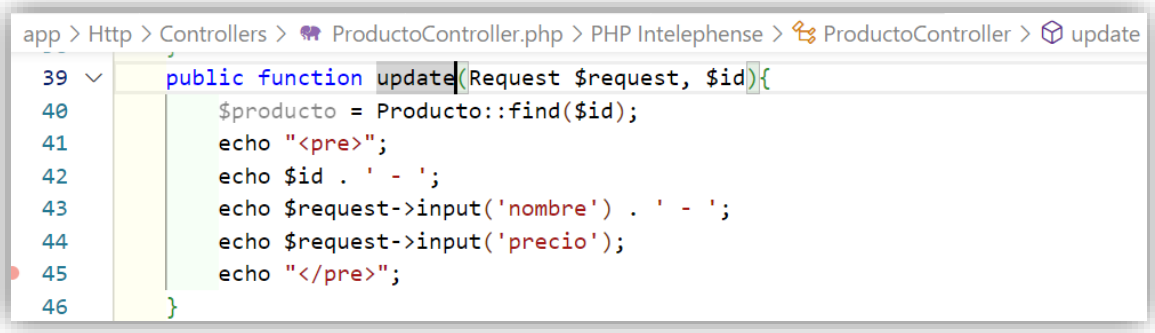


Pero nos hará falta hacer la modificación, y lo haremos en el update, aquí es donde recibimos los datos del formulario **edit** para poder actualizar este registro.



Y aquí vamos a crear otro objeto que se llame producto que va ser igual a la clase Producto, pero ahora lo traeremos de diferente manera, ya no con los paréntesis, sino con :: y vamos a traer una función que el modelo ya contiene y que se llama **find** el cual nos indica que vamos a buscar un registro, entre paréntesis le indicamos que el **id.**

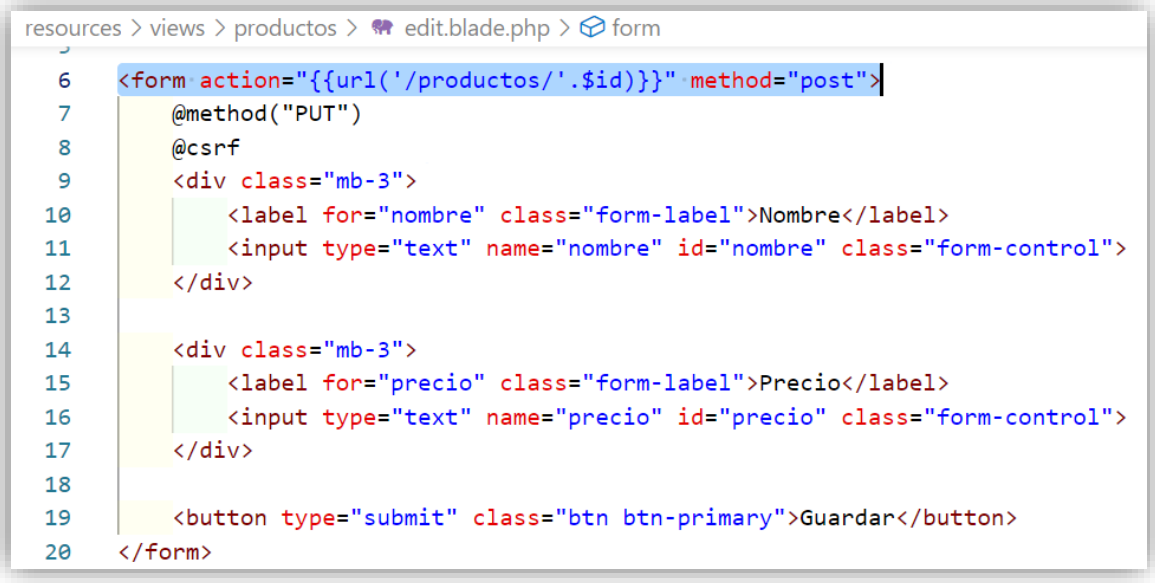
```
$producto = Producto::find($id);
```



Entonces la variable producto ya va traer el resultado de los datos.

