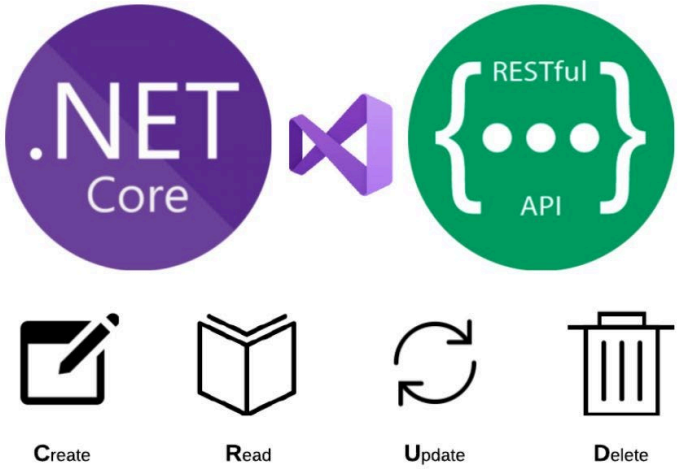
Universidad Nacional de La Matanza

ESCUELA DE FORMACIÓN CONTINUA

Licenciatura en Gestión de Tecnología

Programación avanzada 2

API Rest en Net Core + swagger + microservicios



Integrantes:

Índice

Resúmen

Este documento presenta una investigación sobre la implementación de APIs RESTful utilizando .NET Core, la documentación mediante Swagger bajo el estándar Open API, y el concepto de arquitecturas de microservicios en el contexto de la transformación digital empresarial. Se analiza un caso práctico de un sistema de gestión de librería, demostrando cómo estas tecnologías y patrones arquitectónicos permiten construir aplicaciones escalables, mantenibles y preparadas para los desafíos del entorno digital actual.

En la era de la transformación digital, las organizaciones enfrentan el desafío de construir sistemas de información que sean escalables, flexibles y capaces de evolucionar rápidamente para satisfacer las demandas cambiantes del mercado. Las API REST se han consolidado como el estándar de facto para la comunicación entre sistemas distribuidos, mientras que las arquitecturas de microservicios han emergido como la solución arquitectónica preferida para construir aplicaciones empresariales modernas.

Objetivos de la Investigación

1. **Comprender e implementar una API REST** utilizando .NET Core como plataforma de desarrollo
2. **Diseñar y documentar el contrato de la API** utilizando Swagger bajo el estándar OpenAPI
3. **Analizar el concepto de microservicios** y su relevancia en las arquitecturas escalables
4. **Contextualizar estas tecnologías** en el marco de las transformaciones digitales empresariales

Contexto del Proyecto

Este documento se basa en el proyecto práctico denominado **"Sistema de Gestión de Librería"** (link github), que implementa una API REST para gestionar libros y autores, demostrando los principios fundamentales de diseño RESTful, documentación automática con Swagger, y conceptos aplicables a arquitecturas de microservicios.

Marco Teórico

API REST: Conceptos Fundamentales

REST (Representational State Transfer) es un estilo arquitectónico para sistemas hipermedia distribuidos, propuesto por Roy Fielding en su tesis doctoral del año 2000. Una API REST es una interfaz de programación que se adhiere a los principios REST, permitiendo la comunicación entre sistemas mediante el protocolo HTTP.

Principios Fundamentales de REST

1. **Arquitectura Cliente-Servidor**
    a. Separación clara entre cliente y servidor
    b. El cliente gestiona la interfaz de usuario
    c. El servidor gestiona el almacenamiento y procesamiento de datos
2. **Sin Estado (Stateless)**
    a. Cada petición del cliente contiene toda la información necesaria
    b. El servidor no mantiene contexto de sesión entre peticiones
    c. Mejora la escalabilidad y confiabilidad
3. **Cacheable**
    a. Las respuestas deben definir si son cacheables o no
    b. Mejora la eficiencia y escalabilidad del sistema
4. **Sistema de Capas**
    a. Arquitectura jerárquica con capas intermedias
    b. Permite balanceo de carga, proxies y gateways
5. **Interfaz Uniforme**
    a. Identificación de recursos mediante URIs
    b. Manipulación de recursos a través de representaciones
    c. Mensajes autodescriptivos
    d. HATEOAS (Hypermedia as the Engine of Application State)
6. **Código bajo Demanda (opcional)**
    a. El servidor puede extender funcionalidad del cliente enviando código ejecutable

