Bilbioteca - API

Apartado: Capa API (Remoting y Librerias de Tenaris)

> Se debe usar la versión .NET Framework 4.5 para el desarrollo de todas las partes del backend.

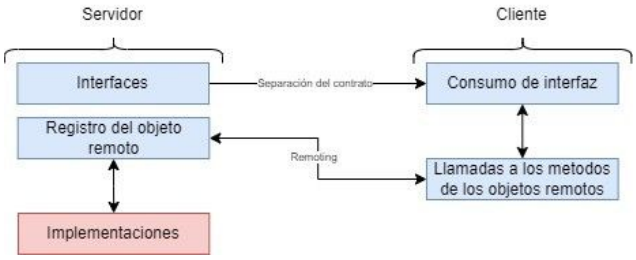
Acceso a la base de datos desde la aplicacion .NET

Tenaris cuenta con su propio Framework .NET llamado Tenaris Library Framework, sus usos son diversos y aplican para muchas areas de su sistema, para este proyecto vamos a centrarnos en los minimos e indispensables para generar la comunicacion con la base de datos y ejecucion de los stored procedures que necesitemos.

Remoting

> Tener en cuenta que esta tecnologia se considera obsoleta ante tecnologias mas modernas como WFC y gRPC, sin embargo, empresas como el cliente de esta capacitación la siguen utilizando.

Por un lado esta la comunicación con la base de datos y por el otro esta la comunicación con el exterior (Frontend o Cliente), de estas dos tareas se encarga el Servidor, ahora vamos a ver una forma de comunicación entre Cliente-Servidor utilizando Remoting.



El rectangulo rojo representa lo que el servidor no comparte (encapsulamiento)

Para quienes nunca hayan escuchado hablar de remoting o de patrones de diseño dificilmente entenderan la imagen, pero esta tomara mucho mas sentido luego.

Primero hablemos de algunos conceptos

1. Interfaz como contrato
   El servidor le brinda al cliente información acerca de los objetos que el servidor utiliza, en otras palabras le comparte información sobre las implementaciones que realiza, simplemente define una listra de operaciones que el objeto puede realizar (Interfaz)
2. Proxy
   Este es un patron de diseño se utiliza para proporcionar una representación de otro objeto y asi controlar el acceso a este. Para este caso no basta con explicarlo verbalmente asi que vamos a verlo con un poco de pseudocodigo.

Imagina dos procesos en paralelo a uno lo llamaremos `cliente` y al otro `servidor`, el `servidor` internamente tiene la siguiente clase

```
internal class RemoteUser : MarshalByRefObject, IRemoteUser
{
    private string _name;
    private string _lastname;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

    public string Lastname
    {
        get { return _lastname; }
        set { _lastname = value; }
    }
}
```

Si prestaste atencion notaras que la clase User implementa una interfaz llamada `IRemoteUser` y otra llamada `MarshalByRefObject` de esta hablaremos en otro momento por ahora centremosnos en `IRemoteUser` el aspecto de esta interfaz es la siguiente

```
public interface IRemoteUser
{
    string Name { get; set; }
    string Lastname { get; set; }
}
```

Cuando hablamos de compartir el contrato de un objeto nos referimos a que la **clase** `RemoteUser` sea **interna** del servidor y la interfaz **publica** para la solucion, en otras palabras, compartirmos la representacion del objeto tal cual queremos que sea visto por los clientes.

El servidor comparte el objeto a travez del “Registro del objeto”, para este caso el protocolo de comunicación es con `HTTP` aunque tambien podria ser a travez de otros protocolos como `TCP`, por ejemplo.

```
static void Main()
{
    HttpChannel channel = new HttpChannel(8085);
    ChannelServices.RegisterChannel(channel, false);
    RemotingConfiguration.RegisterWellKnownServiceType(
        typeof(RemotingLibrary.RemoteUser),
        "RemoteUser.soap",
        WellKnownObjectMode.Singleton
    );
}
```

