

## Programación Web 3

UNLaM - Tecnicatura en Desarrollo Web

Trabajo Práctico de Investigación

Título: *API Rest en Net Core + swagger +  
microservicios + React*

---

|                                |    |
|--------------------------------|----|
| Integrantes                    | 3  |
| Objetivo                       | 3  |
| Situación Actual               | 4  |
| Desarrollo de la Investigación | 5  |
| Conclusiones                   | 15 |
| Referencias/Bibliografía       | 17 |

## Integrantes

- Nava Agustin
- Soengas Sofia
- Marin Damian
- Aquino Romero Jannett Arami
- Foppoli Jennifer

## Objetivo

En primer lugar, se investigará cómo crear una API REST en .NET Core, abarcando definiciones básicas para entender su funcionamiento y aspectos como la configuración del entorno de desarrollo, la creación de controladores, la definición de rutas y el manejo de solicitudes y respuestas HTTP.

Luego, se explorará el uso de Swagger para diseñar y documentar el contrato de la API REST bajo el estándar OpenAPI. Se investigará cómo utilizar esta herramienta para definir los endpoints, los parámetros de entrada y salida, los esquemas de datos y la información adicional necesaria para que el consumidor de la API pueda entender y utilizar correctamente los servicios ofrecidos.

También se profundizará en el concepto de microservicios y su aplicación en arquitecturas escalables. Se analizarán las ventajas y desafíos de utilizar microservicios en el desarrollo de aplicaciones empresariales, así como las mejores prácticas para diseñar, implementar y gestionar estos servicios de forma eficiente y escalable.

Por último, se estudiará cómo realizar peticiones a la API REST implementada desde una aplicación cliente utilizando la tecnología React. Se explorará cómo consumir los endpoints de la API anteriormente creados, enviar y recibir datos, y manejar las respuestas del servidor en la interfaz de usuario de la aplicación React.

En resumen, este trabajo práctico de investigación tiene como objetivo principal explorar en detalle la implementación de una API REST en .NET Core, utilizando Swagger para diseñar su contrato bajo el estándar Open API, y utilizando una aplicación cliente desarrollada en React para analizar cómo interactuar con la API.

## Situación Actual

En la actualidad, el desarrollo de software a través de la tecnología .NET Core ha ganado popularidad debido a su flexibilidad, eficiencia y compatibilidad con múltiples plataformas.

El enfoque de construir APIs REST se ha vuelto muy común para desarrollar aplicaciones web y servicios backend. Esto se debe a que las API REST permiten la comunicación entre diferentes sistemas de forma sencilla y estandarizada.

Una herramienta muy utilizada en el desarrollo de APIs REST en .NET Core es Swagger. Swagger proporciona un conjunto de herramientas que facilitan la generación de documentación interactiva y la creación de contratos de API bien definidos. Esto permite una mejor comunicación entre equipos de desarrollo y clientes de API, además de facilitar la integración y el consumo de servicios por parte de terceros.

Otra herramienta que se utilizará para este trabajo es React, una tecnología muy utilizada en el desarrollo de interfaces de usuario modernas y que es compatible con el desarrollo de aplicaciones que consumen una API REST implementada en .NET Core. Proporciona una forma eficiente y escalable de interactuar con la API y manejar los datos recibidos, lo que lo convierte en una opción popular para el desarrollo de la parte del cliente en aplicaciones web que utilizan esta combinación de tecnologías.

En cuanto al concepto de microservicios, se ha convertido en una arquitectura de referencia para el diseño y desarrollo de aplicaciones empresariales escalables. Los microservicios descomponen una aplicación en componentes más pequeños y autónomos, lo que permite una mayor modularidad, despliegue independiente y facilita el mantenimiento y escalabilidad de la aplicación. Estos componentes se comunican entre sí a través de APIs REST, lo que los hace ideales para trabajar con tecnologías como .NET Core y Swagger.

Para concluir, en el contexto actual del desarrollo de software utilizando tecnologías .NET Core, es importante comprender y dominar la implementación de APIs REST, el diseño de contratos utilizando Swagger bajo el estándar Open API, así como también conocer y aplicar el concepto de microservicios para lograr arquitecturas escalables en el contexto de las transformaciones digitales empresariales.

# Desarrollo de la Investigación

## API Rest en Net Core

### API

*“API (Application Programming Interface) significa interfaz de programación de aplicaciones. En el contexto de las API, la palabra aplicación se refiere a cualquier software con una función distinta. Por otro lado, la interfaz puede considerarse como un contrato de servicio entre dos aplicaciones. Este contrato define cómo se comunican entre sí mediante solicitudes y respuestas.”*

Para empezar a hablar de API REST, debemos entender que es una API. Las API son mecanismos que permiten a dos componentes de software comunicarse entre sí mediante un conjunto de definiciones y protocolos. Es decir, funcionan como un traductor que les permite a dichos componentes entenderse y trabajar juntos. Por ejemplo, el sistema de software de un instituto de meteorología contiene datos meteorológicos diarios. La aplicación meteorológica de su teléfono “habla” con este sistema a través de las API y le muestra las actualizaciones meteorológicas diarias en su teléfono.

### ¿Para qué sirve una API?

Las principales funciones de una API son:

- **Simplificar el trabajo de los desarrolladores** para integrar elementos de las aplicaciones nuevas en arquitecturas ya existentes.
- **Reducir (e incluso eliminar) el trabajo repetitivo**, ya que permite a los desarrolladores usar funciones ya existentes y no tener que crear herramientas desde cero.
- **Favorecer la colaboración** entre los equipos comerciales y los departamentos tecnológicos.
- **Simplificar las tareas** de diseño, gestión y uso de aplicaciones.
- **Propiciar la comunicación** entre productos y servicios.
- **Hacer que sea más sencillo el desarrollo de aplicaciones**, consiguiendo, de este modo, un considerable ahorro de tiempo y dinero.

### Funcionamiento

La arquitectura de las API suele explicarse en términos de cliente y servidor. La aplicación que envía la solicitud se llama cliente, y la que envía la respuesta se llama servidor.

Las API definen cómo deben ser los mensajes que se intercambian, qué información se necesita enviar y recibir, y cómo deben interpretarse estos datos. De esta manera, los diferentes componentes de software pueden colaborar de forma eficiente, utilizando las

funcionalidades que cada uno ofrece, sin necesidad de conocer todos los detalles internos del otro. De esta manera conecta diferentes partes de un software para garantizar el correcto destino de la información. En tal sentido, es un intermediario virtual: envía información de una interfaz a otra. Pueden funcionar de cuatro maneras diferentes, según el momento y el motivo de su creación

- **API de SOAP (Simple Object Access Protocol)**

Se trata de una API menos flexible que era más popular en el pasado.

Es un protocolo con más complejidad en términos de seguridad y comunicación de datos. Dado que este protocolo se ocupa de los mensajes, el objetivo principal es evitar el acceso no autorizado a través de la seguridad de Web Services.

Como es un protocolo, impone reglas integradas que aumentan la complejidad y la sobrecarga, lo cual puede retrasar el tiempo que tardan las páginas en cargarse. Sin embargo, estos estándares también ofrecen normas integradas que pueden ser ideales para el sector empresarial. Los estándares de cumplimiento integrados incluyen la seguridad, la atomicidad, la uniformidad, el aislamiento y la durabilidad (ACID), que forman un conjunto de propiedades que garantizan operaciones confiables de las bases de datos. Por último, el cliente y el servidor intercambian mensajes mediante XML.

- **API de RPC (Remote Procedure Call)**

Se denominan llamadas a procedimientos remotos.

El protocolo RPC gestiona la comunicación entre procesos de manera fiable y requiere un tiempo de procesamiento relativamente corto. Así, se facilita mucho la programación de procesos de comunicación entre ordenadores remotos, porque, por ejemplo, no es necesario tener en cuenta las características más complejas de la red que se utiliza. RPC permite además una modularización coherente: los procesos pueden distribuirse, aligerando así la carga de los ordenadores. Las redes y los sistemas distribuidos funcionan de forma más eficiente gracias al reparto del trabajo mediante el uso de plataformas especializadas para tareas concretas (por ejemplo, servidores de bases de datos), y todas las partes implicadas pueden conectarse a la red desde cualquier lugar del mundo. Otras ventajas son la excelente escalabilidad de las arquitecturas cliente-servidor implementadas, así como la posibilidad de trabajar en la nube fácilmente.

Por otro lado, una de las desventajas de RPC es que no existe un estándar unificado para esta tecnología. Las diferentes implementaciones, la mayoría específicas de cada empresa (por ejemplo, ONC-RPC de Sun), no suelen ser compatibles entre sí. Además, los niveles de transferencia de los sistemas basados en RPC conllevan una cierta pérdida de velocidad, al contrario que las llamadas a procedimiento puramente locales. Como el cliente y el servidor utilizan diferentes entornos de ejecución para sus respectivas rutinas, el uso de recursos (por ejemplo, archivos) también se vuelve más complejo. Por lo tanto, el protocolo RPC no resulta muy adecuado para transferir grandes cantidades de datos.