Vamos a usar ese mismo ejemplo para el de **edit,** aquí vamos a cambiarlo, vamos a enviar otro dato que se llame producto y le enviamos la variable $producto.

Ahora vamos a ver la vista: **resource/views/productos/editblade.php**



Aquí estamos pasando directamente la variable $id, pero ahora tenemos otro que vamos a colocar en el value el nombre y con dos llaves, pero ahora se llama $producto y como es un objeto vamos a traer **descripción**.

También tenemos el precio y lo vamos a traer de la misma forma para mostrarlo en el formulario, pero ahora se llama precio.

resources > views > productos > edit.blade.php > form > div.mb-3 > input#nombre.form-control

```
<form action="{{url('/productos/'.$id)}}" method="post">
    @method("PUT")
    @csrf
    <div class="mb-3">
        <label for="nombre" class="form-label">Nombre</label>
        <input type="text" name="nombre" id="nombre" class="form-control" value="{{
            $producto->descripcion }}">
    </div>

    <div class="mb-3">
        <label for="precio" class="form-label">Precio</label>
        <input type="text" name="precio" id="precio" class="form-control" value="{{
            $producto->precio }}">
    </div>

    <button type="submit" class="btn btn-primary">Guardar</button>
</form>
```

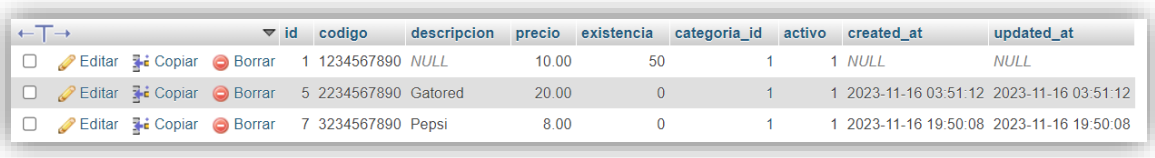
Faltaba modificar en el **update**, note que es muy similar a **store**

app > Http > Controllers > ProductoController.php > PHP Intelephense > ProductoController > update

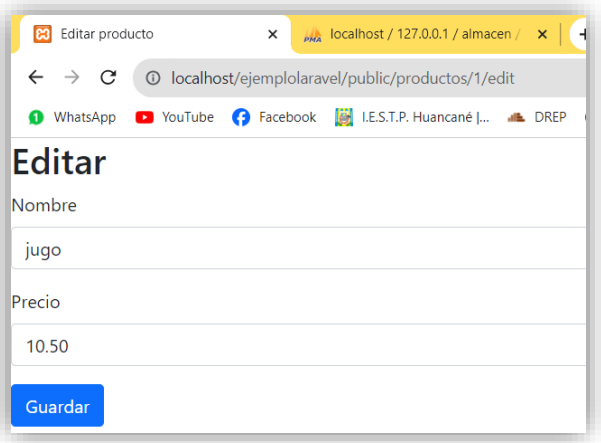
```
    public function store(Request $request)
    {
        $producto = new Producto();
        $producto -> codigo = '3234567890';
        $producto -> descripcion = $request->input('nombre');
        $producto -> precio = $request->input('precio');
        $producto -> existencia = 0;
        $producto -> categoria_id = 1;
        $producto -> activo = 1;
        $producto -> save(); // indica que vamos a guardar todo el registro
        return ("Registro guardado");

    }
    public function edit($id){
        $producto = Producto::find($id);
        return view('productos.edit', ['id'=>$id, 'producto' => $producto]);
    }
    public function update(Request $request, $id){
        $producto = Producto::find($id);
        $producto->descripcion = $request->input('nombre');
        $producto->precio = $request->input('precio');
        $producto->save();
        return "Registro modificado";
    }
    public function destroy($id){
        echo "Registro $id eliminado";
    }
}
```

