Desarrollo de una API REST utilizando .NET y Entity Framework para la base de datos Northwind

Cristian Jurado, Dennis Quisaguano
Universidad Técnica de Ambato
Software
Metodologías Ágiles

29 de mayo de 2025

1. Objetivo General

Desarrollar una API REST utilizando la tecnología .NET 9 (o superior), que permita realizar operaciones CRUD sobre la base de datos Northwind, enfocándose en la tabla *Products*, utilizando Entity Framework como ORM.

2. Objetivos Específicos

- Comprender el uso de .NET y Visual Studio para el desarrollo de servicios Web.
- Establecer una conexión funcional entre Entity Framework y la base de datos Northwind.
- Implementar controladores en la API que gestionen las operaciones CRUD.
- Documentar y probar los endpoints utilizando Swagger.
- Probar la API REST desde un cliente externo como Postman y un archivo HTML personalizado.

3. Introducción

Las API RESTful son una de las herramientas más utilizadas en el desarrollo moderno para conectar sistemas, exponer datos y permitir la interoperabilidad [1]. Este proyecto tuvo como propósito la creación de una API REST básica en .NET, utilizando como fuente de

datos la base Northwind, una base de datos de ejemplo que contiene información relacionada con ventas, productos, clientes y pedidos [2].

Se utilizó Visual Studio como entorno de desarrollo, .NET 9 como tecnología base, y Entity Framework como herramienta para interactuar con la base de datos. Se trabajó con la tabla *Products*, permitiendo realizar operaciones básicas como crear, leer, actualizar y eliminar registros.

4. Marco teórico

4.1. Qué es Nortwind

Northwind API se refiere a una base de datos de ejemplo ficticia utilizada para practicar y aprender en el contexto de la tecnología de desarrollo de software, especialmente en el ecosistema de Microsoft. La base de datos representa una pequeña empresa de comercio, Northwind Traders, que gestiona pedidos, productos, clientes, proveedores, entre otros aspectos. Es una herramienta popular para desarrollar aplicaciones que interactúan con datos, como aplicaciones web o de escritorio [3].

4.2. Qué es Entity Framework

Entity Framework es una tecnología de Microsoft que facilita el acceso y la gestión de datos en aplicaciones, permitiendo a los desarrolladores trabajar con datos como objetos en lugar de escribir directamente consultas SQL. Es un sistema de mapeo objeto-relacional (ORM) que simplifica la interacción entre aplicaciones y bases de datos [3]. Permite convertir datos de bases de datos en objetos del lenguaje de programación, facilitando la manipulación y consulta de datos de manera más intuitiva y orientada a objetos [3].

4.3. Qué es Swagger

Swagger es una especificación y un conjunto de herramientas ampliamente utilizadas en el desarrollo de software para describir, documentar y automatizar APIs web, especialmente aquellas basadas en REST. Su función principal es facilitar la creación, documentación y prueba de APIs, mejorando la comunicación entre desarrolladores y consumidores de servicios. Swagger permite describir de manera clara y estructurada cómo funciona una API, sus endpoints, parámetros y respuestas, lo que facilita la comprensión y el uso por parte de otros desarrolladores[4].

5. Desarrollo

5.1. Restauración de la base de datos Northwind

La base de datos Northwind fue restaurada localmente en SQL Server Management Studio a partir de un archivo .bak. Esta base contiene varias tablas relacionadas entre sí, de las cuales se seleccionó la tabla *Products* para implementar el CRUD.