- **API de REST**

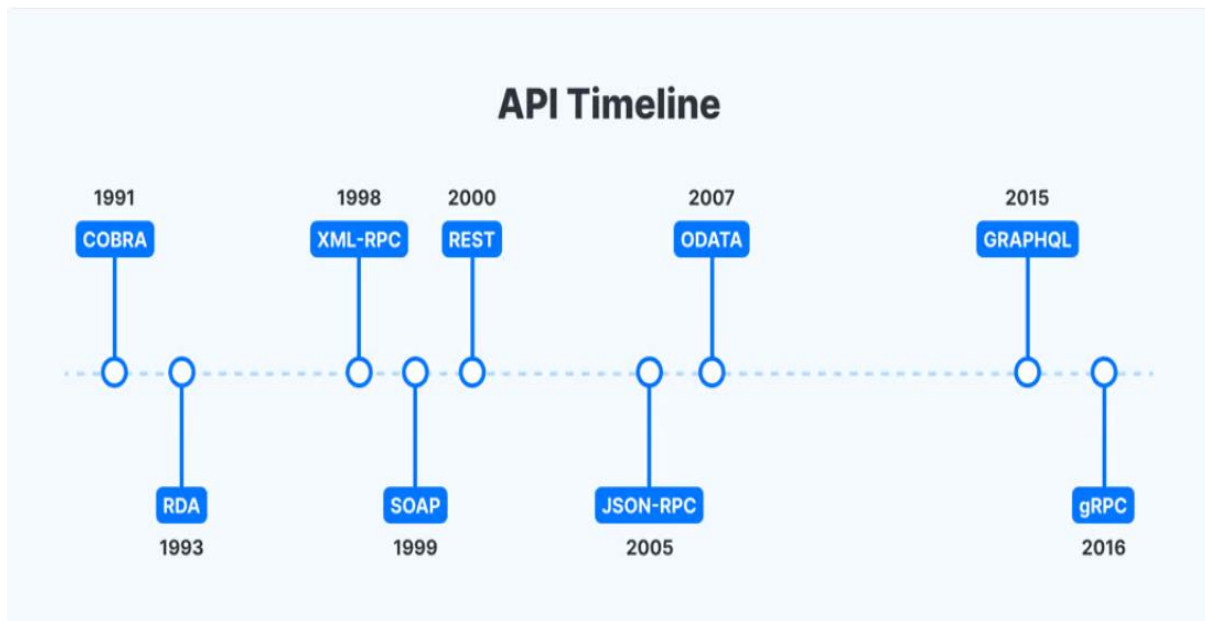
Estas son las más populares y flexibles que se encuentran en la web actualmente. REST es un conjunto de principios arquitectónicos que se ajusta a las necesidades de las aplicaciones móviles y los servicios web ligeros. Dado que se trata de un conjunto de pautas, la implementación de las recomendaciones depende de los desarrolladores. De esta profundizaremos más adelante.

- **API de WebSocket**

WebSocket es un protocolo de red basado en TCP que establece cómo deben intercambiarse datos entre redes. Puesto que es un protocolo fiable y eficiente, es utilizado por prácticamente todos los clientes. El protocolo TCP establece conexiones entre dos puntos finales de comunicación, llamados sockets. De esta manera, el intercambio de datos puede producirse en las dos direcciones.

En las conexiones bidireccionales, como las que crea WebSocket (a veces también websocket o web socket), se intercambian datos en ambas direcciones al mismo tiempo. La ventaja de este intercambio es que se accede de forma más rápida a los datos. En concreto, WebSocket permite así una comunicación directa entre una aplicación web y un servidor WebSocket. En otras palabras: la web que se solicita se muestra en tiempo real.

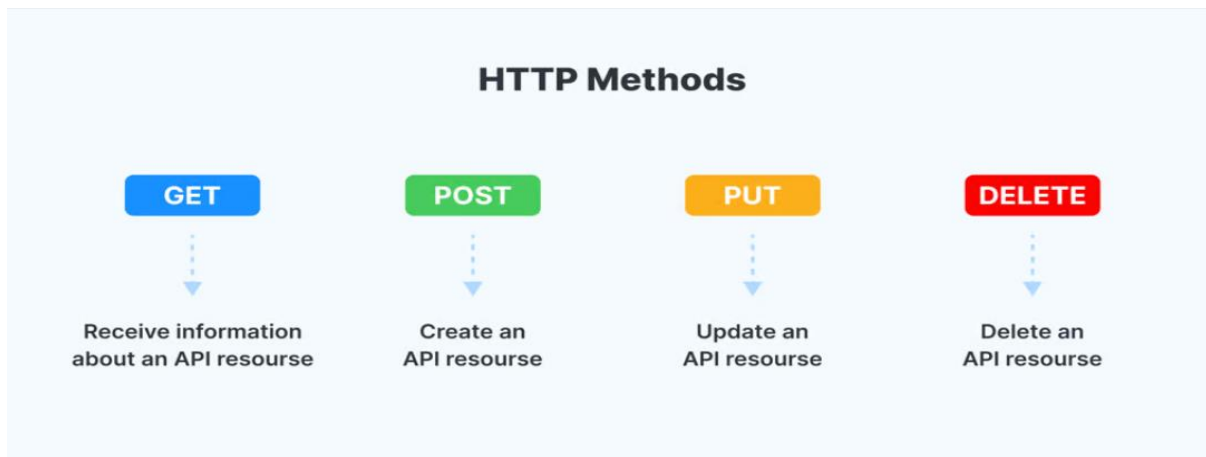
Se envían al servidor mensajes de cadenas simples con datos, y el servidor procesa los datos y las respuestas. La comunicación es más eficiente que HTTP si nos centramos en el tamaño del mensaje, y en la velocidad, especialmente para mensajes de gran tamaño, ya que en HTTP, por ejemplo, tienes que enviar los headers en cada petición. Esto suma bytes.



## API REST

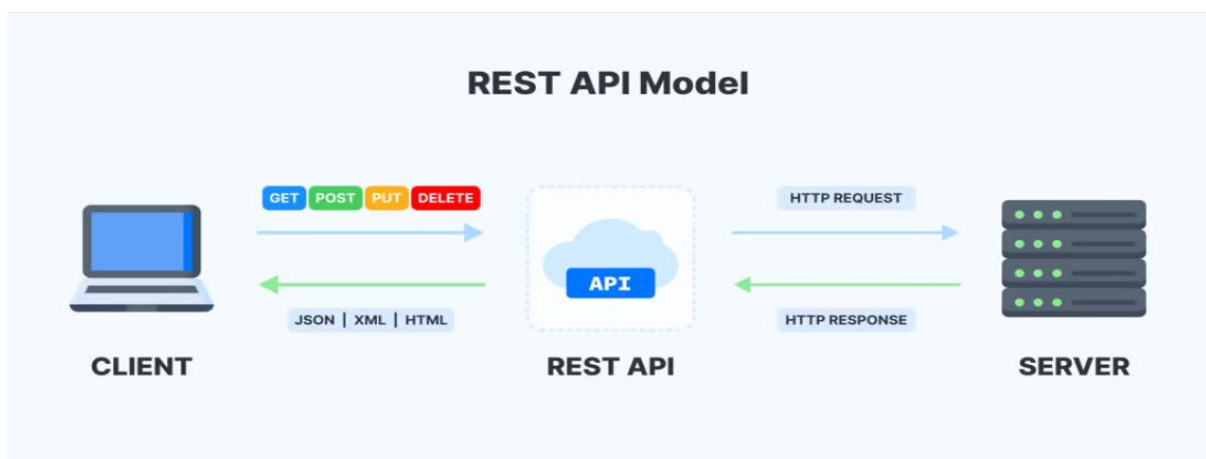
*“REST (Representational state transfer) el cual significa transferencia de estado representacional. REST no es un protocolo ni un estándar, sino más bien un conjunto de límites de arquitectura. “*

Los desarrolladores de las API pueden implementarlo de distintas maneras. Define un conjunto de funciones como GET, PUT, DELETE, etc. que los clientes pueden utilizar para acceder a los datos del servidor. Estos datos son intercambiados mediante HTTP.



Una API REST es una interfaz de comunicación entre sistemas de información que usa el protocolo de transferencia de hipertexto (*hypertext transfer protocol* o HTTP, por sus siglas en inglés) para obtener datos o ejecutar operaciones sobre dichos datos en diversos formatos, como pueden ser XML o JSON.

La principal característica de la API REST es que no tiene estado. La ausencia de estado significa que los servidores no guardan los datos del cliente entre las solicitudes. Las solicitudes de los clientes al servidor son similares a las URL que se escriben en el navegador para visitar un sitio web. La respuesta del servidor son datos simples, sin la típica representación gráfica de una página web.





## Componentes principales de una API REST:

Se basa en el modelo cliente-servidor donde el cliente es el que solicita obtener los recursos o realizar alguna operación sobre dichos datos, mientras que el servidor es aquel ente que entrega o procesa dichos datos a solicitud del cliente.

- Cliente - un cliente o programa lanzado en el lado del usuario (en su dispositivo) iniciando la comunicación.
- Servidor - un servidor que utiliza API como acceso a sus funciones y datos.
- Recurso - cualquier contenido (video, texto, etc.) que el servidor transmite al cliente. Es un concepto crítico en REST API, una abstracción de información. Puede ser cualquier información: documento, imagen, servicio temporal. El estado del recurso en un momento dado se conoce como representación del recurso, que consta de datos, metadatos que describen los datos y enlaces hipermedia para ayudar a los clientes a pasar al siguiente estado.

## Funcionamiento de una API REST

Como ya se mencionó antes, La API REST se comunica a través de solicitudes HTTP y completa las siguientes funciones: crear, leer, actualizar y eliminar datos. También conocidas como operaciones CRUD. REST proporciona la información sobre los recursos solicitados y utiliza métodos para describir qué hacer con un recurso.

La información puede ser entregada al cliente en varios formatos: JSON, HTML, XLT, Python o texto plano. El más popular y usado es JSON porque es legible por humanos y máquinas, y es independiente del lenguaje.

Para acceder a un recurso, un cliente debe realizar una solicitud. Después de recibirlo, el servidor generará una respuesta con datos codificados sobre un recurso.

