**¿Qué es Laravel?**

* Un **framework MVC** (Modelo–Vista–Controlador) que separa claramente la lógica de negocio (Modelos), la presentación (Vistas) y el flujo de control (Controladores).
* Proporciona una **arquitectura organizada**, con convenciones sobre configuración (“convention over configuration”) para acelerar el desarrollo.

**Características principales**

1. **Artisan CLI**
   * Herramienta de línea de comandos que genera controladores, modelos, migraciones, seeds y más en segundos.
2. **Eloquent ORM**
   * ORM “activo récord” que facilita trabajar con bases de datos: define relaciones, consultas y scopes con sintaxis fluida.
3. **Migraciones y Seeders**
   * Versiona el esquema de la base de datos en código, permitiendo crear, modificar y poblar tablas de forma segura.
4. **Blade Templating**
   * Motor de plantillas ligero y expresivo para las vistas, con herencia de layouts y directivas propias (@if, @foreach, etc.).
5. **Middleware y Rutas**
   * Filtra y agrupa rutas con middleware (autenticación, CORS, rate limiting) de manera sencilla.
6. **Colas y Jobs**
   * Desacopla tareas pesadas (envío de correos, procesamiento de imágenes) para ser ejecutadas en segundo plano.
7. **Eventos y Listeners**
   * Sistema de eventos nativo para desacoplar y extender funcionalidades sin tocar el núcleo de la aplicación.
8. **Seguridad incorporada**
   * CSRF protection, hashing de contraseñas, encriptación, políticas de autorización y validaciones centralizadas.

**¿Por qué usar Laravel?**

* **Productividad**: Artisan y los generadores reducen boilerplate.
* **Mantenibilidad**: Arquitectura limpia y modular, fácil de escalar y refactorizar.
* **Comunidad**: Ecosistema con paquetes oficiales (Sanctum, Passport, Telescope) y una gran comunidad que aporta soluciones.
* **Ecosistema completo**: Herramientas de testing (PHPUnit integrado), Vite/Laravel Mix para assets, scheduler para tareas programadas.

¿Por qué Laravel con React?  
Al combinar **Laravel** en el backend con **React** en el frontend, obtenemos una arquitectura moderna, escalable y mantenible. Estas son las razones clave:

1. **Desacoplar responsabilidad:** Laravel gestiona datos, autenticación y rutas; React se centra en la interfaz y la interacción.
2. **Consumir APIs RESTful sin complicaciones:** Laravel crea endpoints listos para JSON y React los llama vía fetch o Axios.
3. **Acelerar el desarrollo:** Artisan y Eloquent automatizan el backend; componentes reutilizables y Vite optimizan el frontend.
4. **Garantizar seguridad y rendimiento:** Laravel ofrece validaciones, colas y protección CSRF; React optimiza el renderizado y la experiencia de usuario.

Así podemos iterar y escalar cada capa de forma independiente, manteniendo el código limpio y modular.

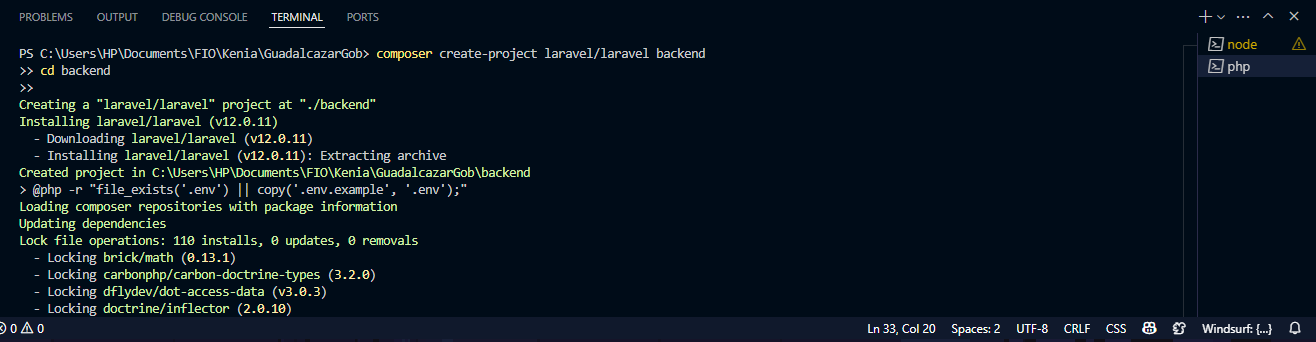
Flujo típico de trabajo

1. Definir modelos y migraciones → php artisan make:model Product -m.
2. Diseñar rutas y controladores → php artisan make:controller ProductController --api.
3. Construir y probar la API con Artisan (php artisan serve) y clientes HTTP (Thunder Client, Postman).
4. Crear vistas o consumir desde un frontend (React, Vue) usando JSON; Laravel gestiona la lógica y expone endpoints listos para consumir.

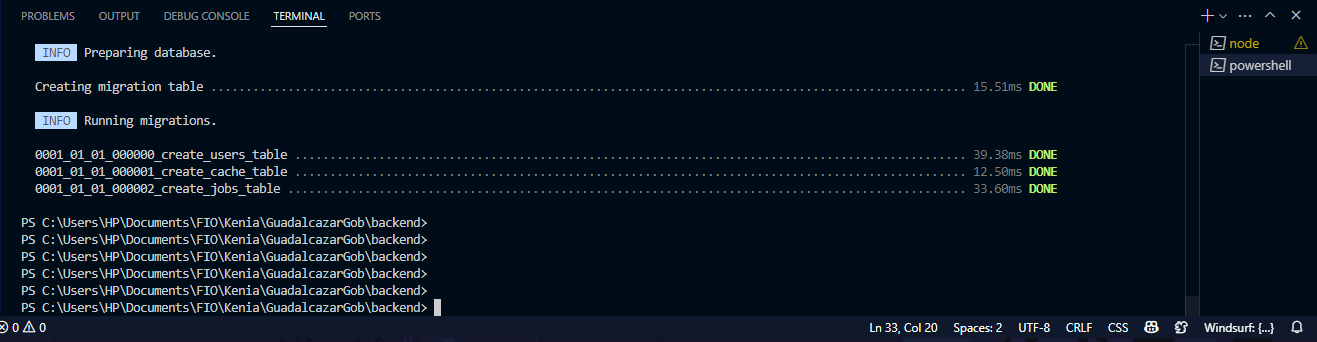
Con Laravel disponemos de un entorno sólido y probado para montar desde APIs sencillas hasta aplicaciones empresariales complejas, manteniendo siempre un equilibrio entre rapidez de desarrollo y buenas prácticas de ingeniería.

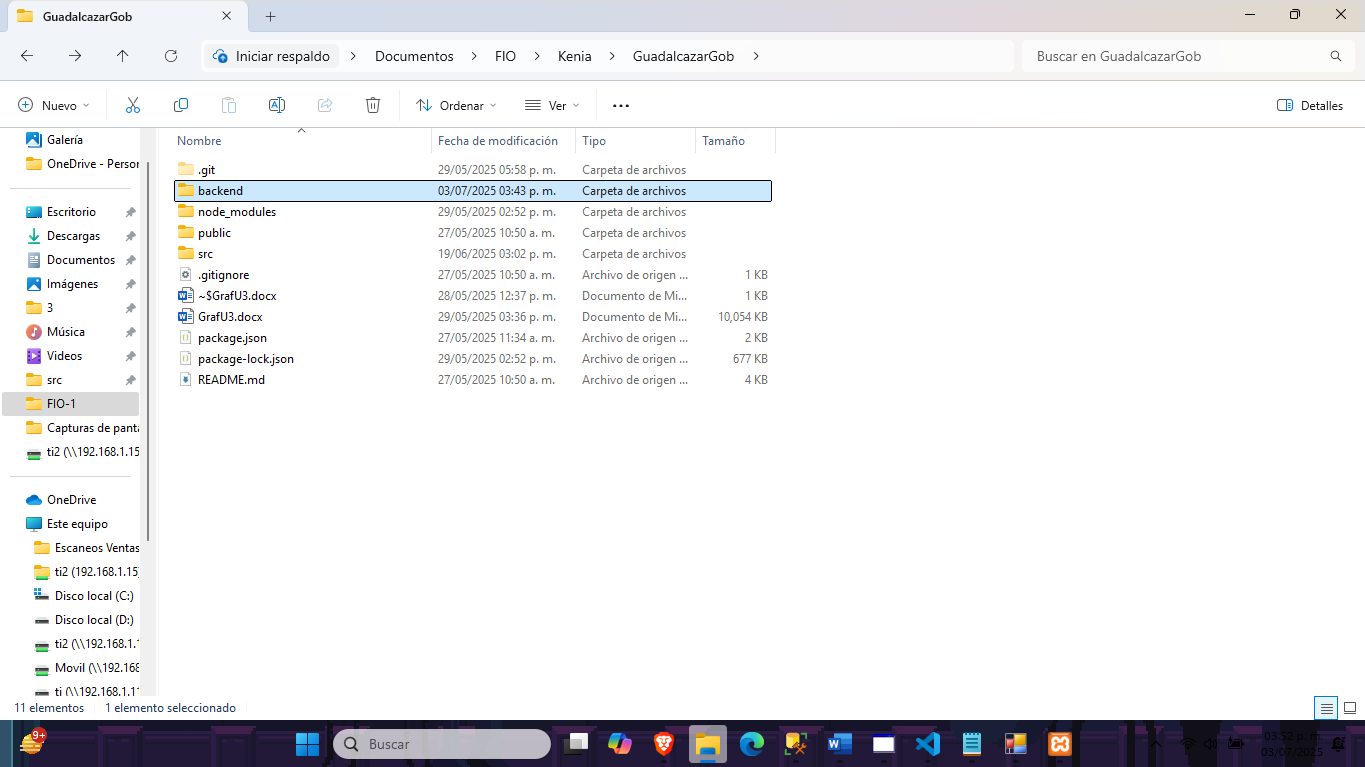
Instalación:

Instalamos Laravel para nuestro backend y creamos la bd llamada “guadalcazargob” desde MySQL.

