AE.U3 REALIZA EL TUTORIAL DE ANGULAR

Nombre y apellidos: Cristina Sandoval Laborde

Curso: 2º DAM

Asignatura: Optativa

Índice

1.Introducción y contextualización

En esta actividad se aborda la transición hacia el desarrollo web moderno mediante Angular. El objetivo es consolidar el uso de arquitecturas basadas en componentes y la comunicación asíncrona con servicios Backend desarrollados en Node.js. Se exploran las ventajas de las aplicaciones de página única (SPA) para mejorar la experiencia de usuario y la eficiencia en la carga de datos.

2.Desarrollo

2.1.Configuración del Routing y providers

Como se menciona en el manual, el Router intercepta las acciones del usuario para mostrar u ocultar componentes sin recargar la página completa. Al centrar los proveedores en appConfig, me aseguro que toda la aplicación tenga acces a las herramientas de navegación y conexión a datos externos desde el comienzo.

En el archivo app.config.ts

He utilizado el modelo de configuración de Angular **(Standalone)** mediante el objeto ApplicationConfig, eliminando asi la necesidad de los antiguos **NgModules**, lo que simplifica la estructura del proyecto.

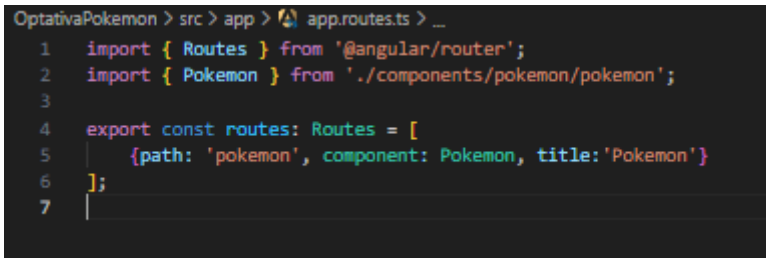
He comenzado añadiendo **provideRouter(routes)** que es el motor de navegación. Este proveedor es el que mapea las URL que el usuario introduce en la barra de direcciones o en los enlaces que hace clic para asociarlos a los componentes de la aplicación.

He añadido **provideHttpClient()** para permitir la comunicación con el servidor, permitiendo así las operaciones CRUD asíncronas mediante el protocolo HTTP.

También he añadido **ProvideClientHydration()** para mejorar el rendimiento de carga y la renderización con el cliente, al permitir que la aplicación se más rápida al “hidratar” el contenido renderizado en el servidor.

Una vez configurados los proveedores globales, he definido el diccionario de rutas en el archivo app.routes.ts. En este archivo es donde se mapea el camino con su componenete correspondiente, permitiendo que Angular separa que vista carga cuando el usuario navega a una URL especifica.

En el archivo app.routes.ts



2.2.Data Binding

El data binding es el mecanismo fundamental de Angular que permite la comunicación o enlace de datos entre el archivo TypeScript del componente y su plantilla HTML.

Según el manual, hay dos formas de gestionar esta comunicación:

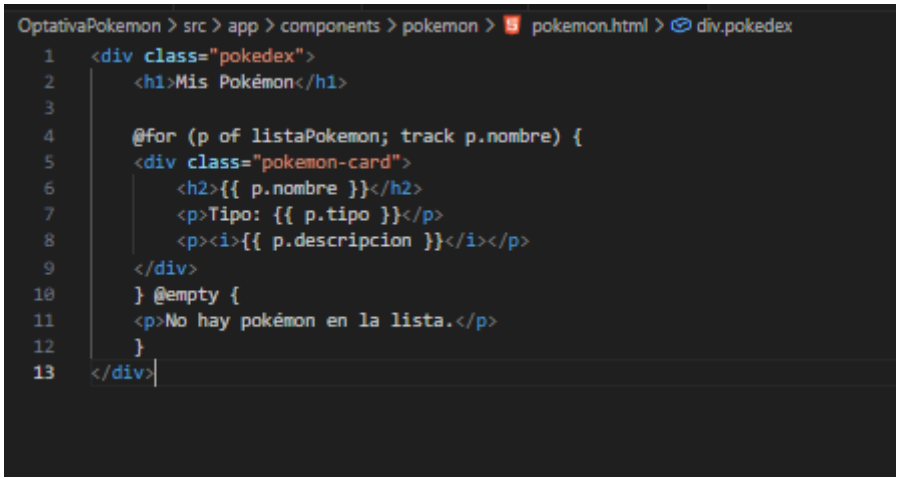
2.2.1. One-way binding (Unidireccional)

Es el envio de datos desde el código hacia la plantilla. Para esto he implementado dos tipos:

- Interpolación{{expresión}}: Que se utiliza para mostrar valores dinámicos. En este proyecto la he usado para pintar el nombre y el tipo de cada Pokémon en la vista.
- Property binding [target]=”expresión”: Que sirve para controlar propiedades de elementos HTML.

Como se observa en la imagen siguiente he definido una interface PokemonData asegurando que los datos que fluyen por el binding tengan la estructura correcta(nombre,tipo y descripción).

En el archivo pokemon.html he usado la interpolación para mostrar los datos del array Pokémon en el navegador de forma dinámica.



En el archivo pokemon.ts

2.2.2. Two-way (Bidireccional)

Permite que la comunicación fluya en ambas direcciones, es decir, los cambios en el código actualizan la vista y las interacciones del usuario en la vista actualizan el código.

Se utiliza la sintaxis “banana in a box” [(target)]=”expresión”

2.3. Directivas Estructurales

Las directivas estructurales son herramientas que permiten modificar la estructura del DOM basándose en los datos del componente.

2.3.1 Iteración de elementos

He utilizado la sintaxis moderna @for para recorrer el array listaPokemon.

He incluido la propiedad track p.nombre, que permite a Angular identificar de forma única cada elemento.

Por cada iteración, se genera una tarjeta que contiene la información del Pokémon y los botones de control.

2.3.2. Visualización condicional

En este proyecto he utilizado la potencia del bloque @empty, que actúa como condicional automático, si la lista de Pokémon no contiene información, Angular oculta el bucle y simplemente muestra un mensaje alterativo (”No hay Pokémon en la lista”).