## La estructura de la solicitud

Incluye cuatro componentes principales: el método HTTP (CRUD que mencionamos anteriormente), puntos finales, encabezados y cuerpo.

- El **método HTTP** describe lo que se debe hacer con el recurso.
  - POST : creación de un recurso.
  - GET : obtener un recurso.
  - PUT : actualizar un recurso.
  - DELETE : eliminación de un recurso.
- El **punto final (Endpoint)** contiene un URI: identificador uniforme de recursos, que indica cómo y dónde se puede encontrar el recurso. Una URL o ubicación uniforme de recursos es el tipo de URI más común y representa una dirección web completa.
- Los **encabezados** contienen los datos relacionados con el cliente y el servidor. Los encabezados incluyen datos de autenticación: clave API, nombre, dirección IP que

pertenece a la computadora en la que está instalado el servidor y también la información sobre el formato de respuesta.

- El **cuerpo** se usa para enviar información adicional al servidor, como los datos que desea agregar.

## Ventajas de una API REST

### 1. Integración

Las API se utilizan para integrar nuevas aplicaciones con los sistemas de software existentes. Esto aumenta la velocidad de desarrollo, ya que no hay que escribir cada funcionalidad desde cero. Puede utilizar las API para aprovechar el código existente.

### 2. Innovación

Sectores enteros pueden cambiar con la llegada de una nueva aplicación. Las empresas deben responder con rapidez y respaldar la rápida implementación de servicios innovadores. Para ello, pueden hacer cambios en la API sin tener que reescribir todo el código.

### 3. Ampliación

Las API presentan una oportunidad única para que las empresas satisfagan las necesidades de sus clientes en diferentes plataformas. Por ejemplo, la API de mapas permite la integración de información de los mapas en sitios web, Android, iOS, etc. Cualquier empresa puede dar un acceso similar a sus bases de datos internas mediante el uso de API gratuitas o de pago.

### 4. Facilidad de mantenimiento

La API actúa como una puerta de enlace entre dos sistemas. Cada sistema está obligado a hacer cambios internos para que la API no se vea afectada. De este modo, cualquier cambio futuro que haga una de las partes en el código no afectará a la otra.

## Puntos de conexión

Los puntos de conexión de las API son los últimos puntos de contacto del sistema de comunicación de las API. Se trata de las URL de servidores, servicios y otras ubicaciones digitales específicas desde las que se envía y recibe información entre sistemas. Los puntos de conexión de las API son fundamentales para las empresas por dos motivos principales:

### 1. Seguridad

Los puntos de conexión de las API hacen que el sistema sea vulnerable a los ataques. La supervisión de las API es crucial para evitar su uso indebido.

### 2. Rendimiento

Los puntos de conexión de las API, especialmente los de alto tráfico, pueden provocar cuellos de botella y afectar al rendimiento del sistema.

## Protección de una API de REST

Todas las API deben protegerse mediante una autenticación y una supervisión adecuadas. Las dos maneras principales de proteger las API de REST son las siguientes:

### **1. Tokens de autenticación**

Se utilizan para autorizar a los usuarios a hacer la llamada a la API. Los tokens de autenticación comprueban que los usuarios son quienes dicen ser y que tienen los derechos de acceso para esa llamada concreta a la API. Por ejemplo, cuando inicia sesión en el servidor de correo electrónico, el cliente de correo electrónico utiliza tokens de autenticación para un acceso seguro.

### **2. Claves de API**

Las claves de API verifican el programa o la aplicación que hace la llamada a la API. Identifican la aplicación y se aseguran de que tiene los derechos de acceso necesarios para hacer la llamada a la API en cuestión. Las claves de API no son tan seguras como los tokens, pero permiten supervisar la API para recopilar datos sobre su uso. Es posible que haya notado una larga cadena de caracteres y números en la URL de su navegador cuando visita diferentes sitios web. Esta cadena es una clave de la API que el sitio web utiliza para hacer llamadas internas a la API.

## Creación de una API REST en NET CORE

ASP.NET Core admite dos enfoques para crear API: un enfoque basado en controlador y API mínimas. Los controladores en un proyecto API son clases que se derivan de ControllerBase. Las API mínimas definen puntos finales con controladores lógicos en lambdas o métodos.

### **API basada en controlador vs API mínima**

El diseño de las API mínimas oculta la clase de host de forma predeterminada y se centra en la configuración y la extensibilidad a través de métodos de extensión que toman funciones como expresiones lambda. Los controladores son clases que pueden tomar dependencias a través de la inyección de constructores o la inyección de propiedades y, en general, siguen patrones orientados a objetos. Las API mínimas admiten la inyección de dependencia a través de otros enfoques, como acceder al proveedor de servicios.

Aquí hay un código de muestra para una API basada en controladores:

```
namespace APIWithControllers;

public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        builder.Services.AddControllers();
        var app = builder.Build();

        app.UseHttpsRedirection();

        app.MapControllers();

        app.Run();
    }
}
```

```
using Microsoft.AspNetCore.Mvc;

namespace APIWithControllers.Controllers;
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
{
    private static readonly string[] Summaries = new[]
    {
        "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot", "Sweltering", "Scorching"
    };

    private readonly ILogger<WeatherForecastController> _logger;

    public WeatherForecastController(ILogger<WeatherForecastController> logger)
    {
        _logger = logger;
    }

    [HttpGet(Name = "GetWeatherForecast")]
    public IEnumerable<WeatherForecast> Get()
    {
        return Enumerable.Range(1, 5).Select(index => new WeatherForecast
        {
            Date = DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
            TemperatureC = Random.Shared.Next(-20, 55),
            Summary = Summaries[Random.Shared.Next(Summaries.Length)]
        })
        .ToArray();
    }
}
```

El siguiente código proporciona la misma funcionalidad en un proyecto de API mínimo. Tenga en cuenta que el enfoque mínimo de API implica incluir el código relacionado en expresiones lambda.

```
namespace MinimalAPI;

public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        var app = builder.Build();

        app.UseHttpsRedirection();

        var summaries = new[]
        {
            "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot", "Sweltering", "Scorching"
        };

        app.MapGet("/weatherforecast", (HttpContext httpContext) =>
        {
            var forecast = Enumerable.Range(1, 5).Select(index =>
                new WeatherForecast
                {
                    Date = DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
                    TemperatureC = Random.Shared.Next(-20, 55),
                    Summary = summaries[Random.Shared.Next(summaries.Length)]
                })
                .ToArray();
            return forecast;
        });

        app.Run();
    }
}
```

Ambos proyectos API se refieren a la siguiente clase:

```
namespace APIWithControllers;

public class WeatherForecast
{
    public DateOnly Date { get; set; }

    public int TemperatureC { get; set; }

    public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);

    public string? Summary { get; set; }
}
```

Las API mínimas tienen muchas de las mismas capacidades que las API basadas en controladores. Admiten la configuración y la personalización necesarias para escalar a varias API, manejar rutas complejas, aplicar reglas de autorización y controlar el contenido de las respuestas de la API. Hay algunas capacidades disponibles con API basadas en controlador que aún no son compatibles o implementadas por API mínimas. Éstas incluyen:

- No hay soporte integrado para el enlace de modelos ( `IModelBinderProvider` , `IModelBinder` ). El soporte se puede agregar con una cuña de unión personalizada.
- No hay soporte incorporado para la validación ( `IModelValidator` ).

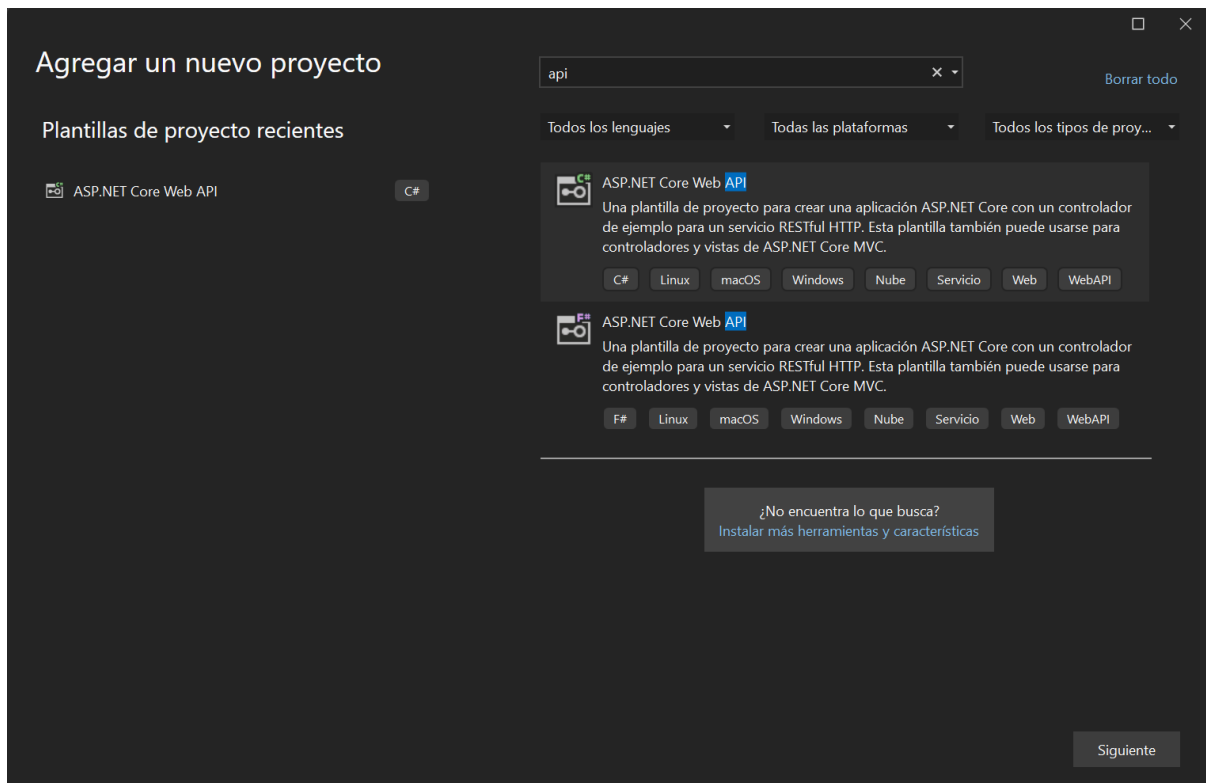
- No hay soporte para las partes de la aplicación o el modelo de la aplicación. No hay manera de aplicar o construir sus propias convenciones.
- No hay soporte de representación de vista incorporado. Recomendamos usar Razor Pages para renderizar vistas.
- Sin soporte para Json Patch.
- Sin soporte para OData.

