

Documentación técnica - Estaciones

Desarrollo web en entorno
servidor

Francisco Javier Arruabarrena Sabroso
Carlos Blanco Hidalgo

Índice

Documentación técnica del módulo	2
Relación de archivos y ubicación y breve descripción de su función	2
Secciones de código relevante / diseño de interfaz de usuario	5
Controlador y lógica principal	5
Helper.....	14
Error 404	16
Rutas.....	16
Vistas	18
Limitaciones de la implementación	22
Fuentes y repositorios.....	23
Dependencias y despliegue	23
Dependencias	23
Despliegue	24

Documentación técnica del módulo

Relación de archivos y ubicación y breve descripción de su función

En este apartado se detallan todos los archivos y carpetas principales del proyecto, la ubicación que ocupan dentro de la estructura de directorios y una breve descripción de su propósito. Cabe destacar que el trabajo mantiene la estructura básica de Laravel, a la que se han ido añadiendo modificaciones y elementos adicionales a lo largo del desarrollo.

- **app/**

Esta carpeta contiene la lógica principal de la aplicación y se subdivide en:

- **Helpers/**: Incluye métodos de ayuda (helpers) que facilitan tareas recurrentes en la aplicación.
- **Http/**: Alberga la carpeta **Controllers/**, donde se encuentran los controladores que gestionan las peticiones y respuestas de la aplicación.
- **Models/**: Contiene todos los modelos que representan y gestionan la interacción con la base de datos.
- **Providers/**: Incluye proveedores de servicios. En este caso, se encuentra un provider que no ha sido modificado, y su función es la de registrar servicios en la aplicación.

- **bootstrap/**

Esta carpeta no ha sido modificada y contiene archivos de configuración y estilos iniciales necesarios para el arranque de la aplicación.

- **config/**

Contiene los archivos de configuración de la aplicación, donde se definen variables y parámetros importantes (por ejemplo, configuración de la base de datos, servicios, etc.).

- **database/**

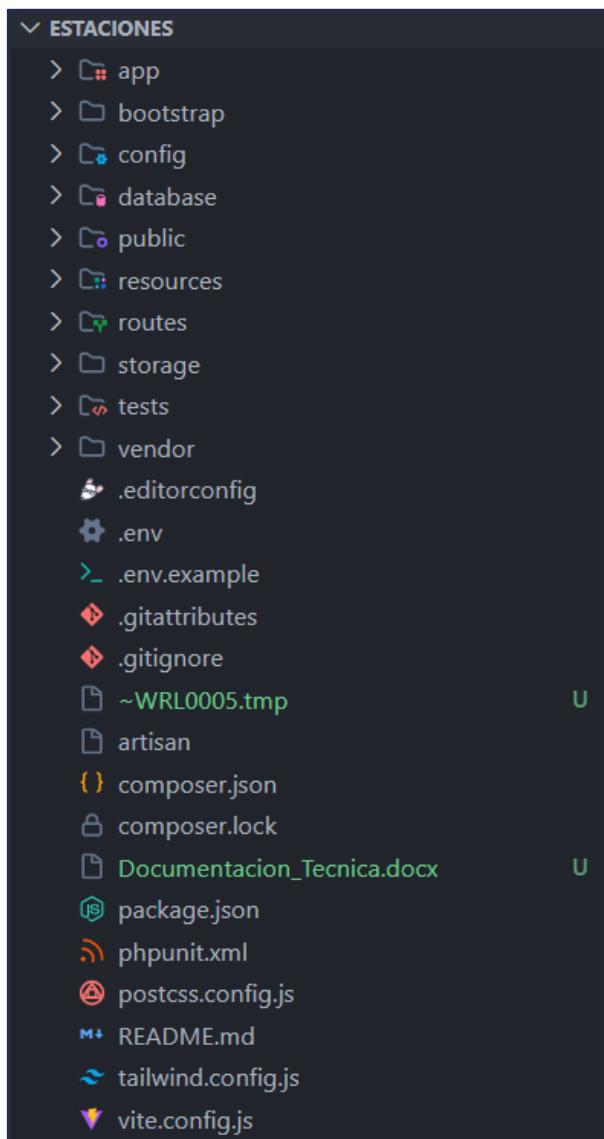
Alberga tres subcarpetas relacionadas con la base de datos:

- **factories/**: Contiene definiciones para la generación de datos de prueba.

- **migrations/**: Incluye las migraciones que definen la estructura de la base de datos.
- **seeders/**: Posee archivos para la inserción de datos de prueba o iniciales en la base de datos.
- **public/**
Es la carpeta pública donde se ubican los archivos accesibles directamente desde el navegador:
 - **css/**: Hojas de estilo utilizadas en el proyecto.
 - **fonths/**: Fuentes empleadas en las vistas de la aplicación.
 - Archivos sueltos como:
 - **index.php**: Archivo de entrada principal de la aplicación.
 - **favicon.ico**: Icono del sitio.
- **resources/**
Esta carpeta contiene los recursos de la aplicación (por ejemplo, vistas y archivos de lenguaje).
 - **views/**: Contiene las vistas para la presentación de la aplicación.
 - **errors/**: Carpeta con vistas para manejar errores.
 - **estaciones/**: Carpeta que contiene las vistas para ver todas las estaciones y para ver una estación en concreto.
- **routes/**
Incluye los archivos que definen las rutas de la aplicación, tanto para las APIs como para las vistas:
 - **api.php**: Rutas específicas para la API.
 - **web.php**: Rutas para la parte web del proyecto.
- **storage/**
Carpeta que gestiona archivos generados y almacenados por la aplicación, como logs, archivos temporales, etc. No se han realizado modificaciones específicas en esta carpeta.
- **tests/**
Contiene los tests y pruebas automatizadas para el proyecto, aunque no se ha profundizado en su contenido.
- **vendor/**
Esta carpeta alberga las dependencias externas instaladas mediante Composer. No se debe modificar manualmente su contenido.
- **Archivos de configuración y documentación en la raíz:**

