



Recovery

CANCEL

MANUAL DE PROGRAMADOR

INTEGRADORA III

Arteaga Gayosso Ángel Jesús

López Navarrijo Laura Michelle

Pueblita Bautista Patrick Guillermo

Sánchez Terán Jonathan Daniel

Asesor. Dr. Téllez Barrientos Omar

Introducción

El siguiente manual de programador presenta MEDICONNECT una aplicación de gestión de citas médicas es un sistema diseñado para facilitar la programación, administración y seguimiento de consultas médicas entre pacientes y profesionales de la salud. Desarrollada con React en el frontend y Laravel en el backend, la plataforma proporciona una interfaz intuitiva y dinámica que permite a los usuarios interactuar de manera eficiente con el sistema.

El objetivo principal de la aplicación es optimizar el proceso de gestión de citas, permitiendo a los pacientes solicitar turnos de manera rápida y sencilla, mientras que los doctores y administradores pueden autorizar, rechazar y gestionar dichas solicitudes con un control detallado. La arquitectura del sistema sigue un modelo cliente-servidor, donde el frontend se encarga de la experiencia del usuario y la comunicación con la API del backend, el cual maneja la lógica de negocio y la persistencia de datos mediante PostgreSQL.

Para garantizar la seguridad y la gestión adecuada de los accesos, el sistema implementa autenticación basada en Laravel Sanctum, asegurando que cada usuario solo pueda acceder a las funciones correspondientes a su rol dentro de la plataforma. Además, la aplicación incorpora funcionalidades avanzadas, como la asignación de medicamentos por parte de los doctores y la generación de recetas médicas, lo que permite un seguimiento integral de la atención brindada. Este documento servirá como una guía detallada para comprender la arquitectura, la configuración y el desarrollo del sistema, proporcionando información clave sobre los componentes esenciales, los endpoints API y las mejores prácticas para su mantenimiento y escalabilidad.

Manual de Programador

Requisitos del Sistema

Para el desarrollo y ejecución de la aplicación de gestión de citas médicas, es necesario contar con un entorno de software y hardware adecuado que permita el correcto funcionamiento del sistema. A continuación, se detallan los requisitos mínimos recomendados.

1. Requisitos de Hardware

Para garantizar un rendimiento óptimo, se recomienda utilizar un equipo con las siguientes especificaciones:

- Procesador: Intel Core i5 o superior.
- Memoria RAM: 8 GB como mínimo (se recomienda 16 GB para un mejor desempeño).
- Espacio en disco: Al menos 10 GB de almacenamiento disponible.

Conexión a Internet para la instalación de dependencias y configuración del entorno.

2. Software Necesario

Backend - Laravel

El sistema requiere la instalación de los siguientes componentes para el desarrollo y ejecución del backend:

- PHP en su versión 8.1 o superior.
- Composer, el administrador de dependencias de PHP.
- Laravel, el framework utilizado para la construcción del backend.
- PostgreSQL, el sistema de gestión de bases de datos.
- pgAdmin, la interfaz gráfica utilizada para la administración de la base de datos.
- Extensión PHP para PostgreSQL (pdo_pgsql).

Frontend - React

Para la implementación del frontend, es necesario contar con las siguientes herramientas y dependencias:

- Node.js en su versión 18.x o superior.
- npm o yarn, los administradores de paquetes de Node.js.
- Vite, herramienta utilizada para la configuración del entorno de desarrollo en React.
- Axios, cliente HTTP que facilita la comunicación con la API.
- React Router, librería utilizada para la navegación entre vistas en la aplicación.

Herramientas Adicionales

- Postman, software para realizar pruebas de la API REST.
- Git, sistema de control de versiones para el seguimiento del desarrollo.
- Visual Studio Code, editor de código recomendado para el desarrollo del proyecto.

Al contar con estos requisitos, el entorno de desarrollo estará listo para la implementación, configuración y ejecución de la aplicación.

Arquitectura del Proyecto

La aplicación de gestión de citas médicas está diseñada bajo una arquitectura basada en el patrón Cliente-Servidor, donde el frontend y el backend operan de manera independiente y se comunican a través de una API REST. Esta separación permite escalabilidad, facilidad de mantenimiento y reutilización de los componentes.

Arquitectura General

La aplicación se compone de tres capas principales:

Capa de Presentación (Frontend - React)

- Desarrollada con React, utilizando Vite como herramienta de configuración.
- Se encarga de la interfaz gráfica y la interacción con el usuario.
- Implementa React Router para la navegación entre vistas.
- Se comunica con el backend mediante Axios para realizar solicitudes a la API REST.
- Gestiona el estado de la aplicación con el uso de useState y useEffect.

Capa de Lógica de Negocio (Backend - Laravel)

- Construida con Laravel, un framework basado en PHP.
- Expone una API REST que permite la comunicación con el frontend.
- Implementa controladores para gestionar las solicitudes HTTP y procesar la lógica del negocio.

Capa de Datos (Base de Datos - PostgreSQL)

- Base de datos relacional gestionada con PostgreSQL.
- Se estructura en tablas que almacenan información sobre usuarios, citas médicas, médicos, pacientes y otros datos relevantes.
- Utiliza migraciones de Laravel para la gestión y versión del esquema de la base de datos.
- Accesible mediante pgAdmin para administración y consultas manuales.

Flujo de Datos en la Aplicación

El flujo de datos en la aplicación sigue el siguiente proceso:

- El usuario accede a la interfaz de React y realiza una acción (ejemplo: solicitar una cita).
- React envía una solicitud HTTP al backend mediante Axios.
- Laravel recibe la solicitud y la procesa a través de los controladores correspondientes.
- Si es necesario, Laravel consulta o actualiza la base de datos utilizando Eloquent ORM.
- Laravel responde con los datos solicitados o con un mensaje de éxito/error.
- React recibe la respuesta y actualiza la interfaz de usuario en tiempo real.

Base de Datos

El diagrama relacional de la base de datos representa la estructura de almacenamiento de la información dentro del sistema, detallando las tablas, sus atributos y las relaciones entre ellas. En este modelo, se organiza la información necesaria para la gestión de citas médicas, garantizando la integridad y consistencia de los datos.

Cada tabla en el diagrama tiene un propósito específico dentro del sistema. Por ejemplo, la tabla de usuarios almacena las credenciales y roles de acceso, mientras que la tabla de pacientes guarda la información personal de quienes solicitan citas médicas.