Métodos HTTP y Operaciones CRUD

Las APIs REST utilizan métodos HTTP estándar para realizar operaciones sobre recursos:

Método HTTP	Operación CRUD	Idempotente	Descripción
GET	Read	Sí	Obtener un recurso o colección
POST	Create	No	Crear un nuevo recurso
PUT	Update	Sí	Actualizar completamente un recurso
PATCH	Update	No	Actualizar parcialmente un recurso

DELETE	Delete	Sí	Eliminar un recurso

Nota: Idempotente significa que una operación puede ejecutarse múltiples veces y el resultado será el mismo que si se ejecutara una sola vez.

Códigos de Estado HTTP

Los códigos de estado HTTP comunican el resultado de la operación:

- ➔ **2xx (Éxito)**: 200 OK, 201 Created, 204 No Content
- ➔ **3xx (Redirección)**: 301 Moved Permanently, 304 Not Modified
- ➔ **4xx (Error del Cliente)**: 400 Bad Request, 401 Unauthorized, 404 Not Found
- ➔ **5xx (Error del Servidor)**: 500 Internal Server Error, 503 Service Unavailable

Diseño de URIs RESTful

Buenas prácticas para el diseño de URIs:

```
✓ GET /api/books # Obtener todos los libros
✓ GET /api/books/{id} # Obtener un libro específico
✓ POST /api/books # Crear un nuevo libro
✓ PUT /api/books/{id} # Actualizar un libro completo
✓ PATCH /api/books/{id} # Actualizar parcialmente un libro
✓ DELETE /api/books/{id} # Eliminar un libro
✓ GET /api/authors/{id}/books # Obtener libros de un autor
```

NET Core como Plataforma de Desarrollo

Evolución de .NET Framework a .NET Core

.NET Framework (2002-2019)

- Plataforma exclusiva de Windows
- Monolítica y acoplada al sistema operativo
- Limitada en escenarios cloud y contenedores

.NET Core (2016-2020) y .NET 5+ (2020-presente)

- Multiplataforma (Windows, Linux, macOS)
- Open Source (licencia MIT)
- Modular y de alto rendimiento
- Optimizado para microservicios y contenedores

- Unificación con .NET 5 en adelante

API en .NET Core

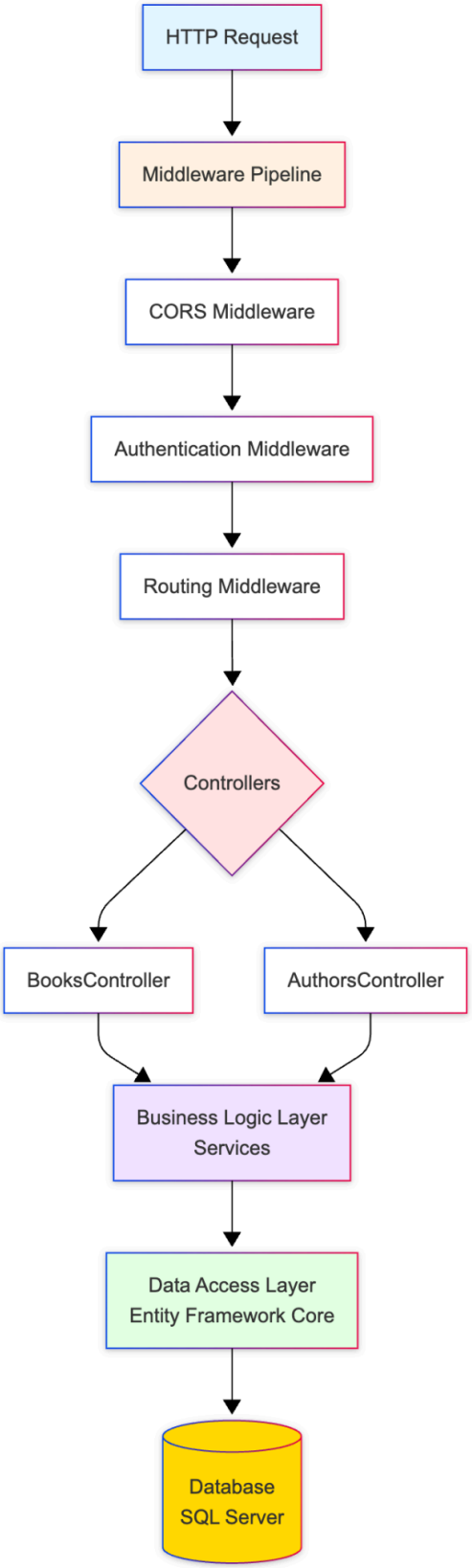
La creación de una API REST en .NET Core generalmente sigue los siguientes pasos:

1. Configuración del Entorno: Instalación del SDK de .NET Core y un IDE como Visual Studio o Visual Studio Code.
2. Creación del Proyecto: Uso de la plantilla webapi de .NET Core.
3. Definición de Modelos: Creación de clases que representen los datos que se expondrán a través de la API.
4. Desarrollo de Controladores: Clases que manejan las solicitudes HTTP y definen los endpoints de la API.

Ventajas de .NET Core para APIs

- ➔ **Alto Rendimiento**
  - Uno de los frameworks más rápidos según TechEmpower Benchmarks
  - Optimización para escenarios de alto tráfico
- ➔ **Desarrollo Multiplataforma**
  - Desarrollo en Windows, Linux o macOS
  - Despliegue flexible en cualquier entorno
- ➔ **Ecosistema Robusto**
  - Entity Framework Core para acceso a datos
  - Middleware extensible y personalizable
  - Inyección de dependencias nativa
- ➔ **Preparado para la Nube**
  - Integración nativa con Azure
  - Compatible con AWS, Google Cloud y otros proveedores
  - Soporte completo para contenedores Docker
- ➔ **Mantenimiento y Soporte**
  - Respaldado por Microsoft
  - Comunidad activa y creciente
  - Actualizaciones regulares y LTS (Long Term Support)

Arquitectura de una API en .NET Core



Ejemplo de Controlador Básico

```
using Microsoft.AspNetCore.Mvc;

namespace MyApi.Controllers
{
    [ApiController]
    [Route("[controller]")]
    public class ProductsController : ControllerBase
    {
        [HttpGet]
        public IEnumerable<string> Get()
        {
            return new string[] { "Product 1", "Product 2", "Product 3" };
        }

        [HttpPost]
        public IActionResult Post([FromBody] string value)
        {
            // Lógica para guardar el nuevo producto
            return Ok($"Producto '{value}' creado.");
        }
    }
}
```

Swagger y OpenAPI Specification

OpenAPI Specification (OAS)

OpenAPI Specification, anteriormente conocida como Swagger Specification, es un estándar independiente del lenguaje para describir APIs RESTful. Permite tanto a humanos como a computadoras descubrir y entender las capacidades de un servicio sin necesidad de acceder al código fuente.

Componentes de OpenAPI

1. Información General

```
penapi: 3.0.0
info:
  title: Library Management API
  version: 1.0.0
  description: API para gestión de libros y autores
```

2. Definición de Endpoints

```
paths:
  /api/books:
    get:
      summary: Obtener todos los libros
      responses:
        '200':
          description: Lista de libros obtenida exitosamente
```

3. Esquemas de Datos

```
components:
  schemas:
    Book:
      type: object
      properties:
        id:
          type: integer
        title:
          type: string
        isbn:
          type: string
```

4. Seguridad

```
components:
  securitySchemes:
    bearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
```

Ventajas de Usar Swagger

1. **Documentación Automática**
    a. Se genera a partir del código
    b. Siempre sincronizada con la implementación
2. **Testing Interactivo**
    a. Interfaz para probar endpoints sin herramientas adicionales
    b. Facilita el desarrollo y debugging
3. **Contract-First Development**
    a. Permite diseñar la API antes de implementarla
    b. Facilita la comunicación entre equipos
4. **Generación de Clientes**

    a. Creación automática de SDKs para diferentes lenguajes
    b. Reduce tiempo de integración
5. **Estandarización**
    a. Formato universal para describir APIs
    b. Facilita integración con herramientas de terceros

Swagger Tools Ecosystem

1. **Swagger Editor**
    a. Editor web para escribir especificaciones OpenAPI
    b. Validación en tiempo real
2. **Swagger UI**
    a. Interfaz web interactiva generada automáticamente
    b. Permite probar endpoints directamente
    c. Documentación visual y fácil de entender
3. **Swagger Codegen**
    a. Generación automática de código cliente y servidor
    b. Soporte para múltiples lenguajes

Integración de Swagger en .NET Core

Para integrar Swagger en un proyecto .NET Core, se suele utilizar el paquete NuGet Swashbuckle.AspNetCore.