- **.env**: Archivo de entorno donde se configuran variables sensibles como el nombre de la base de datos, usuario, contraseña, entre otros.
- **.gitignore**: Define los archivos y carpetas que Git debe ignorar.
- **composer.json**: Archivo que especifica las dependencias y configuraciones necesarias para Composer.
- **.editorconfig**: Configuraciones para mantener la consistencia en el estilo de edición del código.
- **package.json**: Archivo que gestiona las dependencias de Node.js y scripts de desarrollo.
- **README.md**: Documento informativo sobre el proyecto.
- **Documentacion_Tecnica.docx**: Documento actual que contiene la documentación técnica del trabajo.

Nota: Existen otros archivos en la raíz del proyecto, pero se han incluido únicamente aquellos que se consideran relevantes para la comprensión y mantenimiento del sistema.



Secciones de código relevante / diseño de interfaz de usuario

Controlador y lógica principal

El controlador principal del proyecto es `EstacionController.php`, el cual contiene diversos métodos encargados de gestionar la lógica de la aplicación. Entre ellos se incluyen:

- **Métodos para la visualización de datos en las vistas.**
- **Un método para el manejo de errores**, asegurando una respuesta adecuada ante fallos en la aplicación.
- **Métodos específicos de la API**, encargados de procesar las solicitudes y devolver las respuestas correspondientes.

Este controlador actúa como el núcleo de la interacción entre el usuario y la aplicación, coordinando la comunicación entre la interfaz y la lógica del sistema.

Método de manejo de errores

Este es el método de manejo de errores, que verifica si el ID proporcionado es un valor entero válido utilizando el filtro `FILTER_VALIDATE_INT`. Si el ID no es válido, se registra un error en los logs y se aborta la solicitud con un código de estado 400. Se le pasa por parámetro el ID a validar.

```
private function validarIdEntero($id)
{
    if (!filter_var($id, FILTER_VALIDATE_INT)) {
        Log::error("ID inválido recibido: {$id}");
        abort(400, "ID inválido");
    }
    return (int) $id;
}
```

Métodos específicos de la API

El siguiente método obtiene una lista de estaciones que existen en la tabla `estacion_bd` y completa su información con los datos de `estacion_inv`. Devuelve un json con la lista de estaciones filtradas, con el código de acierto número 201. Captura cualquier excepción y devuelve un error 500 en caso de fallo.

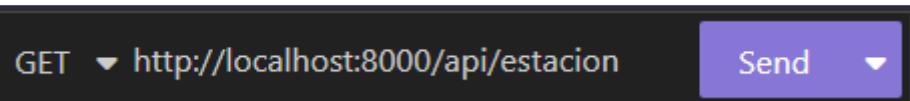
```
public function listarEstaciones()
{
    try {
        // Obtener solo los IDs que existen en estacion_bd
        $idsEstacionesBd = EstacionBd::pluck('id')->toArray();

        // Obtener las estaciones de estacion_inv que coincidan con los IDs de estacion_bd
        $estaciones = EstacionInv::whereIn('id', $idsEstacionesBd)->get()->map(function ($estacionInv) {
            // Obtener el estado desde estacion_bd (relacionado por el mismo ID)
            $estadoBd = EstacionBd::find($estacionInv->id);

            return [
                'id' => $estacionInv->id,
                'nombre' => $estacionInv->nombre,
                'provincia' => $estacionInv->provincia,
                'idema' => $estacionInv->idema,
                'x' => $estacionInv->latitud,
                'y' => $estacionInv->longitud,
                'altitud' => $estacionInv->altitud,
                'estado' => $estadoBd && $estadoBd->estado == 1 ? 'active' : 'inactive', // Tomamos el estado desde estacion_bd
            ];
        });
    }

    return response()->json($estaciones, 201);
} catch (Throwable $e) {
    Log::error('Error al obtener estaciones: ' . $e->getMessage());
    return response()->json(['error' => 'Error al obtener estaciones'], 500);
}
}
```

Como ejemplo de uso, se utilizará **Insomnia** para realizar una solicitud GET a la ruta definida en el archivo **routes** para este tipo de petición.