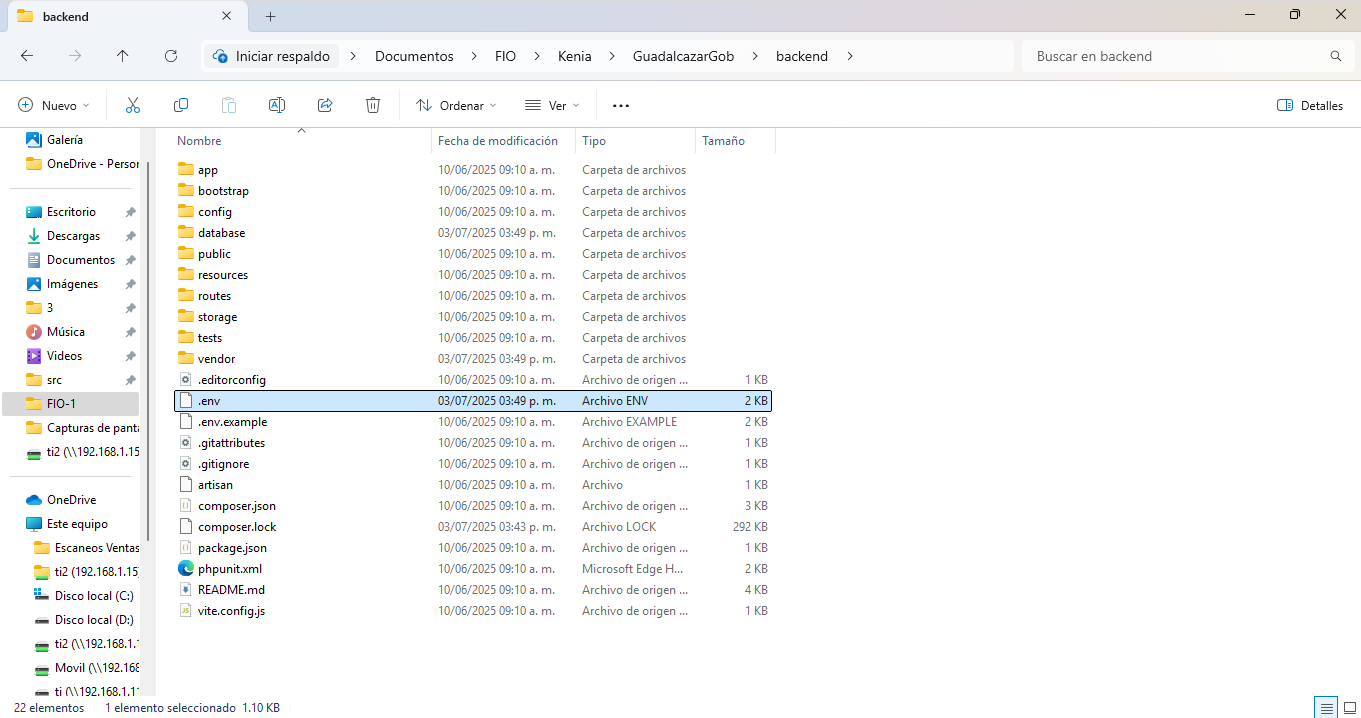


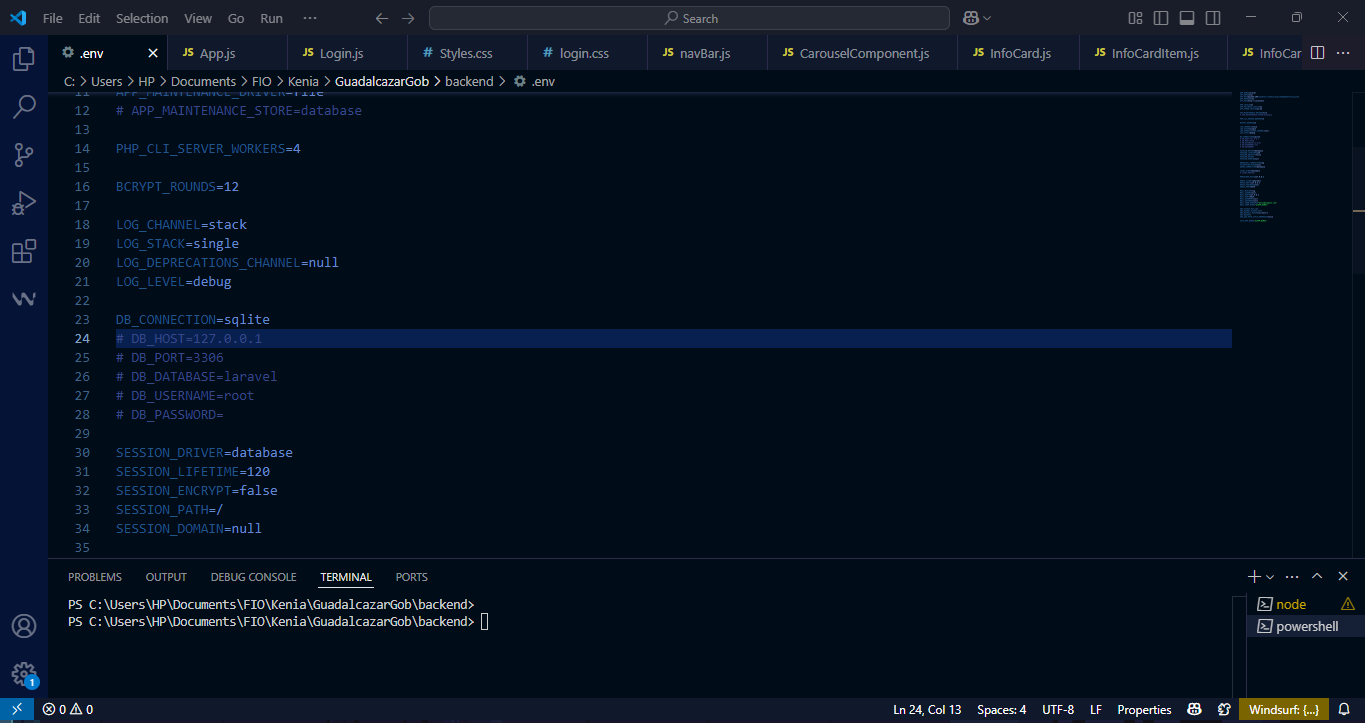
Cuando se haya terminado de instalar laravel debería aparecer algo asi

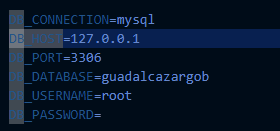


Y en la raíz de nuestro proyecto debería haberse creado una carpeta llamada “backend”  


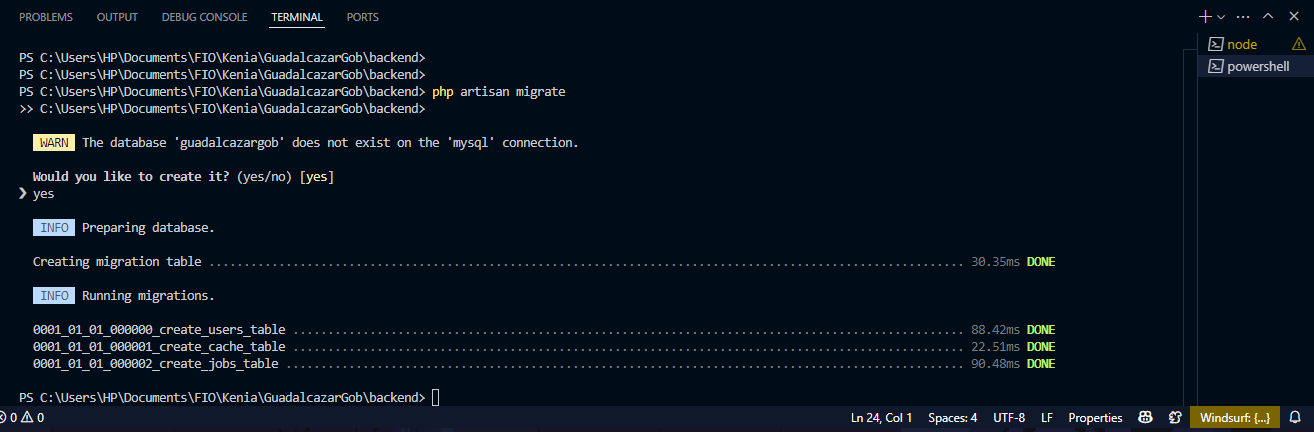
Ahora debemos modificar el archivo .env en la carpeta backend para poder enlazar nuestra base de datos a nuestro proyecto





Ajustamos estos parámetros para nuestras pruebas con XAMPP (Solamente son pruebas eventualmente agregaremos la seguridad y configuración pertinente)  


Y probamos si no hay algún error con el comando  
**#php artisan migrate**  
y si no da error como en la imagen entonces establecimos conexión exitosamente



Ahora creamos el endpoint para login

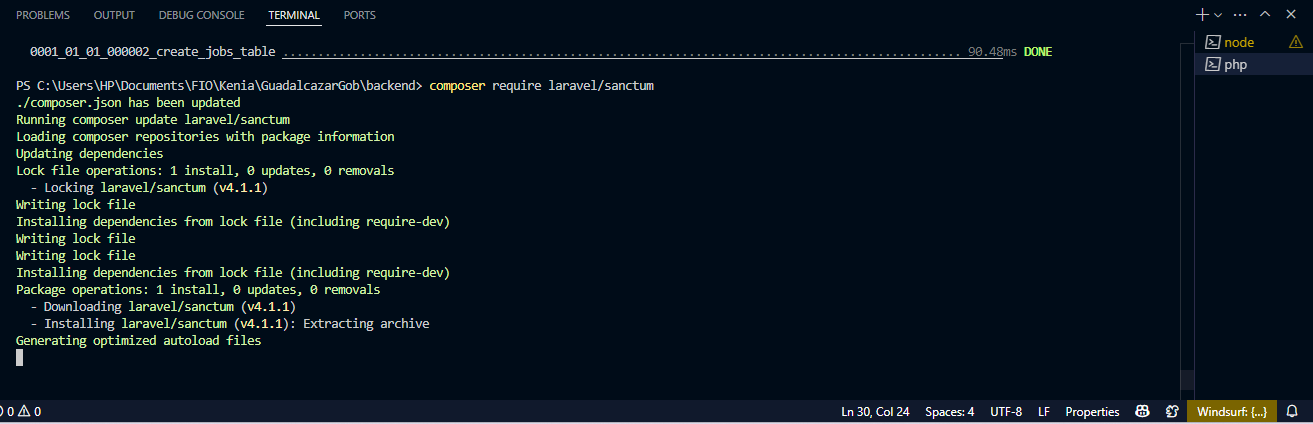
Vamos a usar el sistema de autenticación que trae Laravel (usando API tokens):