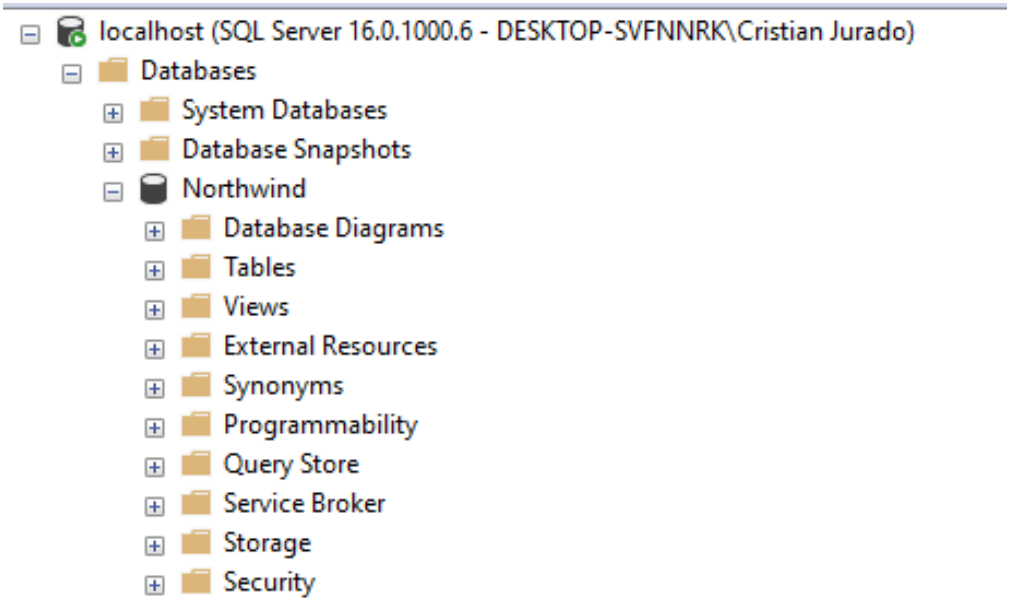


Figura 1: Restauración de la base de datos Northwind en SQL Server

5.2. Creación del proyecto API en Visual Studio

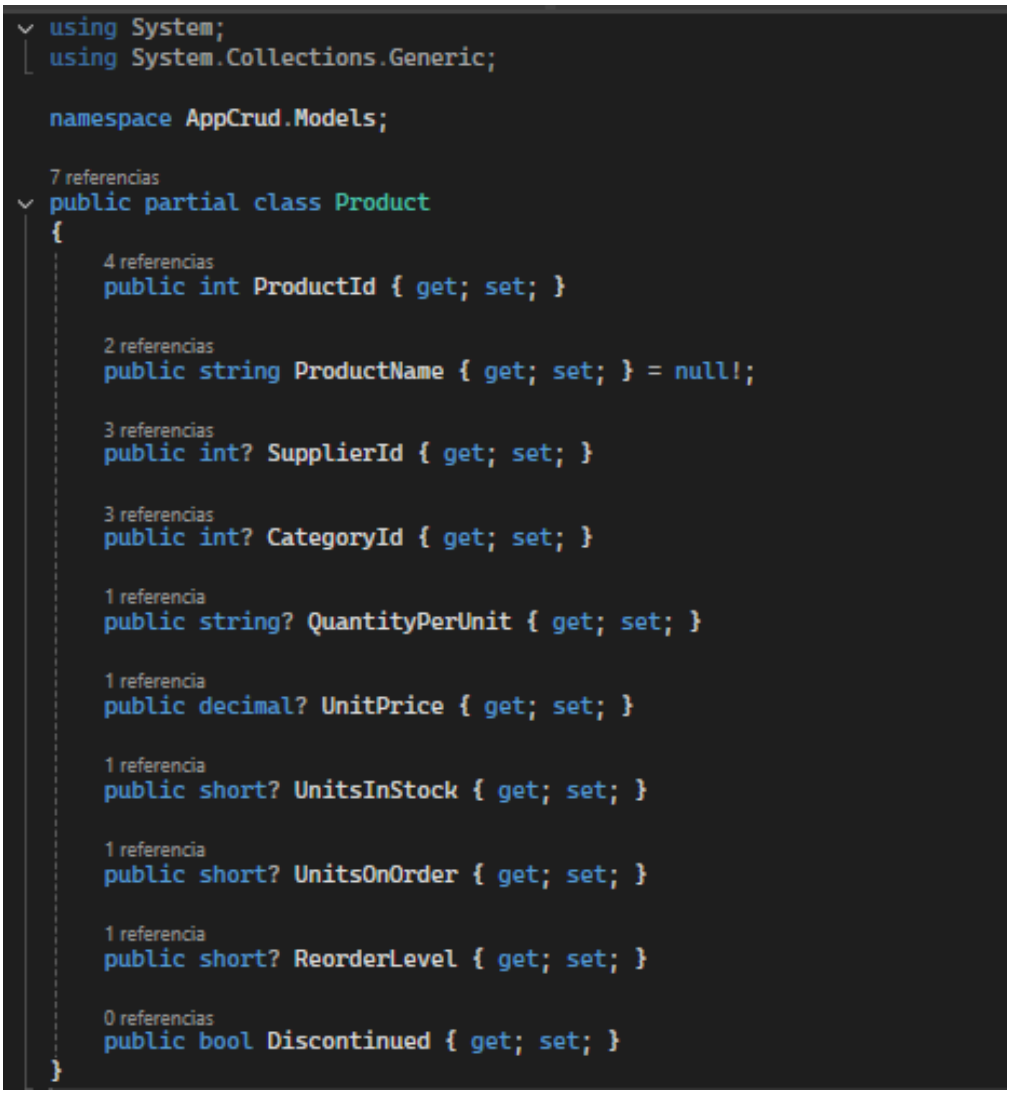
Se creó un nuevo proyecto de tipo *ASP.NET Core Web API*, habilitando el uso de controladores y Swagger desde el inicio. Posteriormente se configuró el archivo `launchSettings.json` para que el API funcionara exclusivamente sobre HTTPS, en el puerto 5268 [5].