1. Instalación: Install-Package Swashbuckle.AspNetCore
2. Configuración en Startup.cs:

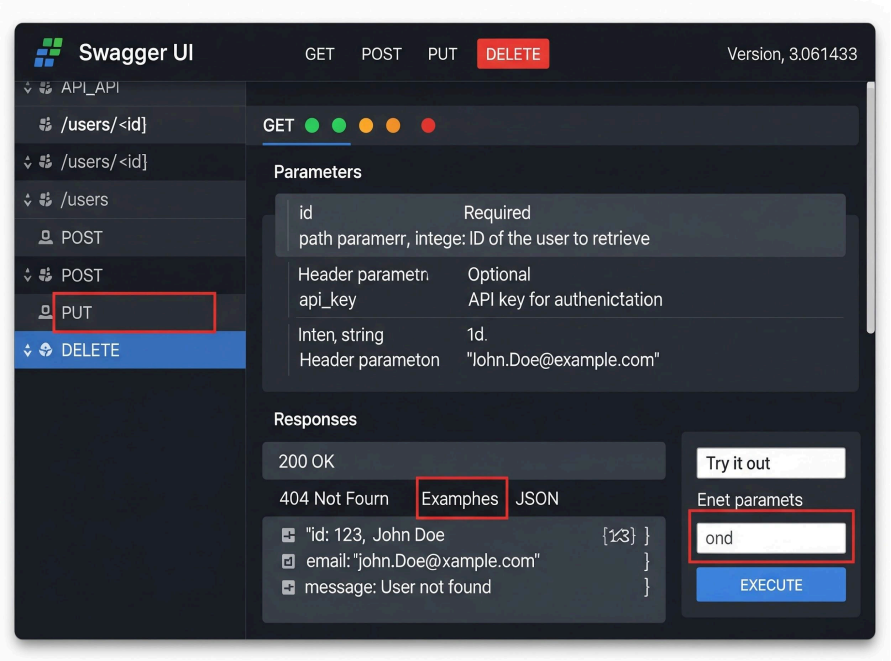
```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo { Title = "My API", Version = "v1"
});
    });
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseSwagger();
        app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json", "My
API v1"));
    }

    app.UseHttpsRedirection();
```

```
    app.UseRouting();
    app.UseAuthorization();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

Una vez configurado, la documentación de Swagger estará accesible en /swagger (por ejemplo, https://localhost:5001/swagger).



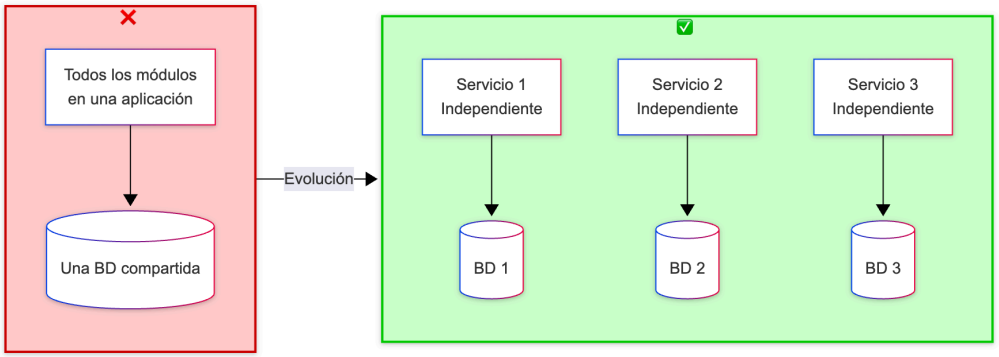
Arquitectura de Microservicios

Definición y Conceptos Básicos

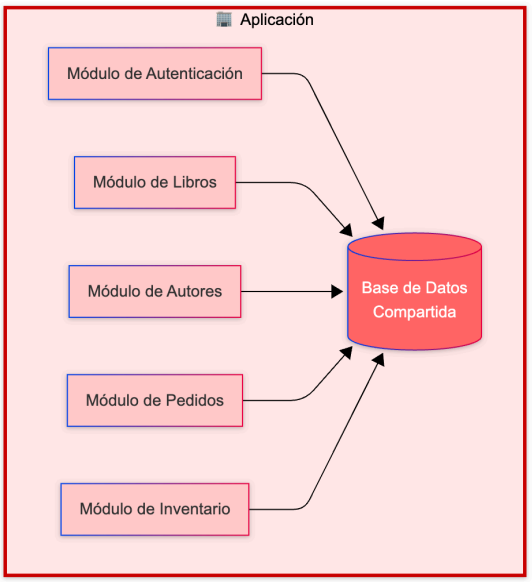
Los microservicios son un estilo arquitectónico que estructura una aplicación como una colección de servicios que son:

- ➔ Altamente mantenibles y testeables
- ➔ Débilmente acoplados
- ➔ Desplegables independientemente
- ➔ Organizados alrededor de capacidades de negocio
- ➔ Propiedad de equipos pequeños

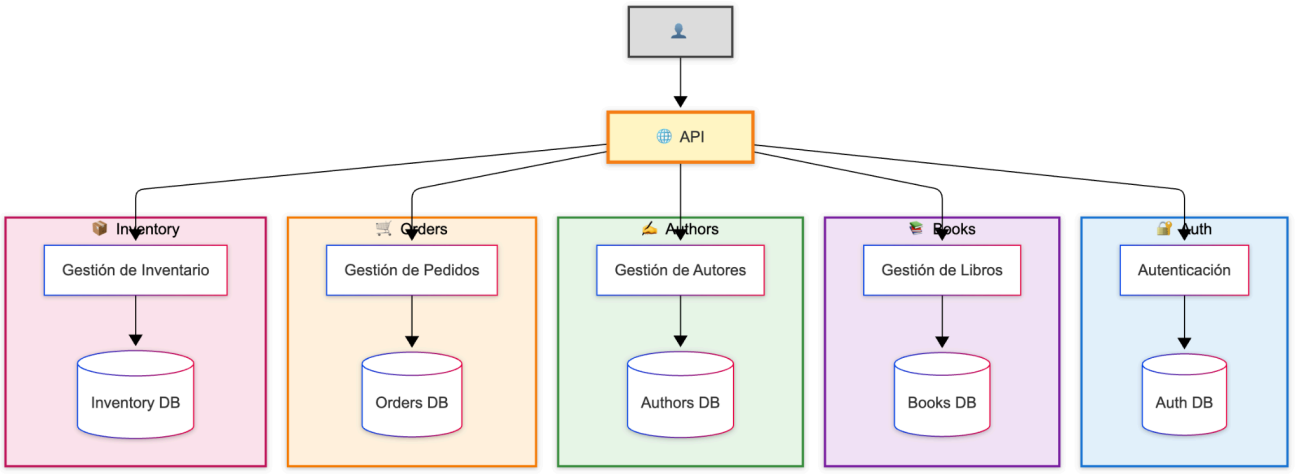
Arquitectura Monolítica vs Microservicios



Arquitectura Monolítica:



Arquitectura de Microservicios:



Microservicios y Escalabilidad

Los microservicios son un enfoque arquitectónico donde una aplicación se construye como una colección de servicios pequeños, autónomos y acoplados de forma débil. Cada servicio se enfoca en una capacidad de negocio específica y puede ser desarrollado, desplegado y escalado independientemente.