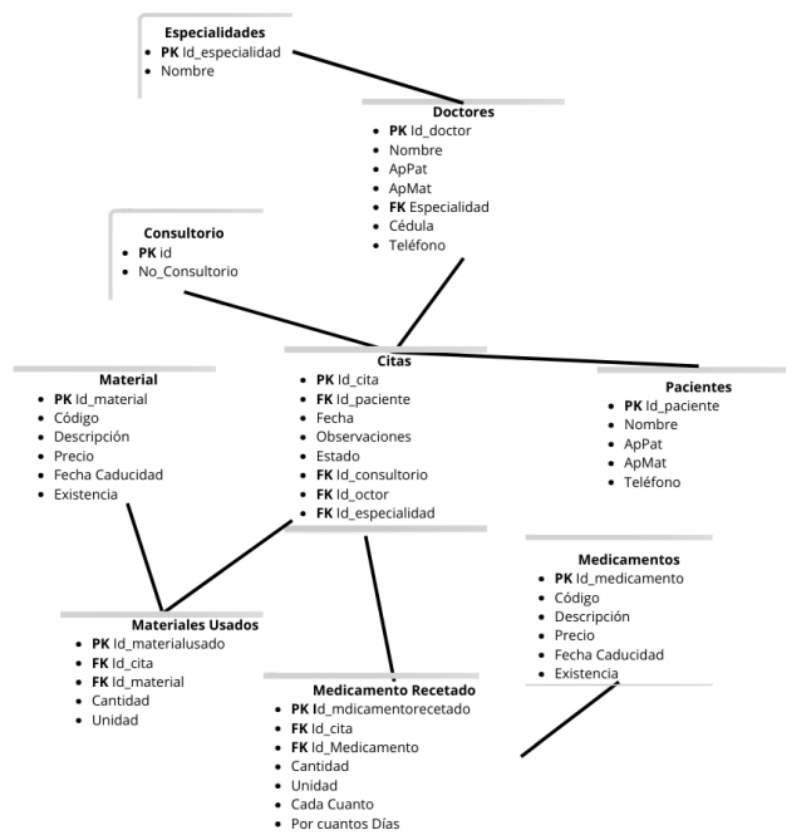


Figura 1.1 Modelo Relacional

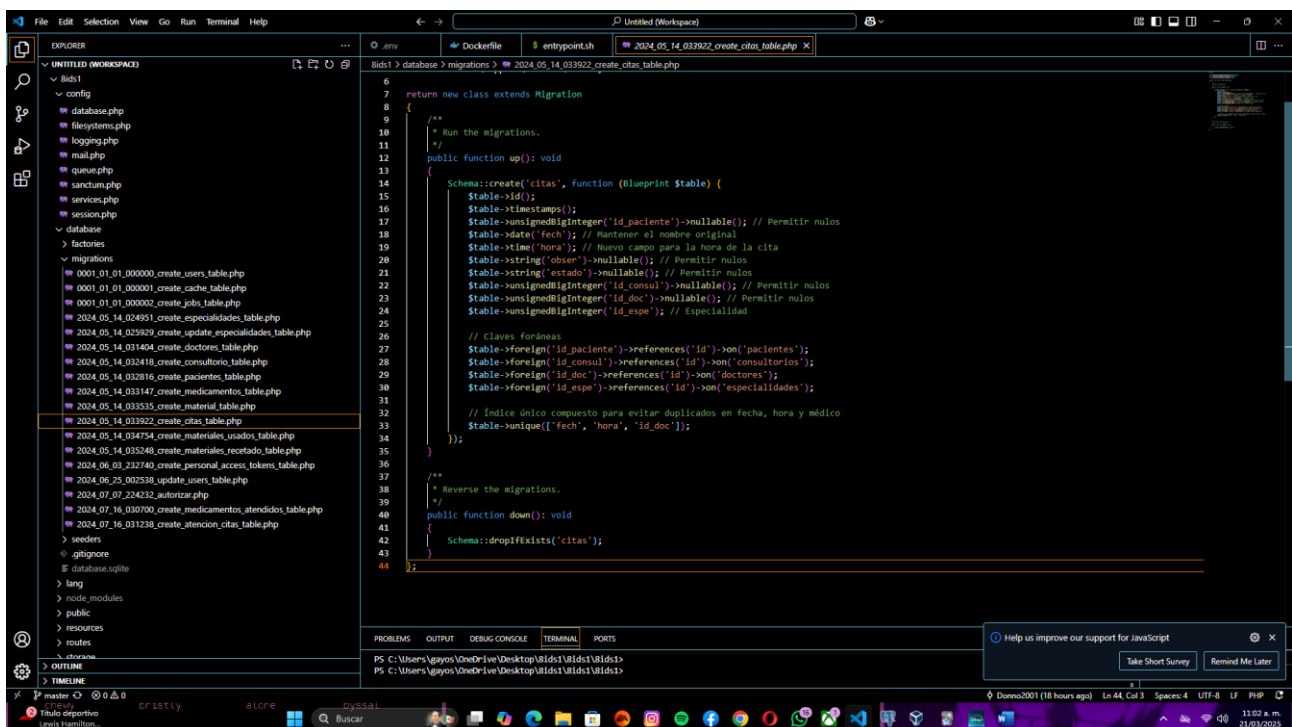
Backend

Migraciones

Las migraciones en el proyecto se utilizan para gestionar de manera eficiente la estructura de la base de datos, permitiendo la creación, modificación y eliminación de tablas de forma controlada y versionada. A través de las migraciones, se asegura que la base de datos esté sincronizada con el código del proyecto en cualquier entorno, ya sea de desarrollo, pruebas o producción.

Cada migración define las transformaciones necesarias en la estructura de la base de datos, como la creación de tablas, la adición de columnas o la modificación de tipos de datos. Estas migraciones se encuentran ubicadas en el directorio `database/migrations` dentro del proyecto Laravel, y pueden ser ejecutadas mediante el uso de comandos proporcionados por el framework.

En este caso, se muestra la migración de la tabla de citas, la cual ha sido implementada en Laravel con sus atributos coherencia del sistema.



```

6  return new class extends Migration
7  {
8
9
10     /**
11      * Run the migrations.
12      */
13     public function up(): void
14     {
15         Schema::create('citas', function (Blueprint $table) {
16             $table->id();
17             $table->timestamps();
18             $table->foreign('id_paciente')->nullable(); // Permitir nulos
19             $table->date('fecha'); // Mantener el nombre original
20             $table->time('hora'); // Nuevo campo para la hora de la cita
21             $table->string('estado')->nullable(); // Permitir nulos
22             $table->unsignedBigInteger('id_consul')->nullable(); // Permitir nulos
23             $table->unsignedBigInteger('id_doc')->nullable(); // Permitir nulos
24             $table->unsignedBigInteger('id_espe')->nullable(); // Especialidad
25
26             // Claves foráneas
27             $table->foreign('id_paciente')->references('id')->on('pacientes');
28             $table->foreign('id_consul')->references('id')->on('consultorios');
29             $table->foreign('id_doc')->references('id')->on('doctores');
30             $table->foreign('id_espe')->references('id')->on('especialidades');
31
32             // Índice único compuesto para evitar duplicados en fecha, hora y médico
33             $table->unique(['fecha', 'hora', 'id_doc']);
34         });
35     }
36
37     /**
38      * Reverse the migrations.
39      */
40     public function down(): void
41     {
42         Schema::dropIfExists('citas');
43     }
44 }

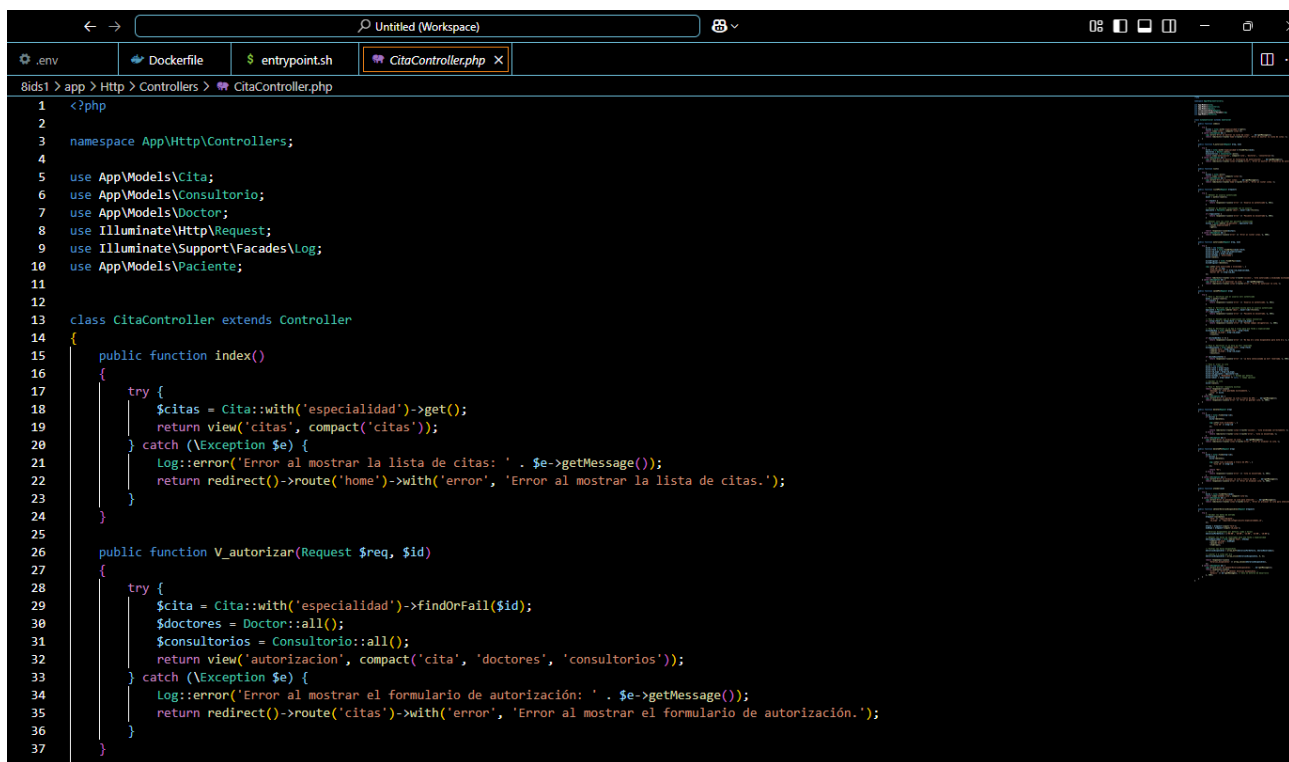
```