## Paso a paso para creación de una API REST en .NET CORE

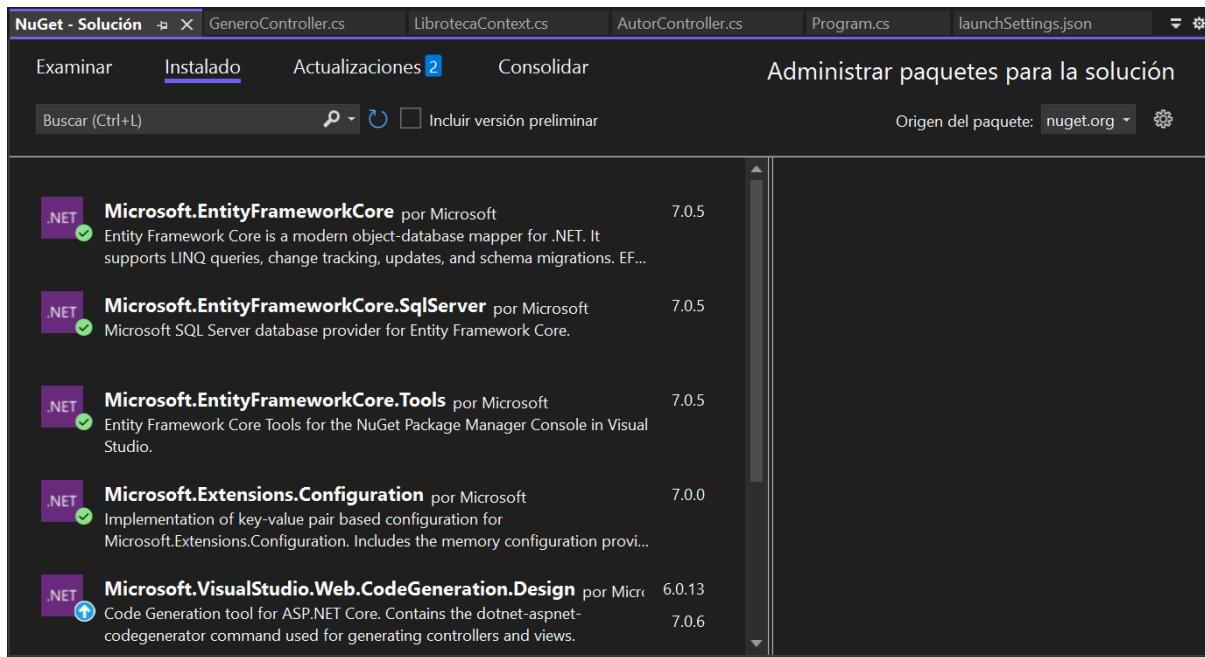
### Entorno y dependencia de paquetes

Lo primero será crear un proyecto con Visual Studio 2022 de tipo Aplicación web ASP.NET Core, será de tipo Core Web API, como se muestra en las siguiente imagen.

Entonces, abrimos el IDE Visual Studio, posteriormente seleccionamos un proyecto existente o creamos uno nuevo. Seleccionamos la opción ASP .NET Core Web API.



Ahora instalaremos los paquetes necesarios para que funcione el Entity Framework en un proyecto .Net Core. Estas dependencias se agregan desde el Nugget como se muestra a continuación:



## Estructura y configuraciones

Una vez instalados todos los paquetes que necesitamos, es momento de crear la estructura de carpetas del proyecto, así que creamos la carpeta Entities, Models para nuestro caso, en donde irán todas las entidades que se corresponderán cada una con una tabla de la base de datos, recordamos que utilizaremos Entity Framework con la técnica Code first.

*“La técnica Code First consiste en crear una BD por completo a partir de nuestras entidades.”*

Luego necesitamos asociar la base de datos para aplicar Database First. Para esto utilizamos un comando dentro de la consola de administración de paquetes. El siguiente comando indica que debe tomar la bdd “libroteca” y debe crear clases en base a ella en la carpeta “Models” (previamente creada) y su respectivo contexto.

```
Scaffold-DbContext "Server=.\SQLEXPRESS; Database=libroteca;Trusted_Connection=True;"  
Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
```

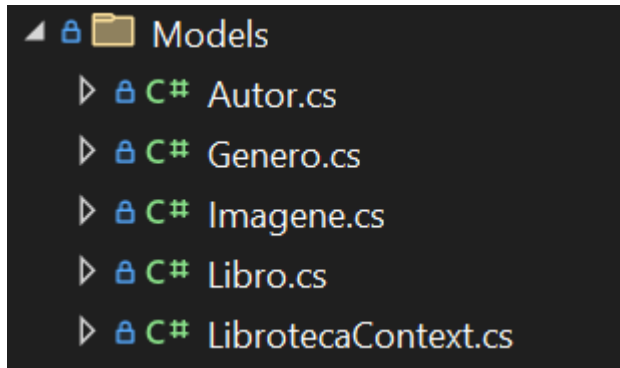
Al ejecutar, recibimos este error:

```
Error Number: -2146893019,State:0,Class:20  
A connection was successfully established with the server, but then an error occurred during the login process. (provider: SSL  
Provider, error: 0 - La cadena de certificación fue emitida por una entidad en la que no se confía.)
```

Lo solucionamos agregando el atributo “Encrypt=False” (Opción no recomendada) a la línea de comando previamente ejecutada:

```
Scaffold-DbContext "Server=.\SQLEXPRESS; Database=libroteca;trusted_connection=true;encrypt=false;"  
Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
```

De esta manera se crearon exitosamente las clases en base a las tablas de Librería, junto con su contexto.



Ejemplo de una de las clases:

```
1  using System;
2  using System.Collections.Generic;
3
4  namespace Libreria.Models;
5
6  13 referencias
7  public partial class Libro
8  {
9      5 referencias
10     public int Id { get; set; }
11
12     1 referencia
13     public string Titulo { get; set; } = null!;
14
15     1 referencia
16     public string Sinopsis { get; set; } = null!;
17
18     1 referencia
19     public string Contenido { get; set; } = null!;
20
21     1 referencia
22     public DateTime FechaEmision { get; set; }
23
24     2 referencias
25     public int AutorId { get; set; }
26
27     2 referencias
28     public int ImagenId { get; set; }
29
30     2 referencias
31     public int GeneroId { get; set; }
32
33     1 referencia
34     public virtual Autor Autor { get; set; } = null!;
35
36     1 referencia
37     public virtual Genero Genero { get; set; } = null!;
38
39     1 referencia
40     public virtual Imagen Imagen { get; set; } = null!;
41 }
```



## Controlador para el Web API

Posteriormente, creamos su respectivo controller, responsable de procesar las peticiones y que instancia al contexto junto con sus respectivos métodos HTTP (GET, POST, PUT, DELETE):

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Threading.Tasks;
5  using Microsoft.AspNetCore.Http;
6  using Microsoft.AspNetCore.Mvc;
7  using Microsoft.EntityFrameworkCore;
8  using Libroteca.Models;
9
10 namespace Libroteca.Controllers
11 {
12     [Route("api/[controller]")]
13     [ApiController]
14     public class LibrotecaController : ControllerBase
15     {
16         private readonly LibrotecaContext _context;
17
18         public LibrotecaController(LibrotecaContext context)
19         {
20             _context = context;
21         }
22
23         // GET: api/Libroteca
24         [HttpGet]
25         public async Task<ActionResult<IEnumerable<Libro>>> GetLibros()
26         {
27             if (_context.Libros == null)
28             {
29                 return NotFound();
30             }
31             return await _context.Libros.ToListAsync();
32         }
33
34         // GET: api/Libroteca/5
35         [HttpGet("{id}")]
36         public async Task<ActionResult<Libro>> GetLibro(int id)
37         {
38             if (_context.Libros == null)
39             {
40                 return NotFound();
41             }
42             var libro = await _context.Libros.FindAsync(id);
43
44             if (libro == null)
45             {
46                 return NotFound();
47             }
48
49             return libro;
50         }
51     }
```

## Otros métodos dentro del controlador