A continuación, se muestra un ejemplo de la respuesta generada para esta solicitud:

```
[  
  {  
    "id": 1,  
    "nombre": "Estación Central",  
    "provincia": "Madrid",  
    "idema": "EC001",  
    "x": 40.4168,  
    "y": -3.70379,  
    "altitud": 667,  
    "estado": "active"  
  },  
  {  
    "id": 2,  
    "nombre": "Estación Norte",  
    "provincia": "Barcelona",  
    "idema": "EN002",  
    "x": 41.3851,  
    "y": 2.1734,  
    "altitud": 12,  
    "estado": "inactive"  
  },  
  {  
    "id": 3,  
    "nombre": "Estación Sur",  
    "provincia": "Sevilla",  
    "idema": "ES003",  
    "x": 37.3886,  
    "y": -5.98233,  
    "altitud": 7,  
    "estado": "inactive"  
  },  
]
```

El siguiente método tiene la función de trasladar una estación desde la tabla *estacion_inv* a *estacion_bd*, siempre que no exista previamente en esta última. Su funcionamiento es el siguiente:

1. **Validación del ID de la estación mediante un método previamente descrito.**
2. **Verificación en la base de datos:**
 - Si la estación ya está en *estacion_bd*, se devuelve un mensaje indicando que la operación no es necesaria.
 - Si la estación no se encuentra en *estacion_inv*, se retorna un error 404.
3. **Traslado de la estación:**
 - Si la estación está en *estacion_inv* pero no en *estacion_bd*, se procede a moverla.
 - En caso de error durante el traslado o la interacción con la base de datos, se devuelve un código de error junto con la descripción del problema.

Este método recibe como parámetro el ID de la estación a mover y garantiza que no haya duplicados en la base de datos.

```

public function moverEstacionAEstacionBd($id): JsonResponse
{
    try {
        Log::info("Intentando mover estación con ID: {$id}");

        // Validamos el ID antes de continuar
        $id = $this->validarIdEntero($id);
        Log::info("ID validado correctamente: {$id}");

        // Buscamos la estación en estacion_bd
        $estacionBd = EstacionBd::find($id);

        // Si la estación ya existe en estacion_bd, retornamos un mensaje indicando que no es necesario moverla
        if ($estacionBd) {
            Log::info("La estación con ID {$id} ya existe en estacion_bd");
            return response()->json(["message" => "La estación ya existe en estacion_bd"], 400);
        }

        // Si no está en estacion_bd, buscamos en estacion_inv
        $estacionInv = EstacionInv::find($id);

        if (!$estacionInv) {
            // Si la estación no se encuentra en estacion_inv, retornamos un error 404
            Log::warning("No se encontró la estación con ID {$id} en estacion_inv");
            return response()->json(["message" => "La estación no existe en estacion_inv"], 404);
        }

        // Si la estación existe en estacion_inv, la insertamos en estacion_bd
        $nuevaEstacionBd = new EstacionBd();
        $nuevaEstacionBd->id = $estacionInv->id;
        $nuevaEstacionBd->estado = $estacionInv->estado; // Suponiendo que el estado es un campo en estacion_inv

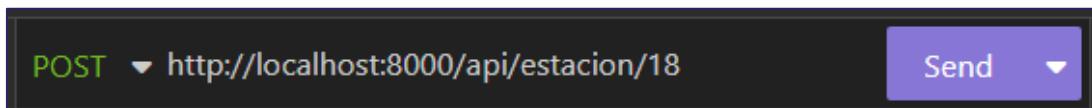
        // Guardamos la nueva estación en estacion_bd
        $nuevaEstacionBd->save();

        Log::info("Estación con ID {$id} movida correctamente de estacion_inv a estacion_bd");

        return response()->json(["message" => "Estación movida correctamente a estacion_bd"], 200);
    } catch (Exception $e) {
        Log::error("Error al mover estación con ID {$id}: " . $e->getMessage());
        return response()->json(["error" => "Error al mover estación con id {$id}"], 500);
    }
}

```

Como ejemplo de uso, se utilizará **Insomnia** para realizar una solicitud **POST** a la ruta definida en el archivo **routes** para este tipo de petición.



A continuación, se muestra un ejemplo de la respuesta generada para esta solicitud:

```
{
  "message": "Estación movida correctamente a estacion_bd"
}
```

El siguiente método recupera los datos de una estación a partir de su ID. Su funcionamiento es el siguiente:

1. **Validación del ID:** Si el ID proporcionado no es válido (verificado mediante un método previamente explicado), se devuelve un error 500.
2. **Búsqueda en la base de datos:**
 - o Si la estación no existe, se retorna un error 404.
 - o Si la estación se encuentra registrada, se genera una respuesta en formato JSON con sus detalles.
3. **Estructura de la respuesta:** Incluye la información completa de la estación, incluyendo su estado, el cual puede ser **active** o **inactive**.

Este método garantiza una respuesta clara y estructurada sobre la existencia y estado de una estación en la base de datos.

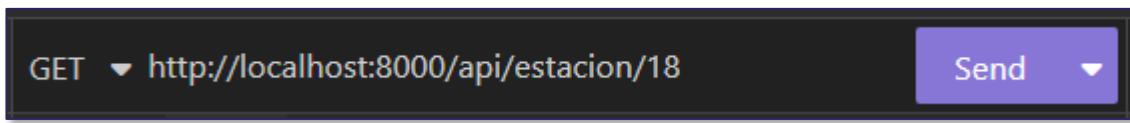
```
public function obtenerEstacion($id)
{
    try {
        // Validamos el ID antes de continuar
        Log::info("Validando ID de la estación: {$id}");
        $id = $this->validarIdEntero($id);

        // Intentamos obtener la estación con el ID proporcionado
        $estacion = EstacionInv::with('estado')->findOrFail($id);

        // Log de éxito con detalles de la estación
        Log::info("Datos de estación obtenidos correctamente: ID: {$estacion->id}, Nombre: {$estacion->nombre}");

        // Retornamos la respuesta
        return response()->json([
            'id' => $estacion->id,
            'nombre' => $estacion->nombre,
            'provincia' => $estacion->provincia,
            'idema' => $estacion->idema,
            'x' => $estacion->latitud,
            'y' => $estacion->longitud,
            'altitud' => $estacion->altitud,
            'estado' => $estacion->estado ? ($estacion->estado->estado == 1 ? 'active' : 'inactive') : 'inactive'
        ], 201);
    } catch (ModelNotFoundException $e) {
        // Si la estación no se encuentra, logueamos el error y retornamos 404
        Log::error("Estación no encontrada para ID {$id}: " . $e->getMessage());
        return response()->json(['error' => 'Estación no encontrada'], 404);
    } catch (Exception $e) {
        // Log de errores generales en caso de que ocurra alguna otra excepción
        Log::error("Error al obtener datos de la estación con ID {$id}: " . $e->getMessage());
        return response()->json(['error' => 'Error al obtener datos de la estación'], 500);
    }
}
```