Instalamos con los siguientes comandos de Laravel Sanctum (para autenticar APIs, desde la carpeta raíz “backend”):

#composer require laravel/sanctum

#php artisan vendor:publish --provider="Laravel\Sanctum\SanctumServiceProvider"

#php artisan migrate



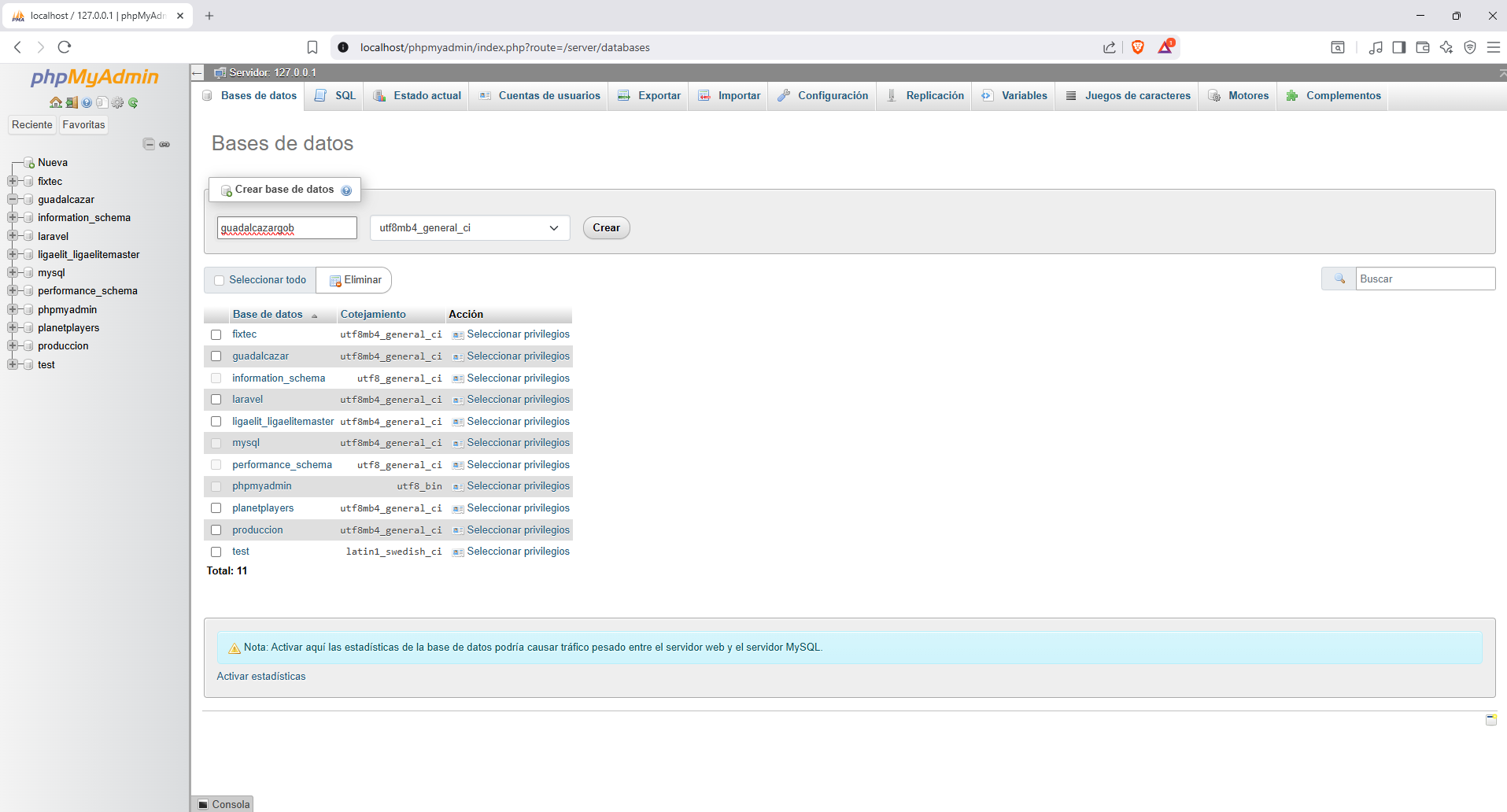
Para pruebas iniciales, usaremos XAMPP como entorno local: así instalamos Apache, MySQL y PHP de golpe, ejecutamos migraciones y seeders, y detectamos problemas de configuración o permisos de forma aislada.

Una vez validado el esquema y la lógica de datos, prepararemos el paso a producción en un servidor dedicado:

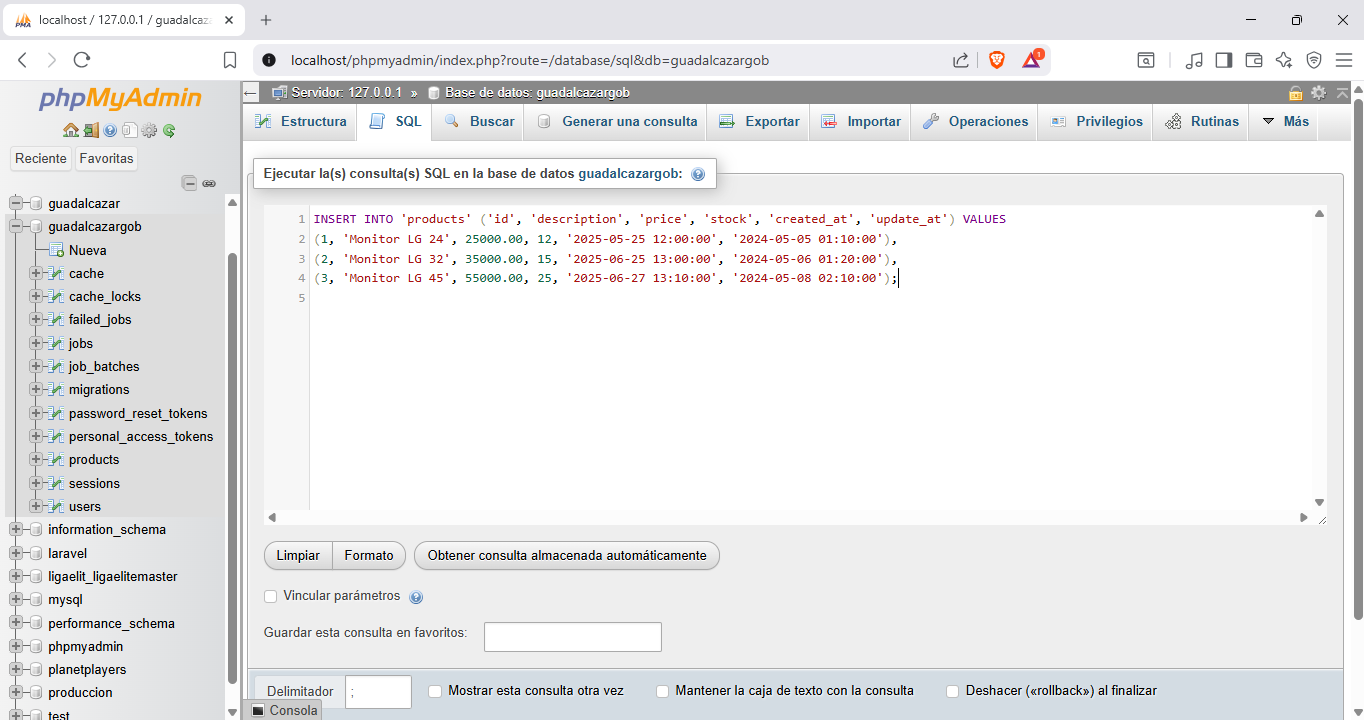
* **Exportar esquema y datos** (con mysqldump o migraciones automáticas).
* **Actualizar .env** para apuntar al nuevo host, puerto, usuario y contraseña.
* **Configurar seguridad:** restricciones de IP, cuentas con privilegios mínimos, backups programados y cifrado SSL/TLS.
* **Ajustar rendimiento:** parámetros de buffers, conexiones máximas y cacheo de consultas según carga prevista.

Este flujo —primero XAMPP en local, luego servidor dedicado— garantiza una migración suave de un entorno de pruebas ágil a uno productivo, robusto y seguro.

Aquí crearemos la base de datos de guadalcazargob para posteriormente enlazarla con nuestro backend.



Insertamos datos de prueba para probarlos desde un complemento de Visual Studio Code que en breve explicare.



Instalamos Thunder Client desde los complementos de visual studio code  
Captura de pantalla de computadora

El contenido generado por IA puede ser incorrecto.

Para que usamos Thunder Client?

Por qué Thunder Client es una extensión ligera para Visual Studio Code diseñada para probar y depurar APIs HTTP/REST de forma rápida y sencilla, sin salir del editor.

Principales usos de Thunder Client

1. Enviar peticiones HTTP

* Soporta todos los métodos: GET, POST, PUT, PATCH, DELETE, etc.
* Permite configurar cabeceras, parámetros de ruta/query, cuerpo (JSON, form-data, x-www-form-urlencoded) y autenticación.

1. Organizar colecciones de peticiones

* Agrupa tus endpoints en carpetas o colecciones (por ejemplo, “Productos”, “Usuarios”, “Autenticación”).
* Guarda cada petición con su propio nombre y descripción.

1. Variables de entorno

* Define entornos (por ejemplo: local, staging, production) con variables como {{baseUrl}}, {{token}}, etc.
* Facilita alternar rápidamente de un servidor a otro sin editar cada URL manualmente.