Figura 2: Configuración del puerto en `launchSettings.json`

5.3. Instalación de Entity Framework y Scaffolding

Se instalaron los paquetes necesarios mediante NuGet: `Microsoft.EntityFrameworkCore.SqlServer` y `Microsoft.EntityFrameworkCore.Tools`. Luego, mediante la consola del administrador de paquetes se utilizó el comando `Scaffold-DbContext` para generar el modelo y el contexto:

```
Scaffold-DbContext "Server=localhost;Database=Northwind;
Trusted_Connection=True;TrustServerCertificate=True;"
Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -Tables Products
```



```
using System;
using System.Collections.Generic;

namespace AppCrud.Models;

7 referencias
public partial class Product
{
    4 referencias
    public int ProductId { get; set; }

    2 referencias
    public string ProductName { get; set; } = null!;

    3 referencias
    public int? SupplierId { get; set; }

    3 referencias
    public int? CategoryId { get; set; }

    1 referencia
    public string? QuantityPerUnit { get; set; }

    1 referencia
    public decimal? UnitPrice { get; set; }

    1 referencia
    public short? UnitsInStock { get; set; }

    1 referencia
    public short? UnitsOnOrder { get; set; }

    1 referencia
    public short? ReorderLevel { get; set; }

    0 referencias
    public bool Discontinued { get; set; }
}
```

Figura 3: Generación del modelo y DbContext con Scaffold

5.4. Implementación del controlador CRUD

Se agregó un nuevo controlador para la tabla `Products`, utilizando el generador automático de Visual Studio. Esto proporcionó los métodos HTTP básicos: `GET`, `POST`, `PUT` y `DELETE`.

