



.Net

Teoría 13

Principio de inversión de dependencias (DIP) y Patrón de Inyección de Dependencias

Principio de inversión de dependencias

Las aplicaciones web **ASP.NET Core** hacen uso intensivo del principio de inversión de dependencias.

Sin embargo este principio es muy poderoso y deberíamos aprovecharlo en cualquier tipo de aplicación



Principio de inversión de dependencias

- El Principio de inversión de dependencias (DIP) es uno de los 5 principios SOLID.
- El principio Open/Close, que vimos cuando introdujimos el tema del polimorfismo, también es uno de los principios SOLID.
- Los principios SOLID facilitan el mantenimiento, extensión y reusabilidad del código.

Principio de inversión de dependencias

La definición original incluye dos recomendaciones:

- 1) Los módulos de alto nivel no deberían depender de los módulos de bajo nivel. Ambos deberían depender de abstracciones.
- 2) Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de abstracciones.



Principio de inversión de dependencias

Reinterpretación en POO



Los clases de alto nivel no deberían depender de clases de menor nivel sino de abstracciones (interfaces o clases abstractas)

Las clases de alto nivel son las que contienen la lógica de negocio, son las más importantes, establecen qué es y qué hace nuestra aplicación

Principio de inversión de dependencias

- Vamos a mostrar por medio de código como podemos implementar el principio de inversión de dependencias utilizando el patrón de Inyección de Dependencias.
- Primero codificaremos una solución que no cumple DIP y luego la iremos transformando gradualmente hasta que lo cumpla.
- Se trata de una aplicación que procesa un cálculo simple y registra los eventos durante su ejecución (log).



Codificando una solución sin DIP



1. Abrir una terminal del sistema operativo
2. Cambiar a la carpeta `proyectosDotnet`
3. Crear la aplicación de consola `CalculoSimple`
4. Abrir `Visual Studio Code` sobre este proyecto



Codificar la clase Logger



```
namespace CalculoSimple;
public class Logger
{
    public void Log(string mensaje)
    {
        Console.WriteLine(mensaje);
    }
}
```



Codificar la clase Calculador



```
namespace CalculoSimple;
class Calculador
{
    Logger _logger = new Logger();
    public void Calcular(int n)
    {
        int resul = (n + 5) * (n + 7);
        _logger.Log($"Fin de Calculo - (resul={resul})");
    }
}
```



Codificar Program.cs de la siguiente manera y ejecutar



```
-----Program.cs-----  
using CalculoSimple;  
Calculador calc = new Calculador();  
calc.Calcular(3);
```

Principio de inversión de dependencias

-----Program.cs-----

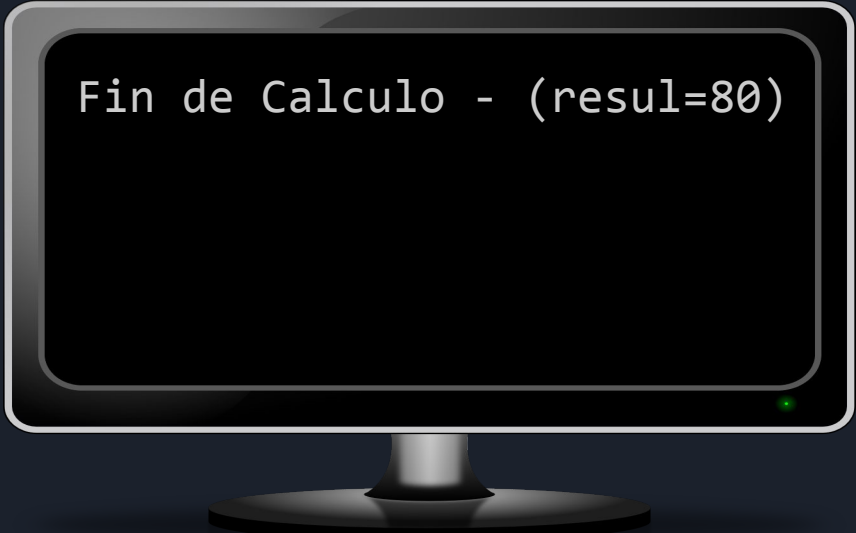
```
using CalculoSimple;  
Calculador calc = new Calculador();  
calc.Calcular(3);
```

-----Calculador.cs-----

```
namespace CalculoSimple;  
class Calculador  
{  
    Logger _logger = new Logger();  
    public void Calcular(int n)  
    {  
        int resul = (n + 5) * (n + 7);  
        _logger.Log($"Fin de Calculo - (resul={resul})");  
    }  
}
```

-----Logger.cs-----

```
namespace CalculoSimple;  
class Logger  
{  
    public void Log(string mensaje)  
    {  
        Console.WriteLine(mensaje);  
    }  
}
```



Fin de Calculo - (resul=80)

Relación entre las clases de la solución anterior

Un objeto **Calculador** usa a un objeto **Logger** invocando un método público de este último

Podemos considerar que la clase **Logger** provee un servicio y que la clase **Calculador** lo consume



Relación entre las clases de la solución anterior



- **Calculador** es cliente de **Logger**
- **Calculador** depende de **Logger**
- Pero **Calculador** es una clase de alto nivel (implementa lógica de negocio) y **Logger** es una clase de bajo nivel

“No se cumple con DIP”



Análisis de Dependencias

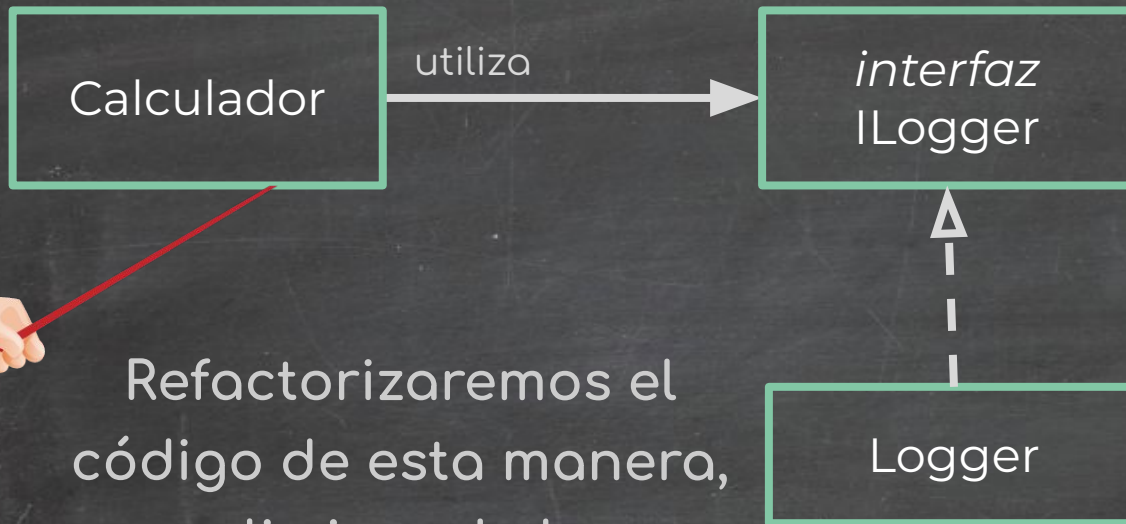
El análisis de las dependencias es estático, se hace sin correr el programa, analizando el código.