Como ejemplo de uso, se utilizará **Insomnia** para realizar una solicitud **GET** a la ruta definida en el archivo **routes** para este tipo de petición.



GET ▾ http://localhost:8000/api/estacion/18

Send ▾

A continuación, se muestra un ejemplo de la respuesta generada para esta solicitud:

```
{
  "id": 18,
  "nombre": "Estación Este",
  "provincia": "Valencia",
  "idema": "EE004",
  "x": 39.4699,
  "y": -0.376288,
  "altitud": 15,
  "estado": "inactive"
}
```

El siguiente método se encarga de **actualizar el estado de una estación** en la base de datos. Su funcionamiento es el siguiente:

1. **Validación del campo 'estado':**
 - Se verifica que el request contenga únicamente este campo y que su valor sea válido.
2. **Comprobación del estado actual:**
 - Si la estación ya tiene el estado que se intenta establecer, se devuelve un mensaje indicando que la actualización no es necesaria.
3. **Actualización en la base de datos:**
 - Si el estado es diferente, se procede a actualizar el registro en la tabla *estacion_bd*.
4. **Manejo de errores:**
 - Si ocurre algún problema durante el proceso, la excepción es capturada y se retorna un mensaje de error adecuado.

Este método garantiza que las actualizaciones sean eficientes, evitando modificaciones innecesarias y proporcionando respuestas claras en caso de error.

```

public function actualizarEstadoEstacion(Request $request, $id)
{
    try {
        // Validar que solo el campo 'estado' esté presente y sea booleano
        $request->validate([
            'estado' => 'required|boolean',
        ]);

        // Buscar la estación en la base de datos
        $estacionBd = EstacionBd::find($id);

        if (!$estacionBd) {
            Log::warning("No se encontró estación con ID {$id} en la base de datos.");
            return redirect()->back()->with('error', "La estación {$id} no existe en estacion_bd");
        }

        // Verificar si el estado actual ya es el mismo que el solicitado
        if ($estacionBd->estado === $request->estado) {
            return redirect()->back()->with('message', "El estado ya está configurado como se desea");
        }

        // Actualizar estado de la estación
        $estacionBd->estado = $request->estado;
        $estacionBd->save();

        return redirect()->back()->with('message', "Estado actualizado correctamente");
    } catch (Exception $e) {
        Log::error("Error al actualizar el estado de la estación con ID {$id}: " . $e->getMessage());
        return redirect()->back()->with('error', "Error al actualizar el estado de la estación");
    }
}

```

Dado que este método no devuelve una respuesta en formato JSON, no es posible probarlo directamente con **Insomnia**. Sin embargo, su funcionamiento se puede verificar al interactuar con las vistas correspondientes, lo que se mostrará más adelante.

El siguiente método elimina una estación de la base de datos en la tabla *estacion_bd* utilizando su ID.

Este método primero valida el ID de la estación y luego intenta localizarla en la tabla *estacion_bd*. Si la estación es encontrada, se procede a eliminarla de la base de datos. En caso de que la estación no se encuentre o ocurra algún error durante el proceso, se devuelve un mensaje adecuado en formato JSON.

```

public function eliminarEstacion($id): JsonResponse
{
    try {
        Log::info("Intentando eliminar estación con ID: {$id}");

        // Validamos el ID antes de continuar
        $id = $this->validarIdEntero($id);
        Log::info("ID validado correctamente: {$id}");

        // Buscamos la estación en la tabla estacion_bd
        $estacionBd = EstacionBd::find($id);

        // Si no se encuentra en estacion_bd, retornamos un error 404
        if (!$estacionBd) {
            Log::warning("No se encontró ninguna estación con ID {$id} en estacion_bd");
            return response()->json(["message" => "La estación {$id} no existe en estacion_bd"], 404);
        }

        Log::info("Estación obtenida de estacion_bd: " . $estacionBd);

        // Intentamos eliminar la estación de la tabla estacion_bd
        $deleted = $estacionBd->delete();

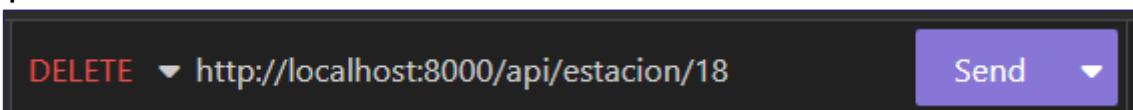
        if ($deleted) {
            Log::info("Estación eliminada de la tabla estacion_bd con ID {$id}");
        } else {
            Log::warning("No se pudo eliminar la estación con ID {$id} de estacion_bd");
        }

        Log::info("Estación con ID {$id} eliminada correctamente");

        return response()->json(["message" => "Estación {$id} eliminada correctamente"], 200);
    } catch (Exception $e) {
        Log::error("Error al eliminar estación con ID {$id}: " . $e->getMessage());
        return response()->json(["error" => "Error al eliminar estación con id {$id}"], 500);
    }
}
}

```

Como ejemplo de uso, se utilizará **Insomnia** para realizar una solicitud **DELETE** a la ruta definida en el archivo **routes** para este tipo de petición.