```
51 // PUT: api/Libroteca/5
52 // To protect from overposting attacks, see https://go.microsoft.com/fwlink/?linkid=2123754
53 [HttpPut("{id}")]
54 0 referencias
55 public async Task<IActionResult> PutLibro(int id, Libro libro)
56 {
57     if (id != libro.Id)
58     {
59         return BadRequest();
60     }
61
62     _context.Entry(libro).State = EntityState.Modified;
63
64     try
65     {
66         await _context.SaveChangesAsync();
67     }
68     catch (DbUpdateConcurrencyException)
69     {
70         if (!LibroExists(id))
71         {
72             return NotFound();
73         }
74         else
75         {
76             throw;
77         }
78     }
79
80     return NoContent();
81 }
82
83 // POST: api/Libroteca
84 // To protect from overposting attacks, see https://go.microsoft.com/fwlink/?linkid=2123754
85 [HttpPost]
86 0 referencias
87 public async Task<ActionResult<Libro>> PostLibro(Libro libro)
88 {
89     if (_context.Libros == null)
90     {
91         return Problem("Entity set 'LibrotecaContext.Libros' is null.");
92     }
93     _context.Libros.Add(libro);
94     await _context.SaveChangesAsync();
95
96     return CreatedAtAction("GetLibro", new { id = libro.Id }, libro);
97 }
98
99 // DELETE: api/Libroteca/5
100 [HttpDelete("{id}")]
101 0 referencias
102 public async Task<IActionResult> DeleteLibro(int id)
103 {
104     if (_context.Libros == null)
105     {
106         return NotFound();
107     }
108     var libro = await _context.Libros.FindAsync(id);
109     if (libro == null)
110     {
111         return NotFound();
112     }
113
114     _context.Libros.Remove(libro);
115     await _context.SaveChangesAsync();
116
117     return NoContent();
118 }
119
120 1 referencia
121 private bool LibroExists(int id)
122 {
123     return (_context.Libros?.Any(e => e.Id == id)).GetValueOrDefault();
124 }
```

Ahora sólo nos queda configurar el contexto en Program.cs, para que en cada request entienda a qué hacemos referencia en el controler:

```
1 using Libroteca.Models;
2
3 var builder = WebApplication.CreateBuilder(args);
4
5 // Add services to the container.
6
7 builder.Services.AddControllers();
8 // Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
9 builder.Services.AddEndpointsApiExplorer();
10 builder.Services.AddSwaggerGen();
11 builder.Services.AddTransient<LibrotecaContext>();
```

Teniendo todo esto funcionando, podemos empezar a consumir la API de forma local a través del puerto 7107. A través del siguiente link, teniendo en cuenta el ejemplo proporcionado: localhost:7107/api/Libroteca.

**Aclaración:** Para los métodos PUT y POST, utilizamos Binding de parámetros complejos (instancias de clases). Que por defecto, se bindean desde el body.

## Documentación de la API

Aunque las API son explicativas, la documentación de las mismas sirve de guía para mejorar su uso. Las API bien documentadas que ofrecen una gama de funciones y casos de uso tienden a ser más populares en una arquitectura orientada a servicios.

La escritura de la documentación completa de la API forma parte del proceso de administración de la API. La documentación de la API puede generarse automáticamente mediante herramientas como **Swagger**, la cual veremos más adelante, o escribirse manualmente. Entre algunas de las prácticas recomendadas se encuentran las siguientes:

- Escribir las explicaciones en un inglés sencillo y fácil de leer. Los documentos generados por las herramientas pueden resultar farragosos y requerir su edición.
- Utilizar ejemplos de código para explicar la funcionalidad.
- Mantener la documentación para que sea precisa y esté actualizada.
- Orientar el estilo de escritura a los principiantes
- Cubrir todos los problemas que la API puede resolver por los usuarios.

## Swagger

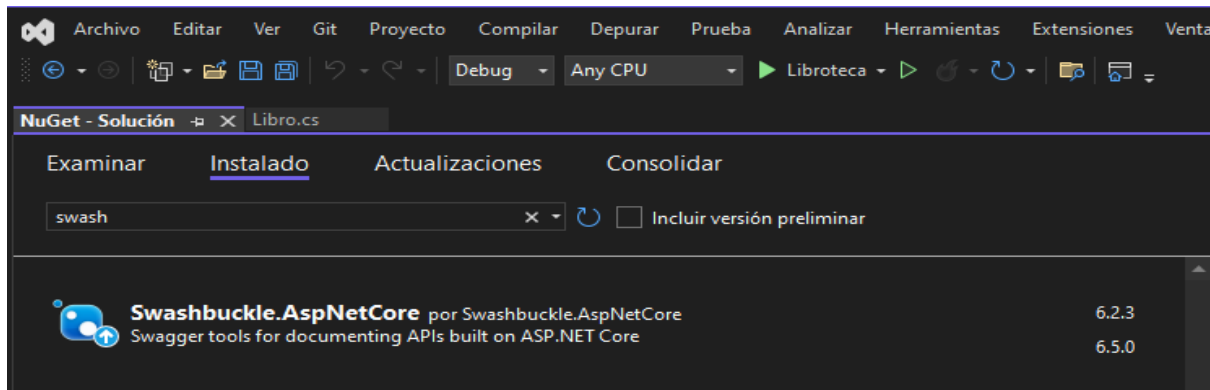
*“Swagger es una serie de reglas, especificaciones y herramientas que nos ayudan a documentar nuestras APIs.”*

Cuando hablamos de Swagger nos referimos a una serie de reglas, especificaciones y herramientas que nos ayudan a documentar nuestras APIs. De esta manera, podemos realizar documentación que sea realmente útil para las personas que la necesitan. Swagger

nos ayuda a crear documentación que todo el mundo entienda. Veamos cómo se implementa esta herramienta con una aplicación de .NET Core para luego explorar en detalle sus funcionalidades

## Configuración en .NET Core

### Instalar el paquete de Swagger Swashbuckle.AspNetCore



### Agregar Swagger middleware

*“Un middleware es una clase que permite manipular una petición (o respuesta) HTTP, actúa en cada request que llega a nuestra aplicación, por lo tanto es parte de la request pipeline.”*

Agregamos el generador de Swagger a la colección de servicios en Program.cs. Para esto utilizaremos la siguiente declaración “builder.Services.AddSwaggerGen();”

```
// Add services to the container.

builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
builder.Services.AddSingleton<LibroContext, LibroContext>();
```

### Habilitar el middleware

```
var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
```

Continuando con la teoría, la principal ventaja es que todo el mundo entiende Swagger, aun si no se tienen vastos conocimientos se puede lograr entender su funcionamiento gracias al

Swagger UI. Otra ventaja es que incluso las máquinas pueden leerlo. Por otro lado, la documentación creada por esta herramienta puede utilizarse directamente para automatizar procesos dependientes de APIs.

Entonces, ¿qué es Swagger? Es una documentación online que se genera sobre una API. Por lo tanto, en esta herramienta podemos ver todos los endpoint que hemos desarrollado en nuestra API. También nos demuestra cómo son los elementos o datos que debemos pasar para hacer que funcione y nos permite probarlos directamente en su interfaz.

Una vez finalizada la configuración que se demostró anteriormente, podemos acceder a Swagger a través de la URL que hayamos declarado en nuestro archivo `launchSettings.json`, es importante agregar `/swagger` al final de nuestra URL de lo contrario no se estará llamando a la herramienta.

### `launchSettings.json`

```
{
  "profiles": {
    "Libroteca": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "launchUrl": "swagger",
      "applicationUrl": "https://localhost:7107;http://localhost:5128",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

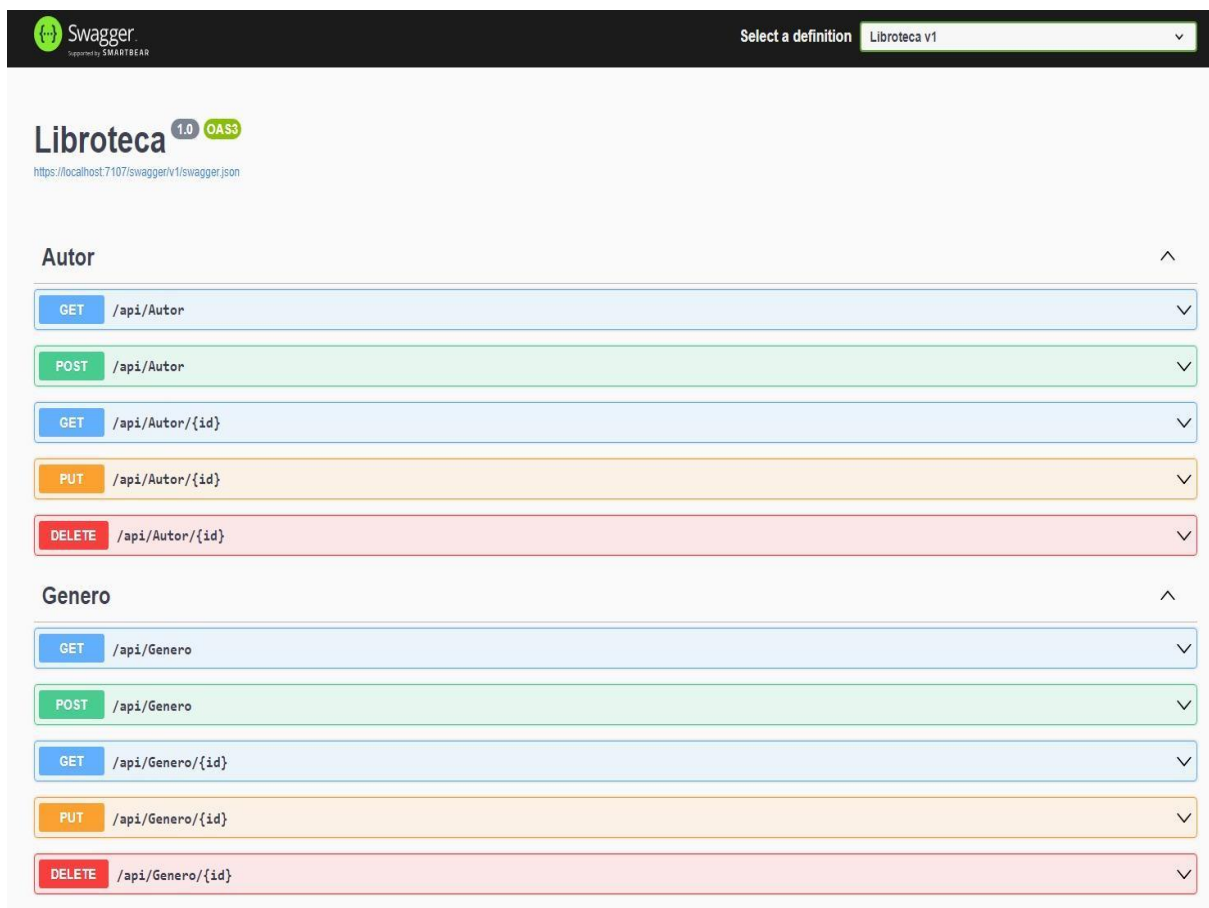
Anteriormente mencionamos que Swagger UI nos facilita el entendimiento de cómo funciona Swagger, veamos de qué se trata.