Figura 1.2 Migración Citas

Controladores

Los controladores en Laravel son responsables de manejar las solicitudes entrantes del usuario y procesarlas, delegando la lógica de negocio a los modelos y pasando los resultados a las vistas para que sean presentados al usuario.

En términos simples, un controlador recibe la solicitud HTTP (como una solicitud de usuario para ver o actualizar una cita), interactúa con los modelos para obtener o modificar los datos, y luego pasa esos datos a las vistas para su presentación. Los controladores también son responsables de validar los datos y gestionar las respuestas de error.



```

1  <?php
2
3  namespace App\Http\Controllers;
4
5  use App\Models\Cita;
6  use App\Models\Consultorio;
7  use App\Models\Doctor;
8  use Illuminate\Http\Request;
9  use Illuminate\Support\Facades\Log;
10 use App\Models\Paciente;
11
12
13 class CitaController extends Controller
14 {
15     public function index()
16     {
17         try {
18             $citas = Cita::with('especialidad')->get();
19             return view('citas', compact('citas'));
20         } catch (\Exception $e) {
21             Log::error('Error al mostrar la lista de citas: ' . $e->getMessage());
22             return redirect()->route('home')->with('error', 'Error al mostrar la lista de citas.');
```

Figura 1.3 Controlador Citas

En el controlador de Citas, se presentan varios métodos esenciales que permiten gestionar las citas médicas. Entre estos métodos, se incluyen funcionalidades para enlistar las citas, autorizar las citas y guardar nuevas citas. Estos métodos se implementan tanto en el backend de Laravel como en las APIs que permiten que los datos sean consumidos por el frontend desarrollado con React. De esta forma, el controlador maneja eficientemente la lógica de negocio, interactuando con la base de datos y facilitando la comunicación con la interfaz de usuario.

```

14 {
102 }
103
104 public function saveAPI(Request $req)
105 {
106     try {
107         // Paso 1: Verificar que el usuario esté autenticado
108         $user = auth()->user();
109         if (!$user) {
110             return response()->json(['error' => 'Usuario no autenticado.'], 401);
111         }
112
113         // Paso 2: Verificar que el paciente exista para el usuario autenticado
114         $paciente = Paciente::where('idusr', $user->id)->first();
115         if (!$paciente) {
116             return response()->json(['error' => 'Paciente no encontrado.'], 404);
117         }
118
119         // Paso 3: Validar que se proporcionen los campos necesarios
120         if (!$req->fech || !$req->hora || !$req->id_espe) {
121             return response()->json(['error' => 'Faltan campos obligatorios.'], 400);
122         }
123
124         // Paso 4: Verificar si ya hay 4 citas para esa fecha y especialidad
125         $citasDelDia = Cita::where('fech', $req->fech)
126             ->where('id_espe', $req->id_espe)
127             ->count();
128
129         if ($citasDelDia >= 4) {
130             return response()->json(['error' => 'No hay más citas disponibles para este día.'], 400);
131         }
132
133         // Paso 5: Verificar si la hora ya está reservada
134         $citaExistente = Cita::where('fech', $req->fech)
135             ->where('hora', $req->hora)
136             ->where('id_espe', $req->id_espe)
137             ->exists();
138
139         if ($citaExistente) {
140             return response()->json(['error' => 'La hora seleccionada ya está reservada.'], 400);
141         }
142
143         // Paso 6: Crear la cita
144         $cita = new Cita();
145         $cita->fech = $req->fech;
146         $cita->hora = $req->hora;
147         $cita->id_espe = $req->id_espe;
148         $cita->id_usr = $user->id;
149         $cita->save();
150     } catch (Exception $e) {
151         return response()->json(['error' => $e->getMessage()], 500);
152     }
153 }

```

Figura 1.4 Apis

Modelos

En Laravel, los modelos representan las entidades que interactúan directamente con la base de datos. Cada modelo corresponde a una tabla específica en la base de datos y facilita la manipulación de los datos de esa tabla.

En el contexto de este proyecto, los modelos gestionan entidades clave como Cita, Paciente y Doctor. Además de representar las tablas de la base de datos, los modelos también definen las relaciones entre las entidades, tales como las relaciones uno a muchos o muchos a muchos, lo que permite obtener de manera sencilla información relacionada, como las citas de un paciente o los doctores asignados a una cita.