La mayoría de las clases que implementemos tendrán dependencias con tipos de la plataforma como `string`, `Array`, `List<T>`, `int`, `double`, `char`, etc. Esto no es problemático, esos tipos son muy estables y además no los modificamos nosotros

El análisis de dependencias que debemos realizar es entre los tipos definidos por nosotros



Refactorización de la solución anterior



Refactorizaremos el código de esta manera, eliminando la dependencia de **Calculador** con **Logger**



Codificar la interfaz ILogger e indicar que la clase Logger implementa esta interfaz



```
-----ILogger.cs-----
```

```
namespace CalculoSimple;
```

```
interface ILogger
```

```
{
```

```
    void Log(string mensaje);
```

```
}
```

```
-----Logger.cs-----
```

```
namespace CalculoSimple;
```

```
class Logger : ILogger
```

```
{
```

```
    public void Log(string mensaje)
```

```
    {
```

```
        Console.WriteLine(mensaje);
```

```
    }
```

```
}
```



En la clase Calculador declarar la variable `_logger` de tipo `ILogger` y ejecutar



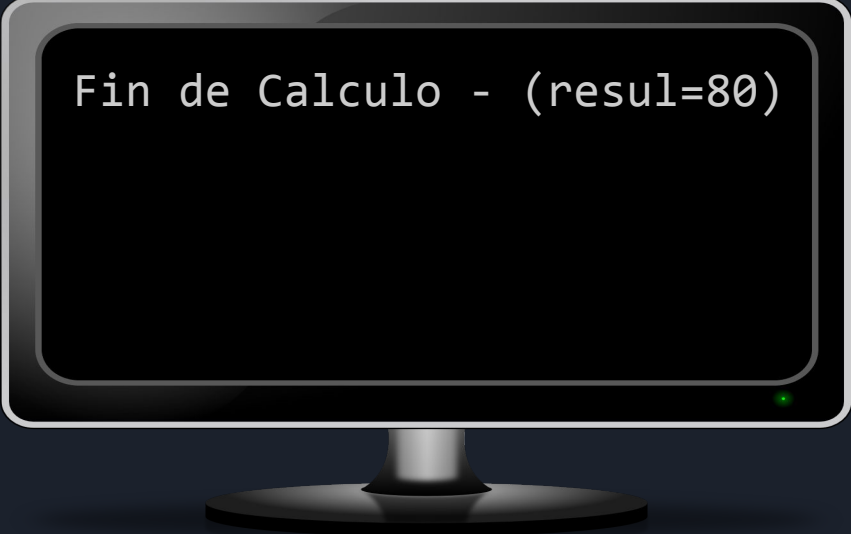
```
namespace CalculoSimple;
class Calculador
{
    ILogger _logger = new Logger();
    public void Calcular(int n)
    {
        int resul = (n + 5) * (n + 7);
        _logger.Log($"Fin de Calculo - (resul={resul})");
    }
}
```

Principio de inversión de dependencias

```
namespace CalculoSimple;
class Calculador
{
    ILogger _logger = new Logger();
    public void Calcular(int n)
    {
        int resul = (n + 5) * (n + 7);
        _logger.Log($"Fin de Calculo - (resul={resul})");
    }
}
```

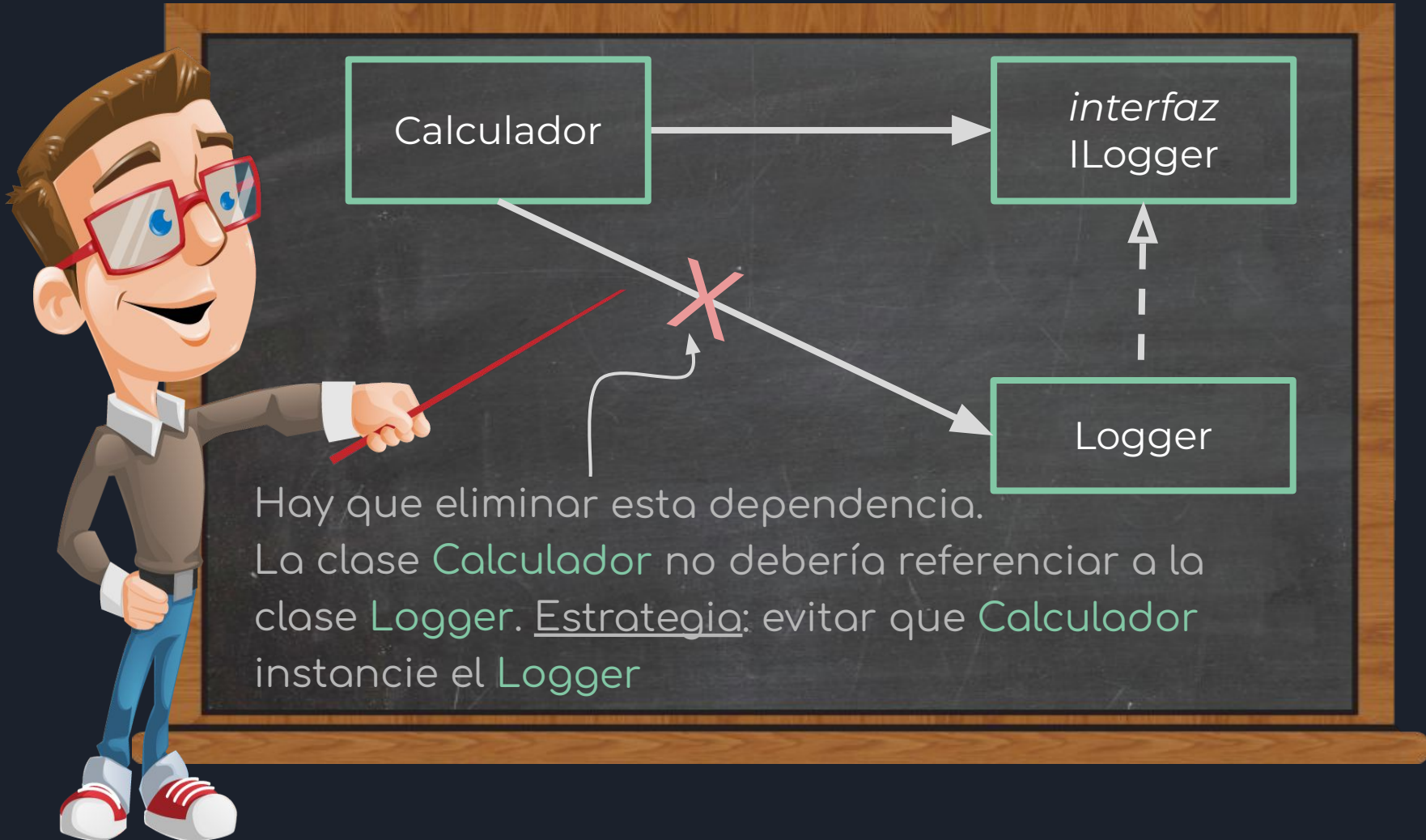
La aplicación está funcionando correctamente pero todavía no eliminamos la dependencia de `Calculador` con `Logger`

Al crear esta instancia se está referenciando directamente a la clase concreta `Logger`, por lo tanto `Calculador` ahora depende de `ILogger` y de `Logger`



Fin de Calculo - (resul=80)

Refactorización de la solución anterior



Principio de inversión de dependencias

- Para evitar que la clase **Calculador** cree una instancia de la clase **Logger** vamos a utilizar el patrón conocido como **Inyección de Dependencias**
- Otra clase debe hacerse responsable de crear la instancia requerida e inyectarla en **Calculador**
- La inyección podría hacerse de varias maneras (por medio de **método**, **propiedad** o **constructor**)
- Vamos a utilizar la **inyección por constructor**



Modificar la clase Calculador (agregar constructor)



```
namespace CalculoSimple;
class Calculador
{
    ILogger _logger;
    public Calculador(ILogger logger)
    {
        _logger = logger;
    }
    public void Calcular(int n)
    {
        int resul = (n + 5) * (n + 7);
        _logger.Log($"Fin de Calculo - (resul={resul})");
    }
}
```

Inyección de dependencia, en este caso una instancia de cualquier clase que implemente `ILogger`