### Swagger UI – La interfaz de usuario de Swagger

Swagger UI permite a cualquier persona, visualizar e interactuar con los recursos de la API. Se genera automáticamente a partir de escribir archivos que cumplan con la especificación OpenAPI, con esto, la documentación visual se crea y facilita la implementación de back-end y el consumo del lado del cliente.

Consiste en una infraestructura de visualización que puede analizar la especificación OpenAPI, hablaremos de esta especificación más adelante, y generar una consola de API para que los usuarios puedan aprender y ejecutar las APIs de forma rápida y sencilla. De esta manera nos encontramos con que todas las APIs creadas en nuestro proyecto están disponibles en una sola página.

Swagger UI utiliza un documento JSON o YAML existente y lo hace interactivo. Crea una plataforma que ordena cada uno de nuestros métodos (get, put, post, delete) y categoriza nuestras operaciones. Cada uno de los métodos es expandible, y en ellos podemos encontrar un listado completo de los parámetros con sus respectivos ejemplos e incluso podemos probar valores de llamada



Swagger  
Supported by SMARTBEAR

Select a definition Libroteca v1

**Libroteca** 1.0 OAS3  
<https://localhost:7107/swagger/v1/swagger.json>

**Autor**

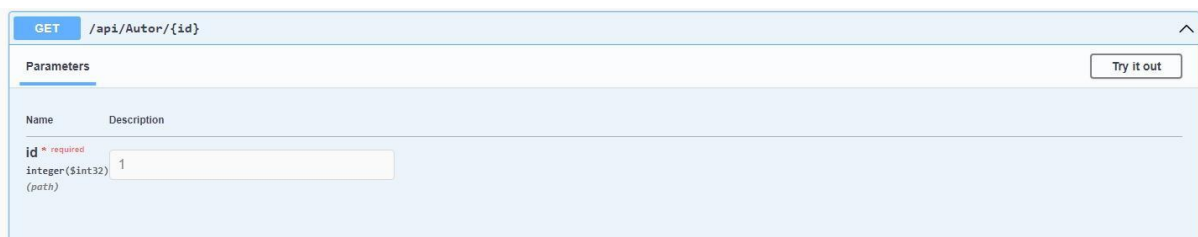
- GET /api/Autor
- POST /api/Autor
- GET /api/Autor/{id}
- PUT /api/Autor/{id}
- DELETE /api/Autor/{id}

**Genero**

- GET /api/Genero
- POST /api/Genero
- GET /api/Genero/{id}
- PUT /api/Genero/{id}
- DELETE /api/Genero/{id}

## Demostración sobre prueba de Endpoint

Para probar los elementos de la API en Swagger, se debe hacer clic sobre el endpoint que se desea probar, aquí se encontrará un botón que dice “try it out”. Aunque previo a esta acción ya se nos indica los parámetros que debemos colocar. En este caso, un identificador.



GET /api/Autor/{id}

Parameters

Try it out

| Name                                      | Description |
|---|-------------|
| id * required<br>integer(int32)<br>(path) | 1           |

Después de esto, Swagger nos permitirá ingresar el valor o valores requeridos y se nos habilitará la posibilidad de hacer clic sobre un nuevo botón azul “Execute” para ejecutar el código editado. Al hacer esto, se obtendrá una instrucción para ejecutar el curl y la respuesta a la acción ejecutada.

## Resultado de la prueba

The screenshot shows the Swagger UI for the endpoint `GET /api/Autor/{id}`. The **Parameters** tab is active, showing a required path parameter `id` of type `integer($int32)` with a value of `1`. The **Execute** button is visible. Below, the **Responses** section shows a `200` status code with a `Content-Type` of `application/json; charset=utf-8`. The response body is a JSON object:

```
{
  "id": 1,
  "nombreApellido": "Joanne Rowling",
  "fechaNacimiento": "1965-07-31T00:00:00",
  "origen": "Reino Unido",
  "libros": []
}
```

The response headers are also displayed:

```
content-type: application/json; charset=utf-8
date: Sun, 04 Jun 2023 09:02:07 GMT
server: Kestrel
```

En Swagger, podemos hacer clic en cada operación para ver los detalles de la solicitud, incluyendo la URL, los parámetros de la ruta, los parámetros de consulta y los posibles encabezados requeridos. También podemos ver ejemplos de carga útil (payload) en formato JSON o XML si se proporcionan en la documentación.

The screenshot shows the Swagger UI for the endpoint `PUT /api/Autor/{id}`. The **Parameters** tab is active, showing a required path parameter `id` of type `integer($int32)` with a value of `id`. The **Request body** tab is also visible, showing a complex JSON payload with a `Content-Type` of `application/json`.

```
{
  "id": 0,
  "nombreApellido": "string",
  "fechaNacimiento": "2023-06-04T09:18:53.853Z",
  "origen": "string",
  "libros": [
    {
      "id": 0,
      "titulo": "string",
      "sinopsis": "string",
      "contenido": "string",
      "fechaEmision": "2023-06-04T09:18:53.853Z",
      "autorId": 0,
      "imagenId": 0,
      "generoId": 0,
      "autor": {
        "id": 0,
        "nombre": "string",
        "libros": [

```

En algunas ocasiones nos podemos encontrar con que los endpoints de Swagger estén bloqueados, si este es el caso, cuando se ingrese a Swagger se visualizará un símbolo de candado en algunos de los endpoint. Esto significa que este endpoint está protegido y se debe pasarle un token para acceder a sus respuestas. La forma en la que se accedera al token varía según el proyecto. Por ello, se recomienda hablar con el equipo para conocer la forma de acceder a estos endpoint protegidos. Normalmente, el acceso se dará ejecutando una serie de acciones con las que obtener un token como respuesta.

### ¿Pero qué es OpenAPI?

Inicialmente comenzó como parte del marco Swagger, siendo conocida como “Especificación swagger”. Swagger presentó las mejores prácticas de construcción de APIs y luego esas mejores prácticas se convirtieron en la especificación OpenAPI, pasando a ser un proyecto separado de colaboración de código abierto de la Fundación Linux, Swagger y algunas otras herramientas que pueden generar código, documentación y casos de prueba con un archivo de interfaz, supervisado por la Iniciativa OpenAPI.

Como ya se explicó antes Swagger trabaja bajo un estándar denominado OpenAPI. Se trata de un estándar global para escribir APIs. Es como una especificación que permite a los desarrolladores de todo el planeta estandarizar el diseño de sus APIs. Además, cumple con toda la seguridad, el versionado, el manejo de errores y otras mejores prácticas al escribir API desde cero. Esto no quiere decir que se tiene que volver a escribir el código de una API existente que no fue diseñada con estas especificaciones ni tampoco exige un proceso de desarrollo específico solo deben ajustarse para cumplir con este estándar global.

### Características

Las características más importantes de OpenAPI son las siguientes:

- Ayuda a establecer un buen diseño de las APIs.
- Documentación completa.
- Testing más rápido gracias a la generación de un sandbox.
- Mejora el Time to market, es el tiempo que transcurre desde que se concibe un producto hasta que está disponible para la venta.
- Generación de un portal de documentación que describe la API, en formato human-readable.

Las aplicaciones implementadas en base a archivos de interfaz OpenAPI pueden generar automáticamente documentación de métodos, parámetros y modelos. Esto ayuda a mantener sincronizados la documentación, las bibliotecas cliente y el código fuente.

Si lo resumimos en pocas palabras, OpenAPI es quien permite descubrir y comprender las capacidades de un servicio o una API, sin necesidad de acceder al código fuente, sin documentación adicional o inspección de las peticiones. Cuando se diseña correctamente

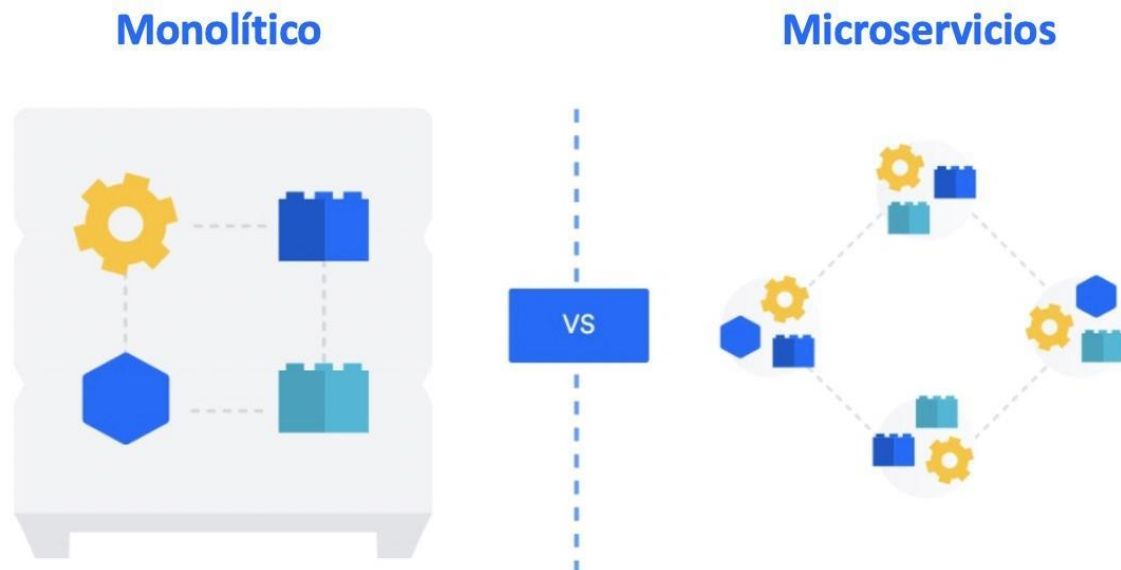


una API a través de esta especificación, un desarrollador puede comprender e interactuar con el servicio mucho más fácil ya que se eliminan las suposiciones al llamar a un servicio.