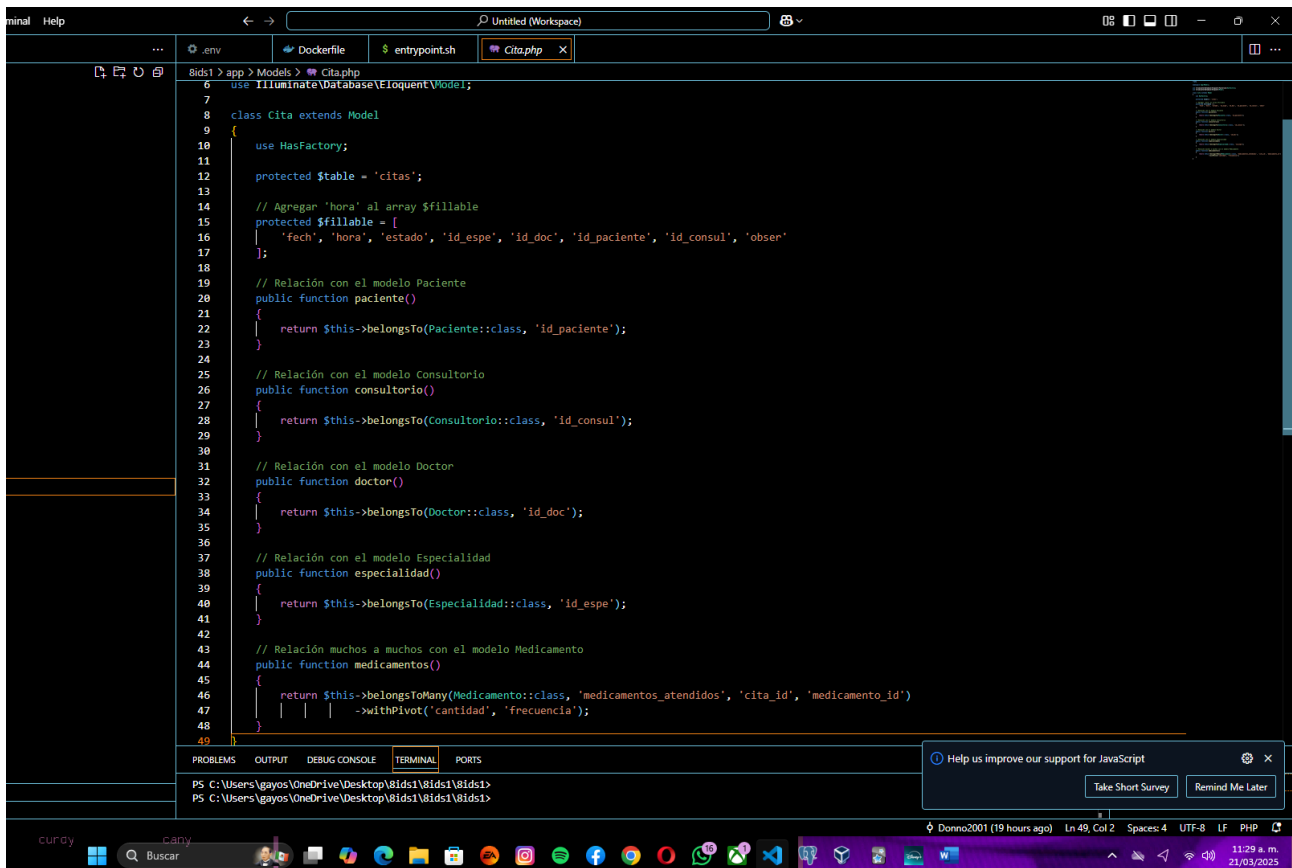
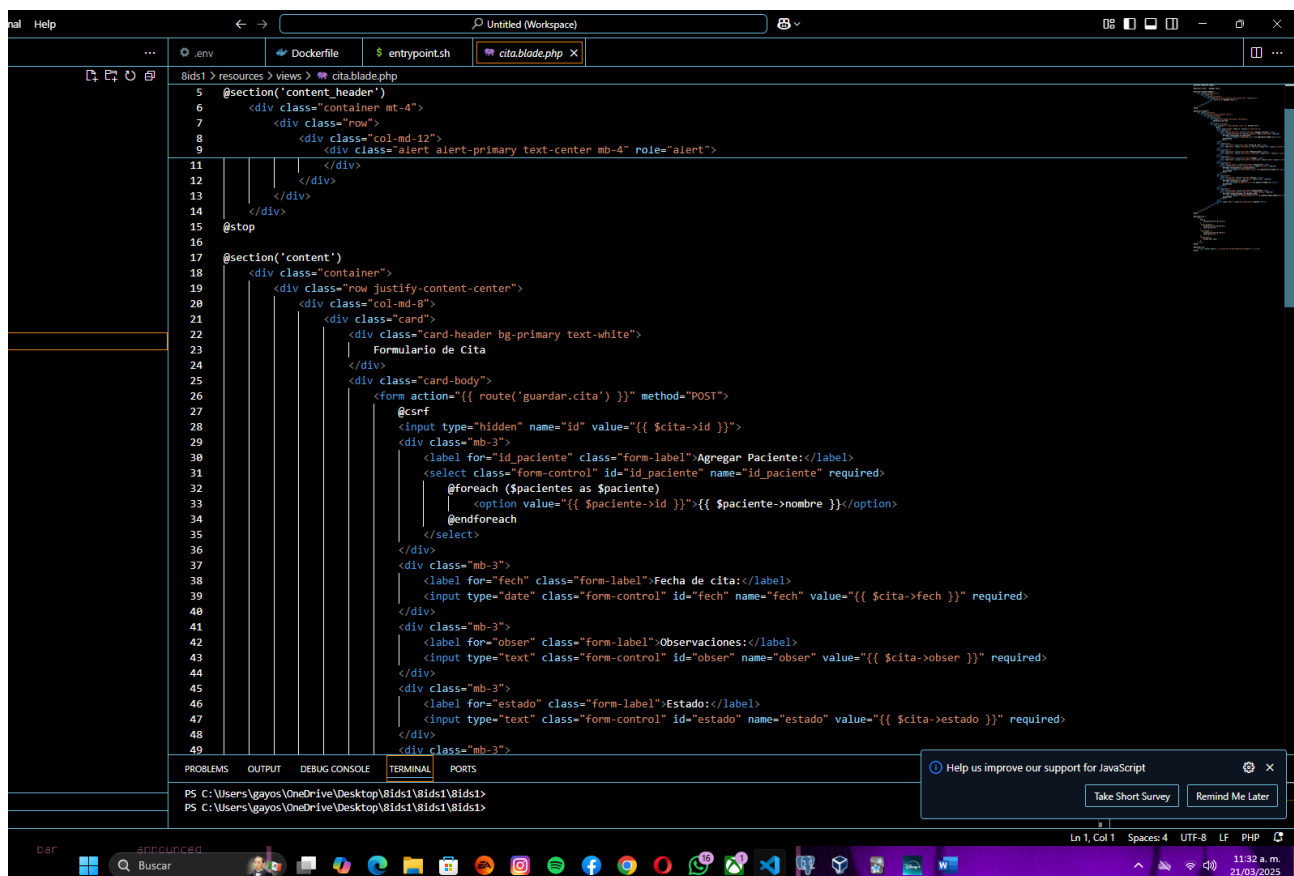


Figura 1.5 Modelo Cita

Vistas

Las vistas en Laravel son responsables de la presentación de los datos al usuario. Son archivos que contienen el código HTML, CSS y JavaScript necesario para mostrar la información de manera estructurada y visualmente atractiva. En el proyecto, las vistas permiten que la lógica de la aplicación se separe de la presentación, lo que facilita la organización y el mantenimiento del código.

En este contexto, las vistas gestionan la interfaz de usuario que interactúa con las APIs y controladores de Laravel, mostrando las citas, permitiendo la autorización de citas, y gestionando otras funcionalidades del sistema. Las vistas se encuentran en el directorio resources/views y están construidas principalmente con Blade, el motor de plantillas de Laravel, que permite una sintaxis sencilla para mezclar PHP con HTML.



```

5 @section('content_header')
6 <div class="container mt-4">
7 <div class="row">
8 <div class="col-md-12">
9 <div class="alert alert-primary text-center mb-4" role="alert">
10
11 </div>
12 </div>
13 </div>
14 </div>
15 @stop
16
17 @section('content')
18 <div class="container">
19 <div class="row justify-content-center">
20 <div class="col-md-8">
21 <div class="card">
22 <div class="card-header bg-primary text-white">
23 Formulario de Cita
24 </div>
25 <div class="card-body">
26 <form action="{{ route('guardar.cita') }}" method="POST">
27 @csrf
28 <input type="hidden" name="id" value="{{ $cita->id }}">
29 <div class="mb-3">
30 <label for="id_paciente" class="form-label">Agregar Paciente:</label>
31 <select class="form-control" id="id_paciente" name="id_paciente" required>
32 @foreach ($pacientes as $paciente)
33 <option value="{{ $paciente->id }}">{{ $paciente->nombre }}</option>
34 @endforeach
35 </select>
36 </div>
37 <div class="mb-3">
38 <label for="fech" class="form-label">Fecha de cita:</label>
39 <input type="date" class="form-control" id="fech" name="fech" value="{{ $cita->fech }}" required>
40 </div>
41 <div class="mb-3">
42 <label for="obser" class="form-label">Observaciones:</label>
43 <input type="text" class="form-control" id="obser" name="obser" value="{{ $cita->obser }}" required>
44 </div>
45 <div class="mb-3">
46 <label for="estado" class="form-label">Estado:</label>
47 <input type="text" class="form-control" id="estado" name="estado" value="{{ $cita->estado }}" required>
48 </div>
49 </div>
50 </div>
51 </div>
52 </div>

```

Figura 1.6 Vista Cita

Frontend

En el proyecto, React interactúa con el backend en Laravel a través de APIs. Esto permite que las solicitudes del usuario, como la creación de citas, la visualización de citas existentes, y la autorización de procedimientos médicos, se realicen de manera eficiente. Los datos se obtienen del servidor Laravel utilizando fetch o Axios, que se encargan de realizar las solicitudes HTTP necesarias y luego actualizar el estado de la aplicación de manera automática. Esto permite que el frontend se mantenga siempre sincronizado con los datos más recientes de la base de datos.