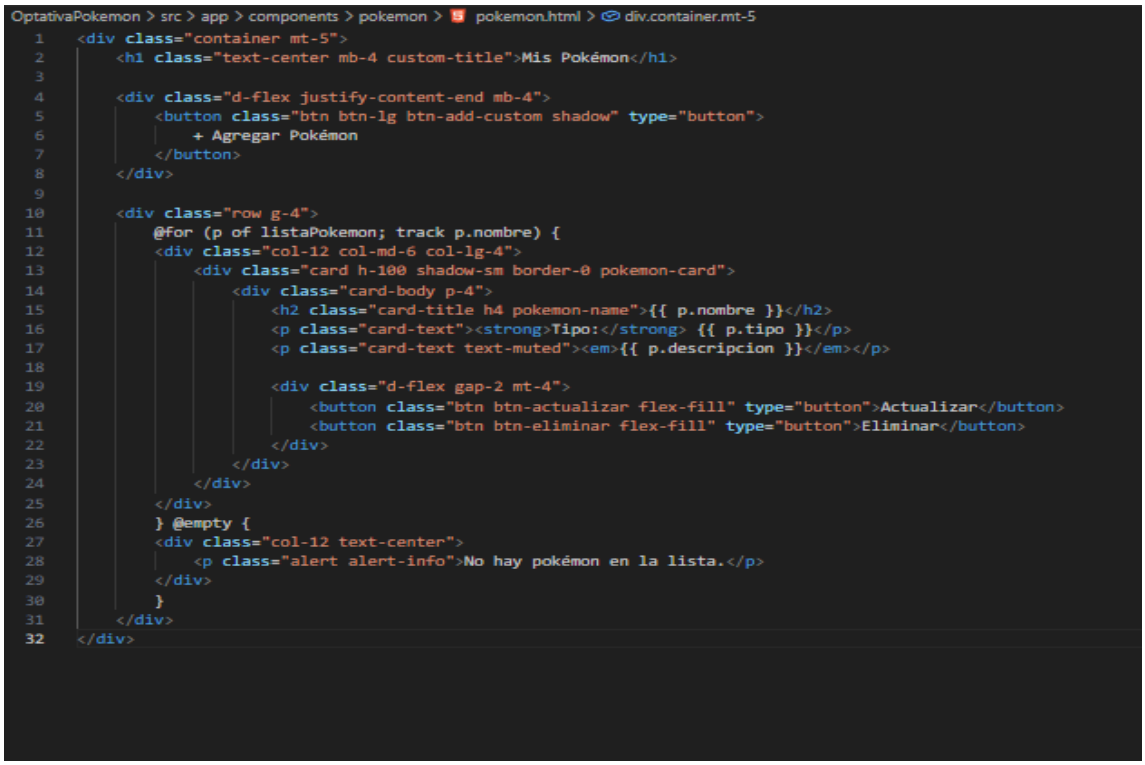
Esto hace que la interfaz siempre proporcione feedback al usuario.

2.3.3. Implementación de CRUD Estático

Dentro de la estructura generada medienta el bucle @for, he integrado los elementos esenciales para las operaciones CRUD(Create, Read,Update,Delete), asegurando que cada elemento de la lista cuente con su porpia interfaz de gestión.

- Read:Esta operación ya es funcional, se realiza mediante la interpolacion de los datos del array listaPokemon dentro de cada tarjeta. Al iterar sobre la lista, Angular extrae propiedades de la interfaz y las muestra en el DOM de forma automática.
- Update/Delete: Ahora tengo la maquetación de los botones específicos para estas acciones dentro de cada tarjeta de pokemon.
    - Estado actual: En esta fase de maquetación y estructura, los botones son solo estéticos. Esta correctamente poscionados y vinculados visualmente a cada registro, pero no tienen ningún tipo de lógica funcional por el momento.

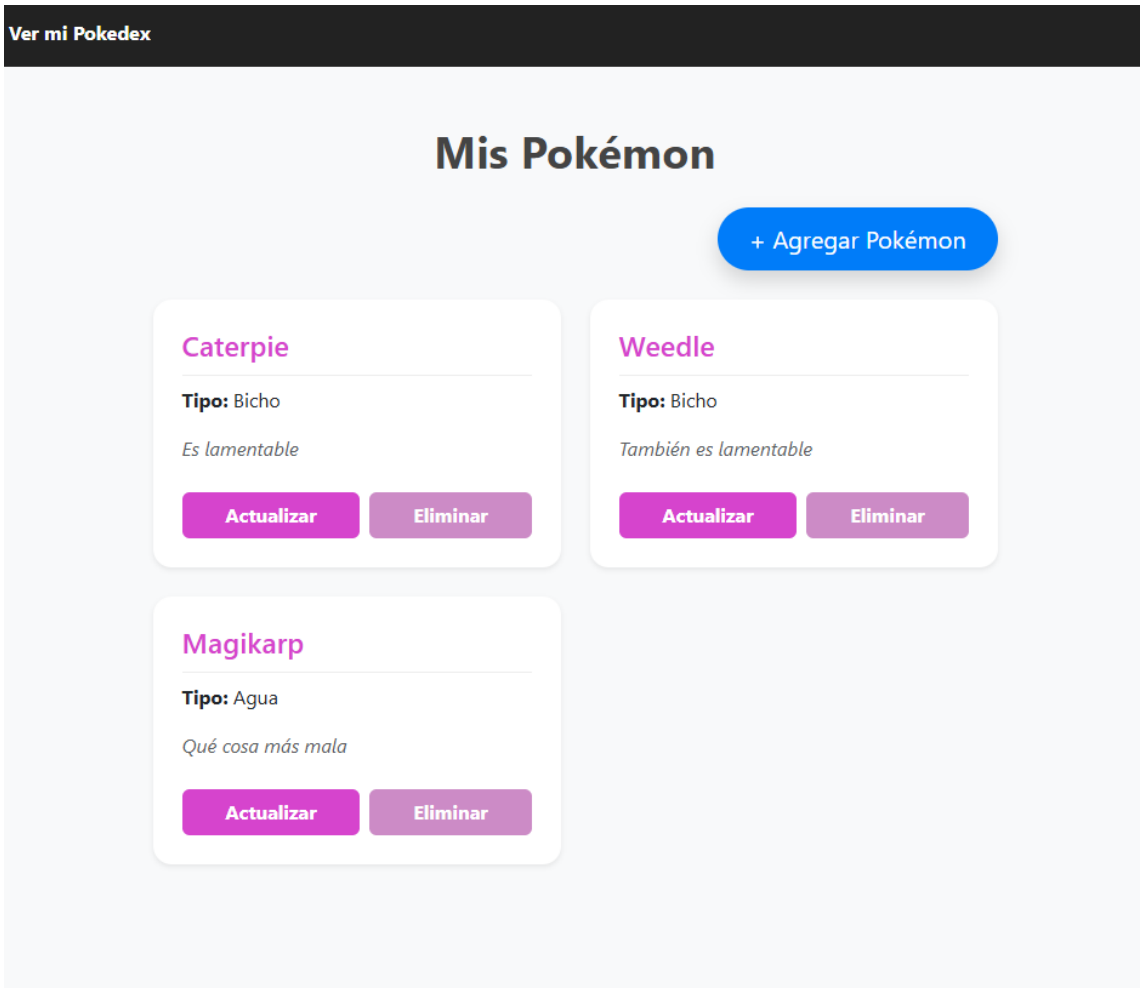
- Fase posterior: La lógica de programación, el manejo de eventos de clic y la comunicación con los servicios para ejecutar cambios reales en la base de datos los implementare en el apartado Desarrollo CRUD Pokemon.



