Documento Técnico: Implementación de la API REST y Cliente-Servidor para la Gestión de Pokémon

Resumen del Proyecto

Este proyecto consiste en una **API REST** que permite gestionar información sobre los Pokémon. La aplicación se estructura en dos partes principales: el **servidor**, que expone la API REST, y el **cliente**, que interactúa con la API para realizar operaciones CRUD sobre los datos de los Pokémon. La base de datos es un archivo **JSON**, y la comunicación entre el cliente y el servidor se realiza a través de HTTP.

Arquitectura del Proyecto

La arquitectura se organiza en tres capas principales:

1. **API REST**: Esta capa es responsable de gestionar las solicitudes HTTP que se reciben del cliente y devolver las respuestas adecuadas. Utiliza un **controlador** que maneja las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre los Pokémon.

2. **Cliente**: La capa del cliente hace uso de peticiones HTTP para interactuar con la API REST. Los clientes pueden ser aplicaciones de consola o interfaces gráficas que consumen la API para gestionar los Pokémon.

3. **Servicios y Manejo de Datos**: Los servicios están diseñados para manejar la lógica de negocio relacionada con los Pokémon. Además, gestionan el acceso y almacenamiento de los datos, en este caso, un archivo JSON. La manipulación de estos datos debe ser segura, utilizando mecanismos de concurrencia y cifrado.
4.

1. API REST: El Servidor

La API REST está construida sobre **ASP.NET Core**, que permite crear servicios web eficientes y escalables. En este servidor, las rutas HTTP están configuradas para permitir operaciones sobre los datos de los Pokémon.

Controlador de Pokémon (`PokemonController.cs`)

El **controlador** es el componente central que gestiona las peticiones HTTP que llegan al servidor. Cada operación HTTP se mapea a un método en el controlador, y este se encarga de ejecutar las operaciones necesarias y devolver una respuesta adecuada. Aquí se utilizan los métodos estándar **GET**, **POST**, **PUT**, y **DELETE** para gestionar los Pokémon.

Cuando el servidor recibe una solicitud, el controlador recibe los datos enviados por el cliente (en formato JSON) y los manipula. Por ejemplo, al hacer una solicitud **GET**, el controlador recupera los Pokémon almacenados en el archivo JSON y los envía como respuesta.

Ejemplo de método GET: Recuperar todos los Pokémon

El método `GetAllPokemons` carga todos los Pokémon almacenados en un archivo JSON y los devuelve en formato JSON como respuesta HTTP. Si no se encuentran datos, la respuesta será un código de estado **404 Not Found**.

```
[HttpGet]
public IActionResult GetAllPokemons()
{
    var pokemons = LoadPokemons(); // Cargar los Pokémon desde el
archivo
    return Ok(pokemons); // Devolver los Pokémon como respuesta
}
```

De manera similar, el controlador puede manejar otras operaciones como crear un nuevo Pokémon (`POST`), actualizar un Pokémon existente (`PUT`), o eliminar un Pokémon (`DELETE`).

Operaciones CRUD:

- **GET**: Obtiene la lista de todos los Pokémon o un Pokémon específico.
- **POST**: Crea un nuevo Pokémon y lo agrega al archivo JSON.
- **PUT**: Actualiza la información de un Pokémon existente.
- **DELETE**: Elimina un Pokémon por su ID.

Modelo de Pokémon (`Pokemon.cs`)

El modelo `Pokemon` define las propiedades que cada Pokémon tiene, como el `Id`, `Name`, `Type`, `Hp` y `Poder`. Estas propiedades son utilizadas para crear, actualizar y devolver los Pokémon a través de la API.

```
public class Pokemon
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Type { get; set; }
    public int Hp { get; set; }
    public int Poder { get; set; }
}
```

Cada una de estas propiedades corresponde a un campo en el archivo JSON que almacena los Pokémon. Este modelo asegura que las respuestas sean consistentes y los datos sean manipulados correctamente.

2. Cliente-Servidor: Interacción con la API

El cliente en este proyecto es el componente que hace las solicitudes HTTP al servidor. Puede ser una aplicación de consola, una aplicación web o cualquier otro tipo de cliente que pueda interactuar con la API REST.