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using AppCrud.Models;

namespace AppCrud.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    1 referencia
    public class ProductsController : ControllerBase
    {
        private readonly NorthwindContext _context;

        0 referencias
        public ProductsController(NorthwindContext context)
        {
            _context = context;
        }

        // GET: api/Products
        [HttpGet]
        0 referencias
        public async Task<ActionResult<IEnumerable<Product>>> GetProducts()
        {
            return await _context.Products.ToListAsync();
        }

        // GET: api/Products/5
        [HttpGet("{id}")]
        0 referencias
        public async Task<ActionResult<Product>> GetProduct(int id)
        {
            var product = await _context.Products.FindAsync(id);

            if (product == null)
            {
                return NotFound();
            }

            return product;
        }
```

Figura 4: Código generado del controlador de Products

```
// PUT: api/Products/5
// To protect from overposting attacks, see https://go.microsoft.com/fwlink/?linkid=2123754
[HttpPut("{id}")]
0 referencias
public async Task<IActionResult> PutProduct(int id, Product product)
{
    if (id != product.ProductId)
    {
        return BadRequest();
    }

    _context.Entry(product).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!ProductExists(id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return NoContent();
}
```

Figura 5: Código generado del controlador de Products

```
    // POST: api/ProductsSSSSS
    // To protect from overposting attacks, see https://go.microsoft.com/fwlink/?linkid=2123754
    [HttpPost]
    0 referencias
    public async Task<ActionResult<Product>> PostProduct(Product product)
    {
        _context.Products.Add(product);
        await _context.SaveChangesAsync();

        return CreatedAtAction("GetProduct", new { id = product.ProductId }, product);
    }

    // DELETE: api/Products/5
    [HttpDelete("{id}")]
    0 referencias
    public async Task<IActionResult> DeleteProduct(int id)
    {
        var product = await _context.Products.FindAsync(id);
        if (product == null)
        {
            return NotFound();
        }

        _context.Products.Remove(product);
        await _context.SaveChangesAsync();

        return NoContent();
    }

    1 referencia
    private bool ProductExists(int id)
    {
        return _context.Products.Any(e => e.ProductId == id);
    }
}
```

Figura 6: Código generado del controlador de Products

5.5. Documentación con Swagger

Swagger fue activado en el archivo `Program.cs`, permitiendo explorar visualmente los endpoints de la API. Esta herramienta facilitó las pruebas iniciales sin necesidad de cliente externo [4].

```
using AppCrud.Models;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);


builder.Services.AddCors(options =>
{
    options.AddPolicy("PermitirTodo", policy =>
    {
        policy.AllowAnyOrigin()
              .AllowAnyMethod()
              .AllowAnyHeader();
    });
});

builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

builder.Services.AddDbContext<NorthwindContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

var app = builder.Build();

app.UseCors("PermitirTodo");