Ventajas de los Microservicios

1. **Escalabilidad Independiente**
    a. Cada servicio puede escalar según sus necesidades
    b. Optimización de recursos y costos
2. **Flexibilidad Tecnológica**
    a. Diferentes servicios pueden usar diferentes tecnologías
    b. Elección de la mejor herramienta para cada problema
3. **Resiliencia**
    a. Fallos aislados no afectan todo el sistema
    b. Implementación de circuit breakers y fallbacks
4. **Desarrollo Ágil**
    a. Equipos independientes trabajando en paralelo
    b. Ciclos de desarrollo y despliegue más rápidos
5. **Facilidad de Mantenimiento**
    a. Código más pequeño y enfocado
    b. Más fácil de entender y modificar

Desafíos de los Microservicios

1. **Complejidad Operacional**
    a. Múltiples servicios para desplegar y monitorear
    b. Necesidad de herramientas de orquestación (Kubernetes)
2. **Comunicación entre Servicios**
    a. Latencia de red
    b. Manejo de fallos en comunicaciones
3. **Gestión de Datos**
    a. Transacciones distribuidas
    b. Consistencia eventual vs consistencia fuerte
4. **Testing**
    a. Pruebas de integración más complejas
    b. Necesidad de entornos de testing robustos
5. **Seguridad**
    a. Mayor superficie de ataque
    b. Necesidad de seguridad en cada servicio

Patrones Comunes en Microservicios

1. **API Gateway**
    a. Punto de entrada único para todas las peticiones
    b. Enrutamiento, autenticación, rate limiting
2. **Service Discovery**
    a. Registro y descubrimiento dinámico de servicios