## Microservicios

*“Los microservicios son un enfoque arquitectónico y organizativo para el desarrollo de software donde el software está compuesto por pequeños servicios independientes que se comunican a través de API bien definidas. “*

Las arquitecturas de microservicios hacen que las aplicaciones sean más fáciles de escalar y más rápidas de desarrollar. Esto permite la innovación y acelera el tiempo de comercialización. La arquitectura de microservicios se considera un reemplazo moderno y flexible del modelo de desarrollo más tradicional de la arquitectura monolítica.



En una arquitectura monolítica, generalmente hay un servidor virtualizado o físico asignado a cada aplicación y esos servidores siempre están en ejecución. La disponibilidad y el escalamiento de la aplicación dependen completamente del hardware subyacente, que es una entidad fija. Los microservicios, por el contrario, pueden operar varias instancias en un solo servidor o en varios servidores, a medida que los recursos escalan de forma dinámica para admitir las demandas de la carga de trabajo. Los microservicios individuales suelen estar en contenedores para mejorar la portabilidad y la escalabilidad.

## Características de los microservicios

- **Autónomos**

Cada servicio componente en una arquitectura de microservicios se puede desarrollar, implementar, operar y escalar sin afectar el funcionamiento de otros servicios. Los servicios no necesitan compartir ninguno de sus códigos o implementaciones con

otros servicios. Cualquier comunicación entre componentes individuales ocurre a través de API bien definidas.

- **Especializados**

Cada servicio está diseñado para un conjunto de capacidades y se enfoca en resolver un problema específico. Si los desarrolladores aportan más código a un servicio a lo largo del tiempo y el servicio se vuelve más complejo, se puede dividir en servicios más pequeños.

## Beneficios de los microservicios

- **Agilidad**

Los microservicios fomentan una organización de equipos pequeños e independientes que se apropian de los servicios. Los equipos actúan en un contexto pequeño y bien comprendido, y están facultados para trabajar de forma más independiente y más rápida. Esto acorta los tiempos del ciclo de desarrollo. Usted se beneficia significativamente del aumento de rendimiento de la organización.

- **Escalado flexible**

Los microservicios permiten que cada servicio se escale de forma independiente para satisfacer la demanda de la característica de la aplicación que respalda. Esto permite a los equipos adecuarse a las necesidades de la infraestructura, medir con precisión el costo de una característica y mantener la disponibilidad si un servicio experimenta un aumento en la demanda.

- **Implementación sencilla**

Los microservicios permiten la integración y la entrega continuas, lo que facilita probar nuevas ideas y revertirlas si algo no funciona. El bajo costo de los errores permite experimentar, facilita la actualización del código y acelera el tiempo de comercialización de las nuevas características.

- **Libertad tecnológica**

Las arquitecturas de microservicios no siguen un enfoque de "diseño único". Los equipos tienen la libertad de elegir la mejor herramienta para resolver sus problemas específicos. Como consecuencia, los equipos que crean microservicios pueden elegir la mejor herramienta para cada trabajo.

- **Código reutilizable**

La división del software en módulos pequeños y bien definidos les permite a los equipos usar funciones para diferentes propósitos. Un servicio escrito para una determinada función se puede usar como un componente básico para otra característica. Esto permite que una aplicación arranque por sí sola, ya que los desarrolladores pueden crear nuevas capacidades sin tener que escribir código desde cero.

- **Resistencia**

La independencia del servicio aumenta la resistencia de una aplicación a los errores. En una arquitectura monolítica, un error en un solo componente, puede provocar un error en toda la aplicación. Con los microservicios, si hay un error en todo el servicio, las aplicaciones lo manejan degradando la funcionalidad sin bloquear toda la aplicación.

## Impacto en la transformación digital

En los últimos años las empresas están migrando los datos y las cargas de trabajo a la nube por todas las ventajas que ofrece esta tecnología y está provocando un aumento de las arquitecturas de microservicios. Una tendencia en las medianas y grandes empresas que actúa como motor de su Transformación Digital al tratarse de arquitecturas muy escalables, flexibles y que permiten la innovación.

La adaptabilidad, la capacidad de cambiar rápida y fácilmente, se ha convertido en un objetivo primordial para las empresas modernas y ha presionado a los equipos tecnológicos para que construyan plataformas más fáciles y menos costosas de cambiar.

Trabajando en estos entornos, estos equipos han sido atraídos cada vez más al estilo de utilizar los microservicios en la arquitectura de software. Lo que los atrae es la promesa de un método para acelerar los cambios en el software, sin introducir peligro innecesario para el negocio.

La forma de hacer los servicios a través de los microservicios es posible en gran parte al favorecer la descentralización de los componentes y datos de software, más específicamente, dividiendo los elementos “monolíticos” en piezas más pequeñas y fáciles de cambiar y desplegando esas piezas en la red. Hacer que esta arquitectura funcione bien requiere un cambio en la forma en que se realiza el trabajo y cómo se regula el trabajo.

En el diseño organizacional, el objetivo es descentralizar la autoridad de decisión. En lugar de tener algunas personas que toman las decisiones de arquitectura y diseño de software para todos en la organización, la descentralización les permitiría distribuir el poder de decisión entre las personas que hacen el trabajo.

Empujar la autoridad de toma de decisiones directamente a los trabajadores les permite producir con más libertad y autonomía. Bajo las circunstancias adecuadas, esto conducirá a cambios mejores y más rápidos. Sin embargo, si su organización se equivoca, una serie de malas decisiones pueden ralentizar la tasa de cambio o peor, terminan dañando su negocio.

Encontrar la estrategia correcta de descentralización es un proceso evolutivo que requiere que las organizaciones se ajusten, analicen y adapten.

## React

*“Es una librería open source de JavaScript para desarrollar interfaces de usuario. Fue lanzada en el año 2013 y desarrollada por Facebook, quienes también la mantienen actualmente junto a una comunidad de desarrolladores independientes y compañías.”*

React cuenta con un óptimo desempeño que se encarga de actualizar y renderizar los cambios realizados de forma automática. Esta cualidad permite a los programadores desarrollar sus códigos sin mayores contratiempos en el modelo en objetos para la representación de documentos (DOM). Entonces podemos ver que react ofrece grandes beneficios en performance, modularidad y promueve un flujo muy claro de datos y eventos, facilitando la planeación y desarrollo de apps complejas.

El secreto de ReactJS para tener un performance muy alto, es que implementa algo llamado Virtual DOM y en vez de renderizar todo el DOM en cada cambio, que es lo que normalmente se hace, este hace los cambios en una copia en memoria y después usa un algoritmo para comparar las propiedades de la copia en memoria con las de la versión del DOM y así aplicar cambios exclusivamente en las partes que varían.

Esto puede sonar como mucho trabajo, pero en la práctica es mucho más eficiente que el método tradicional pues si tenemos una lista de dos mil elementos en la interfaz y ocurren diez cambios, es más eficiente aplicar diez cambios, ubicar los componentes que tuvieron un cambio en sus propiedades y renderizar estos diez elementos, que aplicar diez cambios y renderizar dos mil elementos.

Son más pasos a planear y programar, pero ofrece una mejor experiencia de usuario y una planeación muy lineal.

## Características

Antes de mencionar las principales características, hay que mencionar la más importante la cual es que ReactJS promueve el flujo de datos en un solo sentido, en lugar del flujo bidireccional típico en Frameworks modernos, esto hace más fácil la planeación y detección de errores en aplicaciones complejas, en las que el flujo de información puede llegar a ser muy complejo, dando lugar a errores difíciles de ubicar.

### **1. Lenguaje JSX**

La sintaxis que usa React es el JavaScript XML (JSX), el cual es una combinación del lenguaje HTML y el JavaScript, por lo que también se considera una extensión. Así que la relación que hay entre estos es bastante estrecha. Algunos programadores han afirmado que solamente se diferencian por detalles y que puede coincidir en más de un 90 % el código del JSX con el de JavaScript. Así que las personas que ya dominen este lenguaje se adaptarán muy fácil a React.

## 2. Componentes

Regularmente dentro de la programación web se manejan tres carpetas: la de la estructura de la página web (HTML), la de la presentación de la interfaz (CSS) y la del comportamiento de la misma (JS). Por su lado, React decidió juntar todas estas en un mismo lugar, de manera dinámica, para dar lugar a los componentes. Estos son partes de la interfaz del usuario que es independiente y que se puede reutilizar en otro lugar del sitio web. Esta es una de las características más sobresalientes. Asimismo, existen dos tipos de componentes:

- **Componentes funcionales.** Son los más usados: representan una función de JavaScript, retornan a un elemento JSX de React, comienzan con mayúscula y pueden recibir valores.
- **Componentes de clase.** Se refiere a los componentes que están escritos en JavaScript moderno de una manera más elaborada: retornan a un elemento JSX a través de un método render y también es posible que se les asigne valores.