Estilos CSS

OptativaPokemon > src > app > components > pokemon > pokemon.css > .btn-eliminar

```css
:host {
    --purple-main: #d145d1;
    --pink-accent: #e352e3;
    --dark-bg: #212121;
}

.custom-title {
    font-weight: bold;
    color: #444;
}


.pokemon-card {
    border-radius: 15px !important;
    transition: transform 0.2s;
}

.pokemon-card:hover {
    transform: translateY(-5px);
}

.pokemon-name {
    color: var(--purple-main);
    border-bottom: 1px solid #eee;
    padding-bottom: 10px;
}

/* BOTONES CON  COLORES */
.btn-add-custom {
    background-color: #007bff;

    color: white;
    border-radius: 30px;
    padding: 10px 25px;
}

.btn-actualizar {
    background-color: var(--purple-main);
    color: white;
    font-weight: bold;
}

.btn-eliminar {
    background-color: #c98bc9;
    color: white;
    font-weight: bold;
}

.btn:hover {
    opacity: 0.9;
    color: white;
}
```

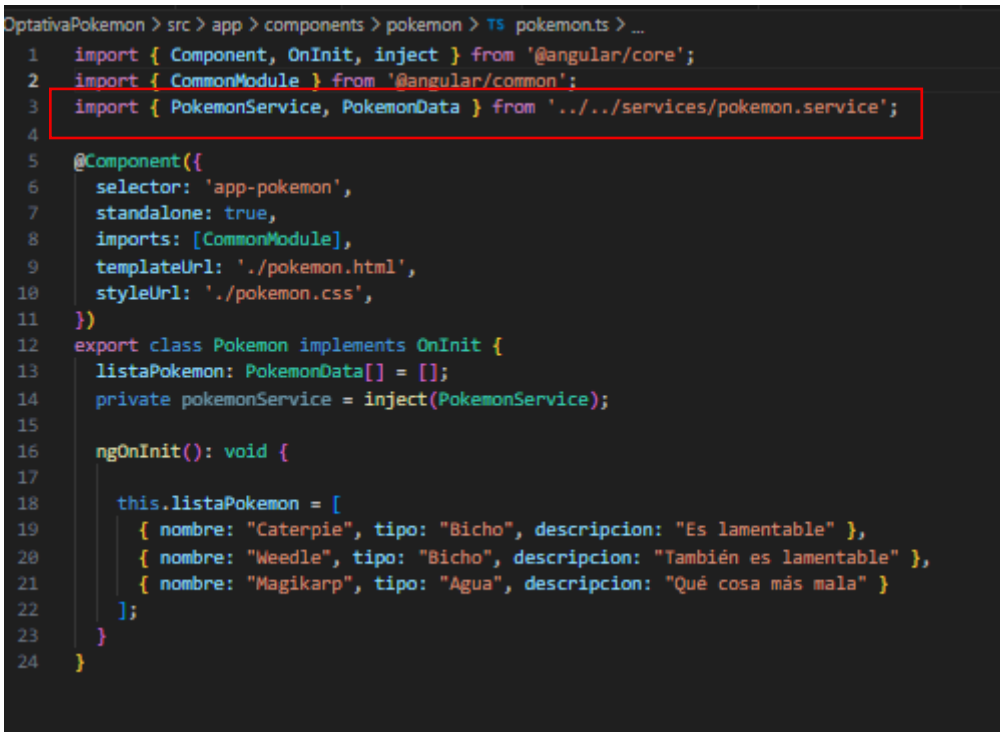


2.4. Integración con el Backend y servicios

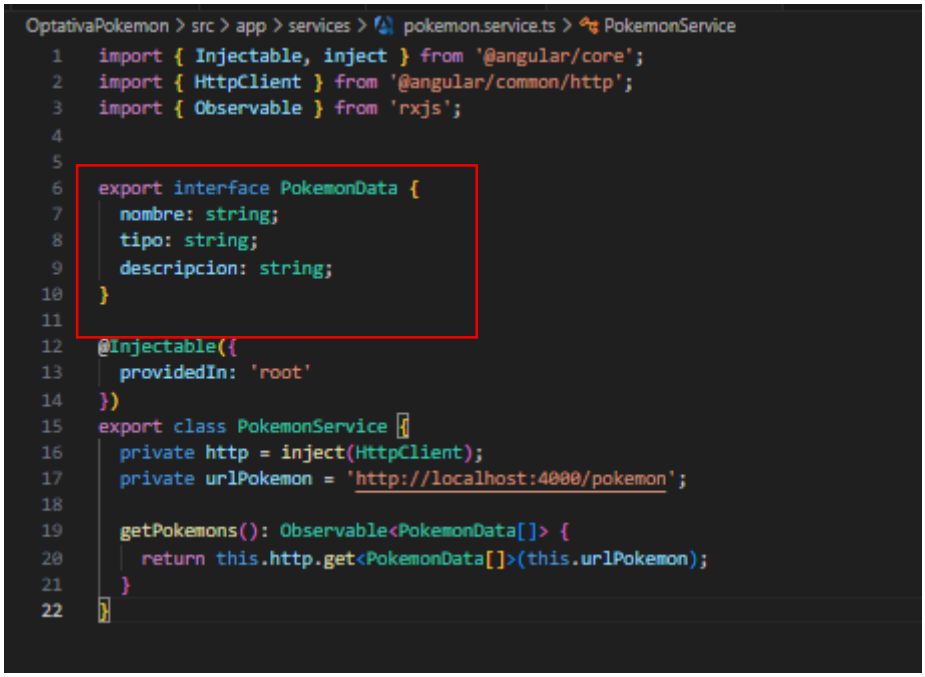
Con el siguiente comando creamos la carpeta service y los archivos pokemon.service.ts y poemon.service.spec.ts

```
Component update sent to client(s).
PS C:\Users\alumno\Desktop\2º Dam\Optativa\Optativa_pokemon\OptativaPokemon> ng generate service services/pokemon.service
CREATE src/app/services/pokemon.service.spec.ts (378 bytes)
CREATE src/app/services/pokemon.service.ts (127 bytes)
PS C:\Users\alumno\Desktop\2º Dam\Optativa\Optativa_pokemon\OptativaPokemon>
```

Ahora hay que sacar la interfaz de pokemon.ts y pasarla a pokemon.service.ts



```
OptativaPokemon > src > app > components > pokemon > TS pokemon.ts > ...
import { Component, OnInit, inject } from '@angular/core';
import { CommonModule } from '@angular/common';
import { PokemonService, PokemonData } from '../../services/pokemon.service';

@Component({
  selector: 'app-pokemon',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './pokemon.html',
  styleUrl: './pokemon.css',
})
export class Pokemon implements OnInit {
  listaPokemon: PokemonData[] = [];
  private pokemonService = inject(PokemonService);

  ngOnInit(): void {

    this.listaPokemon = [
      { nombre: "Caterpie", tipo: "Bicho", descripcion: "Es lamentable" },
      { nombre: "Weedle", tipo: "Bicho", descripcion: "También es lamentable" },
      { nombre: "Magikarp", tipo: "Agua", descripcion: "Qué cosa más mala" }
    ];
  }
}
```



```
OptativaPokemon > src > app > services > pokemon.service.ts > PokemonService
import { Injectable, inject } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';


export interface PokemonData {
  nombre: string;
  tipo: string;
  descripcion: string;
}

@Injectable({
  providedIn: 'root'
})
export class PokemonService {
  private http = inject(HttpClient);
  private urlPokemon = 'http://localhost:4000/pokemon';

  getPokemons(): Observable<PokemonData[]> {
    return this.http.get<PokemonData[]>(this.urlPokemon);
  }
}
```

2.4.1. Consumo del servicio en el componente

En este apartado se detalla la implementación lógica necesario para que el componente Pokemon deje de utilizar datos locales y pase a recibir información de una fuente externa.