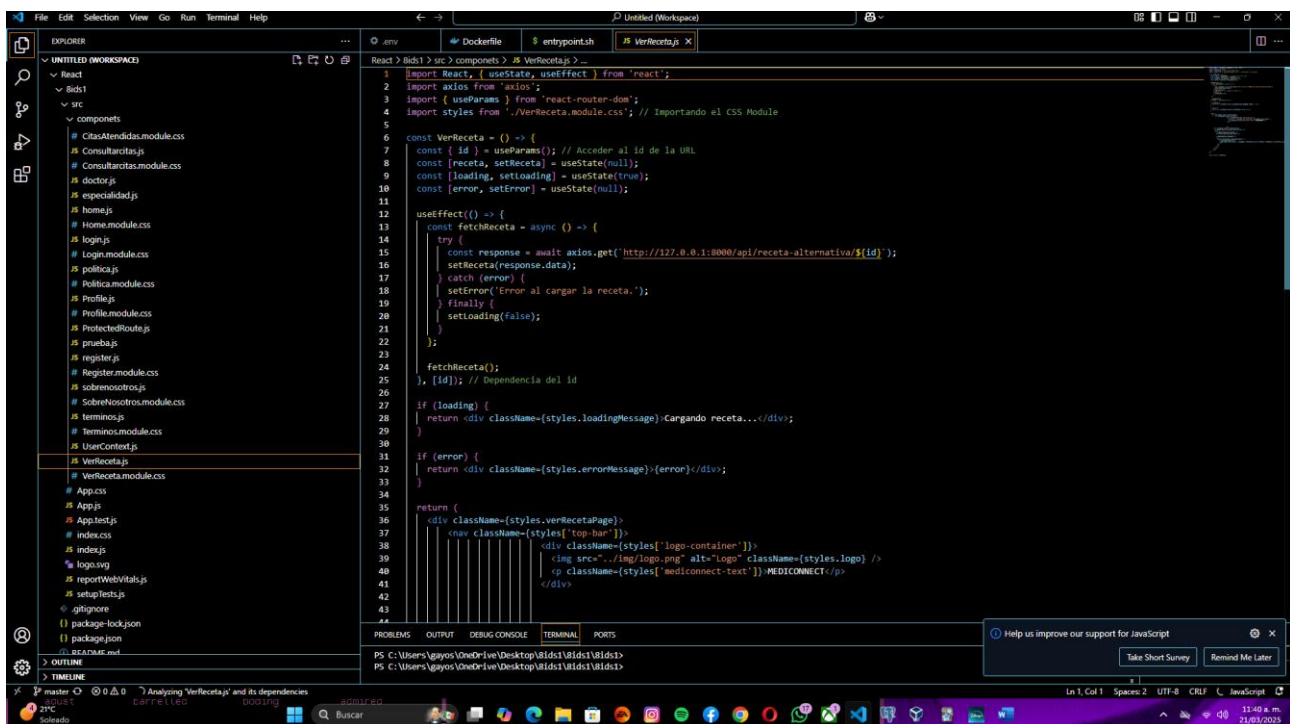


Figura 1.7 React

Vistas

La carpeta components/ en un proyecto React contiene los componentes reutilizables que forman las diferentes partes de la interfaz de usuario. Cada componente es un archivo JavaScript (.js o .jsx) que puede contener lógica y presentación de una sección específica de la aplicación. Los componentes son la unidad básica en React, y se utilizan para dividir la UI en partes pequeñas y manejables.

Cada archivo dentro de la carpeta components/ representa un componente de React que puede ser utilizado en otras partes de la aplicación. Los componentes pueden ser simples, como un botón o una tarjeta, o más complejos, como formularios o listas de elementos. Estos componentes pueden recibir props (propiedades) para personalizar su contenido y comportamiento, y pueden tener su propio estado interno para gestionar cambios interactivos.