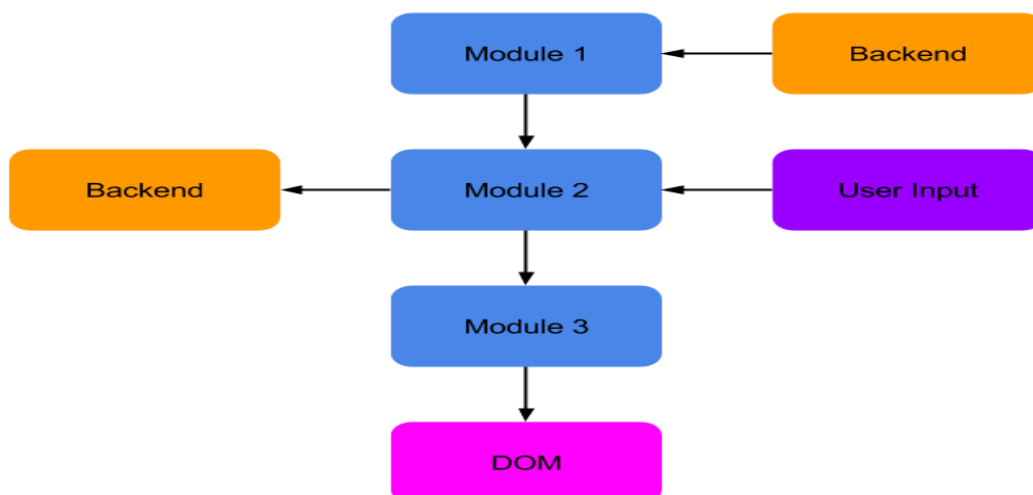
## 3. Ciclos de vida

Dentro de React los componentes pasan por una serie de etapas durante su creación. Regularmente se divide en tres fases esenciales: montaje, utilización y desmontaje.

## 4. Virtual DOM

El virtual DOM es una característica de React que permite que el rendimiento de la página web no se vea afectada por las actualizaciones y que estas, a su vez, se mantengan bien reflejadas en el modelo en objetos para la representación de documentos (DOM).

Normalmente, para que los cambios realizados en los estados de una página web se vean en la interfaz gráfica es necesario un proceso complejo, largo y costoso. Así que el virtual DOM se encarga de actuar como una memoria intermediaria entre las actualizaciones al estado de la página y el DOM real para renderizar los menos cambios posibles.



## Consumo de una API desde React

### Configurar las llamadas a la API

Dentro del proyecto de React, se crea un directorio donde se colocará todo el código relacionado con las llamadas a la API. Dentro de este directorio, se crea el archivo que contendrá las funciones básicas para realizar las llamadas a la API.

```
front > src > helpers > FetchApi.jsx > fetchPOST
1  const baseUrl = 'https://localhost:7107/api'
2
3  export const fetchGET = async (url) => {
4    try {
5      const response = await fetch(baseUrl+url, {
6        method: "GET",
7        headers: {
8          "Content-Type": "text/plain",
9        },
10     });
11     const jsonData = await response.json();
12
13     return jsonData;
14   } catch (err) {
15     console.error(err);
16   }
17 }
18
19 export const fetchPOST = async (url, data) => {
20   try {
21     const response = await fetch(baseUrl+url, {
22       method: "POST",
23       headers: {
24         'Content-Type': 'application/json',
25       },
26       body: JSON.stringify(data),
27     });
28     const jsonData = await response.json();
29     return jsonData;
30   } catch (err) {
31     console.error(err);
32   }
33 }
34
```

## Crear componentes en React para consumir la API

Luego se puede crear un componente que utilice todas las funciones de la API. En el siguiente ejemplo se crearon tres métodos, el primero retorna un listado de todos los libros, el segundo nos solicita un parámetro id para obtener la información de un libro, y el último es una función para agregar un nuevo libro a la base de datos. Un paso importante es declarar las funciones como “export” para luego poder importarlas en otros componentes.

```
librosEndpoints.jsx X
front > src > endpoints > librosEndpoints.jsx > addLibro
1  import { fetchGET, fetchPOST } from "../helpers/FetchApi";
2
3  export const getLibros = async () => {
4    const response = await fetchGET('/Libroteca');
5    return response;
6  }
7
8  export const getDetalleLibro = async (id) => {
9    const response = await fetchGET(`/Libroteca/${id}`);
10   return response;
11 }
12
13 export const addLibro = async (libro) => {
14   const response = await fetchPOST(`/Libroteca`, libro);
15   return response;
16 }
```

## Uso de las funciones desde otro componente

Habiendo creado las funciones anteriores ahora podemos llamarlas y utilizarlas desde cualquier componente. El siguiente componente utiliza el “useEffect” para llamar a getDetalleLibro y obtener la información de ese objeto cuando el componente se monta.

```
JS detail.js X
front > src > views > detail > JS detail.js > ...
1  import React, { useEffect, useState } from 'react';
2  import FadeDownAnimation from '../../components/animations/fade-down';
3
4  import './detail.css';
5  import { useLocation } from 'react-router-dom';
6  import { getDetalleLibro } from '../../endpoints/librosEndpoints';
7
8  function Detail() {
9    const [libro, setLibro] = useState(null);
10    const location = useLocation();
11    const libroId = location.state.libroId;
12
13    useEffect(() => {
14      if(!libroId) return;
15
16      const traerDetalleLibro = async () => {
17        setLibro(await getDetalleLibro(libroId));
18      }
19      console.log(libroId);
20      traerDetalleLibro();
21    }, []);
22 }
```

## Mostrando los valores en pantalla

```
JS details.js X
front > src > views > detail > JS details.js > ...

22
23   return (
24     <div className="detail">
25       <div className="detail-container container">
26         <FadeDownAnimation>
27           <div className="card shadow-lg border-0">
28             <div className="card-body my-5">
29               {libro?
30                 <div className="row">
31                   <div className="col-lg-5 col-md-5 col-sm-6">
32                     <div className="white-box text-center">
33                       </img>
35                     </div>
36                   </div>
37                   <div className="col-lg-7 col-md-7 col-sm-6">
38                     <div className="me-5">
39                       <h2 className="titulo">{libro.titulo}</h2>
40                       <h3 className="text-muted fs-5 mb-3 autor">{libro.autor.nombreApellido}</h3>
41                       <hr className="border opacity-100"></hr>
42                     </div>
43
44                     <div className="sinopsis-container rounded mt-3 me-5">
45                       <span className="fs-4 fw-bold sinopsis">Sinopsis</span>
46                       <p className="contenido mt-3">{libro.sinopsis}</p>
47                     </div>
48
49                     <span className="fs-5 fw-bold categoria">Género</span>
50                     <p className="border border-dark text-center rounded-pill p-1 mt-3 w-25">{libro.genero.nombre}</p>
51                   </div>
52                 </div>
53               : <div className="row"></div>
54             </div>
55           </div>
56         </div>
57       </FadeDownAnimation>
58     </div>
59   </div>
60 );
61 }
```



## Conclusiones

En conclusión, este proyecto de investigación se abordó la implementación de una API REST en .NET Core utilizando Swagger para diseñar y documentar el contrato de la API bajo el estándar Open API, así como el uso de React como aplicación cliente para interactuar con la API. Además, se exploró el concepto de microservicios y su aplicación en arquitecturas escalables.

Por lo que podemos decir que las API REST son interfaces de comunicación entre sistemas que utilizan HTTP para obtener datos o ejecutar operaciones sobre ellos. Proporcionan un conjunto de funciones como GET, PUT, DELETE, etc., y se basan en el principio de transferencia de estado representacional (REST).

El uso de Swagger nos facilita la generación de documentación interactiva y la definición de contratos bien definidos para las APIs que generemos. Esto nos ayuda a mejorar la comunicación entre los equipos de desarrollo y los clientes de la API, y facilita la integración y el consumo de servicios por parte de terceros.

Se utilizó React como tecnología para desarrollar la interfaz de usuario de la aplicación cliente que consume la API REST implementada en .NET Core. Encontramos que este framework proporciona una forma eficiente y escalable de interactuar con la API y manejar los datos recibidos.

Por último, se destacó el concepto de microservicios como una arquitectura escalable para el desarrollo de aplicaciones empresariales. Los microservicios descomponen una aplicación en componentes más pequeños y autónomos, lo que permite una mayor modularidad, despliegue independiente y facilita el mantenimiento y escalabilidad de la aplicación.

## Referencias/Bibliografía

- <https://aws.amazon.com/es/what-is/api/>
- <https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-7.0&tabs=visual-studio>
- <https://appmaster.io/es/blog/que-es-la-api-rest-y-en-que-se-diferencia-de-otros-tipos>
- <https://www.redhat.com/es/topics/api/what-is-a-rest-api>
- <https://blog.hubspot.es/website/que-es-api-rest>
- <https://keepcoding.io/blog/que-es-swagger/>
- <https://www.chakray.com/es/swagger-y-swagger-ui-por-que-es-imprescindible-para-tusapis/#:~:text=Cuando%20hablamos%20de%20Swagger%20nos,que%20todo%20el%20mundo%20entienda.>
- <https://aws.amazon.com/es/microservices/>
- <https://www.intel.la/content/www/xl/es/cloud-computing/microservices.html>
- <https://www.ibm.com/es-es/topics/microservices>
- <https://platzi.com/blog/react-js-de-javascript/>
- <https://blog.hubspot.es/website/que-es-react>