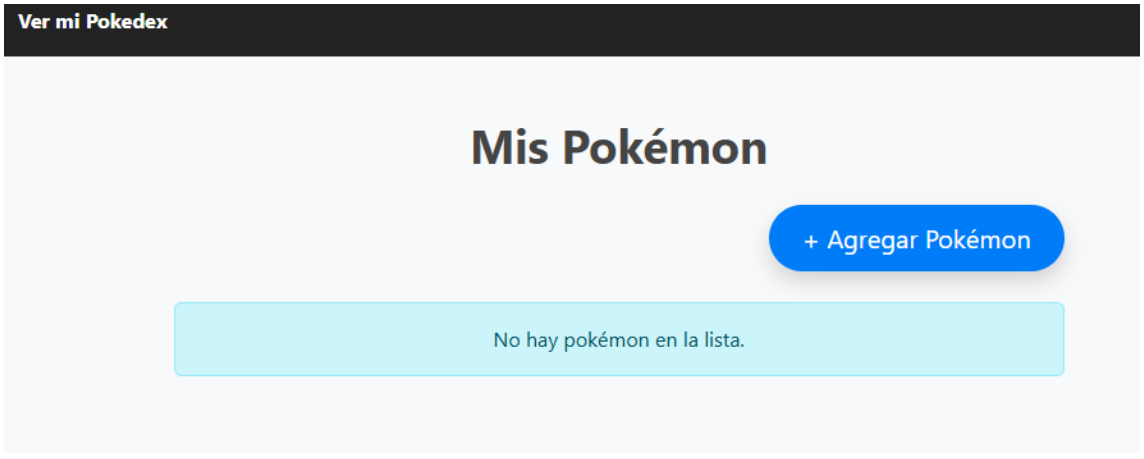
Para conectar la vista con la base de datos, he implementado el patrón Observer mediante el método **.subscribe()**

He utilizado la función **inject(PokemonService)** para dar acceso al componente a los métodos de comunicación HTTP.

La petición se dispara automáticamente al cargar el componente, asegurando que el usuario vea la lista actualizada de imnediato gracias al uso de **ngOnInit**

Al recibir la respuesta del servidor, al variable **listaPokemon** se actualiza.

Al probarlo podemos comprobar que funciona, porque todavía no me he conectado a la base de datos y por eso me sale el mensaje de “No hay Pokémon en la lista”.



2.5 Configuración del entorno y servidor

Estoy utilizando Node.js con el framework Express, utilizando un proyecto anterior que ya teníamos de pokemon.

Dependencias principales: express,mongoose,dotenv

Puerto del servidor: Configurado en el puerto 4000

Variables de Entorno: Uso de un archivo .env para proteger las credenciales de conexión al clúster.

Actualización del archivo pokemon.ts en angular

OptativaPokemon > src > app > components > pokemon > TS pokemon.ts > Pokemon > agregarPokemon

```ts
import { Component, OnInit, inject } from '@angular/core';
import { CommonModule } from '@angular/common';
import { PokemonService, PokemonData } from '../../services/pokemon.service';

@Component({
  selector: 'app-pokemon',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './pokemon.html',
  styleUrl: './pokemon.css',
})
export class Pokemon implements OnInit {
  // Array para los datos de la base de datos
  listaPokemon: PokemonData[] = [];

  // Inyectamos el servicio
  private pokemonService = inject(PokemonService);

  ngOnInit(): void {
    this.obtenerPokemons();
  }

  obtenerPokemons(): void {
    this.pokemonService.getPokemons().subscribe({
      next: (datos) => {
        this.listaPokemon = datos;
      },
      error: (e) => {
        console.error('Error al recuperar los Pokémon:', e);
      }
    });

  }

  agregarPokemon(): void {
    // Objeto directo sin validación
    const nuevoPokemon: PokemonData = {
      nombre: "Pikachu",
      tipo: "Eléctrico",
      descripcion: "Ratón eléctrico amarillo"
    };

    this.pokemonService.postPokemon(nuevoPokemon).subscribe({
      next: (res) => {
        console.log('Pokémon guardado:', res);
        this.obtenerPokemons();
      },
      error: (e) => console.error('Error al guardar:', e)
    });
  }


}
```

Actualización del archivo pokemon.service

2.5.1 Transformación a API REST (Modificación de 01-express.js)

Para que Angular pueda procesar la información, he modificado la lógica del servidor en el archivo 01-express.js.

He sustituido el envío de vistas renderizadas (HTML) por el método res.json(), permitiendo que el servidor actúe como una API de datos pura.

Compatibilidad con Angular: Angular espera recibir objetos o arrays de datos para que sus directivas, como @for, puedan iterar sobre ellos en el frontend.

2.5.2 Configuración de CORS

Al tener el servidor de Angular en un puerto (4200) y el servidor de Node.js en otro (puerto 4000), he tenido que habilitar la política de intercambio de recursos de origen cruzado (CORS).

2.5.3 Implementación del modelo y rutas de datos

He vinculado el esquema de Mongoose definido en ejercicios anteriores con las rutas del servidor.