A continuación, se muestra un ejemplo de la respuesta generada para esta solicitud:

```
{
    "message": "Estación 18 eliminada correctamente"
}
```

Métodos para la visualización de datos

El siguiente método muestra la ficha de una estación basada en el ID proporcionado.

Este método obtiene la información de la estación desde una fuente externa (probablemente una API o servicio), decodifica la respuesta en formato JSON y pasa los datos a la vista correspondiente. Si no se encuentra la estación, se aborta la solicitud con un error 404.

- 1. Obtención de datos:** Llama al método obtenerEstacion para obtener los datos de la estación en formato JSON.
- 2. Decodificación de la respuesta:** Convierte la respuesta JSON en un array utilizando json_decode.
- 3. Verificación de la estación:** Si la estación no se encuentra o se presenta un error en la respuesta, se aborta la solicitud y se devuelve un error 404.
- 4. Renderizado de la vista:** Si los datos son válidos, se pasa la información de la estación a la vista estaciones.FichaEstacion para su presentación.

Este método retorna la vista con los detalles de la estación o un error 404 si no se encuentra la estación.

```
public function fichaEstacion($id)
{
    // Llamar al método obtenerEstacion para obtener los datos en formato JSON
    $response = $this->obtenerEstacion($id);

    // Decodificar la respuesta JSON para convertirla en un array
    $estacion = json_decode($response->getContent(), true);

    // Verificar si la estación fue encontrada y la información fue procesada correctamente
    if (isset($estacion['error'])) {
        abort(404, 'Estación no encontrada');
    }

    // Pasar los datos como un array a la vista
    return view('estaciones.FichaEstacion', compact('estacion'));
}
```

El siguiente método muestra una lista de estaciones, sin paginación, obteniendo los datos completos.

Este método obtiene los datos de las estaciones llamando a la función listarEstaciones, que devuelve los datos como un array. Luego, pasa esta lista completa de estaciones a la vista correspondiente para su visualización.

1. **Obtención de datos:** Llama a la función listarEstaciones para obtener todos los datos de las estaciones en formato array.
2. **Pasaje de datos a la vista:** La lista de estaciones obtenida se pasa a la vista estaciones.ListaEstaciones para su presentación.

A diferencia de una lista paginada, este método devuelve todos los elementos sin limitación o segmentación en múltiples páginas.

```
public function index()
{
    // Llamar a la función listarEstaciones y obtener los datos como array
    $estaciones = $this->listarEstaciones()->getData();

    // Retornar la vista pasando las estaciones como array
    return view('estaciones.ListaEstaciones', compact('estaciones'));
}
```

Helper

El siguiente código define un **helper** llamado EstadoHelper que contiene un método estático llamado obtenerEstado. Un helper en Laravel es una clase que contiene funciones auxiliares o métodos que pueden ser llamados de forma sencilla desde cualquier parte del código. Los helpers permiten simplificar tareas repetitivas y hacer que el código sea más limpio y organizado.

```
<?php

namespace App\Helpers;

class EstadoHelper
{
    public static function obtenerEstado($estado)
    {
        if (is_object($estado) || is_array($estado)) {
            $estado = $estado['estado'];
        }

        return $estado === 0 || $estado === null ? 'Inactive' : ($estado === 1 ? 'Active' : $estado);
    }
}
```

Antes de continuar, cabe mencionar que este helper no se ha utilizado solo en una de las vistas.

Explicación del código

1. **Namespace y declaración de la clase:**
 - La clase EstadoHelper está dentro del espacio de nombres App\Helpers, lo que significa que está organizada dentro de esa carpeta en el proyecto.
2. **Método estático obtenerEstado:**
 - Este método recibe un parámetro \$estado que puede ser un valor simple, un array o un objeto.
3. **Comprobación del tipo de \$estado:**
 - El método primero verifica si \$estado es un objeto o un array con la condición `is_object($estado) || is_array($estado)`. Si es así, intenta acceder al valor de la clave `estado` dentro del array u objeto con `$estado['estado']`.
4. **Retorno del estado:**
 - Después de obtener el valor de \$estado, se evalúa su valor:
 - Si \$estado es igual a 0 o es null, se retorna la cadena 'Inactive' (inactivo).
 - Si \$estado es igual a 1, se retorna 'Active' (activo).
 - Si el valor de \$estado es diferente de 0 o 1, se retorna el valor de \$estado tal cual, sin modificaciones.

¿Qué hace el helper?

Este helper proporciona una forma sencilla de interpretar el estado de un elemento (por ejemplo, una estación) basado en su valor, devolviendo 'Active' o 'Inactive' según corresponda. Si el estado no es 0, 1, o null, el método simplemente devuelve el valor original de \$estado.

