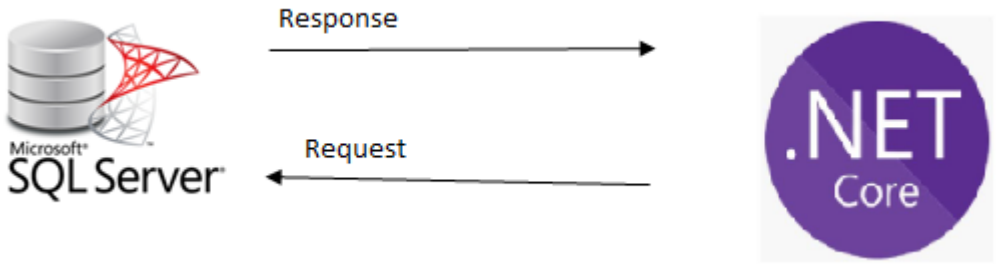
Solución Prueba Técnica Desarrollador de Software.

Interfaz de comunicación – Api Rest **SQL SERVER Y .NET CORE**

Una API REST (Application Programming Interface, o Interfaz de Programación de Aplicaciones, en español) es una forma de permitir que dos sistemas se comuniquen entre sí a través de una interfaz basada en HTTP. Una API REST utiliza métodos HTTP estándar, como GET, POST, PUT y DELETE, para realizar operaciones en un sistema.

Base de datos Backend



Como crear la interfaz de comunicación. Entre SQL SERVER Y .NET CORE

Para hacer un CRUD (Create, Read, Update, Delete, o Crear, Leer, Actualizar, Eliminar, en español) estilo banco en .NET Core con Entity Framework, necesitarías seguir los siguientes pasos:

- Crea un proyecto de .NET Core y agrega Entity Framework como una dependencia.
  Ejem:

```
dotnet new web -o BankAPI
cd BankAPI
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

- Crea un modelo para representar tus entidades de banco, como cuentas, transacciones y préstamos. Asegúrate de utilizar anotaciones de Entity Framework para especificar la configuración de la base de datos.
  Ejem:

```
public class BankContext : DbContext
{
    public BankContext(DbContextOptions<BankContext> options)
        : base(options)
    { }

    public DbSet<Account> Accounts { get; set; }
    public DbSet<Transaction> Transactions { get; set; }
    public DbSet<Loan> Loans { get; set; }
}
```

```
public class Account
{
    [Key]
    public int AccountId { get; set; }
    public string AccountNumber { get; set; }
    public decimal Balance { get; set; }
}

public class Transaction
{
    [Key]
    public int TransactionId { get; set; }
    public int AccountId { get; set; }
    public Account Account { get; set; }
    public decimal Amount { get; set; }
    public string Description
}
```
Regenerate response

- Crea un contexto de Entity Framework para gestionar la conexión a la base de datos y el seguimiento de cambios en tus entidades.

```
services.AddDbContext<BankContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
```

- Crea un controlador para manejar las solicitudes HTTP. Utiliza el contexto de Entity Framework para realizar operaciones CRUD en tus entidades de banco.

```
[Route("api/[controller]")]
[ApiController]
public class AccountsController : ControllerBase
{
    private readonly BankContext _context;

    public AccountsController(BankContext context)
    {
        _context = context;
    }
```

- Crea vistas para mostrar al usuario las opciones de crear, leer, actualizar y eliminar. Utiliza el controlador para manejar las solicitudes del usuario y realizar las operaciones correspondientes en la base de datos.

```
// GET: api/Accounts
[HttpGet]
public async Task<ActionResult<IEnumerable<Account>>> GetAccounts()
{
    return await _context.Accounts.ToListAsync();
}

// GET: api/Accounts/5
[HttpGet("{id}")]
public async Task<ActionResult<Account>> GetAccount(int id)
{
    var account = await _context.Accounts.FindAsync(id);

    if (account == null
```

Estos son solo algunos pasos básicos para hacer un CRUD estilo banco en .NET Core con Entity Framework. Hay muchos detalles adicionales que podrías considerar, como la validación de entrada del usuario y la seguridad de la aplicación.

Interfaz de comunicación – Api Rest **.NET CORE Y ANGULAR**

- Utiliza el comando ng generate service para crear un servicio en Angular:

```
ng generate service my-service
```

- En el archivo del servicio generado, agrega la lógica que quieras incluir en el servicio. Por ejemplo, podrías tener un método que haga una llamada HTTP para obtener datos de una API externa:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class MyService {

  constructor(private http: HttpClient) { }

  getData(): Observable<any> {
    return this.http.get<any>('https://my-api.com/data');
  }
}
```

- Inyecta el servicio en el componente o en el módulo donde quieras utilizarlo:

```
import { MyService } from './my.service';

@Component({
  ...
})
export class MyComponent {
  constructor(private myService: MyService) { }

  ngOnInit() {
    this.myService.getData().subscribe(data => {
      console.log(data);
    });
  }
}
```