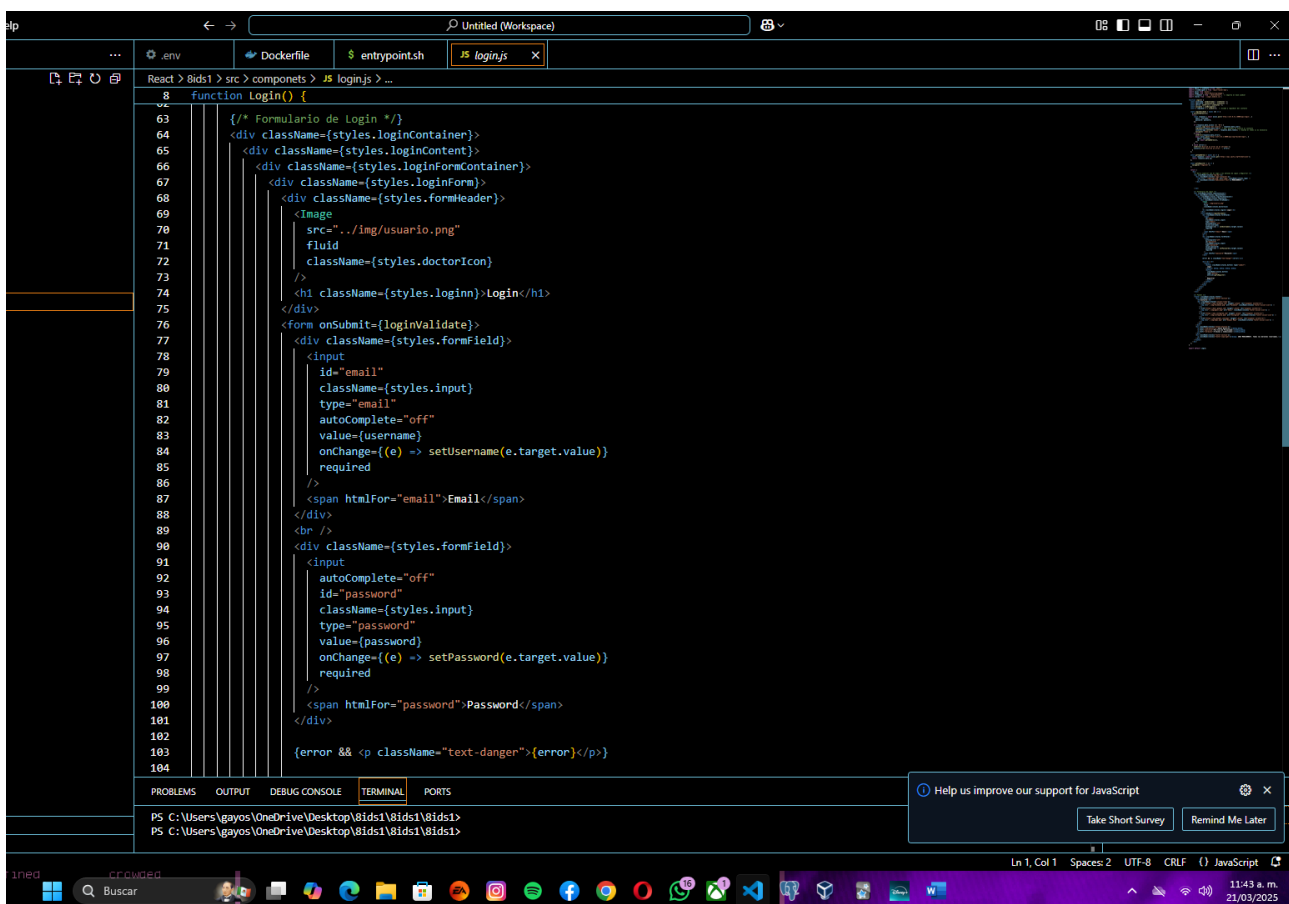


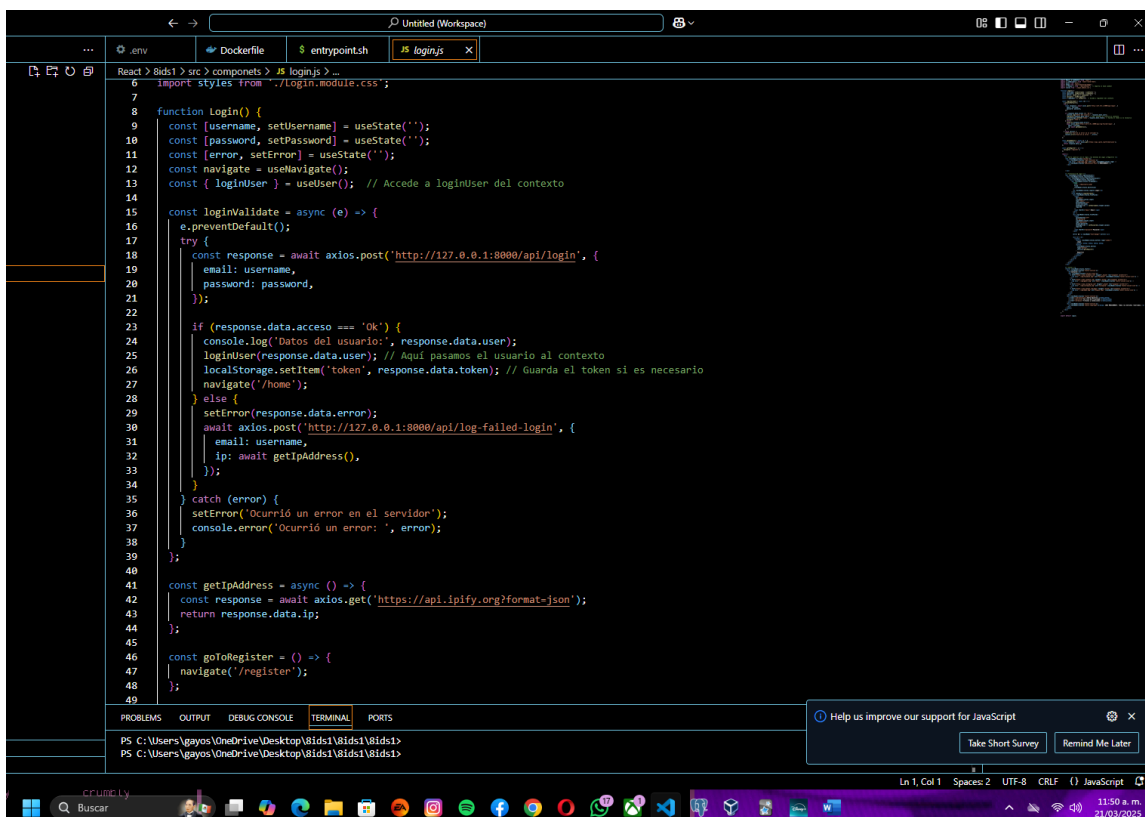
Figura 1.8 Vista Login

APIS

React consume las APIs de Laravel para interactuar con el backend y obtener datos, así como enviar información para su procesamiento. Laravel actúa como el servidor, proporcionando las rutas API que permiten a React realizar solicitudes HTTP (GET, POST, PUT, DELETE) y obtener o enviar datos de la base de datos.

Para consumir las APIs de Laravel desde React, se utilizan herramientas como Axios o la función nativa fetch de JavaScript. Estas herramientas permiten enviar solicitudes HTTP a las rutas definidas en Laravel, y luego manejar las respuestas dentro de React.

En el frontend, React realiza una solicitud HTTP a estas rutas utilizando Axios o fetch. Al hacer una solicitud GET, React recibe los datos en formato JSON, que luego se procesan y se utilizan para actualizar el estado de la aplicación, lo que genera la actualización en la interfaz de usuario.



```

6 import styles from './Login.module.css';
7
8 function login() {
9   const [username, setUsername] = useState('');
10  const [password, setPassword] = useState('');
11  const [error, setError] = useState('');
12  const navigate = useNavigate();
13  const { loginUser } = useUser(); // Accede a loginUser del contexto
14
15  const loginValidate = async (e) => {
16    e.preventDefault();
17    try {
18      const response = await axios.post('http://127.0.0.1:8000/api/login', {
19        email: username,
20        password: password,
21      });
22
23      if (response.data.acceso === 'Ok') {
24        console.log('Datos del usuario:', response.data.user);
25        loginUser(response.data.user); // Aquí pasamos el usuario al contexto
26        localStorage.setItem('token', response.data.token); // Guarda el token si es necesario
27        navigate('/home');
28      } else {
29        setError(response.data.error);
30        await axios.post('http://127.0.0.1:8000/api/log-failed-login', {
31          email: username,
32          ip: await getIpAddress(),
33        });
34      }
35    } catch (error) {
36      setError('Ocurrió un error en el servidor');
37      console.error('Ocurrió un error: ', error);
38    }
39  };
40
41  const getIpAddress = async () => {
42    const response = await axios.get('https://api.ipify.org?format=json');
43    return response.data.ip;
44  };
45
46  const goToRegister = () => {
47    navigate('/register');
48  };
49

```

Figura 1.9 Apis en Login

Configuración de despliegue

El proceso de despliegue del proyecto en Render fue estructurado de manera eficiente para asegurar un despliegue exitoso y funcional. Primero, se configuró el proyecto para ser subido a GitHub, facilitando la gestión del código fuente y el control de versiones. Con el repositorio en GitHub listo, se procedió a crear un archivo Dockerfile para contenerizar la aplicación, asegurando que la aplicación Laravel y su entorno de ejecución estuvieran correctamente configurados y pudieran ejecutarse de manera consistente en cualquier entorno.