1. Historial y respuestas

* Mantiene un registro de las peticiones enviadas, con fecha y hora.
* Visualiza respuestas en formatos:
* JSON (con resaltado de sintaxis y árbol plegable).
* Texto, HTML, o incluso vistas en bruto.
* Muestra tiempos de respuesta y tamaños de payload.

1. Autenticación simplificada

* Soporta esquemas comunes: Bearer Token, Basic Auth, API Key, OAuth2 (parcial).
* Guarda credenciales por entorno.

1. Tests y scripts (en versiones Pro)

* Permite definir assertions (comprobaciones) para automatizar validaciones de respuesta.
* Ejecutar pequeños snippets de JavaScript antes o después de la petición.

1. Integración con VS Code

* Sin cambiar de aplicación: todo ocurre dentro del mismo editor.
* Atajos de teclado para abrir y ejecutar tus peticiones.

¿Por qué usar Thunder Client?

* Ligero y rápido: ocupa pocos megas y consume menos recursos que herramientas más pesadas como Postman.
* Flujo más ágil: no necesitas cambiar de ventana, acelera el desarrollo al mantener todo en VS Code.
* Ideal para microservicios: cuando trabajas con APIs complejas o muchas rutas, te ayuda a mantenerlas organizadas.
* Gratuito y open source: la versión básica cubre la mayoría de necesidades sin costo.

A continuación, utilizaremos Thunder Client para validar que el backend responda correctamente: así podremos confirmar que cada endpoint existe, devuelve los datos esperados y maneja errores o validaciones como diseñamos. Una vez que comprobamos que la API funciona, pasamos a definir y ejecutar nuestras migraciones de Laravel para crear las tablas en la base de datos, revisando esquemas, relaciones y poblando datos de prueba con seeders o factories. Con la base de datos y la lógica del servidor ya validadas, el siguiente paso es integrar estas rutas con el frontend: construiremos componentes o páginas que consuman la API de forma dinámica, mostrando listados, formularios de creación/edición y vistas detalladas. De este modo seguimos un flujo incremental y modular—primero aseguramos el correcto funcionamiento de la capa de datos y negocio, y luego enlazamos esa lógica a las vistas, lo que nos permite detectar y corregir cualquier fallo de comunicación cliente-servidor antes de avanzar a la siguiente fase de diseño visual o interacción.

Ejemplo:

Con esta petición lo que hacemos es pedirle que nos traiga los datos que insertamos desde la bd.

Texto

El contenido generado por IA puede ser incorrecto.

Ahora le pediremos que cree un registro, cambiando el método por POST

Captura de pantalla de computadora

El contenido generado por IA puede ser incorrecto.

Nos da el status 200 de ok eso significa que si inserto los datos ahora lo que haremos es volver a enviar una solicitud GET para verificarlo visualmente.

Interfaz de usuario gráfica, Aplicación

El contenido generado por IA puede ser incorrecto.

Verificación visual:

Captura de pantalla de computadora

El contenido generado por IA puede ser incorrecto.

Y si como podemos ver esta agregado.

Ahora enviaremos una solicitud solo para que nos devuelva un solo id en este caso el 2 modificando la petición quitando {id} por el numero 2 que queremos.

Captura de pantalla de computadora

El contenido generado por IA puede ser incorrecto.

Ahora la solicitud de eliminar

Captura de pantalla de un celular

El contenido generado por IA puede ser incorrecto.

Eliminaremos el id 4 que creamos al principio de nuestras pruebas

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.



Volvemos a traer todos los id para ver si fue eliminado correctamente

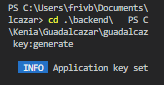
Captura de pantalla de computadora

El contenido generado por IA puede ser incorrecto.

Para correr laravel correctamente necesitamos generar una key de lo contrario nos generara un error como este  
**Illuminate\Encryption\MissingAppKeyException**

**No application encryption key has been specified.**

Esto significa que Laravel no tiene configurada una clave de cifrado (APP\_KEY) en nuestro archivo .env



Al ejecutar este comando sucederán dos cosas:

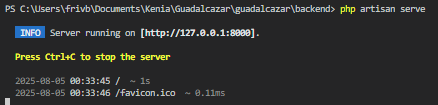
* **Generará una clave como: base64:largo\_valor\_aquí**
* **La insertará automáticamente en el archivo .env como APP\_KEY=...**

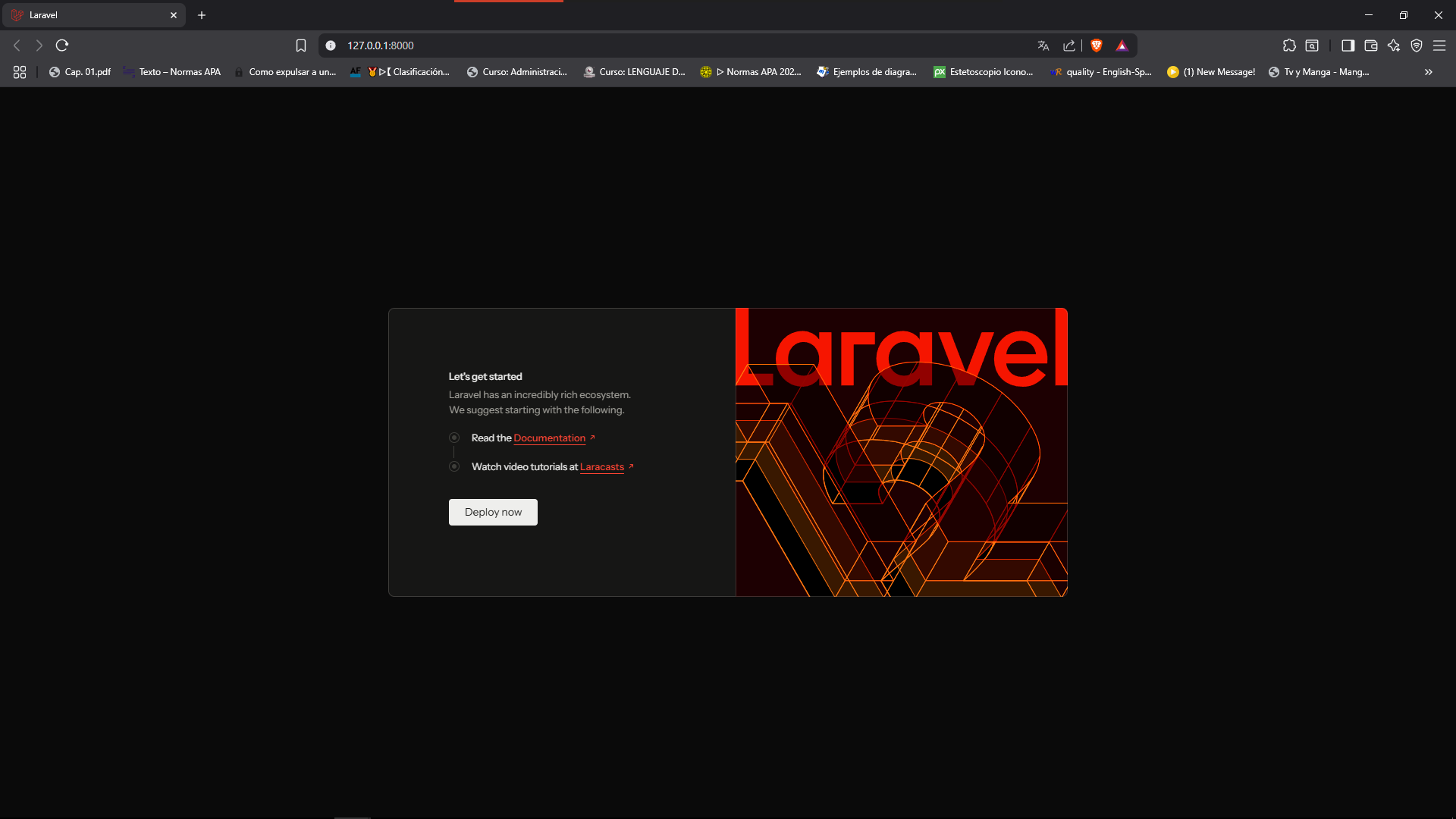
Laravel necesita esta clave para poder cifrar/de-cifrar datos como sesiones, cookies, etc.

Nunca debemos compartir la APP\_KEY en público si la aplicación ya está en producción pero aquí pondremos la que se genero ya que es parte del proyecto y de igual manera se podría generar una nueva mas adelante si hubiera algún inconveniente

**APP\_KEY=base64:eKgXuGWvCmgH85n0Frp5Pmnm3pj/8als5VdtlOoa4yI=**

Y después



Para correr el servidor y en el navegador web nos debe aparecer algo asi  


Ese comando **debe estar activo** en la consola mientras desarrollamos de lo contrario nos aparecerá un mensaje como “No se puede acceder a este sitio”.

Ahora utilizaremos



Después vamos a buscar ese modelo que creamos y vamos a poner el siguiente código con su respectiva explicación comentada en el mismo código.  
Texto

El contenido generado por IA puede ser incorrecto.

Despues vamos a auth.php en config en nuestro proyecto

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

Y vamos a configurar el “guards” para poder manejar las autenticaciones en laravel y tambien los providers para poder hacer referencia a nuestra clase o modelo usuarios

En la parte de Authenticacion Guards agregamos el siguiente código:  
Texto

El contenido generado por IA puede ser incorrecto.

Y en la parte del código de “User Providers” agregamos lo siguiente :

Texto

El contenido generado por IA puede ser incorrecto.

Ahora configuraremos el middlware para que pueda trabajar con el guard usuarios

En nuestro caso aun no estaba creado asi que vamos a la terminal y en nuestra carpeta backend ejecutamos lo siguiente  
Texto

El contenido generado por IA puede ser incorrecto.

Esto generara un archivo en:

backend/app/Http/Middleware/Authenticate.php

Pantalla de un video juego

El contenido generado por IA puede ser incorrecto.

El middleware en laravel es un filtro que se ejecuta antes o después de que una petición HTTP llegue a tu controlador. Por ejemplo:

* Verificar si un usuario está autenticado.
* Filtrar acceso según roles.
* Registrar logs.
* Modificar headers o la respuesta.