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();
app.Run();
```

Figura 7: Interfaz de Swagger mostrando los endpoints CRUD

5.6. Pruebas con Postman y cliente HTML

Para validar el correcto funcionamiento de la API, se realizaron pruebas en Postman y también se desarrolló un archivo HTML personalizado con Bootstrap, tablas y botones, que interactúa con la API mediante JavaScript y fetch.

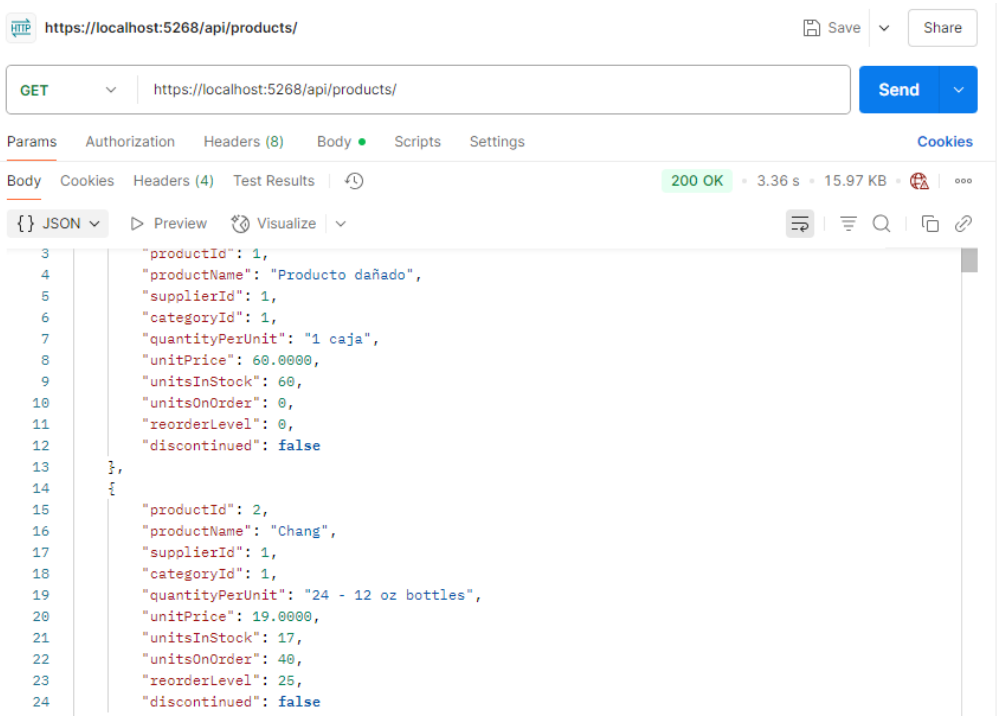


Figura 8: Prueba de endpoints en Postman: GET

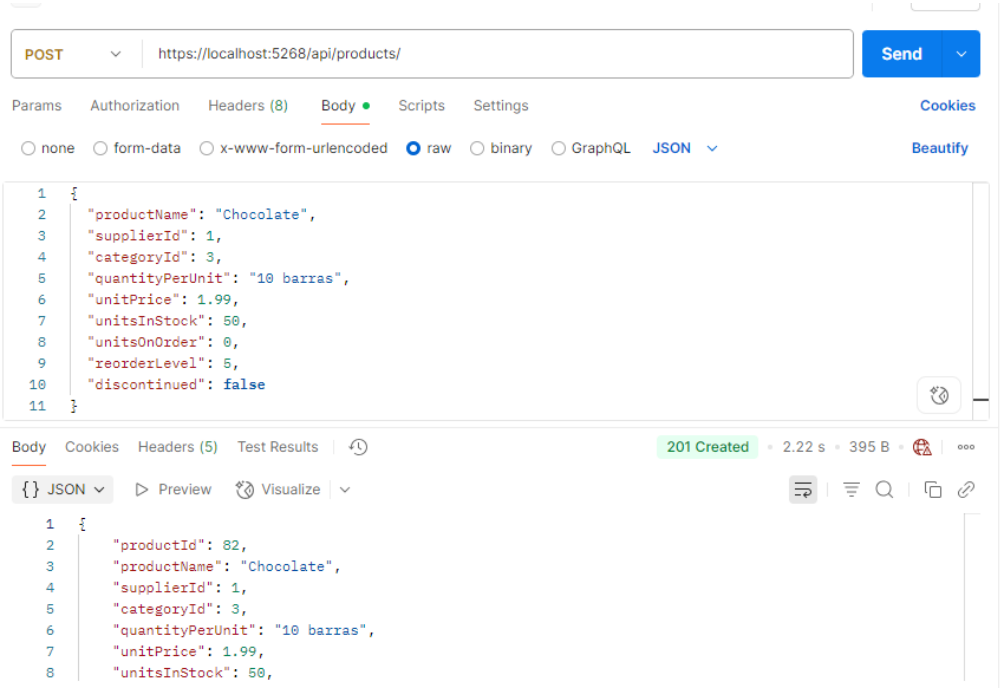


Figura 9: Prueba de endpoints en Postman: POST

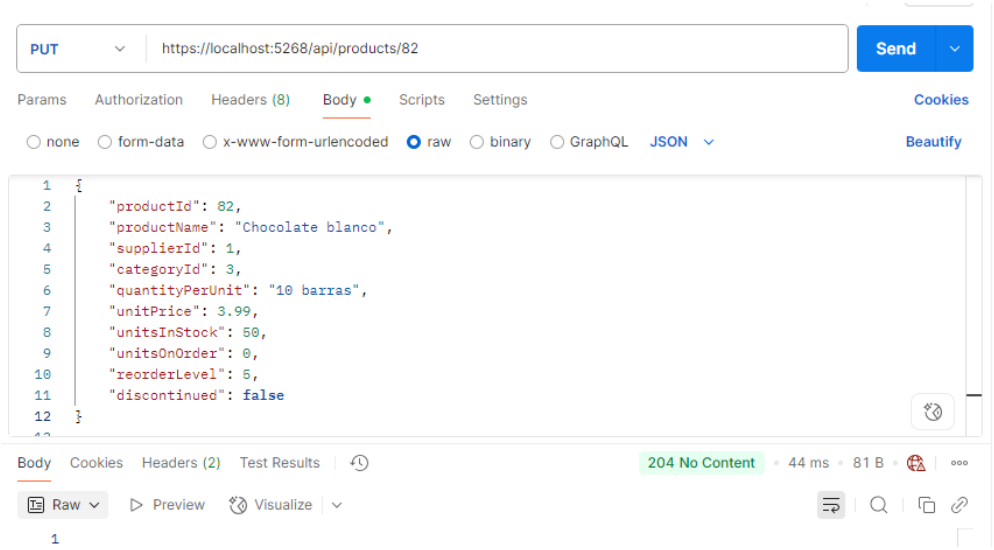


Figura 10: Prueba de endpoints en Postman: PUT

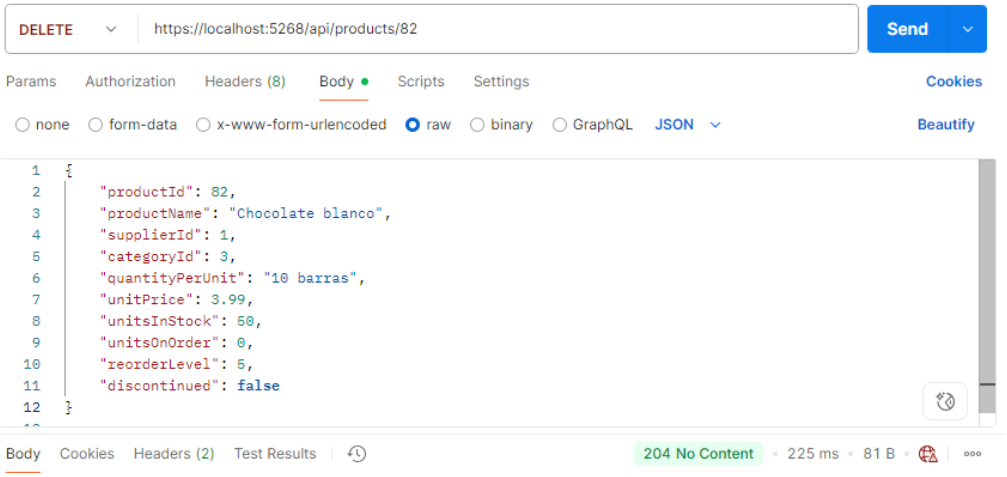


Figura 11: Prueba de endpoints en Postman: DELETE

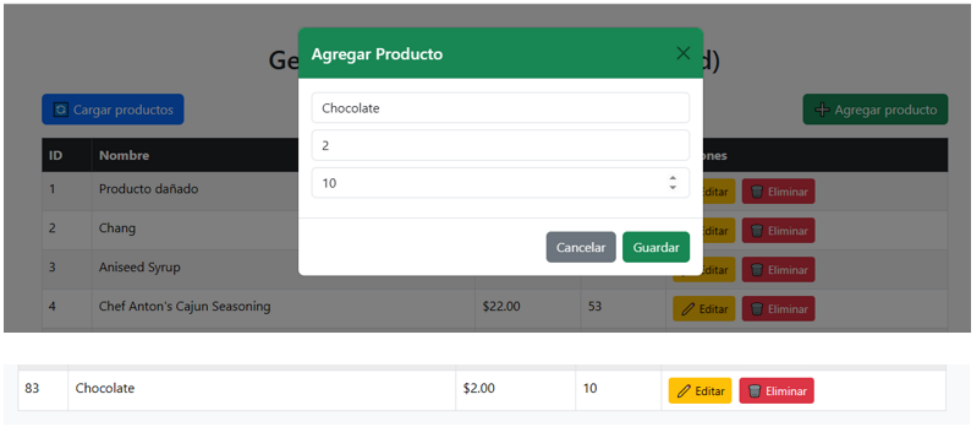