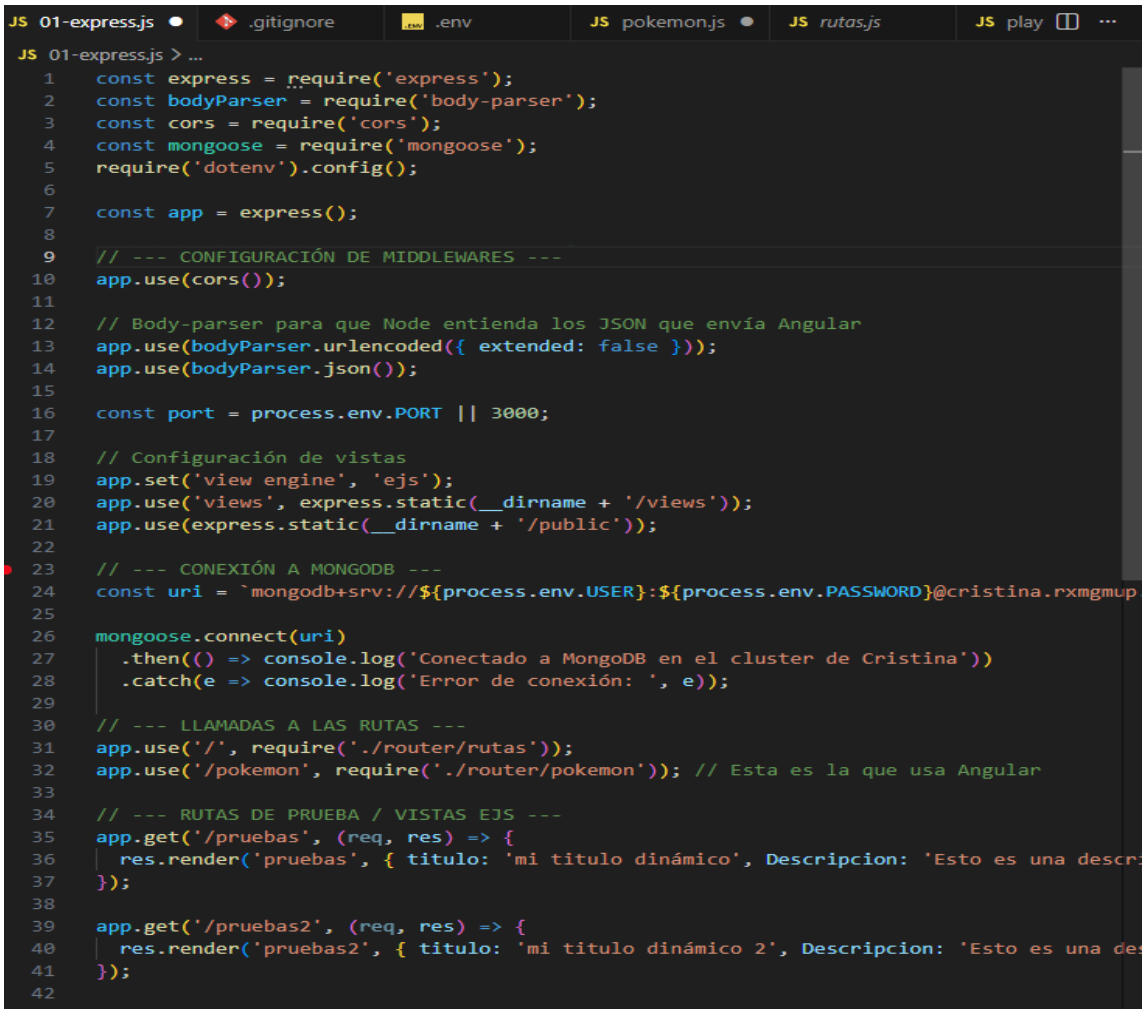
El servicio de Angular (pokemon.service.ts) realiza una petición GET a esta ruta, recibe el JSON y lo inyecta en el componente mediante la suscripción al Observable.

2.5.4 Estado final de la integración

El servidor se conecta al **cluster de Cristina** usando mongoose y las variables de .env.

Angular solicita los datos al cargar el componente mediante ngOnInit.

Si la base de datos devuelve un array vacío, el bloque @empty muestra el mensaje de feedback informativo: *"No hay pokémon en la lista"*.

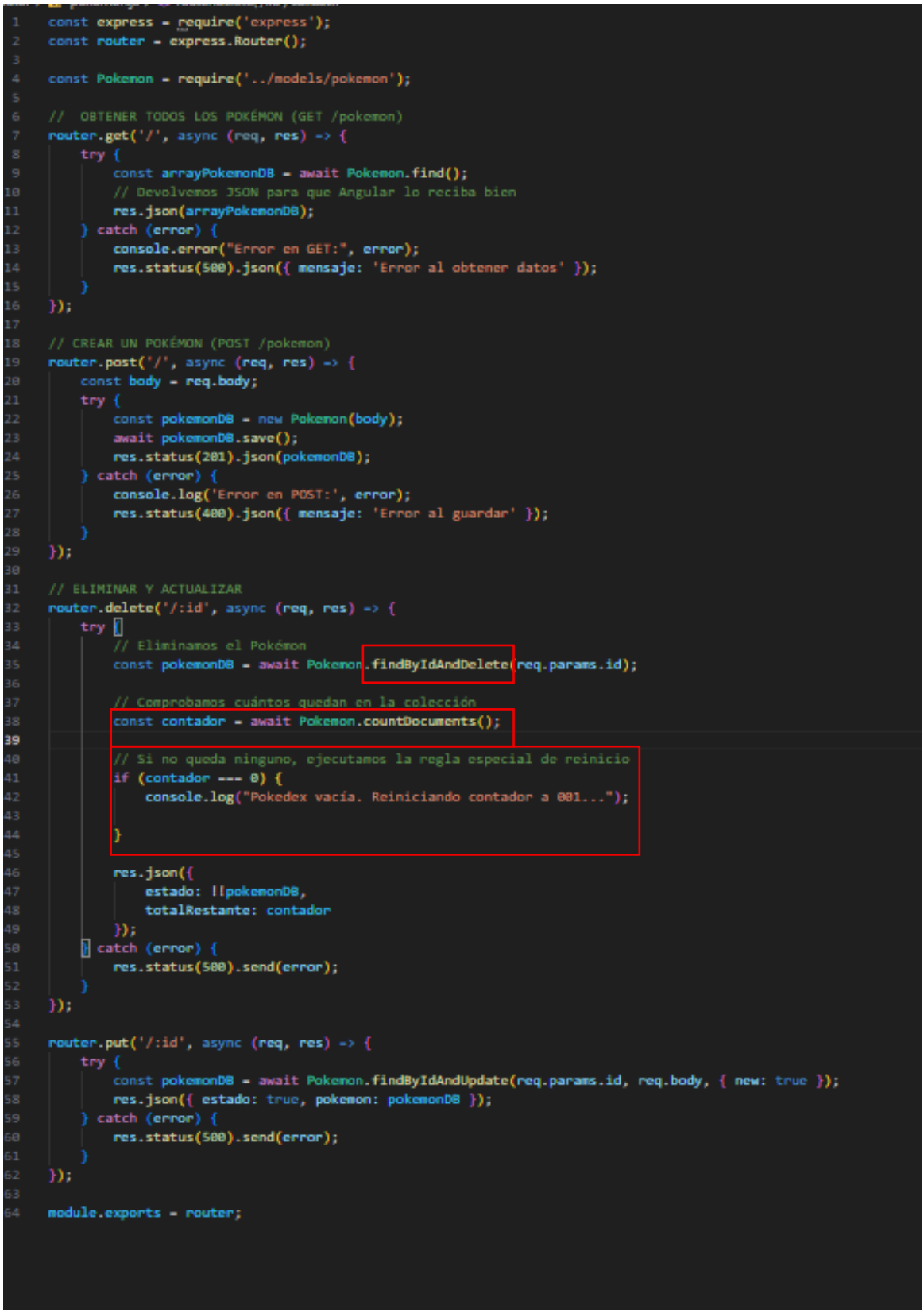


2.6 Desarrollo CRUD

2.6.1.Implementación de la API y Reglas de Negocio (Backend)

Para que el CRUD sea funcional y profesional, he configurado el servidor Express para que actúe como una API JSON. Una de las aportaciones adicionales más importantes es la **gestión automática del contador de IDs**:

- **Ruta de Borrado:** Se ha implementado en router/pokemon.js utilizando findByIdAndDelete.
- **Lógica de Reinicio:** Al eliminar un registro, el servidor ejecuta countDocuments() para verificar si la colección está vacía
- **Regla Especial:** Si el conteo es 0, el sistema está programado para reiniciar la secuencia de IDs al valor **001**, asegurando que la Pokedex siempre comience con un orden lógico tras una limpieza total