Ahora lo modificaremos de la siguiente manera:

Texto

El contenido generado por IA puede ser incorrecto.

Ahora vamos al archivo web.php que esta en la carpeta routes y agregamos el siguiente código:

Texto

El contenido generado por IA puede ser incorrecto.

Nota. Aquí aun no creamos el LoginController pero igual lo agregamos para crearlo después.

**Git y GitHub**

Durante el desarrollo de este proyecto, utilicé **Git** y **GitHub** como herramientas fundamentales para el control de versiones, la gestión del código fuente y la organización del flujo de trabajo. Estas herramientas me permitieron mantener un historial detallado de todos los cambios realizados, asegurar la integridad del proyecto, y facilitar la recuperación ante errores o fallos inesperados.

**Git** es un sistema de control de versiones distribuido, ampliamente utilizado en el desarrollo de software. Su principal ventaja es que permite registrar y administrar cambios en el código a lo largo del tiempo, lo que resulta especialmente útil cuando se trabaja en proyectos de mediano o largo plazo. Gracias a Git, pude documentar cada modificación relevante mediante *commits*, lo que me permitió tener un control preciso sobre el avance del proyecto, entender qué se cambió, cuándo y por qué.

Por otro lado, utilicé **GitHub** como plataforma para alojar el repositorio remoto del proyecto. GitHub no solo me permitió almacenar mi código en la nube, sino también contar con una copia de seguridad centralizada y segura. Esto fue muy útil para acceder al proyecto desde diferentes dispositivos y para asegurar que el trabajo no se perdiera en caso de una falla en mi equipo local. Además, GitHub ofrece herramientas visuales para revisar el historial de commits, comparar versiones de archivos y gestionar ramas de desarrollo.

Para ejecutar los comandos de Git utilicé **Git Bash**, una terminal que me permitió interactuar con Git desde la línea de comandos. A través de esta herramienta realicé acciones como:

* **Inicializar el repositorio** con git init.
* **Subir el proyecto** por primera vez a GitHub usando git remote add origin y git push -u origin master.
* **Registrar cambios** con git add y git commit -m.
* **Actualizar mi repositorio local** con los últimos cambios usando git pull.
* **Explorar el historial de cambios** con git log y git diff.

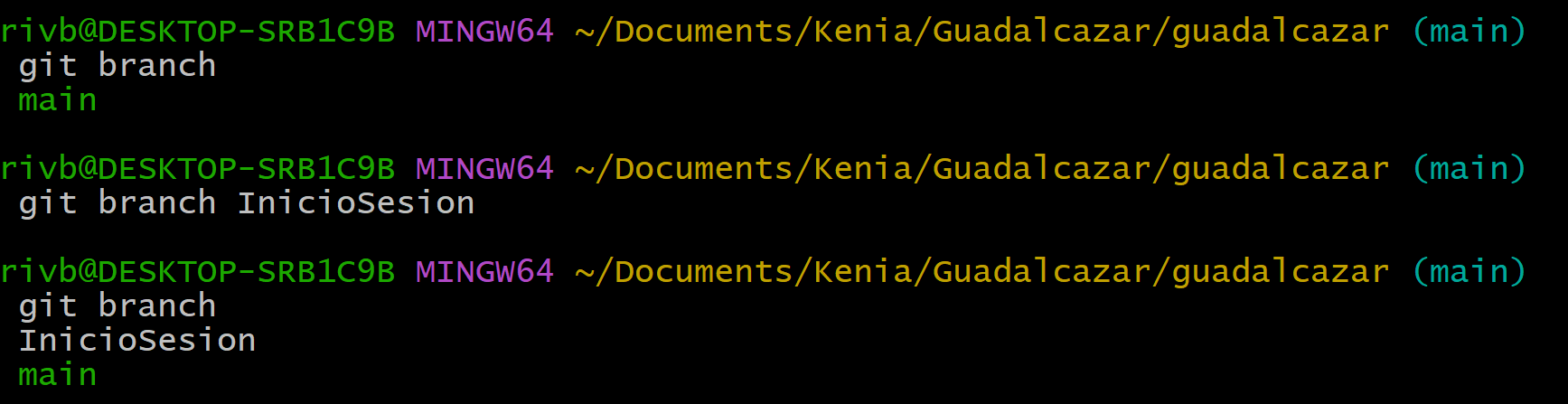
Además, una de las funcionalidades más importantes que utilicé fue la posibilidad de **crear ramas (branches)**. Esto me permitió trabajar en nuevas características o probar soluciones alternativas sin afectar el código principal. Por ejemplo, cuando quise implementar una nueva funcionalidad que no estaba seguro si funcionaría correctamente, creé una rama separada, realicé las pruebas correspondientes y, una vez validado que todo funcionaba bien, integré los cambios al código principal mediante git merge.

En ocasiones, cuando una modificación causaba errores o fallos en el sistema, Git me permitió **volver fácilmente a una versión anterior estable** utilizando comandos como git checkout o git reset. Esta capacidad fue fundamental para evitar pérdida de tiempo o retrocesos mayores, y me dio la tranquilidad de poder experimentar sin comprometer el proyecto completo.

Además de todas estas funciones, Git también me ayudó a mantener un flujo de trabajo más organizado y profesional. Cada cambio quedó documentado y justificado en el historial, lo cual facilita el mantenimiento del código y la comprensión del mismo a futuro.

El uso de Git y GitHub fue esencial para el desarrollo eficiente y ordenado de este proyecto. Aunque fui el único desarrollador involucrado, estas herramientas me ofrecieron grandes ventajas como control de versiones, respaldo automático, gestión de ramas y recuperación ante errores. Aprender a utilizarlas correctamente no solo me permitió mejorar mi productividad, sino que también fortaleció mis habilidades como desarrollador, preparándome para futuros trabajos en equipo y proyectos más complejos.

Aquí algunos ejemplos:

Aquí utilizamos para crear el inicio de sesión ya que íbamos a modificar cosas muy sensibles del código y queríamos probar sin tener miedo a corromper el proyecto  


Con git branch cambiamos de rama a la que usaremos de prueba

Texto

El contenido generado por IA puede ser incorrecto.

Ahora aquí una prueba de el como guardamos cambios en nuestra nueva rama  
Texto

El contenido generado por IA puede ser incorrecto.

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use Illuminate\Support\Facades\Auth;

use App\Models\User;

use Illuminate\Validation\ValidationException; // Importa esta clase para manejar errores de validación de forma más limpia

use Illuminate\Support\Facades\Hash;

class AuthController extends Controller

{

    public function login(Request $request)

    {

        // 1. Validar los datos de entrada

        $credentials = $request->validate([

            'email' => 'required|string|email',

            'password' => 'required|string',

        ]);

        // 2. Intentar autenticar al usuario

        if (!Auth::attempt($credentials)) {

            // Si la autenticación falla, lanza una excepción de validación

            // Laravel la convertirá automáticamente en una respuesta JSON 422

            throw ValidationException::withMessages([

                'email' => ['Las credenciales proporcionadas son incorrectas.'],

            ]);

        }

        #$request->session()->regenerate();

        // 3. Si la autenticación es exitosa, Laravel ya ha establecido la cookie de sesión.

        // No necesitas generar un token aquí para la autenticación de SPA.

        // Simplemente devuelve una respuesta sin contenido (o un mensaje de éxito simple).

        return response()->noContent(); // Responde con un estado 204 (No Content)

    }

    // Opcional: Un método para obtener los datos del usuario autenticado

    public function user(Request $request)

    {

        return response()->json($request->user());

    }

    // Opcional: Un método para cerrar sesión

    public function logout(Request $request)

    {

        Auth::guard('web')->logout(); // Cierra la sesión web

        $request->session()->invalidate(); // Invalida la sesión

        $request->session()->regenerateToken(); // Regenera el token CSRF

        return response()->noContent();

    }

    public function register(Request $request)

    {

        // 1. Validar los datos de entrada

        $request->validate([

            'name' => 'required|string|max:255',

            'email' => 'required|string|email|max:255|unique:users',

            'password' => 'required|string|min:8|confirmed', // 'confirmed' valida que password\_confirmation sea igual

        ]);

        // 2. Crear el nuevo usuario

        $user = User::create([

            'name' => $request->name,

            'email' => $request->email,

            'password' => Hash::make($request->password),

        ]);

        // 3. Devolver una respuesta de éxito

        return response()->json([

            'message' => '¡Registro exitoso!',

            'user' => $user,

        ], 201); // El estado 201 indica que se ha creado un nuevo recurso

    }

}

    // Puedes añadir más métodos según sea necesario, como para restablecer contraseñas, etc.

**Flujo Completo de Login: Frontend (React) y Backend (Laravel)**

El flujo de login es una coreografía precisa entre tu frontend (React) y tu backend (Laravel). La cookie es la pieza clave que permite que ambos se recuerden después del primer "saludo".

A continuación se describe el flujo completo, paso a paso.

**1. El Intento de Login (Frontend)**

Todo comienza cuando el usuario rellena el formulario y presiona "Iniciar Sesión".

* **Componente Login.js**:
  1. Captura el email y la contraseña.
  2. Hace una llamada a la API usando Axios: axios.post('/api/login', { email, password });.
  3. Gracias a la configuración axios.defaults.withCredentials = true, esta petición le dice al navegador: "Permite que se usen cookies con este servidor".

**2. Verificación de Credenciales (Backend)**

La petición llega a Laravel.

* **Ruta routes/api.php**:
  + La ruta Route::post('/login', ...) recibe la petición y la dirige al AuthController.
* **Controlador AuthController.php**:
  1. El método login valida los datos.
  2. La función Auth::attempt() hace el trabajo pesado: toma la contraseña enviada, la hashea y la compara con la contraseña hasheada que tienes en tu base de datos para ese email.
  3. **Si las credenciales son incorrectas**, lanza un error y el proceso se detiene aquí.
  4. **Si las credenciales son correctas**, el proceso continúa.