Ejemplo de uso

```
$estado = EstadoHelper::obtenerEstado(1); // Devuelve 'Active'
$estado = EstadoHelper::obtenerEstado(0); // Devuelve 'Inactive'
$estado = EstadoHelper::obtenerEstado(null); // Devuelve 'Inactive'
$estado = EstadoHelper::obtenerEstado(['estado' => 1]); // Devuelve 'Active'
```

En resumen, este helper facilita la conversión de valores numéricos o nulos de un estado en representaciones legibles como "Active" o "Inactive", mejorando la legibilidad y reutilización del código.

Error 404

Hemos descubierto que al crear una carpeta llamada errors dentro de la carpeta views en resources, y agregar un archivo como 404.blade.php dentro de ella, podemos manejar de manera personalizada los errores. Cuando se retorna un error 404, por ejemplo, automáticamente se muestra esta vista personalizada, lo cual resulta ser una característica bastante interesante. Cabe destacar que se puede hacer con cualquier código de error.

En nuestro caso, hemos creado una vista sencilla para el error 404, que es la siguiente:

¡Oops! La página o recurso que buscas no existe.

El ID proporcionado no es válido o está fuera del rango permitido (1-100).

Por favor, asegúrate de ingresar un ID válido entre 1 y 100 y existente en la base de datos.

Consideramos que este recurso es muy útil para personalizar la gestión de errores en la aplicación.

Rutas

En la carpeta routes, encontramos varios archivos que definen las rutas de la aplicación.

api.php

Uno de estos archivos es api.php, que contiene las rutas para las funciones relacionadas con la API.

Dentro de este archivo, se definen las rutas asociadas a las estaciones, que se gestionan mediante el controlador EstacionController. Las rutas se detallan de la siguiente manera:

- **GET /estacion:** Esta ruta llama al método listarEstaciones del controlador EstacionController, que se encarga de devolver todas las estaciones registradas en la base de datos.
- **GET /estacion/{id}:** Esta ruta llama al método obtenerEstacion, que permite obtener los detalles de una estación específica identificada por su ID.
- **POST /estacion/{id}:** Esta ruta utiliza el método moverEstacionAEstacionBd para agregar una nueva estación desde una tabla temporal llamada estacion_inv a la tabla principal estacion_bd.
- **PUT /estacion/{id}:** Aquí, se llama al método actualizarEstadoEstacion, que permite cambiar el estado de una estación (por ejemplo, de activa a inactiva o viceversa). Esta ruta está asociada con el nombre estaciones.actualizarEstado para poder referenciarla en otros lugares del código.
- **DELETE /estacion/{id}:** Finalmente, esta ruta se usa para eliminar una estación de la base de datos, específicamente de la tabla estacion_bd, utilizando el método eliminarEstacion.

Estas rutas permiten interactuar con las estaciones de manera eficaz, gestionando desde su visualización hasta su modificación o eliminación dentro de la base de datos.

```
// Rutas relacionadas con "Estaciones"
Route::get('/estacion', [EstacionController::class, 'listarEstaciones']); // Obtener todas las estaciones
Route::get('/estacion/{id}', [EstacionController::class, 'obtenerEstacion']); // Obtener una estación específica por su ID
Route::post('/estacion/{id}', [EstacionController::class, 'moverEstacionAEstacionBd']); // Añadir una nueva estación desde estacion_inv a estacion_bd
Route::put('/estacion/{id}', [EstacionController::class, 'actualizarEstadoEstacion'])->name('estaciones.actualizarEstado'); // Cambiarle el estado a una estación
Route::delete('/estacion/{id}', [EstacionController::class, 'eliminarEstacion']); // Eliminar una estación por su ID de estacion_bd
```

web.php

Estas rutas están relacionadas con la parte de las vistas en la aplicación. Son las siguientes:

1. **Route::get('/estaciones', [EstacionController::class, 'index']);**
Esta ruta carga la vista que muestra una lista de todas las estaciones disponibles.
2. **Route::get('estaciones/{id}', [EstacionController::class, 'fichaEstacion'])->name('estaciones.ficha');**
Esta ruta muestra la información detallada de una estación específica, identificada por su ID. Se puede acceder a esta ruta mediante el nombre estaciones.ficha.
3. **Route::get('estaciones', [EstacionController::class, 'index'])->name('estaciones.index');**
Similar a la primera, esta ruta carga la vista con todas las estaciones. La diferencia es que esta ruta tiene un nombre específico, estaciones.index, que se puede utilizar para referenciarla de manera más fácil en el código.

En resumen, estas rutas permiten acceder a diferentes vistas de estaciones: una lista general de todas las estaciones y una vista individual de una estación específica.

```
Route::get('/estaciones', [EstacionController::class, 'index']);
Route::get('estaciones/{id}', [EstacionController::class, 'fichaEstacion'])->name('estaciones.ficha');
Route::get('estaciones', [EstacionController::class, 'index'])->name('estaciones.index');
```

Vistas

Se han desarrollado dos vistas principales para la gestión de las estaciones:

- **Listado de Estaciones:** Muestra todas las estaciones registradas en la tabla estacion_bd de la base de datos.
- **Ficha de Estación:** Permite ver los detalles de una estación en particular y cambiar su estado.