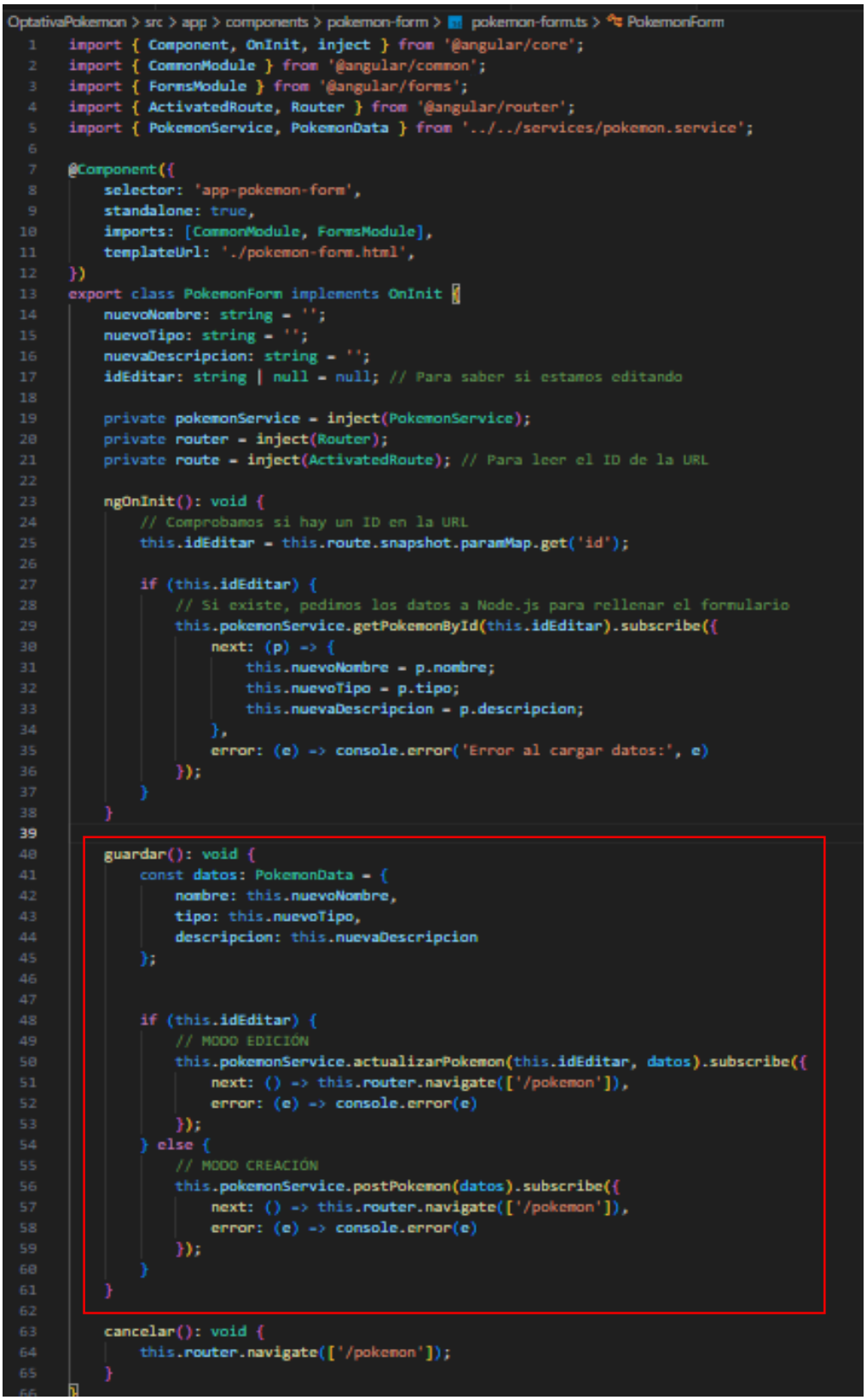


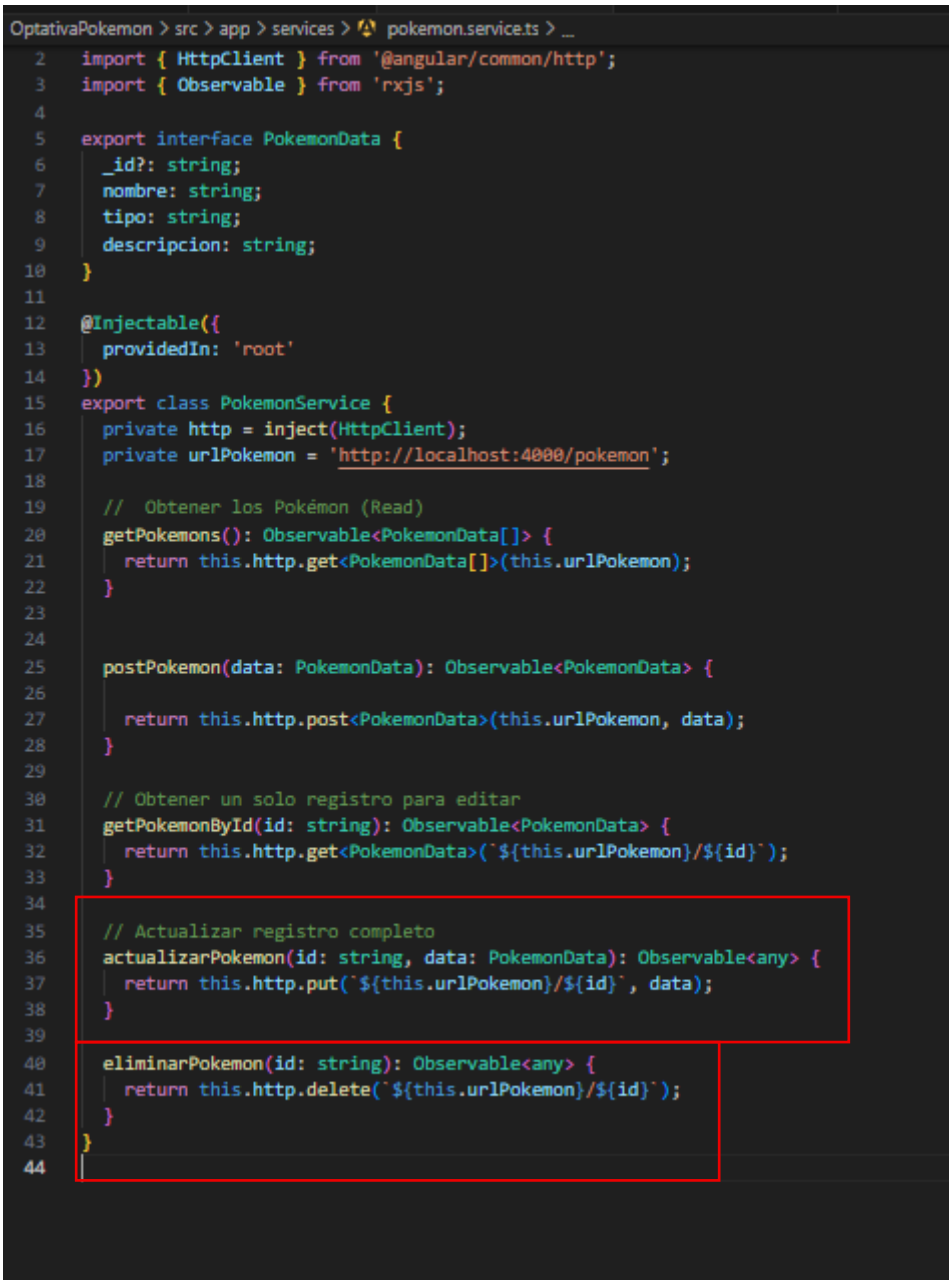
```
const express = require('express');
const router = express.Router();

const Pokemon = require('../models/pokemon');

//  OBTENER TODOS LOS POKÉMON (GET /pokemon)
router.get('/', async (req, res) => {
    try {
        const arrayPokemonDB = await Pokemon.find();
        // Devolvemos JSON para que Angular lo reciba bien
        res.json(arrayPokemonDB);
    } catch (error) {
        console.error("Error en GET:", error);
        res.status(500).json({ mensaje: 'Error al obtener datos' });
    }
});

// CREAR UN POKÉMON (POST /pokemon)
router.post('/', async (req, res) => {
    const body = req.body;
    try {
        const pokemonDB = new Pokemon(body);
        await pokemonDB.save();
        res.status(201).json(pokemonDB);
    } catch (error) {
        console.log('Error en POST:', error);
        res.status(400).json({ mensaje: 'Error al guardar' });
    }
});

// ELIMINAR Y ACTUALIZAR
router.delete('/:id', async (req, res) => {
    try {
        // Eliminamos el Pokémon
        const pokemonDB = await Pokemon.findByIdAndDelete(req.params.id);

        // Comprobamos cuántos quedan en la colección
        const contador = await Pokemon.countDocuments();

        // Si no queda ninguno, ejecutamos la regla especial de reinicio
        if (contador === 0) {
            console.log("Pokedex vacía. Reiniciando contador a 001...");

        }

        res.json({
            estado: !!pokemonDB,
            totalRestante: contador
        });
    } catch (error) {
        res.status(500).send(error);
    }
});

router.put('/:id', async (req, res) => {
    try {
        const pokemonDB = await Pokemon.findByIdAndUpdate(req.params.id, req.body, { new: true });
        res.json({ estado: true, pokemon: pokemonDB });
    } catch (error) {
        res.status(500).send(error);
    }
});

module.exports = router;
```