Modificar el método Main de la clase Program y ejecutar



```
using CalculoSimple;
```

```
ILogger logger = new Logger();
```

```
Calculador calc = new Calculador(logger);
```

```
calc.Calcular(3);
```

Inyección de
dependencia

Principio de inversión de dependencias

-----Program.cs-----

```
using CalculoSimple;
ILogger logger = new Logger();
Calculador calc = new Calculador(logger);
calc.Calcular(3);
```

-----Calculador.cs-----

```
namespace CalculoSimple;
class Calculador
{
    ILogger _logger;
    public Calculador(ILogger logger)
    {
        _logger = logger;
    }
    public void Calcular(int n)
    {
        int resul = (n + 5) * (n + 7);
        _logger.Log($"Fin de Calculo - (resul={resul})");
    }
}
```

-----ILogger.cs-----

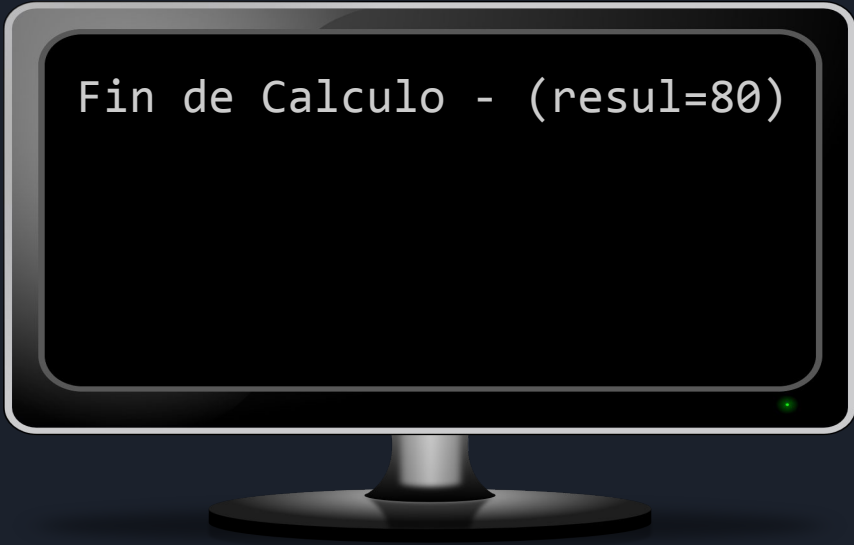
```
namespace CalculoSimple;
public interface ILogger
{
    void Log(string mensaje);
}
```

-----Logger.cs-----

```
namespace CalculoSimple;
public class Logger : ILogger
{
    public void Log(string mensaje)
    {
        Console.WriteLine(mensaje);
    }
}
```

Hemos desacoplado completamente las clases **Calculador** y la clase **Logger**.

Ahora resulta fácil inyectar en **Calculador** una instancia de otra clase que implemente **ILogger**. **Calculador** no lo advertirá, seguirá funcionando convenientemente sin requerir ninguna modificación (**principio Open/Close**)



Fin de Calculo - (resul=80)

La clase que contiene a Main es de bajo nivel



- La clase **Program** se consideran una clase de muy **bajo nivel**.
- Es el **punto de entrada** inicial del sistema. Nada, salvo el sistema operativo, depende de ella. Por lo tanto se toleran las dependencias con otras clases de la aplicación sin afectar el principio de inversión de dependencias.
- En esta clase podemos establecer las configuraciones iniciales y luego entregar el control a módulos abstractos de alto nivel.



Principio de inversión de dependencias

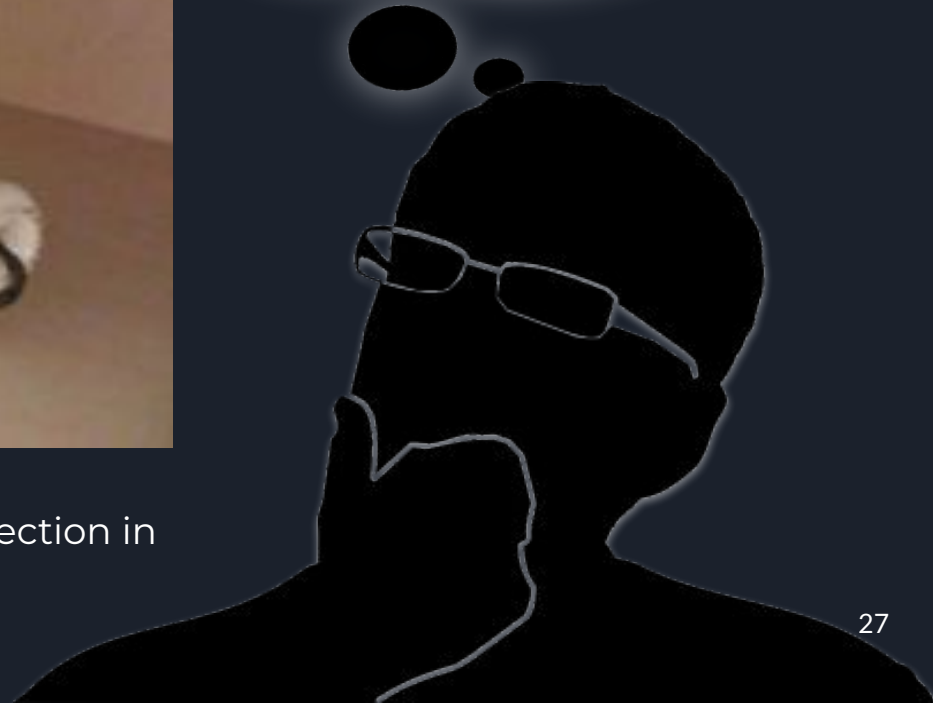
- El Principio de inversión de dependencias favorece el acoplamiento débil en el código
- El acoplamiento es el grado de dependencia que existe entre los módulos (clases o estructuras en POO)
- El acoplamiento débil hace más fácil la modificación o reemplazo de un módulo por otro