Realizando solicitudes HTTP desde el Cliente

El cliente se comunica con la API a través de solicitudes HTTP. Utilizando **HttpClient** en C#, se pueden hacer solicitudes **GET**, **POST**, **PUT**, y **DELETE**. Estas solicitudes son enviadas a la URL del servidor y la API responde con los resultados esperados.

Ejemplo de Cliente que hace una solicitud GET:

En este caso, el cliente envía una solicitud **GET** para obtener todos los Pokémon. La respuesta es un JSON con la lista de Pokémon.

```
public async Task GetPokemons()
{
    var response = await
client.GetStringAsync("http://localhost:5062/api/pokemon");
    Console.WriteLine(response); // Muestra los Pokémon obtenidos
}
```

Creación de un Nuevo Pokémon (POST)

Para crear un nuevo Pokémon, el cliente debe enviar una solicitud **POST** con los datos del Pokémon en formato JSON. La solicitud se realiza al endpoint `/api/pokemon` con los datos en el cuerpo de la petición.

```
public async Task CreatePokemon()
{
    var newPokemon = new
    {
        Name = "Pikachu",
        Type = "Electric",
        Hp = 100,
        Poder = 50
    };
```

```
    var json = JsonConvert.SerializeObject(newPokemon);
    var content = new StringContent(json, Encoding.UTF8,
"application/json");

    var response = await
client.PostAsync("http://localhost:5062/api/pokemon", content);
    if (response.IsSuccessStatusCode)
    {
        Console.WriteLine("Pokémon creado exitosamente");
    }
}
```

Este código crea un nuevo Pokémon y lo envía al servidor para que sea almacenado.

3. Servicios: Manipulación de Datos

El manejo de datos, como la carga y el almacenamiento de los Pokémon, se realiza en un servicio específico. Este servicio maneja la lectura y escritura del archivo **pokemons.json**, que actúa como base de datos.

Servicio de Pokémon (`PokemonService.cs`)

Este servicio tiene la responsabilidad de leer los Pokémon desde el archivo JSON y escribir los cambios de vuelta en el archivo. Utiliza **Newtonsoft.Json** para serializar y deserializar los datos.

Ejemplo de carga de Pokémon:

El servicio carga los datos de los Pokémon de la siguiente manera. Si el archivo no existe, se devuelve una lista vacía.

```
public List<Pokemon> GetPokemons()
{
    if (!File.Exists(_filePath))
        return new List<Pokemon>();

    var json = File.ReadAllText(_filePath);
```

```
    return JsonConvert.DeserializeObject<List<Pokemon>>(json) ?? new
List<Pokemon>();
}
```

De manera similar, el servicio también maneja la escritura de los Pokémon actualizados en el archivo.

Seguridad y Manejo de Concurrencia

Es importante destacar que el acceso al archivo JSON debe ser seguro, ya que varias solicitudes podrían intentar modificar los datos al mismo tiempo. Se implementa un mecanismo de **bloqueo** utilizando un objeto de bloqueo (`_lock`) para evitar problemas de concurrencia cuando múltiples hilos intentan leer o escribir en el archivo al mismo tiempo.

```
public void SavePokemons(List<Pokemon> pokemons)
{
    lock (_lock)
    {
        var json = JsonConvert.SerializeObject(pokemons,
Formatting.Indented);
        File.WriteAllText(_filePath, json);
    }
}
```

Este mecanismo asegura que solo un hilo pueda acceder al archivo a la vez, evitando que se corrompan los datos.

4. Configuración de la Aplicación

La configuración de la aplicación se maneja a través del archivo `launchSettings.json`, donde se define la URL local del servidor (por ejemplo, `http://localhost:5062`). Este archivo permite configurar el entorno de desarrollo y otras variables necesarias para ejecutar la aplicación de manera adecuada.

```
{
  "profiles": {
    "http": {
      "applicationUrl": "http://localhost:5062",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
```

```
    },
    "https": {
      "applicationUrl":
"https://localhost:7149;http://localhost:5062",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Conclusión

Este proyecto demuestra cómo construir una API REST que maneja operaciones CRUD sobre los Pokémon, con un cliente que interactúa con la API a través de solicitudes HTTP. Además, se implementa un servicio para la gestión de los datos, asegurando su integridad y seguridad. La arquitectura modular facilita el mantenimiento del sistema y permite ampliaciones futuras, como la incorporación de bases de datos o funcionalidades adicionales.