2.6.2. Implementación de Operaciones CRUD (Frontend)

Una vez establecidas las reglas en el servidor, he procedido a activar la lógica en Angular para que los botones, anteriormente estéticos, realicen cambios reales en la base de datos.

- **Create (Creación)**: He desarrollado un componente específico (PokemonForm) que captura los datos mediante un formulario dinámico.
- **Update (Actualización)**: He implementado una lógica de edición integral. Al pulsar "Actualizar", el sistema redirige al usuario a una nueva página donde se recuperan los datos actuales de MongoDB para ser modificados simultáneamente.
- **Delete (Borrado)**: Se ha vinculado el botón de eliminación con una alerta de confirmación nativa. Tras la aprobación del usuario, se invoca el método DELETE del servicio, disparando la lógica de reinicio de IDs en el backend si la colección queda vacía.


OptativaPokemon > src > app > components > pokemon-form > pokemon-form.ts > PokemonForm
import { Component, OnInit, inject } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { ActivatedRoute, Router } from '@angular/router';
import { PokemonService, PokemonData } from '../../services/pokemon.service';
@Component({
selector: 'app-pokemon-form',
standalone: true,
imports: [CommonModule, FormsModule],
templateUrl: './pokemon-form.html',
})
export class PokemonForm implements OnInit {
nuevoNombre: string = '';
nuevoTipo: string = '';
nuevaDescripcion: string = '';
idEditar: string | null = null; // Para saber si estamos editando
private pokemonService = inject(PokemonService);
private router = inject(Router);
private route = inject(ActivatedRoute); // Para leer el ID de la URL
ngOnInit(): void {
// Comprobamos si hay un ID en la URL
this.idEditar = this.route.snapshot.paramMap.get('id');
if (this.idEditar) {
// Si existe, pedimos los datos a Node.js para rellenar el formulario
this.pokemonService.getPokemonById(this.idEditar).subscribe({
next: (p) => {
this.nuevoNombre = p.nombre;
this.nuevoTipo = p.tipo;
this.nuevaDescripcion = p.descripcion;
},
error: (e) => console.error('Error al cargar datos:', e)
});
}
}
guardar(): void {
const datos: PokemonData = {
nombre: this.nuevoNombre,
tipo: this.nuevoTipo,
descripcion: this.nuevaDescripcion
};
if (this.idEditar) {
// MODO EDICIÓN
this.pokemonService.actualizarPokemon(this.idEditar, datos).subscribe({
next: () => this.router.navigate(['/pokemon']),
error: (e) => console.error(e)
});
} else {
// MODO CREACIÓN
this.pokemonService.postPokemon(datos).subscribe({
next: () => this.router.navigate(['/pokemon']),
error: (e) => console.error(e)
});
}
}
cancelar(): void {
this.router.navigate(['/pokemon']);
}
}



2.6.3. Sincronización y Detección de Cambios

Para garantizar que la aplicación cargue los datos "del tirón" tras realizar una operación CRUD, he aplicado técnicas avanzadas de renderizado:

- **Refresco Automático**: Tras cada inserción, edición o borrado, el componente ejecuta una nueva suscripción al método getPokemons(), actualizando la galería morada de forma instantánea.
- **ChangeDetectorRef**: He integrado la detección de cambios manual para asegurar que, tras recibir el array de datos desde el puerto 4000, la interfaz de Angular se "hidrate" correctamente y elimine el estado de carga inicial.

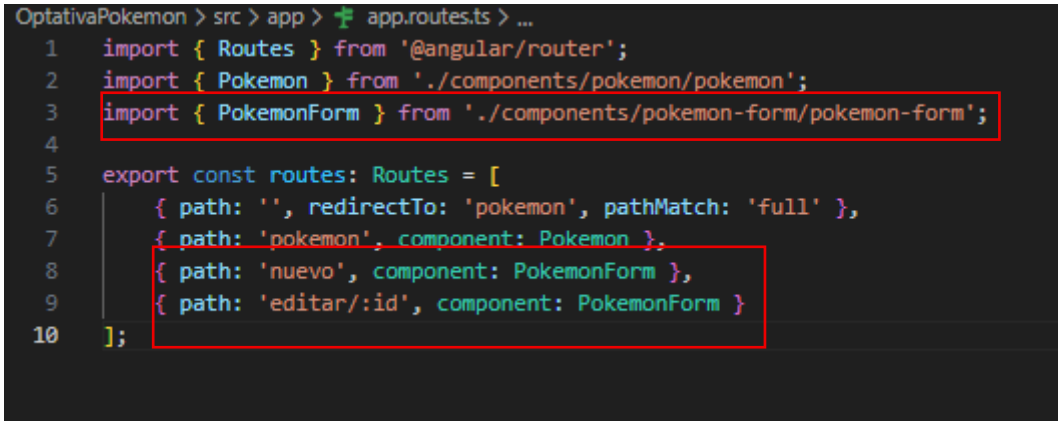
OptativaPokemon > src > app > components > pokemon > pokemon.ts > Pokemon > obtenerPokemons