Diseño fuertemente acoplado Una muy mala idea

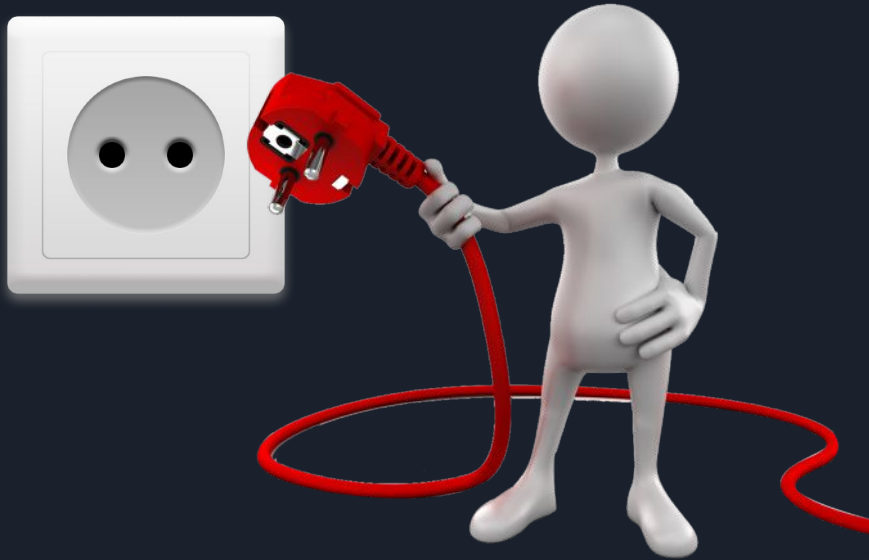


Imagen extraída del libro “Dependency Injection in .NET” de Mark Seemann

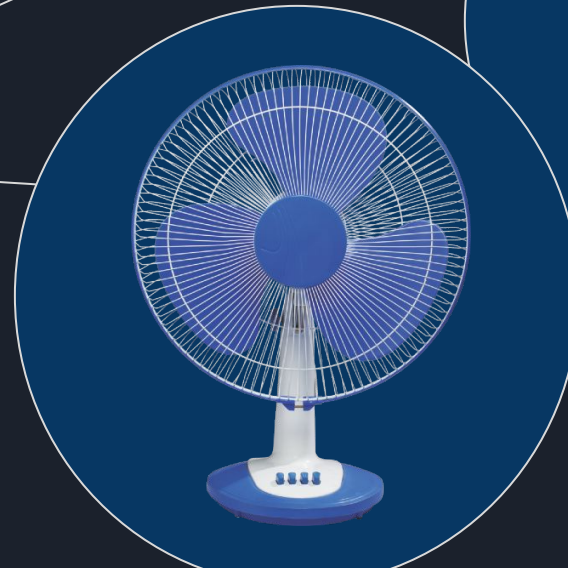
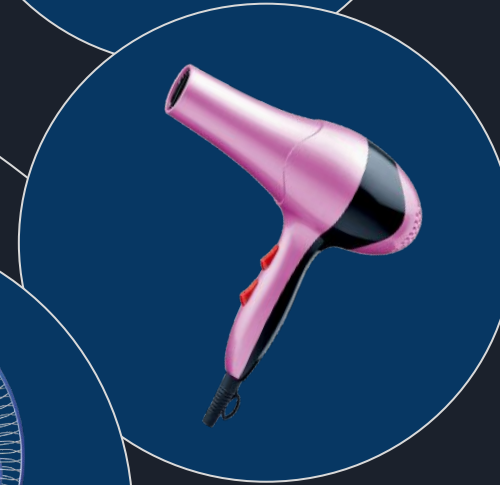
¿Y si queremos usar otra cosa distinta de este secador de pelo ?



Diseño débilmente acoplado



Con una interfaz adecuada resulta fácil reemplazar el secador por otro secador o algo distinto, sólo hace falta que posea la misma ficha para poder conectarlo al tomacorriente



Principio de inversión de dependencias

- Al igual que en una pared podemos conectar distintos dispositivos eléctricos por medio del tomacorriente (interfaz), a la clase `Calculador` podemos conectarle (inyectando) distintas clases que implementen la interfaz `ILogger`
- Por ejemplo podríamos querer que los mensajes de log se graben en un archivo en el disco (la consola no siempre está disponible, por ejemplo en una aplicación de escritorio `WinForm` o `WPF`)

Principio de inversión de dependencias

Alcanza con agregar la clase `LoggerArchivo`

```
class LoggerArchivo : ILogger
{
    public void Log(string mensaje)
    {
        using var sw = new StreamWriter("registro.log", true);
        sw.WriteLine($"{DateTime.Now} {mensaje}");
    }
}
```

Y configurar su uso adecuadamente en `Programs.cs`

```
using CalculoSimple;
ILogger logger = new LoggerArchivo();
Calculador calc = new Calculador(logger);
calc.Calcular(3);
```

Principio de inversión de dependencias

Alcanza con agregar la clase `Logger`

```
class LoggerArchivo : ILogger
{
    public void Log(string mensaje)
    {
        using var sw = new StreamWriter("log.txt");
        sw.WriteLine($"{DateTime.Now} {mensaje}");
    }
}
```

Esta es la única
modificación de código
que se realizó
Todo lo demás es
código nuevo

Y configurar su uso adecuadamente en

```
using CalculoSimple;
ILogger logger = new LoggerArchivo();
Calculador calc = new Calculador(logger);
calc.Calcular(3);
```



Principio de inversión de dependencias

Aplicar **DIP** hace que el código sea mantenible. Los programas pequeños, como el anterior son inherentemente mantenibles, por ello aplicar **DIP** en ejemplos simples tiende a parecer una ingeniería excesiva.

Cuanto mayor sea el tamaño del código, más visibles serán los beneficios de **DIP**





Principio de inversión de dependencias Configuración de las dependencias

- Idealmente para cambiar el comportamiento de la aplicación deberíamos:
 - 1) Crear nuevas clases (dependencias) que implementen determinadas interfaces
 - 2) Configurar adecuadamente la elección de las dependencias que se utilizarán
- Es conveniente agrupar la configuración de las dependencias en el código para facilitar cualquier cambio que se requiera.



Principio de inversión de dependencias Configuración de las dependencias

- Para concentrar en nuestro código la configuración de las dependencias podemos delegar en una única clase la creación de las instancias de todas las dependencias.
- Como las dependencias pueden ser consideradas servicios, vamos a llamar a esa clase **ProveedorServicios**
- Vamos a considerar a **Calculador** también una dependencia para mostrar que es conveniente centralizar la creación de todas las dependencias



Agregar la clase ProveedorServicios y la interfaz ICalculador



```
-----ProveedorServicios.cs-----  
namespace CalculoSimple;  
class ProveedorServicios  
{  
    public ILogger GetLogger()  
        => new Logger();  
    public ICalculador GetCalculador()  
        => new Calculador(this.GetLogger());  
}
```

```
-----ICalculador.cs-----  
namespace CalculoSimple;  
interface ICalculador  
{  
    void Calcular(int n);  
}
```

```
-----Calculador.cs-----  
namespace CalculoSimple;  
class Calculador : ICalculador  
{  
    ...  
}
```

Área concentrada del
código dónde se
configuran las
dependencias

ProveedorServicios concentra la creación de todas las dependencias

```
class ProveedorServicios
{
    public ILogger GetLogger()
        => new Logger();
    public ICalculador GetCalculador()
        => new Calculador(this.GetLogger());
}
```

Para crear un **Calculador** se necesita un **Logger**. Esto no es problema para **ProveedorServicios** pues también puede autoproporcionárselo





Modificar Program.cs y ejecutar



```
using CalculoSimple;  
var proveedor = new ProveedorServicios();  
ICalculador calc = proveedor.GetCalculador();  
calc.Calcular(3);
```



Contenedor de Inyección de Dependencias

- En lugar de la clase `ProveedorServicios` utilizada en el ejemplo anterior, usaremos un `contenedor de inyección de dependencias`.
- Un contenedor de inyección de dependencias (`DI Container`) facilita configurar y obtener las dependencias que se usarán en la aplicación. También permite especificar si una dependencia debe usarse como `singleton` o debe crearse un nuevo objeto cada vez que se utilice

Contenedor de Inyección de Dependencias

Con “**Singleton**” nos referimos a una clase de la cual se va a instanciar un único objeto, por lo tanto la aplicación trabajará siempre con la misma instancia en cualquier parte del código.