    b. Ejemplos: Consul, Eureka, etcd
3. **Circuit Breaker**
    a. Prevención de fallos en cascada
    b. Implementación: Polly, Hystrix
4. **Event-Driven Architecture**
    a. Comunicación asíncrona mediante eventos
    b. Message brokers: RabbitMQ, Apache Kafka
5. **Database per Service**
    a. Cada servicio tiene su propia base de datos
    b. Independencia y encapsulamiento de datos
6. **CQRS (Command Query Responsibility Segregation)**
    a. Separación de operaciones de lectura y escritura
    b. Optimización independiente
7. **Saga Pattern**
    a. Manejo de transacciones distribuidas
    b. Orquestación o coreografía de operaciones

Arquitecturas Escalables en la Transformación Digital Empresarial

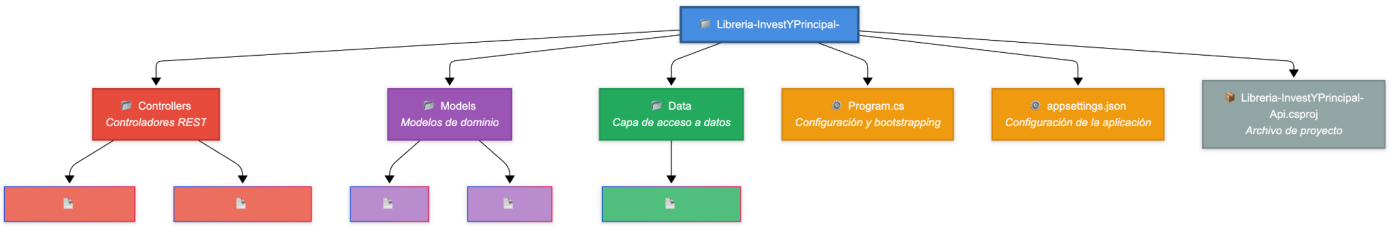
La transformación digital empresarial impulsa la necesidad de arquitecturas que puedan adaptarse rápidamente a la evolución del mercado y manejar volúmenes crecientes de datos y usuarios. Los microservicios, combinados con API REST y herramientas como Swagger, son fundamentales para lograr esta escalabilidad.

Implementación Práctica: Sistema de Gestión de Librería

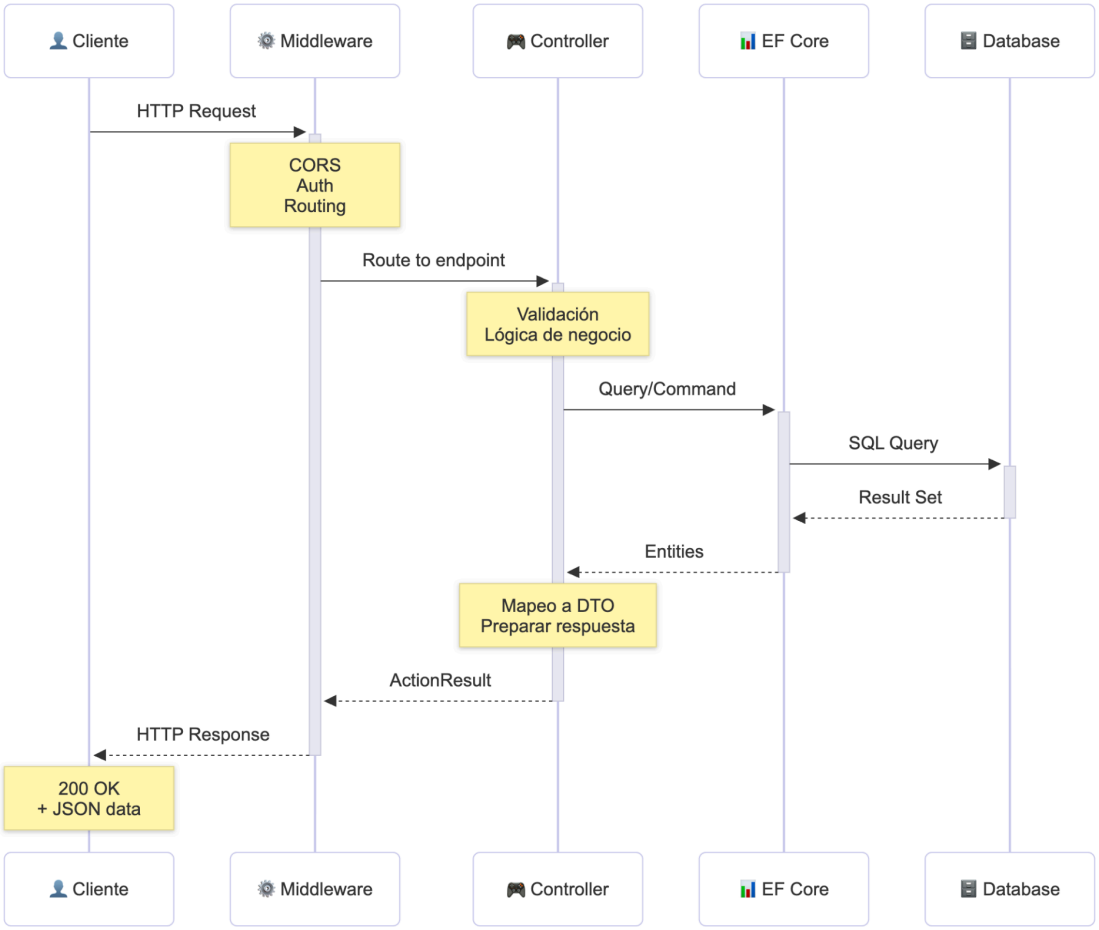
Arquitectura del Proyecto

El proyecto **"Sistema de Gestión de Librería"** implementa una API REST para gestionar libros y autores, siguiendo los principios RESTful y las mejores prácticas de .NET Core.4.1.1.

Estructura del Proyecto



Flujo de Datos



Configuración de la API REST

Archivo Program.cs

El archivo `Program.cs` es el punto de entrada de la aplicación y contiene toda la configuración necesaria:

```
using Microsoft.EntityFrameworkCore;
var builder = WebApplication.CreateBuilder(args);
```

```
// ==========================================
// Configuración de Servicios (Dependency Injection Container)
// ==========================================
// 1. Controladores
builder.Services.AddControllers();
// 2. Explorador de endpoints para Swagger
builder.Services.AddEndpointsApiExplorer();
// 3. Generador de Swagger/OpenAPI
builder.Services.AddSwaggerGen();
// 4. Contexto de base de datos con Entity Framework Core
builder.Services.AddDbContext
```

Conclusiones

La implementación de API REST con .NET Core, el diseño de contratos robustos con Swagger/OpenAPI y la adopción de una arquitectura de microservicios son pilares fundamentales para las empresas que buscan innovar y escalar en la era digital. Aunque presenta desafíos, los beneficios en términos de agilidad, flexibilidad y resiliencia superan con creces las complejidades, permitiendo a las organizaciones construir sistemas capaces de soportar las demandas de un mercado en constante cambio. La transformación digital exige arquitecturas que no solo respondan al presente, sino que también estén preparadas para el futuro, y la combinación de estas tecnologías ofrece precisamente esa capacidad.

Referencias Bibliográficas

- Richardson, L., & Amundsen, M. (2013). *RESTful Web APIs*. O'Reilly Media.
- Microsoft Docs. (n.d.). *Overview of ASP.NET Core*.
- OpenAPI Initiative. (n.d.). *OpenAPI Specification*.
- Newman, S. (2015). *Building Microservices*. O'Reilly Media.
- Fowler, M. (2014). *Microservices*.
- Código fuente: https://github.com/Lenh22/Libreria-InvestYPrincipal-Api