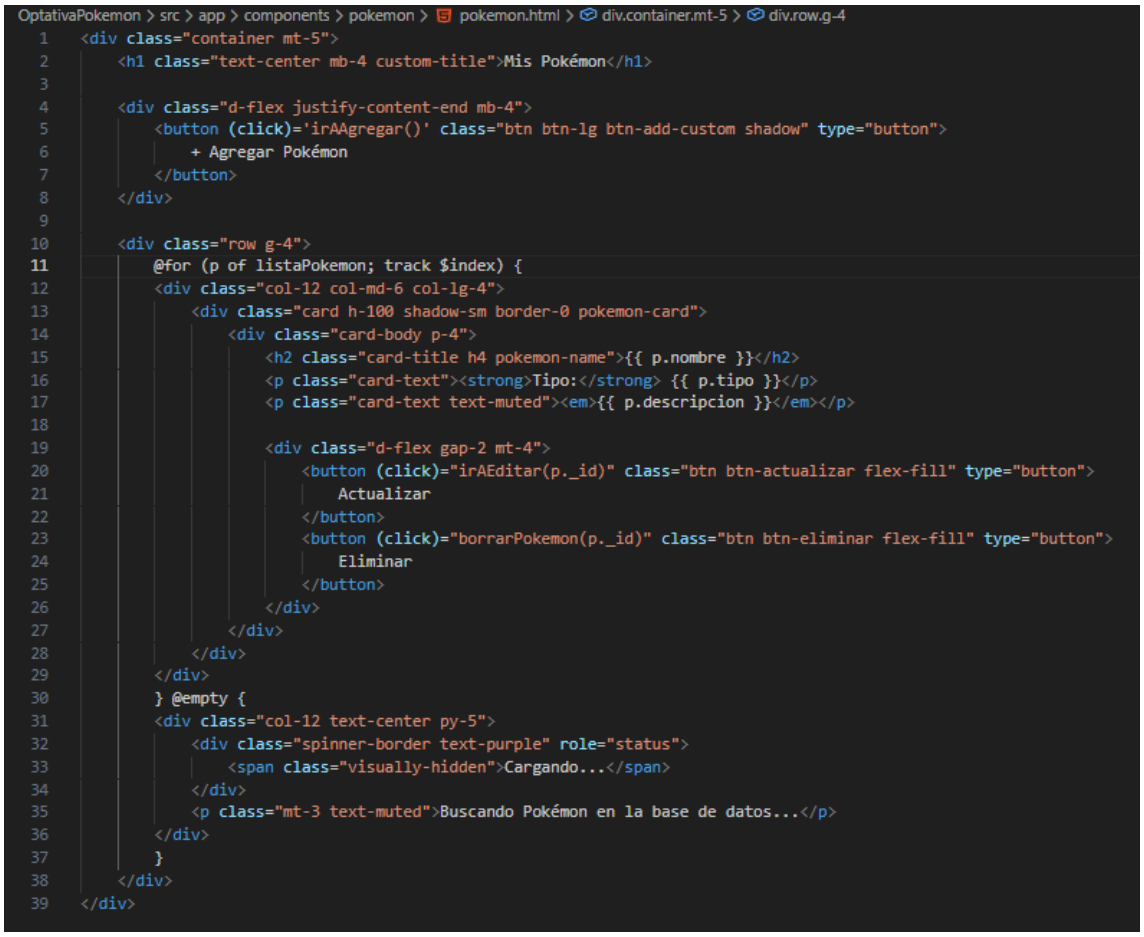
```ts
import { Component, OnInit, inject, PLATFORM_ID, ChangeDetectorRef } from '@angular/core';
import { CommonModule, isPlatformBrowser } from '@angular/common'; // Importamos estas utilidades
import { Router } from '@angular/router';
import { PokemonService, PokemonData } from '../../services/pokemon.service';

@Component({
  selector: 'app-pokemon',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './pokemon.html',
  styleUrl: './pokemon.css',
})
export class Pokemon implements OnInit {
  listaPokemon: PokemonData[] = [];

  // Inyectamos el ID de la plataforma para evitar fallos de carga inicial
  private platformId = inject(PLATFORM_ID);
  private pokemonService = inject(PokemonService);
  private router = inject(Router);
  private cd = inject(ChangeDetectorRef);

  ngOnInit(): void {
    if (isPlatformBrowser(this.platformId)) {
      this.obtenerPokemons();
    }
  }

  obtenerPokemons(): void {
    this.pokemonService.getPokemons().subscribe({
      next: (datos) => {
        // 1. Forzamos una nueva referencia del array
        this.listaPokemon = [...datos];
        console.log("¡Datos recibidos!", this.listaPokemon);

        // 2. OBLIGAMOS a la interfaz a refrescarse
        this.cd.detectChanges();
      },
      error: (e) => console.error("Error:", e)
    });
  }

  irAAgregar(): void {
    this.router.navigate(['/nuevo']);
  }

  irAEditar(id: string | undefined): void {
    if (id) this.router.navigate(['/editar', id]);
  }

  borrarPokemon(id: string | undefined): void {
    if (id && confirm('¿Eliminar este Pokémon?')) {
      this.pokemonService.eliminarPokemon(id).subscribe({
        next: () => this.obtenerPokemons(),
        error: (e) => console.error(e)
      });
    }
  }
}
```

2.7. Navegación y Experiencia de Usuario

He transformado la arquitectura de una aplicación estática a una **Single Page Application (SPA)** con navegación por rutas:

- **Rutas Dinámicas**: En el archivo app.routes.ts, he definido caminos para el listado (/pokemon), la creación (/nuevo) y la edición parametrizada (/editar/:id).
- **Navegación Programática**: Utilizando el servicio Router, la aplicación redirige al usuario automáticamente a la lista principal una vez que el servidor confirma que el Pokémon ha sido guardado o actualizado con éxito.
- **Feedback Visual**: Se mantiene el uso del bloque @empty que, en combinación con el contador del backend, informa al usuario cuando la Pokedex ha sido reiniciada a sus valores por defecto.





3.Conclusión

La realización de este proyecto **me ha permitido consolidar** la integración de un ecosistema **Full-Stack** moderno, utilizando **Angular** como frontend y **Node.js** como backend.

A través de la implementación de componentes **Standalone** y el uso de **ApplicationConfig**, **he comprobado** que esta arquitectura supone una evolución mucho más eficiente respecto a los modelos tradicionales, ya que **me ha simplificado** enormemente la inyección de dependencias críticas, como el cliente HTTP y el enrutador. Asimismo, al desarrollar **mi propia API REST**, **he logrado** tener un control total sobre las reglas de negocio, destacando especialmente la gestión automatizada de IDs y la persistencia de datos en el clúster de **MongoDB Atlas**.

En definitiva, **he conseguido crear** una aplicación reactiva, escalable y con una experiencia de usuario fluida, cumpliendo con los estándares de calidad que exige el desarrollo web profesional hoy en día.

4.Enlaces github

https://github.com/crisanlab94/2-Dam/tree/main/Optativa/03Express

https://github.com/crisanlab94/2-Dam/tree/main/Optativa/Optativa_pokemon