Figura 12: Cliente HTML con CRUD: Agregar

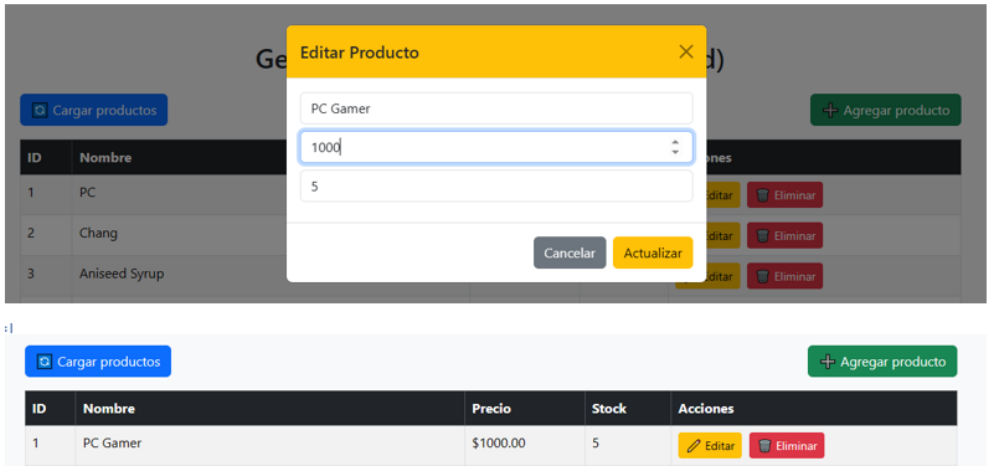


Figura 13: Cliente HTML con CRUD: Editar

Figura 14: Cliente HTML con CRUD: Eliminar

6. Conclusiones

A través del desarrollo de esta API REST, se logró consolidar el conocimiento práctico en tecnologías modernas como .NET y Entity Framework. Se demostró la capacidad de conectar una base de datos existente como Northwind, exponer sus datos mediante una API y facilitar el consumo desde aplicaciones externas como Postman y HTML.

El uso de Swagger y Visual Studio permitió una implementación fluida, mientras que la modularidad de .NET facilitó la organización del código y su mantenimiento.

7. Link al trabajo

`https://drive.google.com/file/d/1gTINfyMj2QyDTySgIy3QVQ_fxVd2GOuJ/view?usp=sharing`

8. Referencias Bibliográficas

Referencias

[1] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. Tesis doctoral que define REST.

[2] Microsoft Docs. Introduction to asp.net core web api, 2023. Documentación oficial de Microsoft.

[3] Bogdan Popa, I. Popescu, Radu-Lucian Constantinescu, Anca Albita, Alexandru Toma, and Mariş Cătălin-Vlădut. Describing and applying the entity framework from simple applications to complex scenarios. case study. *2024 25th International Carpathian Control Conference (ICCC)*, pages 01–06, 2024.

[4] SmartBear Software. Openapi specification (oas) 3.0 documentation, 2023. Especificación de Swagger para documentar APIs REST.

[5] Alexander Nielsen. *Building Web APIs with ASP.NET Core*. Packt Publishing, 2021. Libro práctico para crear APIs seguras y escalables con ASP.NET Core.