**3. Creación de la Sesión y la Cookie (Backend)**

Este es el momento más importante. Una vez que las credenciales son validadas:

1. **$request->session()->regenerate()**: Laravel realiza una acción de seguridad crucial. Destruye la sesión temporal que el usuario tenía y crea un **ID de sesión nuevo y seguro**.
2. **Almacenamiento del Lado del Servidor**: Laravel guarda este nuevo ID de sesión en el servidor (en la carpeta storage/framework/sessions/ por defecto) y lo asocia con el ID del usuario que acaba de iniciar sesión. Es como un registro interno que dice: ID\_Sesion\_ABC pertenece al Usuario\_ID\_5.
3. **Envío de la Cookie**: Laravel prepara la respuesta para el frontend. Antes de enviarla, añade una cabecera HTTP llamada **Set-Cookie**. Esta cabecera contiene la cookie laravel\_session, cuyo valor es el nuevo ID de sesión cifrado.

La cookie es como la **llave 🔑** de una casa. El servidor se queda con la **cerradura 🔒** (el registro de la sesión) y le entrega la llave al navegador.

**4. Actualización del Estado (Frontend)**

El navegador recibe la respuesta del backend.

* **El Navegador**:
  1. Ve la cabecera Set-Cookie y automáticamente guarda la cookie laravel\_session en su almacén interno. A partir de ahora, la incluirá en todas las futuras peticiones a tu backend.
* **Componente Login.js (continuación)**:
  1. La llamada a /api/login fue exitosa. Ahora, para saber *quién* ha iniciado sesión, hace una segunda llamada: axios.get('/api/user').
  2. El navegador adjunta automáticamente la cookie recién guardada a esta nueva petición.
* **El Backend (de nuevo)**:
  1. La petición a /api/user es interceptada por el middleware auth:sanctum.
  2. sanctum mira la cookie, la descifra, busca el ID de sesión en su registro interno y confirma que pertenece a un usuario válido.
  3. Como la llave encaja en la cerradura, permite el paso y el AuthController devuelve los datos del usuario como JSON.
* **Componente Login.js (final)**:
  1. Recibe los datos del usuario.
  2. Llama a la función setUser(datosDelUsuario) del AuthContext.
  3. El estado global de la aplicación se actualiza. La app ahora "sabe" que hay un usuario conectado, y componentes como la barra de navegación cambian para mostrar "Bienvenido, Carlos" y el botón de "Cerrar Sesión".

**5. La Magia de Refrescar la Página (Frontend + Backend)**

Aquí es donde la cookie demuestra su verdadero valor.

1. **El Usuario Refresca la Página**:
   * Toda la aplicación de React se reinicia. El estado user en AuthContext vuelve a ser null temporalmente.
2. **useEffect en AuthContext.js**:
   * Tan pronto como la app carga, el useEffect que programamos se dispara y hace la llamada axios.get('/api/user').
3. **El Navegador y la Cookie**:
   * El navegador no ha olvidado la cookie. La adjunta a la petición.
4. **El Backend Responde**:
   * El backend recibe la petición, sanctum valida la cookie, identifica al usuario y devuelve sus datos, exactamente como en el paso anterior.
5. **Sincronización Final**:
   * El useEffect recibe los datos del usuario y llama a setUser(datosDelUsuario).
   * El estado del frontend se restaura al estado de "sesión iniciada" antes de que el usuario siquiera note algo.

**Diagrama de Flujo: ¿Cómo Funciona la Autenticación?**

Este es el viaje que hacen tus datos desde que el usuario hace clic en "Entrar" hasta que puede crear un nuevo servicio.

**Fase 1: El Proceso de Login**

1. **💻 Frontend (Login.js)**: El usuario llena el formulario y hace clic en "Entrar". Se llama a la función login() de tu AuthContext.
2. **🧠 Frontend (AuthContext.js)**: La función login usa **apiClient** para hacer 2 peticiones a Laravel:
   * **Petición A: GET /sanctum/csrf-cookie**. Su único propósito es pedirle a Laravel una cookie de seguridad (CSRF).
   * **Petición B: POST /api/login**. Envía el email y la contraseña.
3. **🐘 Backend (Laravel)**:
   * Laravel recibe la petición de login, valida la cookie de seguridad y, con Auth::attempt, verifica las credenciales en la base de datos.
   * Si son correctas, Laravel crea una **sesión** y envía una respuesta al navegador con una cabecera Set-Cookie para establecer la **cookie de sesión**. Esta es la "credencial" real.
4. **🍪 Navegador**: El navegador recibe la respuesta y **guarda automáticamente la cookie de sesión**. Esta cookie es segura y tu código JavaScript no puede leerla directamente.
5. **✅ Frontend (AuthContext.js)**:
   * Después del login exitoso, la función login llama a checkAuthStatus().
   * checkAuthStatus() hace una petición GET /api/user. El navegador **automáticamente adjunta la cookie de sesión** a esta petición.
   * Laravel recibe la petición, ve la cookie, identifica al usuario y devuelve sus datos (nombre, email, etc.).
   * Tu AuthContext recibe los datos del usuario, los guarda en el estado (setUser(datos)) y isAuthenticated se vuelve true. ¡Tu aplicación ahora sabe que el usuario está conectado!

**Fase 2: Petición a una Ruta Protegida (Ej: Crear un Servicio)**

1. **💻 Frontend (InfoCard.js)**: El usuario, ya autenticado, llena el formulario para agregar un servicio y hace clic en "Agregar". Se llama a la función handleAddCard().
2. **🧠 Frontend (apiClient)**: La función handleAddCard usa **apiClient** para hacer una petición POST /api/infocards. Como apiClient está configurado con withCredentials: true, el navegador automáticamente adjunta la **cookie de sesión** a la petición.
3. **🛡️ Backend (Middleware auth:sanctum)**:
   * La ruta /api/infocards está protegida por el middleware auth:sanctum.
   * Este middleware actúa como un guardia de seguridad. Revisa la petición, encuentra la cookie de sesión, verifica que sea válida y, si lo es, deja pasar la petición al controlador. Si no hay cookie o es inválida, devuelve el error 401 Unauthorized.
4. **👨‍🍳 Backend (InfocardController)**: Como el middleware dio el visto bueno, la petición llega al controlador. El código se ejecuta, se crea el nuevo servicio en la base de datos y se devuelve como JSON.
5. **✅ Frontend (InfoCard.js)**: Recibe la respuesta del controlador con el nuevo servicio y actualiza la lista en la pantalla para que el usuario vea el cambio al instante.

**Guía de Archivos de Configuración Clave**

Estos son los archivos que tuvimos que tocar y que son cruciales para que todo funcione.

**Frontend (React)**

* 🎨 **src/apiClient.js**
  + **Propósito**: El corazón de la comunicación con tu API. Centraliza la configuración de Axios (baseURL, withCredentials: true) para asegurar que todas las peticiones se hagan de la misma forma y siempre lleven las cookies.
* 🧠 **src/AuthContext.js**
  + **Propósito**: El cerebro del estado de autenticación de tu aplicación. Guarda la información del usuario, sabe si está autenticado o no y provee las funciones (login, logout, register) que los componentes pueden usar.

**Backend (Laravel)**

* 🔑 **.env**
  + **Propósito**: El archivo de las "llaves maestras". Contiene todas las contraseñas y configuraciones sensibles.
  + **Claves importantes**: APP\_URL, DB\_CONNECTION (toda la configuración de la base de datos), y SANCTUM\_STATEFUL\_DOMAINS (le dice a Laravel qué dominio de frontend tiene permiso para usar sesiones).
* 🚦 **config/cors.php**
  + **Propósito**: El control de tráfico. Define qué dominios externos (allowed\_origins) pueden hacer peticiones a tu API y si pueden usar credenciales (supports\_credentials => true).
* 🏗️ **bootstrap/app.php** (El que no encontrabas)
  + **Propósito**: El "director de orquesta" en Laravel moderno. Aquí se arma la aplicación y, lo más importante para nosotros, se **añaden los middlewares a los grupos de rutas**. Aquí le dijimos que el grupo 'api' debía usar sesiones (StartSession::class).
* 🗺️ **routes/api.php**
  + **Propósito**: El mapa de tu API. Define todas las rutas (/login, /user, /infocards) y les asigna los middlewares que las protegen, como auth:sanctum.

Espero que este resumen te sirva como una guía de referencia. ¡Has hecho un gran trabajo al seguir todos los pasos hasta el final!

[Navegador / SPA React]

|

| POST /api/login (envía email + password)

v

[Route en routes/api.php]

|

|----> /login ------------------> [AuthController@login]

| |

| | Verifica credenciales

| v

| Genera token Sanctum

| |

| v

| Devuelve JSON { token }

|

| GET /api/user (opcional)

v

[Route protegida middleware auth:sanctum]

|

v

[AuthController@user] ---------> Devuelve datos del usuario autenticado

|

v

[Navegador / SPA React] ---------> Guarda token en localStorage / cookies

**Archivos modificados / involucrados**

1. **routes/api.php**
   * Aquí se definen tus rutas públicas y protegidas:
   * Route::post('/login', [AuthController::class, 'login']); // pública
   * Route::post('/register', [AuthController::class, 'register']); // pública
   * Route::middleware('auth:sanctum')->group(function () {
   * Route::post('/logout', [AuthController::class, 'logout']); // protegida
   * Route::get('/user', [AuthController::class, 'user']); // protegida
   * });
2. **app/Http/Controllers/AuthController.php**
   * Aquí implementas los métodos: login, register, logout, user.
3. **config/sanctum.php**
   * Configuración de Sanctum (si usas cookies o tokens).
4. **front-end React**
   * Peticiones fetch/axios a las rutas /api/login, /api/register, /api/user.
   * Guardar el token en localStorage o cookies según la configuración.

**Guía Definitiva de Archivos y Configuraciones**

Aquí se incluyen los archivos de configuración, los de lógica, los de rutas y hasta los de depuración que usamos.

**Configuración Principal del Backend (Laravel)**

* 🔑 **.env**: El archivo de contraseñas y configuraciones vitales.
  + **Su rol**: Define la URL de la app, las credenciales de la base de datos y, crucialmente, SANCTUM\_STATEFUL\_DOMAINS para decirle a Laravel qué frontend es de confianza.
* 🚦 **config/cors.php**: El "control de fronteras" de tu API.
  + **Su rol**: Permite que tu frontend (localhost:3000) se comunique con tu backend, especialmente con la opción supports\_credentials => true.
* 🏗️ **bootstrap/app.php**: El "director de orquesta" en Laravel moderno.
  + **Su rol**: Aquí le dimos "poderes de sesión" a nuestras rutas de API al añadir StartSession::class al grupo api, corrigiendo el error 500.