Singleton es también un patrón de diseño





Contenedor de Inyección de Dependencias

- .Net provee un contenedor de inyección de dependencias por medio de las clases `ServiceCollection` y `ServiceProvider`
- Para utilizar estas clases en una aplicación de consola es necesario instalar un paquete `NuGet`
- `NuGet` es un administrador de paquetes gratuito y de código abierto diseñado para .Net



Contenedor de Inyección de Dependencias



- En la terminal del sistema operativo (o en la que provee el **Visual Studio Code**) posicionarse en la carpeta del proyecto (donde se encuentra el archivo **CalculoSimple.csproj**) y tipear el siguiente comando:

```
dotnet add package Microsoft.Extensions.Hosting
```

Este es el nombre del paquete
que se requiere instalar



Modificar Program.cs de la siguiente manera y ejecutar



```
using Microsoft.Extensions.DependencyInjection;
using CalculoSimple;
var servicios = new ServiceCollection();
servicios.AddTransient<ICalculador, Calculador>();
servicios.AddTransient<ILogger, Logger>();
var proveedor = servicios.BuildServiceProvider();
var calc = proveedor.GetService<ICalculador>();
calc?.Calcular(3);
```

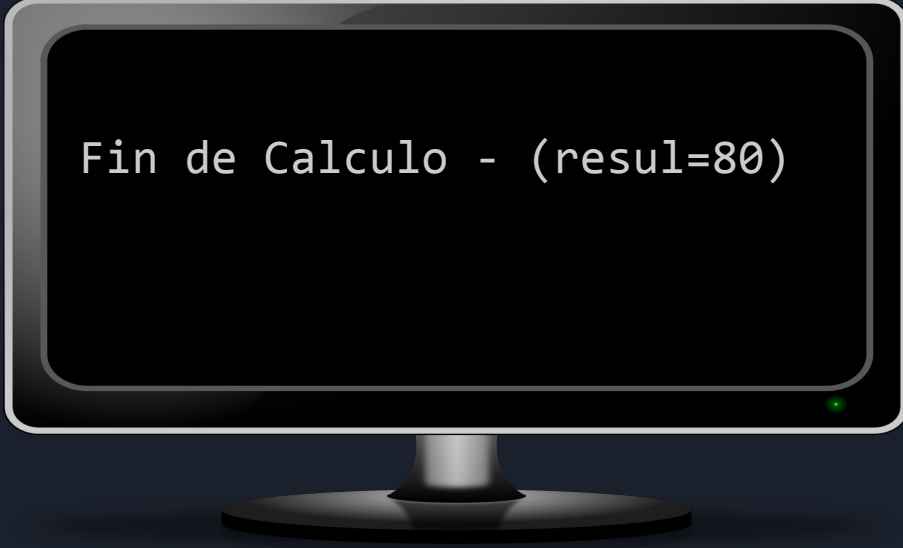
Agregar esta
directiva using

Se registran los
servicios y se construye
el proveedor

La clase
ProveedorServicios
ya no es necesaria

-----Program.cs-----

```
using Microsoft.Extensions.DependencyInjection;
using CalculoSimple;
var servicios = new ServiceCollection();
servicios.AddTransient<ICalculador, Calculador>();
servicios.AddTransient<ILogger, Logger>();
var proveedor = servicios.BuildServiceProvider();
var calc = proveedor.GetService<ICalculador>();
calc?.Calcular(3);
```



Fin de Calculo - (resul=80)

Contenedor de Inyección de Dependencias descripción del código presentado

```
servicios.AddTransient<ICalculador, Calculador>();  
servicios.AddTransient<ILogger, Logger>();
```

Se registran los servicio `ICalculador` y `ILogger` en la colección de servicios, indicando que cuando se requiera un `ICalculador` debe proveerse una nueva instancia de la clase `Calculador` y cuando se requiera un `ILogger` debe proveerse una nueva instancia de la clase `Logger`

Contenedor de Inyección de Dependencias descripción del código presentado

```
var proveedor = servicios.BuildServiceProvider();  
  
var calc = proveedor.GetService<ICalculador>();
```

Se obtiene el proveedor de servicios a partir de la colección de servicios.

Se instancia y devuelve un objeto de la clase `Calculador`. No debemos preocuparnos por las dependencias que requiere `Calculador`, serán provistas por el contenedor

Contenedor de Inyección de Dependencias

Para que el **contenedor** pueda proveer los servicios requeridos se necesita:

1. Haber registrado el servicio y todas sus dependencias en el contenedor.
2. Utilizar en todos los casos inyección por medio del constructor.



Tiempo de vida de los servicios en un contenedor

- `servicios.AddTransient<ILogger, Logger>();`
Registra el servicio `Logger` como transitorio. El proveedor devolverá un nuevo objeto cada vez que se lo requiera.
- `servicios.AddSingleton<ILogger, Logger>();`
Registra el servicio `Logger` como singleton. El proveedor devolverá siempre el mismo objeto cada vez que se lo requiera.



Ejercicio práctico: Se requiere numerar las líneas de log



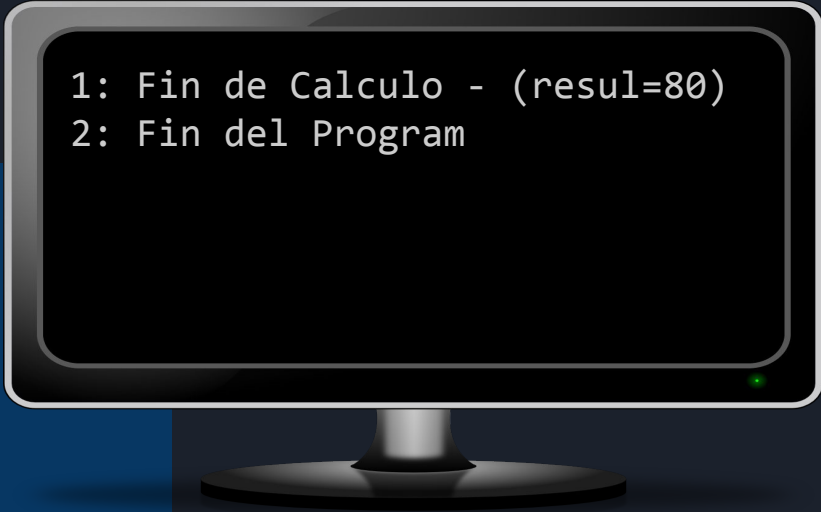
- Agregar un nuevo servicio **LoggerNum** que implemente la interfaz **ILogger** y que enumere las líneas que va imprimiendo en la consola
- Usar este nuevo servicio registrándolo como **singleton** para poder compartir la misma instancia en todo el programa
- En el método **Main**, luego de realizar el cálculo requerido, usar el servicio para imprimir un mensaje de Log con el string “Fin del programa”


```
class Program
{
    static void Main(string[] args)
    {
        var servicios = new ServiceCollection();
        servicios.AddTransient<ICalculador, Calculador>();
        servicios.AddSingleton<ILogger, LoggerNum>();
        var proveedorDeServicios
            = servicios.BuildServiceProvider();

        ICalculador calc = proveedorDeServicios.GetService<ICalculador>();
        calc.Calcular(3);
        ILogger logger = proveedorDeServicios.GetService<ILogger>();
        logger.Log("Fin del Program");
    }
}
```

Se necesita siempre acceder a la misma instancia del servicio de log, por eso lo registramos como Singleton

```
class LoggerNum : ILogger
{
    private int _n;
    public void Log(string mensaje)
    {
        Console.WriteLine($"{++_n}: {mensaje}");
    }
}
```



```
1: Fin de Calculo - (resul=80)
2: Fin del Program
```

Contenedor de Inyección de Dependencias

Hemos cambiado el comportamiento del programa agregando nuevo código y modificando sólo el área donde se registran los servicios

ii Principio OPEN/CLOSE !!



Tiempo de vida de los servicios en un contenedor de DI

- Los servicios también se pueden registrar dentro de un scope (alcance o ámbito). Resulta útil en las aplicaciones web [ASP. NET Core](#)

```
servicios.AddScoped<IservicioA, ServicioA>();
```

- Se devuelve la misma instancia dentro del mismo ámbito. Para una aplicación Blazor Server se crea un ámbito por cada conexión SignalR, Las instancias se compartirán entre páginas y componentes para un mismo usuario, pero no entre diferentes usuarios y no entre diferentes pestañas del mismo explorador.