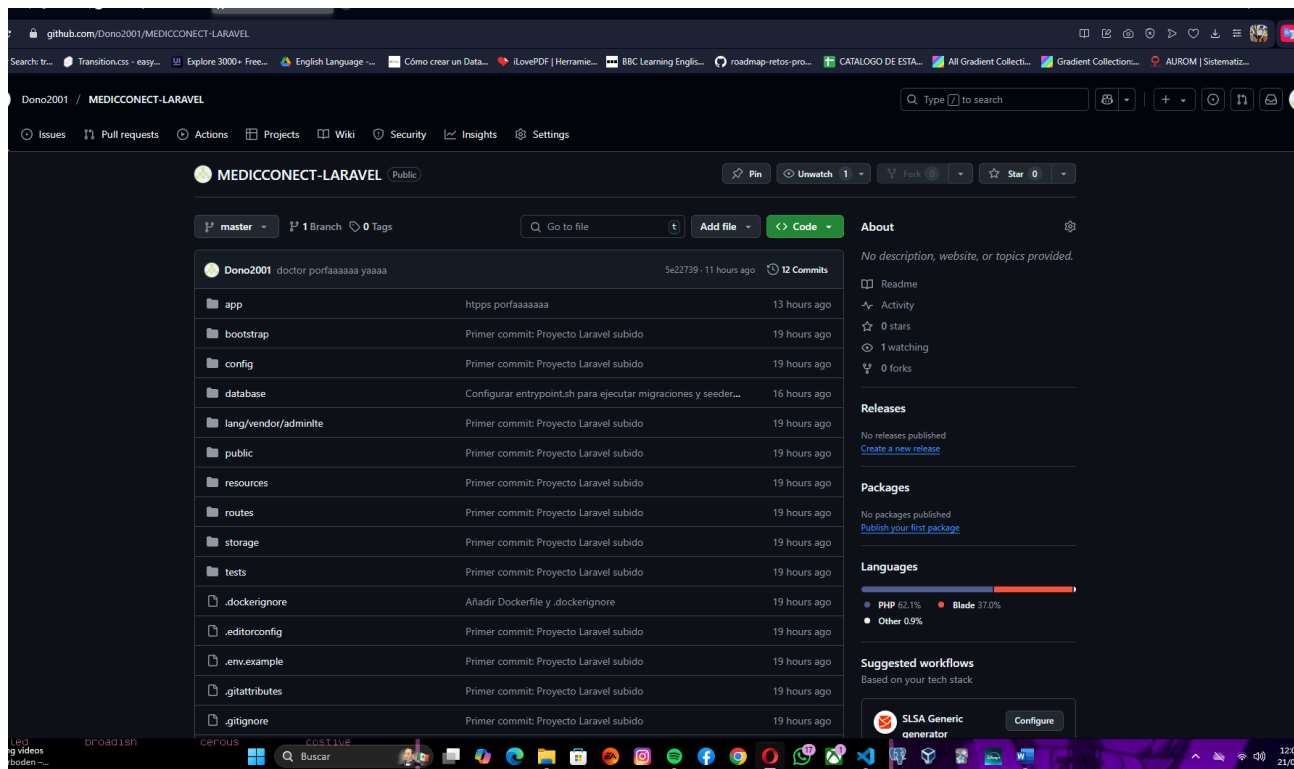


Figura 1.10 Repositorio

El archivo **Dockerfile** se construyó sobre una base de **PHP con Apache** (versión 8.2), en la que se instaló las dependencias necesarias, como **Git**, **unzip**, y las extensiones PHP requeridas por **Laravel** y **PostgreSQL**. Además, se incluyó **Composer** para gestionar las dependencias de PHP. Los archivos del proyecto fueron copiados al contenedor, configurando el directorio raíz de **Apache** para que apunte a la carpeta **public** de Laravel, como es usual en las aplicaciones de este framework.

```

1 # Usar una imagen base de PHP con Apache
2 FROM php:8.2-apache
3
4 # Establecer el directorio de trabajo
5 WORKDIR /var/www/html
6
7 # Instalar dependencias del sistema
8 RUN apt-get update && apt-get install -y \
9     git \
10    unzip \
11    libzip-dev \
12    libpng-dev \
13    libonig-dev \
14    libxml2-dev \
15    libpq-dev
16
17 # Instalar extensiones de PHP necesarias para Laravel y PostgreSQL
18 RUN docker-php-ext-install pdo pdo_pgsql zip mbstring exif pcntl bcmath gd
19
20 # Instalar Composer
21 COPY --from=composer:latest /usr/bin/composer /usr/bin/composer
22
23 # Copiar los archivos del proyecto
24 COPY . .
25
26 # Configurar el document root de Apache
27 ENV APACHE_DOCUMENT_ROOT /var/www/html/public
28 RUN sed -ri -e 's!/var/www/html!${APACHE_DOCUMENT_ROOT}!g' /etc/apache2/sites-available/*.conf
29 RUN sed -ri -e 's!/var/www/html!${APACHE_DOCUMENT_ROOT}!g' /etc/apache2/apache2.conf /etc/apache2/conf-available/*.conf
30
31 # Instalar dependencias de Composer (sin incluir las dependencias de desarrollo)
32 RUN composer install --optimize-autoloader --no-dev
33
34 # Ajustar permisos en el directorio storage y bootstrap/cache
35 RUN chmod -R 775 /var/www/html/storage /var/www/html/bootstrap/cache && \
36     chown -R www-data:www-data /var/www/html/storage /var/www/html/bootstrap/cache
37
38 # Copiar el script de entrypoint
39 COPY entrypoint.sh /entrypoint.sh
40 RUN chmod +x /entrypoint.sh
41
42 # Habilitar módulos de Apache necesarios
43 RUN a2enmod rewrite
44

```

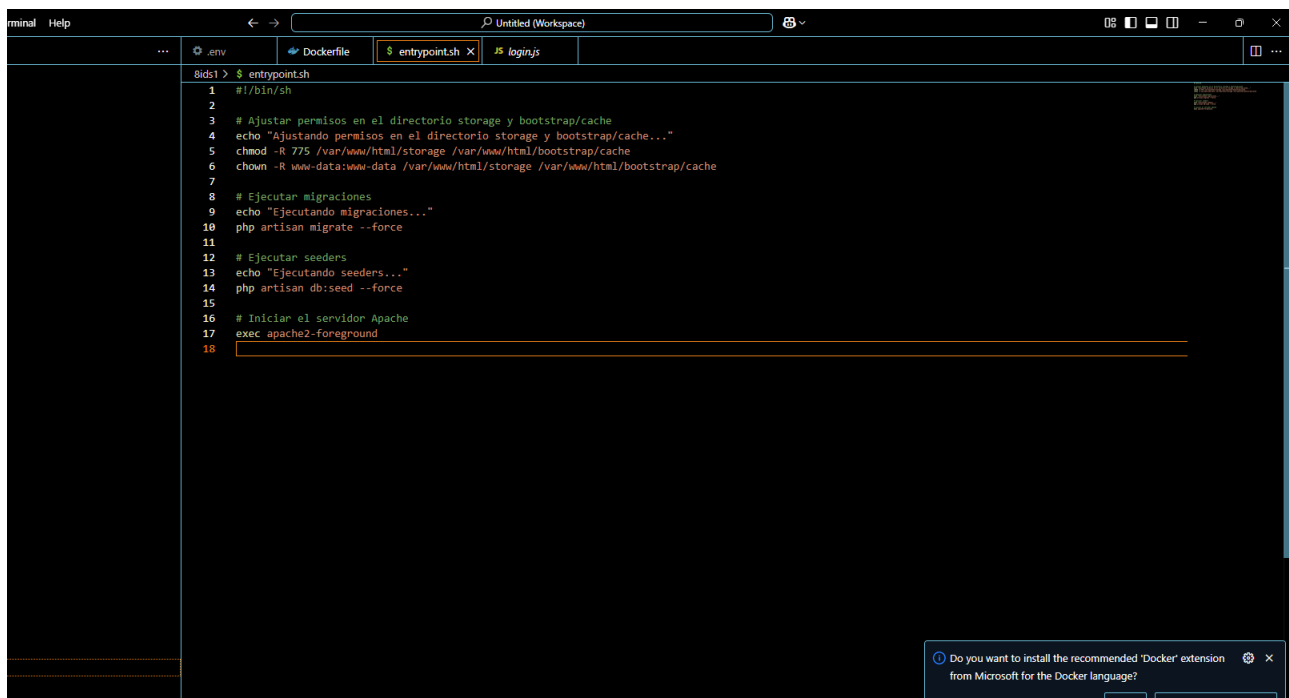
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\lgayos\OneDrive\Desktop\8ids1\8ids1>
PS C:\Users\lgayos\OneDrive\Desktop\8ids1\8ids1>

Figura 1.11 Archivo Docker

Dentro de este archivo, también se incluyó la instalación de las dependencias de Composer con los parámetros adecuados para optimizar el autoloading y evitar incluir dependencias de desarrollo. Para garantizar el funcionamiento adecuado del sistema de almacenamiento de Laravel, se ajustaron los permisos de las carpetas **storage** y **bootstrap/cache**, dándoles los permisos necesarios para su funcionamiento dentro del contenedor.