**Estructura de la Base de Datos (Laravel)**

* 🧾 **database/migrations/...\_create\_users\_table.php**: El "plano arquitectónico" de tu tabla de usuarios.
  + **Su rol**: Define las columnas que tendrá la tabla users en la base de datos. Es el primer paso antes de poder guardar usuarios. Se ejecuta con php artisan migrate.
* 🦸 **app/Models/User.php**: El modelo que representa a un usuario.
  + **Su rol**: Es la conexión directa con la tabla users. Aquí se añade el "superpoder" use HasApiTokens; de Sanctum, que le permite al modelo manejar tokens y autenticación de API.

**Lógica del Backend (Laravel)**

* 👨‍🍳 **app/Http/Controllers/AuthController.php**: El "chef" que prepara las respuestas de autenticación.
  + **Su rol**: Contiene el código PHP que realmente hace el trabajo de login (verificar contraseña), register (crear usuario), logout (destruir sesión) y user (devolver datos).

**Rutas del Backend (Laravel)**

* 🗺️ **routes/api.php**: El mapa de tu API.
  + **Su rol**: Define todas las URLs de tu API (/api/login, /api/infocards) y las protege con el guardia middleware('auth:sanctum').
* 🕸️ **routes/web.php**: El mapa para rutas web tradicionales.
  + **Su rol**: En nuestro caso, su rol fue **no ser usado** para la API. Lo revisamos para asegurarnos de que la lógica estuviera en api.php, que es la convención correcta para una SPA.

**Configuración y Lógica del Frontend (React)**

* ⚙️ **src/apiClient.js**: El "mensajero" oficial de tu aplicación.
  + **Su rol**: Centraliza la configuración de Axios (baseURL, withCredentials: true) para que todas las peticiones al backend se hagan de forma correcta y consistente.
* 🧠 **src/AuthContext.js**: El "cerebro" que recuerda quién eres.
  + **Su rol**: Mantiene el estado de autenticación (user, isAuthenticated) de forma global para que toda la aplicación sepa si el usuario ha iniciado sesión.

**Herramientas de Depuración**

* 🔍 **storage/logs/laravel.log**: El "diario de errores" del servidor.
  + **Su rol**: Fue una herramienta crucial. Nos permitió diagnosticar el **error 500** al mostrarnos el mensaje exacto (Session store not set on request), lo que nos llevó a la solución en bootstrap/app.php.

¡Ahora sí! Esta lista es el panorama completo de todas las piezas que conforman tu sistema de autenticación. Excelente trabajo al llegar hasta aquí.

**1️⃣ InfoCard.js**

* **Propósito:** Es el componente principal que maneja toda la lógica de las tarjetas de información (“InfoCards”).
* **Funciones principales:**
  + Traer los datos del backend usando apiClient.get('/api/infocards').
  + Agregar, editar y eliminar tarjetas usando apiClient.post, put y delete.
  + Mantener el estado de las tarjetas en items con useState.
  + Renderizar el formulario (InfoCardForm) para agregar nuevas tarjetas y la lista de tarjetas (InfoCardList) que se muestra en la interfaz.

**Resumen:** Controla los datos y sirve de “puente” entre el backend y los componentes hijos.

**2️⃣ InfoCardItem.jsx**

* **Propósito:** Representa una tarjeta individual.
* **Funciones principales:**
  + Muestra la información de la tarjeta: título, descripción e imagen.
  + Permite editar la tarjeta directamente en el componente usando un modo de edición (isEditing).
  + Permite eliminar la tarjeta llamando a la función onDelete pasada desde InfoCard.js.

**Resumen:** Es la tarjeta individual que se puede ver, editar o eliminar.

**3️⃣ InfoCardForm.jsx**

* **Propósito:** Formulario para agregar nuevas tarjetas de información.
* **Funciones principales:**
  + Captura los datos de la tarjeta: título, descripción y URL de imagen.
  + Al enviar el formulario (handleSubmit), llama a onAdd (función pasada desde InfoCard.js) para agregar la tarjeta al backend y actualizar el estado.
  + Limpia el formulario después de enviar los datos.

**Resumen:** Permite a los usuarios crear nuevas tarjetas.

**4️⃣ InfoCardList.jsx**

* **Propósito:** Lista de todas las tarjetas.
* **Funciones principales:**
  + Recibe un array de tarjetas (items) y funciones onDelete y onEdit desde InfoCard.js.
  + Renderiza cada tarjeta usando el componente InfoCardItem.

**Resumen:** Se encarga de mostrar todas las tarjetas existentes en una vista de lista.

**📦 Relación entre ellos**

InfoCard.js ← Controla todo

├── InfoCardForm.jsx ← Agregar nuevas tarjetas

└── InfoCardList.jsx ← Lista de tarjetas

└── InfoCardItem.jsx ← Cada tarjeta editable/eliminable

En pocas palabras:

* InfoCard → Lógica y estado.
* InfoCardForm → Crear tarjetas.
* InfoCardList → Mostrar lista de tarjetas.
* InfoCardItem → Cada tarjeta individual con opciones de editar y eliminar.

Perfecto, aquí tienes un **diagrama visual simplificado del flujo de tu componente InfoCard** y sus subcomponentes:

┌───────────────────────┐

│ Backend API │

│ (ej. /api/infocards)│

└─────────▲────────────┘

│

GET / POST / PUT / DELETE

│

┌─────────┴─────────┐

│ InfoCard.js │ ← Componente principal, controla el estado

│ - useState(items) │

│ - useEffect(fetch)│

│ - handleAddCard │

│ - handleEdit │

│ - handleDelete │

└───────┬───────┬────┘

│ │

│ │

┌─────────────┘ └──────────────┐

│ │

┌─────────────────────┐ ┌─────────────────────┐

│ InfoCardForm.jsx │ │ InfoCardList.jsx │

│ - Formulario para │ │ - Mapea items y │

│ agregar tarjetas │ │ renderiza InfoCardItem │

│ - onAdd → InfoCard │ │ - onEdit / onDelete → InfoCard │

└─────────────────────┘ └───────────┬─────────┘

│

┌──────────┴─────────┐

│ InfoCardItem.jsx │

│ - Muestra tarjeta │

│ - Edita / Elimina │

│ - Estado local │

└───────────────────┘

**Explicación del flujo:**

1. InfoCard.js carga los datos desde el **backend** usando apiClient.get.
2. Los datos (items) se pasan a:
   * InfoCardList.jsx → renderiza todas las tarjetas.
   * InfoCardForm.jsx → permite agregar nuevas tarjetas.
3. Cada tarjeta individual es un InfoCardItem.jsx, que puede:
   * Editarse → llama a handleEdit en InfoCard.js.
   * Eliminarse → llama a handleDelete en InfoCard.js.
4. Todo cambio (agregar, editar, eliminar) actualiza el **estado central items** en InfoCard.js y se refleja en la UI.

**Crear secciones administrables**

En el desarrollo de aplicaciones web modernas, un aspecto clave es la capacidad de contar con secciones administrables, es decir, módulos donde el administrador pueda crear, editar o eliminar información de manera dinámica. Para lograrlo, el proceso se puede dividir en dos grandes fases: preparar el Backend para manejar los datos y construir el Frontend para que los usuarios interactúen con ellos.

Fase 1: Backend (Laravel)

El Backend es la base sobre la cual se construye todo el sistema. Aquí se define cómo se almacenan los datos, cómo se validan y de qué manera se exponen al Frontend a través de una API.

**1. Creación de la Migración**

El primer paso consiste en definir la estructura de la base de datos. Laravel ofrece el comando:

**#php artisan make:migration create\_nombre\_en\_plural\_table**

Este comando genera un archivo de migración en el que se definen las columnas que tendrá la tabla (por ejemplo: id, title, description, etc.). Estas columnas se agregan dentro de la función up(). Una vez ejecutada la migración, la tabla queda lista en la base de datos.

**2. Creación del Modelo**

El modelo actúa como un puente entre la base de datos y el resto de la aplicación. Para crearlo se utiliza:

**# php artisan make:model NombreEnSingular**

En este archivo se define la propiedad $fillable, que indica qué columnas pueden ser editadas mediante asignación masiva. Además, se pueden añadir "Accessors" para transformar automáticamente ciertos campos, como convertir la ruta de una imagen en una URL completa.

**3. Creación del Controlador**

El controlador contiene la lógica que permite interactuar con los datos. El comando para generarlo es:

**# php artisan make:controller Api/NombreController --api**

Dentro del controlador se implementan los métodos CRUD:

* index(): devuelve todos los registros, ideal para mostrar listados en la parte pública.
* show(): devuelve un solo registro, útil para vistas de detalle.
* store(): permite guardar nuevos registros, incluyendo validación de campos y subida de imágenes.
* destroy(): elimina un registro, y si corresponde, también borra el archivo asociado en el almacenamiento.

**4. Definición de Rutas en la API**

El archivo routes/api.php es el lugar donde se configuran las rutas. Aquí se definen dos tipos de accesos:

* **Públicos**, que solo permiten index y show.
* **Protegidos**, que requieren autenticación con Sanctum para realizar acciones de administración como store, update o destroy.

Esto se logra dividiendo las rutas en dos declaraciones de Route::apiResource(...), una sin restricciones y otra dentro del middleware auth:sanctum

**5. Ejecución de la Migración**

Finalmente, se construye la tabla en la base de datos ejecutando:

**# php artisan migrate**

Con esto, la infraestructura del Backend queda lista para ser consumida por el Frontend.

**Fase 2: Frontend (React) 🎨**

Una vez que el Backend está listo y ofrece la información mediante una API, llega el turno de construir el **Frontend**, es decir, la parte visible para los usuarios. En esta fase, el objetivo es doble: por un lado, ofrecer a los visitantes una experiencia clara y amigable para consultar el contenido; y por otro, proporcionar al administrador un **panel de control** desde donde pueda gestionar esa información de manera eficiente.