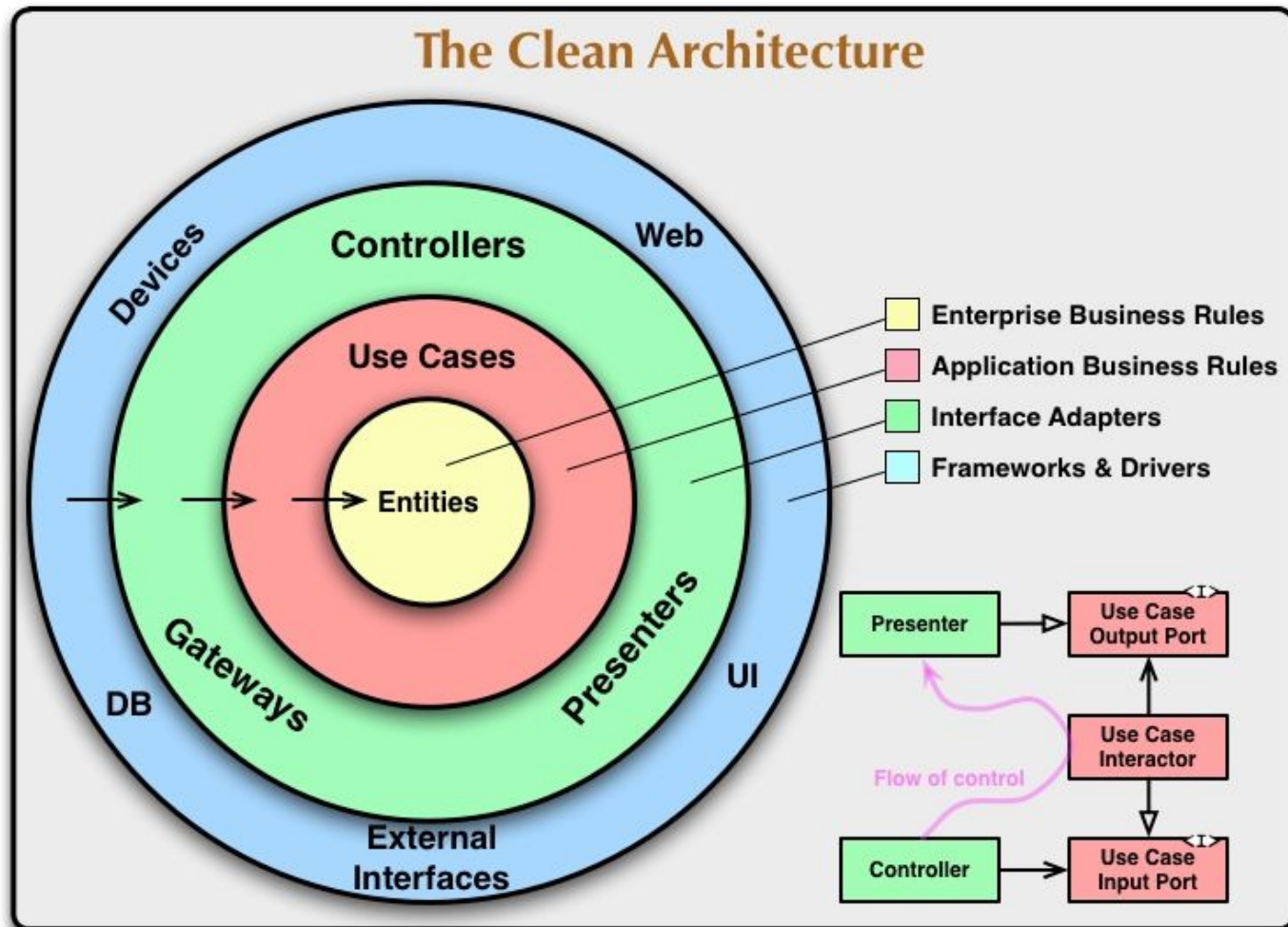
Arquitectura Limpia

Para que nuestras
aplicaciones puedan escalar
fácilmente



Arquitectura limpia

- Se refiere a organizar el proyecto para que sea fácil de entender y cambiar a medida que el proyecto crece.
- En los últimos años han aparecido varias ideas con respecto a la arquitectura como **Arquitectura Hexagonal** y **Arquitectura de Cebolla**.
- En su blog, Robert C. Martin (el tío Bob) expone una idea general de una Arquitectura Limpia



<https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>

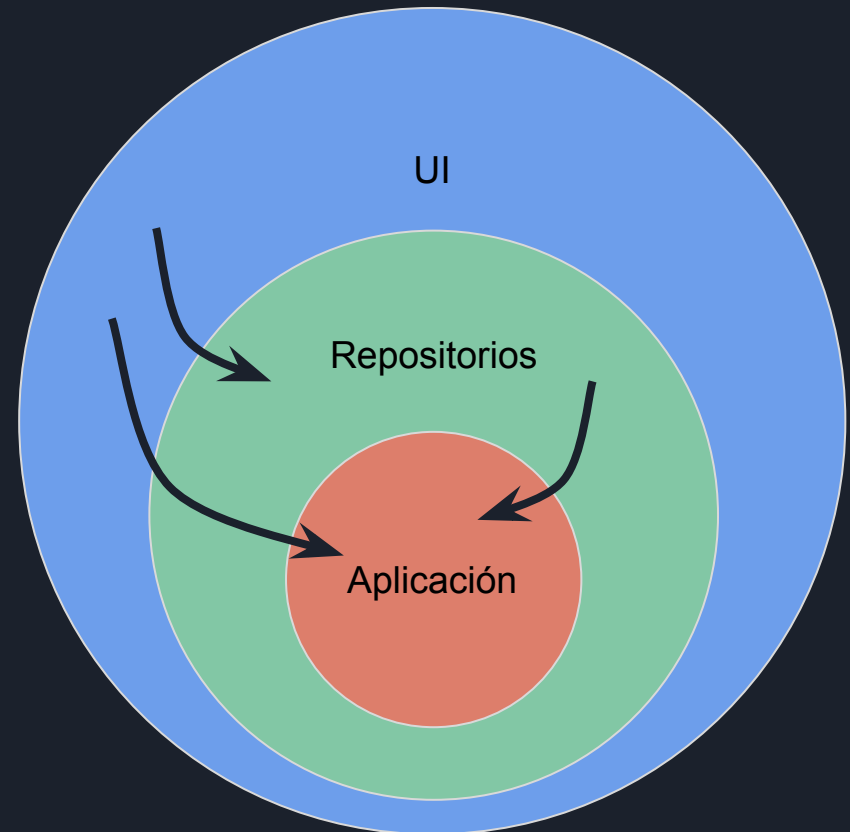
Separa los elementos de un diseño en niveles de anillo. Los anillos externos dependen de los internos y nunca al revés. El código en las capas internas no puede tener conocimiento de las funciones en las capas externas.

Arquitectura limpia

Para trabajar en este curso proponemos la siguiente versión simplificada

Aplicación y Repositorios serán proyectos de biblioteca de clases

UI será un proyecto ejecutable (por ej. una aplicación de consola o Blazor)