No olvide guardar el archivo, luego regresamos a la vista, le vamos a colocar jugo y el precio a 10.50

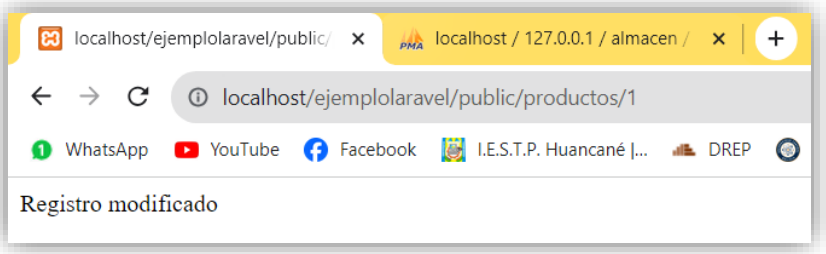
Pero antes revisemos la tabla, en el registro 1 vemos que si tiene precio, pero no tiene **descripción**.



	id	codigo	descripcion	precio	existencia	categoria_id	activo	created_at	updated_at
Editar Copiar Borrar	1	1234567890	NULL	10.00	50	1	1	NULL	NULL
Editar Copiar Borrar	5	2234567890	Gatored	20.00	0	1	1	2023-11-16 03:51:12	2023-11-16 03:51:12
Editar Copiar Borrar	7	3234567890	Pepsi	8.00	0	1	1	2023-11-16 19:50:08	2023-11-16 19:50:08



Presionamos en guardar, y nos muestra el registro guardado.



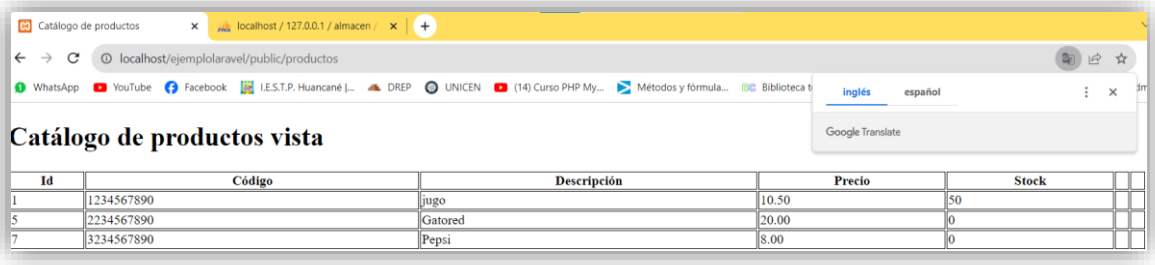
Vamos a la base de datos para cerciorarnos la modificación.

	id	codigo	descripcion	precio	existencia	categoria_id	activo	created_at	updated_at
Editar Copiar Borrar	1	1234567890	jugo	10.50	50	1	1	NULL	2023-11-16 20:37:15
Editar Copiar Borrar	5	2234567890	Gatored	20.00	0	1	1	2023-11-16 03:51:12	2023-11-16 03:51:12
Editar Copiar Borrar	7	3234567890	Pepsi	8.00	0	1	1	2023-11-16 19:50:08	2023-11-16 19:50:08

Para el destroy vamos a buscar el registro que queremos eliminar y con el objeto que se llama $producto llamamos a delete

```
app > Http > Controllers > ProductoController.php > ...
46      }
47      public function destroy($id){
48          $producto = Producto::find($id);
49          $producto->delete();
50          echo "Registro $id eliminado";
51      }
52  }
53
```

Veamos los productos:



Vamos a hacer un cambio en el index del Controller, en vez de enviarle la variable $productos, vamos a hacer el llamado directo a la función de la clase **Producto :: all( )**

Con esta instrucción le estamos diciendo que traiga todos los registros de la clase Producto, en este caso de nuestra tabla productos.