1. **Creación de la Vista Pública (Lista)**

El primer componente es la **lista pública**, la página que cualquier visitante puede ver para explorar el contenido de la nueva sección.

* **Archivo:** NombreEnPlural.jsx
* **Lógica:** Utiliza el hook useEffect para hacer una petición GET al endpoint /api/nombre-en-plural mediante un cliente de peticiones (apiClient). La respuesta se almacena en un estado local con useState.
* **Vista:** A partir de los datos en el estado, se generan tarjetas (cards) para cada elemento, mostrando su información básica (título, imagen pequeña, etc.). Además, cada tarjeta incluye un enlace (<Link>) que lleva al detalle del elemento (/nombre-en-plural/${item.id}).

De esta forma, el visitante puede navegar fácilmente por la sección y elegir qué elemento desea explorar en profundidad.

1. **Creación de la Vista de Detalle**

El segundo componente es la **vista de detalle**, destinada a mostrar toda la información de un solo registro.

* **Archivo:** NombreEnSingularDetalle.jsx
* **Lógica:** Aquí se aprovecha el hook useParams para obtener el id desde la URL. Luego, con useEffect, se realiza una petición GET al endpoint /api/nombre-en-plural/{id}.
* **Vista:** Se muestran todos los datos disponibles del elemento seleccionado: título, imagen en tamaño completo, descripción extendida, ubicación o fecha si corresponde.

Este componente complementa la lista pública, ofreciendo a los usuarios una experiencia más completa e informativa.

**Creación del Componente de Administración**

El tercer paso consiste en implementar el **panel de administración** para esa sección. Este es un espacio restringido, diseñado únicamente para el administrador.

* **Archivo:** AdminNombreEnPlural.jsx
* **Lógica:**
  + Con useEffect se obtiene la lista completa de elementos existentes para mostrarlos en la interfaz de gestión.
  + Se crea un formulario controlado con useState, donde cada campo (título, descripción, imagen, etc.) está vinculado a un estado.
  + La función handleSubmit empaqueta los datos en un FormData y realiza una petición POST al API para crear un nuevo registro.
  + La función handleDelete solicita confirmación al usuario y, en caso afirmativo, envía una petición DELETE al API para eliminar un elemento específico.
* **Vista:** El panel muestra un formulario de creación en la parte superior y, debajo, una lista de registros ya creados, cada uno acompañado de su botón “Eliminar”.

Con este componente, el administrador obtiene control total sobre la información, pudiendo crear y borrar registros de manera rápida y segura.

1. **Creación del Componente de Administración**

El tercer paso consiste en implementar el **panel de administración** para esa sección. Este es un espacio restringido, diseñado únicamente para el administrador.

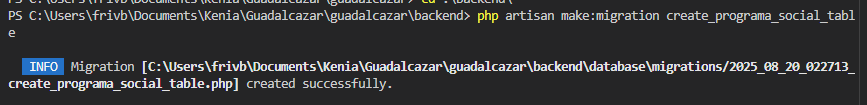
* **Archivo:** AdminNombreEnPlural.jsx
* **Lógica:**
  + Con useEffect se obtiene la lista completa de elementos existentes para mostrarlos en la interfaz de gestión.
  + Se crea un formulario controlado con useState, donde cada campo (título, descripción, imagen, etc.) está vinculado a un estado.
  + La función handleSubmit empaqueta los datos en un FormData y realiza una petición POST al API para crear un nuevo registro.
  + La función handleDelete solicita confirmación al usuario y, en caso afirmativo, envía una petición DELETE al API para eliminar un elemento específico.
* **Vista:** El panel muestra un formulario de creación en la parte superior y, debajo, una lista de registros ya creados, cada uno acompañado de su botón “Eliminar”.

Con este componente, el administrador obtiene control total sobre la información, pudiendo crear y borrar registros de manera rápida y segura.

Aquí un ejemplo de como creamos las paginas, prácticamente todas son muy parecidas incluso algunas en ciertas cosas son exactamente lo mismo solo varían un poco como por ejemplo la sección de deportes que se le agrego un formulario para que los participantes se puedan inscribir a los eventos deportivos.

**Creando la Estructura para "Noticias Culturales"**

Paso 1: Migración de la Base de Datos



Abrimos el nuevo archivo en database/migrations/ y define la tabla:  
Texto

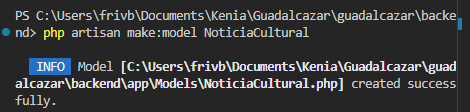
El contenido generado por IA puede ser incorrecto.

Después creamos la tabla en la base de datos:  
Texto

El contenido generado por IA puede ser incorrecto.

Paso 2: Modelo NoticiaCultural

Creamos el modelo correspondiente en la terminal:

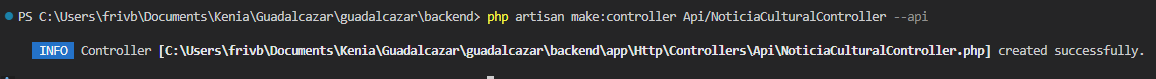


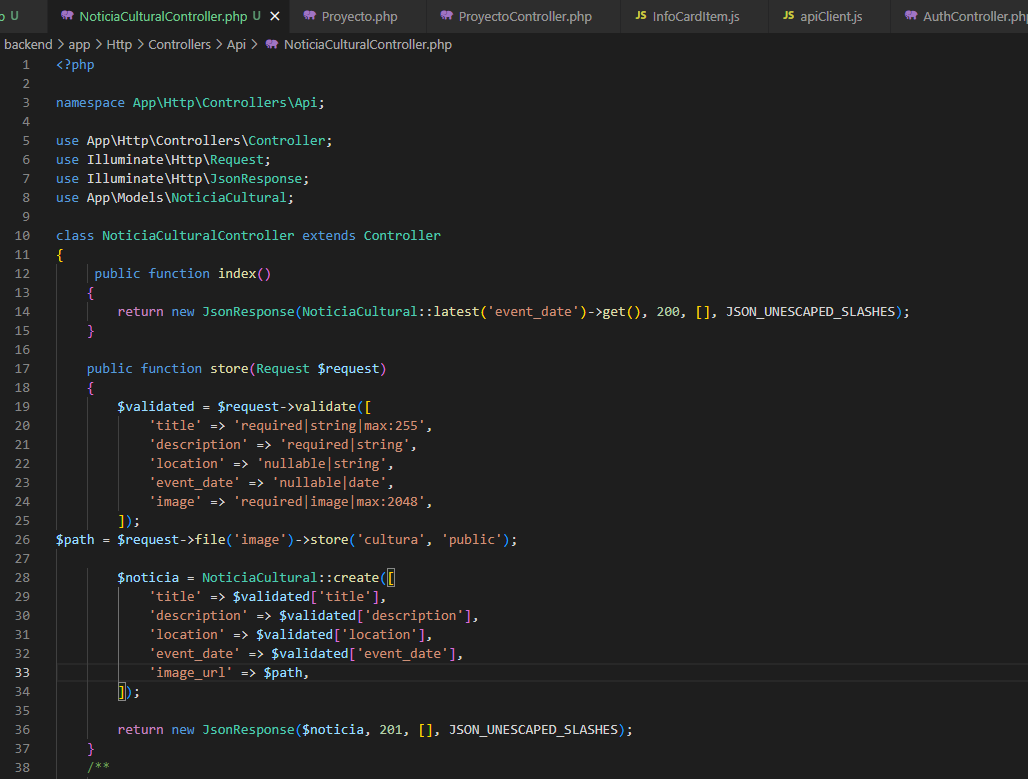
Abrimos app/Models/NoticiaCultural.php y lo configúramos:

Texto

El contenido generado por IA puede ser incorrecto.

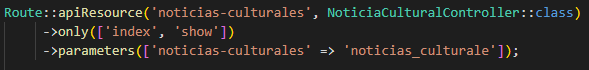
Creamos el controlador para manejar la lógica.

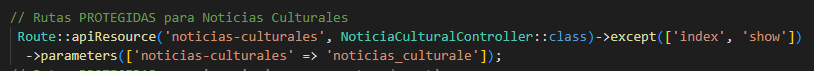


Abrimos app/Http/Controllers/Api/NoticiaCulturalController.php y configuramos el código:  


Ahora en routes/api.php y añadimos las nuevas rutas:

Primero agregamos  


En nuestras rutas publicas que no necesitan autenticación:  


Y en las rutas protegidas agregamos:  


**Fase 2: Frontend - Páginas Públicas y de Administración**

Paso 1: Componente Público Cultura.js

Creamos el archivo en nuestra carpeta de componentes para la vista publica  
Texto

El contenido generado por IA puede ser incorrecto.

Paso 2: Panel de Administración AdminCultura.js

Creamos el archivo AdminCultura.js para el panel de control:

Texto

El contenido generado por IA puede ser incorrecto.

Texto

El contenido generado por IA puede ser incorrecto.

Como funciona:

* **useEffect**: Al cargar la página, se hace una petición a la API para obtener todas las noticias culturales y se muestran en la lista.
* **handleSubmit**: Al enviar el formulario, se empaquetan todos los datos (incluida la imagen) en un FormData y se envían a la API para crear un nuevo registro. Si tiene éxito, actualiza la lista al instante.
* **handleDelete**: Al hacer clic en "Eliminar", pide confirmación y luego envía la petición a la API para borrar el registro. Si tiene éxito, lo quita de la lista.

Paso 3: Añadir las Rutas en App.js

Importamos  




En las rutas publicas:  
Pantalla de un video juego

El contenido generado por IA puede ser incorrecto.

Y en las rutas protegidas:

Logotipo

El contenido generado por IA puede ser incorrecto.

Resultado de la parte administrativa:

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

Resultado de la parte publica:  
Texto

El contenido generado por IA puede ser incorrecto.

Probando su funcionamiento:

En la vista del administrador rellenamos el formulario.

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

El contenido generado por IA puede ser incorrecto.

Y al momento de guardarlo  
Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

En la vista pública:  
Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

DashboardAdmin

Paso 1: Crear el Componente del Panel

Aqui unificaremos toda la parte administrativa

Paso 2: Actualizar las Rutas en App.js