A continuación, se muestra un fragmento de código relevante de la vista que lista las estaciones:

```
<tbody>
    @forelse ($estaciones as $estacion)
        <tr>
            <td>{{ $estacion->id }}</td>
            <td>{{ $estacion->nombre }}</td>
            <td>{{ $estacion->provincia }}</td>
            <td>{{ $estacion->x }}</td>
            <td>{{ $estacion->y }}</td>
            <td>{{ $estacion->altitud }}</td>
            <td>{{ \App\Helpers\EstadoHelper::obtenerEstado($estacion->estado) }}</td>
            <td id="ficha"><a class="button"
                href="{{ route('estaciones.ficha', ['id' => $estacion->id]) }}">Ver ficha</a>
            </td>
        </tr>
    @empty
        <tr>
            <td colspan="8">No hay estaciones disponibles.</td>
        </tr>
    @endforelse
</tbody>
```

Explicación del código:

- Recorrido de datos:** El bloque `@forelse` itera sobre la colección de estaciones (`$estaciones`). Si la colección está vacía, se muestra un mensaje indicando que no hay estaciones disponibles.
- Visualización de información:** Para cada estación, se muestran los siguientes campos:
 - ID, Nombre, Provincia, Coordenadas (x, y) y Altitud.**
 - Estado:** Se utiliza el helper `EstadoHelper::obtenerEstado` para convertir el valor numérico del estado en una representación legible (por ejemplo, "Active" o "Inactive").
- Acceso a la ficha de la estación:** En la última columna, se incluye un enlace (botón) que dirige a la vista de la ficha individual de la estación. Este enlace utiliza la función `route` con el nombre `estaciones.ficha` y le pasa el ID de la estación como parámetro.

En resumen, esta vista no solo presenta de manera ordenada los datos de cada estación, sino que también permite acceder a una vista más detallada donde se pueden gestionar aspectos como el estado de la estación.

A continuación, se muestra un ejemplo visual de la vista:

LISTA DE ESTACIONES

ID	Nombre	Provincia	Latitud	Longitud	Altitud	Estado	Ver ficha
1	Estación Central	Madrid	40.4168	-3.70379	667	inactive	<button>Ver ficha</button>
2	Estación Norte	Barcelona	41.3851	2.1734	12	inactive	<button>Ver ficha</button>
3	Estación Sur	Sevilla	37.3886	-5.98233	7	inactive	<button>Ver ficha</button>
4	Estación Este	Valencia	39.4699	-0.376288	15	inactive	<button>Ver ficha</button>
5	Estación Oeste	La Coruña	43.3623	-8.41154	19	inactive	<button>Ver ficha</button>
17	Estación Este	Valencia	39.4699	-0.376288	15	active	<button>Ver ficha</button>
18	Estación Este	Valencia	39.4699	-0.376288	15	inactive	<button>Ver ficha</button>
19	Estación Este	Valencia	39.4699	-0.376288	15	active	<button>Ver ficha</button>
20	Estación Este	Valencia	39.4699	-0.376288	15	active	<button>Ver ficha</button>
21	Estación Este	Valencia	39.4699	-0.376288	15	active	<button>Ver ficha</button>
27	Estación Pito	Valencia	39.4699	-0.376288	15	inactive	<button>Ver ficha</button>

Ahora se muestra el código relevante de la vista de FichaEstacion.php, que muestra la información de una estación permitiendo cambiar su estado.

```

<body>
    <h1>Ficha de la Estación</h1>

    <div id="contenedorP">
        <div id="contenedorH">
            <h1 id="estacion">{{ $estacion['nombre'] }}</h1>
            <p><strong>ID:</strong> <span>{{ $estacion['id'] }}</span></p>
            <p><strong>Provincia:</strong> <span>{{ $estacion['provincia'] }}</span></p>
            <p><strong>Latitud:</strong> <span>{{ $estacion['x'] }}</span></p>
            <p><strong>Longitud:</strong> <span>{{ $estacion['y'] }}</span></p>
            <p><strong>Altitud:</strong> <span>{{ $estacion['altitud'] }}</span></p>
            <p><strong>Estado:</strong>
                <span>{{ \App\Helpers\EstadoHelper::obtenerEstado($estacion['estado']) }}</span>
            </p>
        </div>
        <!-- Formulario para actualizar el estado de la estación -->
        <form action="{{ route('estaciones.actualizarEstado', ['id' => $estacion['id']]) }}" method="POST">
            @csrf
            @method('PUT') <!-- Esto indica que la solicitud es PUT, pero no es necesario para validar el estado -->

            <label for="estado">Estado:</label>
            <select name="estado" id="estado">
                <option value="0" {{ $estacion['estado'] == 0 ? 'selected' : '' }}>Inactivo</option>
                <option value="1" {{ $estacion['estado'] == 1 ? 'selected' : '' }}>Activo</option>
            </select>

            <button class="button" type="submit">Actualizar Estado</button>
        </form>
    </div>

    <a class="button" href="{{ route('estaciones.index') }}>Volver a la lista de estaciones</a>
</body>
```

Este código representa la vista para mostrar la ficha de una estación y permitir actualizar su estado, centrándose en la lógica funcional:

- **Visualización de datos de la estación:** Se extraen y muestran los datos principales de la estación (nombre, ID, provincia, coordenadas, altitud y estado) a partir del arreglo \$estacion. Para el estado, se utiliza el helper EstadoHelper::obtenerEstado para convertir el valor numérico en una cadena legible ("Inactivo" o "Activo").
- **Formulario de actualización:** Se incluye un formulario que envía una solicitud PUT a la ruta responsable de actualizar el estado de la estación. Este formulario:
 - Utiliza protección CSRF.
 - Presenta un campo de selección para elegir el nuevo estado, mostrando la opción correspondiente al estado actual como seleccionada.
 - Al enviarse, se actualiza el estado de la estación en la base de datos mediante la ruta estaciones.actualizarEstado.
- **Navegación:** Se proporciona un enlace que permite volver a la lista de estaciones, facilitando la navegación entre las diferentes vistas.