Por otro lado el cliente obtendra la referencia del objeto `User` a travez de Remoting, el aspecto del codigo que se tendra en el cliente es apuntando a la interfaz del objeto y no a la implementacion que el cliente tiene es el siguiente, notese que para obtener la referencia tiene que usar una petición `HTTP`.

```
IRemoteUser remoteUser = (IRemoteUser)Activator.GetObject(
  typeof(IRemoteUser),
  "http://localhost:8085/RemoteUser.soap"
);
```

```
5 remoteUser.Name = "Test"; // Ejecuta el setter del modelo en el servidor
6 Console.WriteLine(remoteUser.Name);; //Ejecuta el get del modelo en el servidor y Devuelve "Test"
```

Centremosnos en la linea `5`, cuando establecemos el valor de la propiedad `Name` del `remoteUser`, aqui realmente lo que estamos haciendo es pidiendole a nuestro proceso `servidor` que establezca esa propiedad y no es el `cliente` el que esta cambiando el valor del propiedad realmente. A esta practica donde delegamos la responsabilidad en que otro proceso se encargue del objeto manipulado por referencia es conocida como `Proxy`

3. Transmision de llamadas a travez de serialización, deserialización y ejecucución en el servidor
   a. Serialización desde el cliente al servidor: Cuando invocas un método en el proxy (que implementa la interfaz), el proxy serializa esa llamada (incluyendo el nombre del método y cualquier argumento) y la envía al servidor.
   b. Deserialización, ejecución en el servidor y serialización de la respuesta: El servidor recibe la llamada serializada, la deserializa y determina qué método debe ser invocado en el objeto real. Luego, ejecuta ese método en el objeto real, toma el resultado, lo serializa y lo envía de vuelta al cliente.
   c. Deserialización de la respuesta en el cliente: El proxy en el cliente recibe la respuesta serializada, la deserializa y devuelve el resultado al código cliente como si la operación se hubiera realizado localmente.

> El compartir solo la interfaz asegura que el cliente sabe qué operaciones están disponibles y cómo llamarlas, pero no necesita conocer (ni debería conocer por razones de seguridad y encapsulación) la implementación real detrás de esas operaciones. Esto proporciona un desacoplamiento entre el cliente y el servidor, permitiendo que el servidor cambie la implementación real sin afectar a los clientes, siempre y cuando la interfaz (el contrato) se mantenga intacta.

El proyecto deberia tener tentativamente esta estructura.

```
- Proyecto
  - Controllers
    - LibroController.cs
    - ...
  - Repositories
    - ILibroRepository.cs
    - LibroRepository.cs
    - ...
  - Services
    - ILibroService.cs
    - LibroService.cs
    - ...
  - Models
    - Libro.cs
    - ...
  - DbContext
    - MyDbContext.cs
  - Startup.cs
  - ...
```

A continuación, te explico brevemente el propósito de cada carpeta:

- **Controllers**: Aquí se encuentran los controladores de tu API. Cada controlador es responsable de manejar las solicitudes HTTP y coordinar las acciones adecuadas utilizando los servicios correspondientes.
- **Repositories**: En esta carpeta se encuentran las interfaces y las implementaciones de los repositorios. Los repositorios son responsables de interactuar con la base de datos y realizar operaciones CRUD en las entidades.
- **Services**: Aquí se encuentran las interfaces y las implementaciones de los servicios. Los servicios contienen la lógica de negocio de tu aplicación y utilizan los repositorios para acceder a los datos.

- **Models**: Esta carpeta contiene las definiciones de tus entidades, como la clase "Producto" en nuestro ejemplo. Las entidades representan los objetos de tu dominio.
- **DbContext**: Aquí se encuentra la clase que hereda de DbContext y representa el contexto de la base de datos. Esta clase se encarga de mapear las entidades a las tablas de la base de datos y proporciona la funcionalidad para interactuar con la base de datos.
- **Startup.cs**: Este archivo contiene la configuración inicial de tu aplicación, donde se registran los servicios, se configuran las rutas de acceso y se realizan otras configuraciones necesarias.