Ahora ya podemos eliminar toda esa línea de Sql.

```
$productos = DB::select('SELECT * from productos where activo = 1');
```

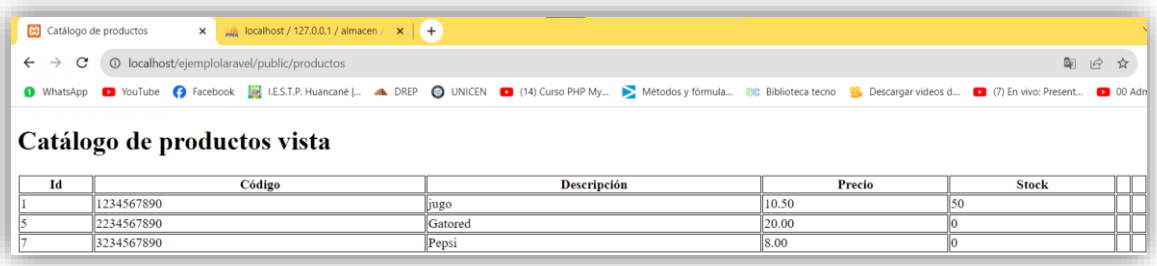
Quedaría así:

```
app > Http > Controllers > ProductoController.php > PHP Intelephense > ProductoController > create
8   class ProductoController extends Controller
9   {
10      public function index()
11      {
12          return view('productos.index', ['lista' => Producto::all()]);
13      }
```

Esa sería la forma sencilla de hacer un select de todos los registros.

Muy bien comprobemos si está bien la codificación:

http://localhost/ejemplolaravel/public/productos



No vemos ningún cambio porque está haciendo lo mismo, pero ya no tenemos ninguna consulta con el select para mostrar los productos.

Como se dieron cuenta trabajar con un ORM es mucho más sencillo y rápido, sin embargo, en ocasiones si necesitaremos hacer algunas consultas que no nos vaya a salir.

BIBLIOGRAFÍA

Otwell, T. (2023). *Laravel documentation (Version 9.x)*. Laravel LLC. Recuperado de https://laravel.com/docs/9.x

Dockins, K. (2017). *Design patterns in PHP and Laravel*. GitHub. Recuperado de https://github.com/Jessinra/READING-general-programming-books/blob/master/PHP/Frameworks/Design%20Patterns%20in%20PHP%20and%20Laravel%20-%20Kelt%20Dockins.pdf

Laravel Community. (2023). *Laravel documentation (PDF, EPUB, MOBI versions)*. GitHub. Recuperado de https://github.com/driade/laravel-book

Eyadhamza. (2023). *Laravel roadmap: Learning path and structured guide*. GitHub. Recuperado de https://github.com/Eyadhamza/LaravelRoadmap

Ebook Foundation. (2023). *Free programming books: Laravel resources*. GitHub. Recuperado de https://github.com/EbookFoundation/free-programming-books

Fundamentos Arquitectónicos de Laravel

Fundamentos Arquitectónicos de Laravel es una obra orientada al aprendizaje práctico y estructurado del desarrollo de aplicaciones web utilizando uno de los frameworks más importantes del ecosistema PHP.El contenido del libro aborda desde los conceptos fundamentales hasta la implementación de funcionalidades esenciales como rutas, controladores, vistas, migraciones y el uso del ORM Eloquent, proporcionando una guía clara y progresiva para la construcción de aplicaciones web modernas.A través de ejemplos aplicados y explicaciones paso a paso, esta obra facilita la comprensión de la arquitectura Modelo-Vista-Controlador (MVC), permitiendo al lector desarrollar soluciones eficientes, escalables y alineadas con las buenas prácticas del desarrollo de software.

Sobre los autores

Los autores cuentan con formación en ingeniería informática y sistemas, con experiencia en el desarrollo de software y la enseñanza universitaria, contribuyendo a la formación de nuevos profesionales en el área tecnológica mediante la elaboración de material académico actualizado y aplicado.