En resumen, este código se encarga de presentar de forma clara la información detallada de una estación y de ofrecer una interfaz sencilla para modificar su estado.

A continuación, se muestra como se ve visualmente dicha vista.

FICHA DE LA ESTACIÓN

ESTACIÓN NORTE	
ID:	2
PROVINCIA:	BARCELONA
LATITUD:	41.3851
LONGITUD:	2.1734
ALTITUD:	12
ESTADO:	INACTIVE

ESTADO: ACTUALIZAR ESTADO

[VOLVER A LA LISTA DE ESTACIONES](#)

Limitaciones de la implementación

- **Requisitos de Versión de PHP:** La aplicación requiere PHP en la versión ^8.2, lo que significa que solo podrá ejecutarse en entornos que tengan PHP 8.2 o superior. Esto limita la compatibilidad en servidores o entornos que utilicen versiones anteriores de PHP.
- **Dependencia del Framework Laravel:** Se utiliza Laravel Framework en la versión ^11.31, por lo que la aplicación depende de las características y convenciones de esta versión. Cualquier cambio significativo en futuras versiones mayores de Laravel podría requerir modificaciones en el código.
- **Biblioteca de Manipulación de Bases de Datos:** La dependencia doctrine/dbal en la versión ^4.2 se utiliza para operaciones avanzadas con bases de datos. Esto puede implicar limitaciones en cuanto a funcionalidades y compatibilidad con otros paquetes que requieran versiones distintas de esta biblioteca.
- **Limitaciones en el Entorno de Desarrollo:** Las dependencias de desarrollo (como laravel/sail, laravel/pint, nunomaduro / collision, entre otras) están configuradas para facilitar el proceso de desarrollo y pruebas. Sin embargo, estas herramientas no se aplican en producción, lo que puede requerir configuraciones adicionales para adaptar el entorno de despliegue.
- **Sistema de Autenticación:** El uso de laravel/sanctum para la autenticación limita la aplicación a este método de autenticación basado en tokens. Si se requiere implementar otro sistema de autenticación, podría ser necesario realizar ajustes en la configuración y en el código.
- **Dependencias Adicionales para Pruebas y Generación de Código:** Herramientas como reliese/laravel y mockery/mockery facilitan la generación de código y la realización de pruebas. No obstante, su integración añade una capa de dependencia que puede restringir la flexibilidad en el manejo de ciertos procesos de desarrollo o personalización.

Estas limitaciones deben tenerse en cuenta al desplegar y mantener la aplicación, ya que pueden influir en la compatibilidad y en la evolución futura del proyecto.

Fuentes y repositorios

- 🔗 **Repositorio GitHub:** <https://github.com/jarasa03/Estaciones>
- 🔗 Se ha sacado información también de la documentación oficial de Laravel: <https://laravel.com>

Dependencias y despliegue

Dependencias

El proyecto utiliza varias dependencias esenciales para su funcionamiento, definidas en el archivo composer.json:

1. **PHP:** Se requiere PHP en versión ^8.2, lo que garantiza el uso de las funcionalidades modernas y seguras del lenguaje.
2. **Laravel Framework:** La aplicación se basa en Laravel (versión ^11.31), aprovechando sus componentes y convenciones para el desarrollo rápido y estructurado de aplicaciones web.
3. **Doctrine DBAL:** Se incluye doctrine/dbal (^4.2) para operaciones avanzadas de bases de datos, como modificaciones complejas en la estructura de las tablas.
4. **Laravel Sanctum:** Utilizado para la autenticación mediante tokens, facilitando el desarrollo de APIs seguras.
5. **Laravel Tinker:** Proporciona una consola interactiva para la depuración y experimentación con el código.

Además, se han configurado dependencias de desarrollo para pruebas, generación de código, análisis y formateo, entre ellas:

- **Faker:** Para la generación de datos de prueba.
- **Laravel Sail y Pint:** Para el entorno de desarrollo y la estandarización del código.
- **PHPUnit y Mockery:** Para la realización de pruebas automatizadas.
- **Reliese Laravel:** Para la generación automática de modelos a partir de la base de datos.

El autoload está configurado mediante PSR-4, lo que permite una carga automática de las clases del proyecto, y se incluye explícitamente el helper EstadoHelper.php.

Despliegue

Para desplegar y ejecutar la aplicación, se deben seguir los siguientes pasos:

1. Clonar el repositorio

git clone https://github.com/jarasa03/Estaciones

cd Estaciones

2. Instalar las dependencias

composer install

Esto descargará todas las librerías y paquetes necesarios según lo definido en el composer.json.

3. Configurar variables de entorno

cp .env.example .env

Esto copia el archivo de ejemplo y edita las variables necesarias (como la configuración de la base de datos, claves de API, etc.). Luego, ajusta los valores según el entorno de despliegue.

4. Generar la clave de la aplicación

php artisan key:generate

Laravel necesita una clave única para la aplicación, se genera con el comando anterior.

5. Migraciones

Configura la base de datos y ejecuta las migraciones para crear la estructura necesaria:

php artisan migrate

6. Iniciar el servidor

php artisan serve

Este comando inicia la aplicación en <http://localhost:8000>.