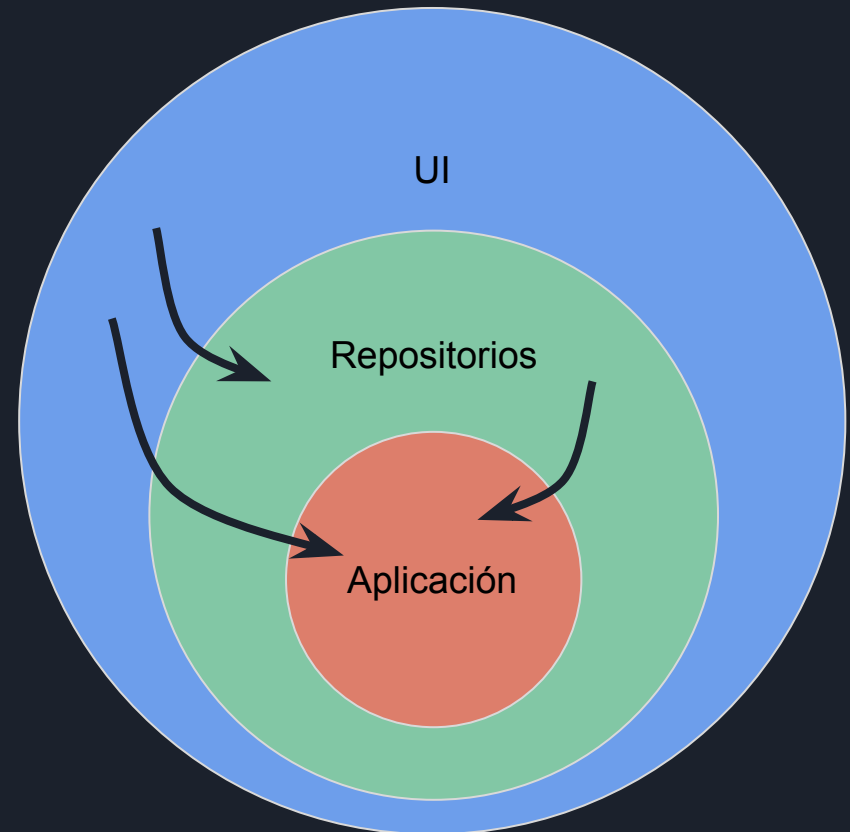


Arquitectura limpia

UI hará referencia a los
proyectos Repositorios y
Aplicación

Repositorios hará
referencia a Aplicación

Aplicación no hará
referencia a ningún otro
proyecto





Codificando una solución con diseño de Arquitectura Limpia



- Abrir una terminal del sistema operativo
- Cambiar a la carpeta `proyectosDotnet`
- Crear la solución `AL` con el siguiente comando:
`dotnet new sln -o AL`
- Cambiar a la carpeta `AL`
`cd AL`
- Crear los proyectos siguientes proyectos:
`dotnet new classlib -o AL.Aplicacion`
`dotnet new classlib -o AL.Repositorios`
`dotnet new blazorserver --no-https -o AL.UI`



Luego de ejecutar estos comando abrir Visual Studio Code (en la carpeta de la solución AL)



- Agregar los 3 proyectos a la solución

```
dotnet sln add AL.Aplicacion
dotnet sln add AL.Repositorios
dotnet sln add AL.UI
```
- Establecer las referencias entre proyectos para que `AL.UI` conozca a los otros 2 proyectos

```
dotnet add AL.UI reference AL.Aplicacion
dotnet add AL.UI reference AL.Repositorios
```
- Establecer la referencia para que `Al.Repositorios` conozca a `Al.Aplicacion`

```
dotnet add AL.Repositorios reference AL.Aplicacion
```



En el proyecto AL.Aplicacion crear la carpeta Entidades y dentro la clase Cliente



The screenshot shows the Visual Studio Code interface with the following components:

- EXPLORER:** A file tree on the left showing the project structure. The 'AL' folder is expanded, revealing subfolders: '.vscode', 'AL.Aplicacion', 'bin', 'Entidades', 'obj', 'AL.Aplicacion.csproj', 'AL.Repositorios', 'AL.UI', and 'AL.sln'. The 'Entidades' folder is selected, and 'Cliente.cs' is listed within it.
- CLIENTE.cs - AL - Visual Studio Code:** The main editor window showing the code for 'Cliente.cs'. The file path is 'AL.Aplicacion > Entidades > C# Cliente.cs > {} AL.Aplicacion.Entidades'. The code is as follows:

```
1 namespace AL.Aplicacion.Entidades;
2
3 0 references
4 public class Cliente
5 {
6     0 references
7     public int Id { get; set; }
8     0 references
9     public string Nombre { get; set; } = "";
10    0 references
11    public string Apellido { get; set; } = "";
12 }
```
- Callout Box:** A white box with a black border and a pointer to the namespace line. It contains the text: 'Establecer el namespace adecuado teniendo en cuenta el proyecto y la carpeta donde se localiza'.
- STATUS BAR:** The bottom bar shows 'Ln 1, Col 5', 'Spaces: 4', 'UTF-8', 'LF', 'C#', and icons for search, run, and other functions.



En el proyecto AL.Aplicacion crear la carpeta Interfaces y dentro la interfaz IRepositoryCliente



The screenshot shows the Visual Studio Code interface with the following details:

- Explorer Panel:** Displays the project structure. The 'AL' folder is expanded, showing subfolders: '.vscode', 'AL.Aplicacion', 'bin', 'Entidades', and 'Interfaces'. The 'Interfaces' folder is selected, and the file 'IRepositoryCliente.cs' is highlighted.
- Code Editor:** Shows the content of 'IRepositoryCliente.cs'. The file path is 'AL.Aplicacion > Interfaces > IRepositoryCliente.cs'. The code is as follows:

```
1 using AL.Aplicacion.Entidades;
2
3 namespace AL.Aplicacion.Interfaces;
4 public interface IRepositoryCliente
5 {
6     List<Cliente> GetClientes();
7     Cliente? GetCliente(int id);
8     void ModificarCliente(Cliente cliente);
9     void EliminarCliente(int id);
10    void AgregarCliente(Cliente cliente);
11 }
12
```
- Bottom Panel:** Shows the 'OUTLINE' and 'TIMELINE' views. The status bar at the bottom indicates 'Ln 1, Col 31', 'Spaces: 4', 'UTF-8', 'LF', and 'C#'.



En el proyecto AL.Aplicacion crear la carpeta UseCases y dentro la clase AgregarClienteUseCase



The screenshot shows the Visual Studio Code interface with the file explorer on the left and the code editor in the center. The file explorer shows the project structure with folders like .vscode, AL.Aplicacion, bin, Entidades, Interfaces, obj, and UseCases. The UseCases folder is expanded, showing the file AgregarClienteUseCase.cs. The code editor shows the implementation of the class, with a callout box highlighting the dependency injection.

```
using AL.Aplicacion.Entidades;
using AL.Aplicacion.Interfaces;

namespace AL.Aplicacion.UseCases;

0 references
public class AgregarClienteUseCase
{
    2 references
    private readonly IRepositoryCliente _rCliente;

    0 references
    public AgregarClienteUseCase(IRepositoryCliente rCliente)
    {
        _rCliente = rCliente;
    }

    0 references
    public void Ejecutar(Cliente cliente)
    {
        _rCliente.AgregarCliente(cliente);
    }
}
```

Inyección de dependencias!



En el proyecto AL.Aplicacion dentro de la carpeta UseCases codificar la clase ListarClientesUseCase



```
using AL.Aplicacion.Entidades;
using AL.Aplicacion.Interfaces;

namespace AL.Aplicacion.UseCases;

public class ListarClientesUseCase
{
    private readonly IRepositoryCliente _rCliente;

    public ListarClientesUseCase(IRepositoryCliente rCliente)
    {
        _rCliente = rCliente;
    }

    public List<Cliente> Ejecutar()
    {
        return _rCliente.GetClientes();
    }
}
```

Inyección de dependencias!



En el proyecto AL.Aplicacion dentro de la carpeta UseCases codificar la clase ObtenerClienteUseCase



```
using AL.Aplicacion.Entidades;
using AL.Aplicacion.Interfaces;

namespace AL.Aplicacion.UseCases;

public class ObtenerClienteUseCase
{
    private readonly IRepositoryCliente _rCliente;

    public ObtenerClienteUseCase(IRepositoryCliente rCliente)
    {
        _rCliente = rCliente;
    }

    public Cliente? Ejecutar(int id)
    {
        return _rCliente.GetCliente(id);
    }
}
```

Inyección de dependencias!