Además, se creó un script de **entrypoint** para ejecutar tareas necesarias al iniciar el contenedor, como las migraciones de base de datos y los seeders, asegurando que la base de datos estuviera configurada antes de ejecutar la aplicación. En cuanto a la configuración de **Apache**, se habilitaron los módulos necesarios, como el módulo **rewrite** para permitir el correcto funcionamiento de las rutas de Laravel.



```

1  #!/bin/sh
2
3  # Ajustar permisos en el directorio storage y bootstrap/cache
4  echo "Ajustando permisos en el directorio storage y bootstrap/cache..."
5  chmod -R 775 /var/www/html/storage /var/www/html/bootstrap/cache
6  chown -R www-data:www-data /var/www/html/storage /var/www/html/bootstrap/cache
7
8  # Ejecutar migraciones
9  echo "Ejecutando migraciones..."
10 php artisan migrate --force
11
12 # Ejecutar seeders
13 echo "Ejecutando seeders..."
14 php artisan db:seed --force
15
16 # Iniciar el servidor Apache
17 exec apache2-foreground
18

```

Figura 1.12 Entrypoint

Una vez preparado el contenedor, se procedió a crear la base de datos en **Render**. Se declararon las variables de entorno necesarias para la aplicación dentro de la interfaz de Render, como las credenciales de la base de datos, claves de API y demás configuraciones específicas del entorno de producción. Finalmente, se subió el proyecto a **Render**, configurando los pasos de despliegue y asegurando que todos los parámetros estuvieran correctamente establecidos para que el proyecto funcionara sin problemas. Este enfoque permitió un despliegue automatizado, eliminando la necesidad de gestionar servidores manualmente y asegurando que la aplicación Laravel estuviera operativa en la nube con una configuración óptima para producción.

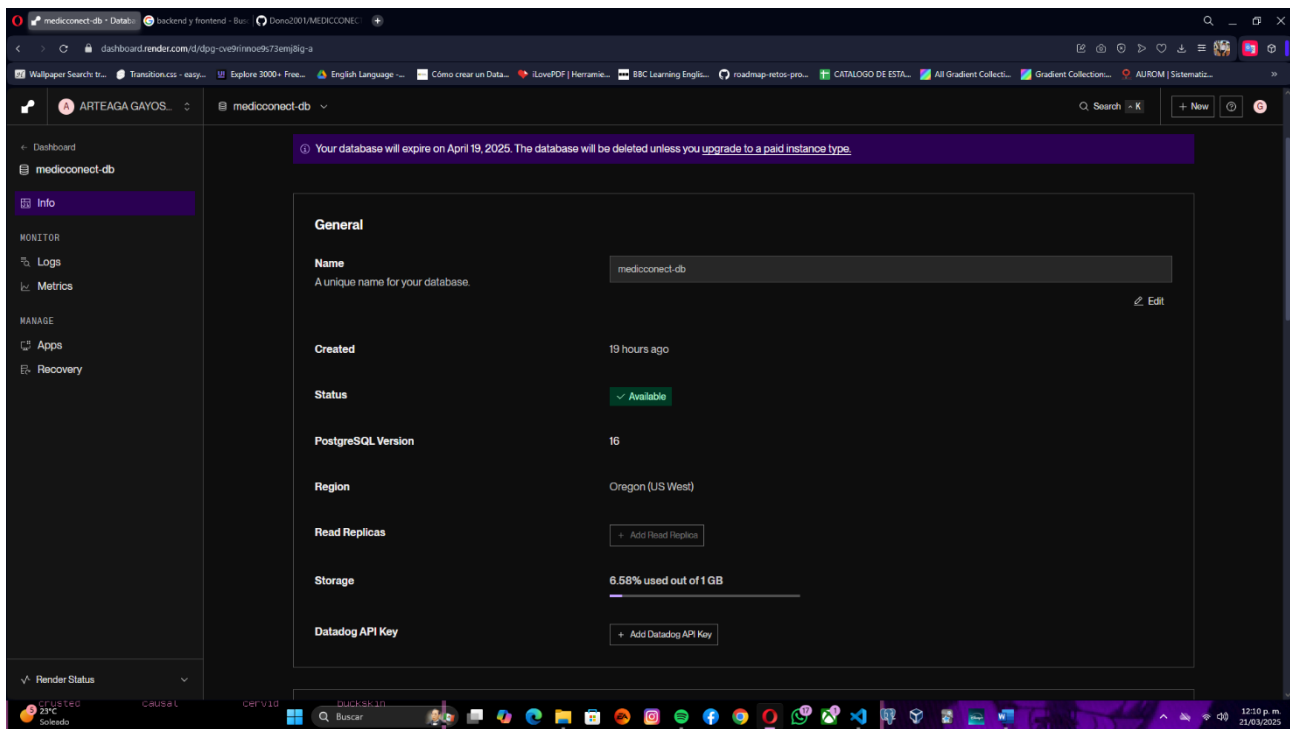


Figura 1.13 BD en Render

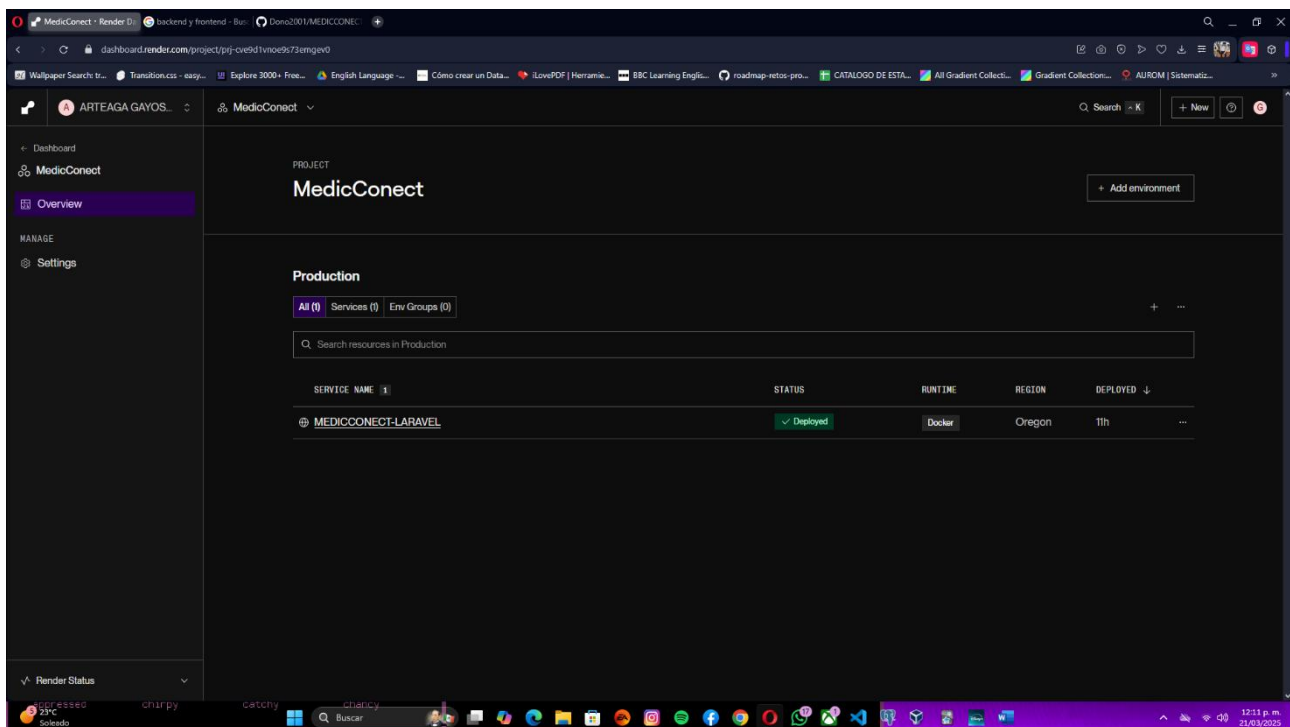


Figura 1.14 Despliegue Backend