En el proyecto AL.Aplicacion dentro de la carpeta UseCases codificar la clase EliminarClienteUseCase



```
using AL.Aplicacion.Interfaces;

namespace AL.Aplicacion.UseCases;

public class EliminarClienteUseCase
{
    private readonly IRepositoryCliente _rCliente;

    public EliminarClienteUseCase(IRepositoryCliente rCliente)
    {
        _rCliente = rCliente;
    }
    public void Ejecutar(int id)
    {
        _rCliente.EliminarCliente(id);
    }
}
```

Inyección de
dependencias!



En el proyecto AL.Aplicacion dentro de la carpeta UseCases codificar la clase ModificarClienteUseCase



```
using AL.Aplicacion.Entidades;
using AL.Aplicacion.Interfaces;

namespace AL.Aplicacion.UseCases;

public class ModificarClienteUseCase
{
    private readonly IRepositoryCliente _rCliente;

    public ModificarClienteUseCase(IRepositoryCliente rCliente)
    {
        _rCliente = rCliente;
    }
    public void Ejecutar(Cliente cliente)
    {
        _rCliente.ModificarCliente(cliente);
    }
}
```

Inyección de dependencias!



En el proyecto AL.Repositorios crear la clase RepositorioClienteMock



The screenshot shows the Visual Studio Code interface with the file Explorer on the left, the Explorer view in the center, and the code editor on the right. The Explorer view shows the project structure with folders like .vscode, AL.Aplicacion, AL.Repositorios, bin, obj, AL.Repositorios.csproj, RepositorioClienteMock.cs, AL.UI, and AL.sln. The code editor shows the file RepositorioClienteMock.cs with the following code:

```
1 using AL.Aplicacion.Entidades;
2 using AL.Aplicacion.Interfaces;
3
4 namespace AL.Repositorios;
5
6 public class RepositorioClienteMock : IRepositoryCliente
7 {
8     public void AgregarCliente(Cliente cliente)
9     {
10     }
11 }
```

A callout box with a white border and a shadow points to the `IRepositoryCliente` interface in the code. The text in the callout box is:

RepositorioClienteMock debe implementar la interfaz IRepositoryCliente

Inyectaremos en la aplicación este repositorio Mock para hacer algunas pruebas. Lo llamamos Mock porque es un sustituto de algún repositorio que va a utilizar la aplicación realmente.

```
public Cliente? GetCliente(int id)
{
    Cliente? c = _listaClietes.SingleOrDefault(c => c.Id == id);
    if (c != null)
    {
        return Clonar(c);
    }
    return null;
}
public List<Cliente> GetClientes()
{
    List<Cliente> lista = new List<Cliente>();
    foreach (var c in _listaClietes)
    {
        lista.Add(Clonar(c));
    }
    return lista;
}
public void ModificarCliente(Cliente cliente)
{
    var cli = _listaClietes.SingleOrDefault(c => c.Id == cliente.Id);
    if (cli != null)
    {
        cli.Apellido = cliente.Apellido;
        cli.Nombre = cliente.Nombre;
    }
}
}
```

```
using AL.Aplicacion.Entidades;
using AL.Aplicacion.Interfaces;
namespace AL.Repositorios;
public class RepositorioClienteMock : IRepositoryCliente
{
    private readonly List<Cliente> _listaClietes = new List<Cliente>(){
        new Cliente(){Id=1,Nombre="Alberto",Apellido="García"},
        new Cliente(){Id=2,Nombre="Ana",Apellido="Perez"}
    };//hemos hardcodeado dos clientes en la lista

    private Cliente Clonar(Cliente c) //se van a devolver copias de los cliente guardados
    {
        return new Cliente() {
            Id = c.Id,
            Nombre = c.Nombre,
            Apellido = c.Apellido
        };
    }
    public void AgregarCliente(Cliente cliente) {
        cliente.Id = _listaClietes.Count() == 0 ? 1 : _listaClietes.Max(c => c.Id) + 1;
        _listaClietes.Add(cliente);
    }
    public void EliminarCliente(int id) {
        var cliente = _listaClietes.SingleOrDefault(c => c.Id == id);
        if (cliente != null)
        {
            _listaClietes.Remove(cliente);
        }
    }
}
```

Contenedor de Inyección de Dependencias en Blazor

La aplicación Blazar que usaremos como interfaz de usuario ya viene con un **DI container** integrado.

Registraremos nuestros servicios en la clase program por medio de `builder.Services` que devuelve un `IServiceCollection`





Modificar la clase Program de la aplicación AL.UI



```
using Microsoft.AspNetCore.Components;  
using Microsoft.AspNetCore.Components.Web;  
using AL.UI.Data;
```

```
//agregamos estas directivas using  
using AL.Repositorios;  
using AL.Aplicacion.UseCases;  
using AL.Aplicacion.Interfaces;
```

```
var builder = WebApplication.CreateBuilder(args);
```

```
// Add services to the container.  
builder.Services.AddRazorPages();  
builder.Services.AddServerSideBlazor();  
builder.Services.AddSingleton<WeatherForecastService>();
```

```
//agregamos estos servicios al contenedor DI  
builder.Services.AddTransient<AgregarClienteUseCase>();  
builder.Services.AddTransient<ListarClientesUseCase>();  
builder.Services.AddTransient<EliminarClienteUseCase>();  
builder.Services.AddTransient<ModificarClienteUseCase>();  
builder.Services.AddTransient<ObtenerClienteUseCase>();  
builder.Services.AddScoped<IRepositorioCliente, RepositorioClienteMock>();
```

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




Agregar las directivas @using en el archivo _imports.razor de la aplicación AL.UI



```
AL.UI > @ _Imports.razor
1 @using System.Net.Http
2 @using Microsoft.AspNetCore.Authorization
3 @using Microsoft.AspNetCore.Components.Authorization
4 @using Microsoft.AspNetCore.Components.Forms
5 @using Microsoft.AspNetCore.Components.Routing
6 @using Microsoft.AspNetCore.Components.Web
7 @using Microsoft.AspNetCore.Components.Web.Virtualization
8 @using Microsoft.JSInterop
9 @using AL.UI
10 @using AL.UI.Shared
11
12 @using AL.Aplicacion.Entidades
13 @using AL.Aplicacion.Interfaces
14 @using AL.Aplicacion.UseCases
15 @using AL.Repositorios
16
```



Codificar un componente razor llamado ListadoClientes.razor



```
@page "/listadoclientes"  
@inject ListarClientesUseCase ListarClientesUseCase
```

```
@code {  
    List<Cliente> _lista = new List<Cliente>();  
    protected override void OnInitialized()  
    {  
        _lista = ListarClientesUseCase.Ejecutar();  
    }  
}
```

Este método se invoca una vez creada la instancia de la clase que representa el componente

Con la directiva `@inject` directiva inyectamos en la propiedad `ListarClientesUseCase` un objeto de tipo `ListarClientesUseCase`



Agregar el siguiente código a ListadoClientes.razor



```
@page "/listadoclientes"
@inject ListarClientesUseCase ListarClientesUseCase

<table class="table">
  <thead>
    <tr>
      <th>ID</th>
      <th>NOMBRE</th>
      <th>APELLIDO</th>
    </tr>
  </thead>
  <tbody>
    @foreach (var cli in _lista)
    {
      <tr>
        <td>@cli.Id</td>
        <td>@cli.Nombre</td>
        <td>@cli.Apellido</td>
      </tr>
    }
  </tbody>
</table>
@code {
  ...
}
```

...




Modificar el componente NavMenu.razor que está en la carpeta Shared y ejecutar



. . .

```
<div class="nav-item px-3">  
  <NavLink class="nav-link" href="fetchdata">  
    <span class="oi oi-list-rich" aria-hidden="true"></span> Fetch data  
  </NavLink>  
</div>
```

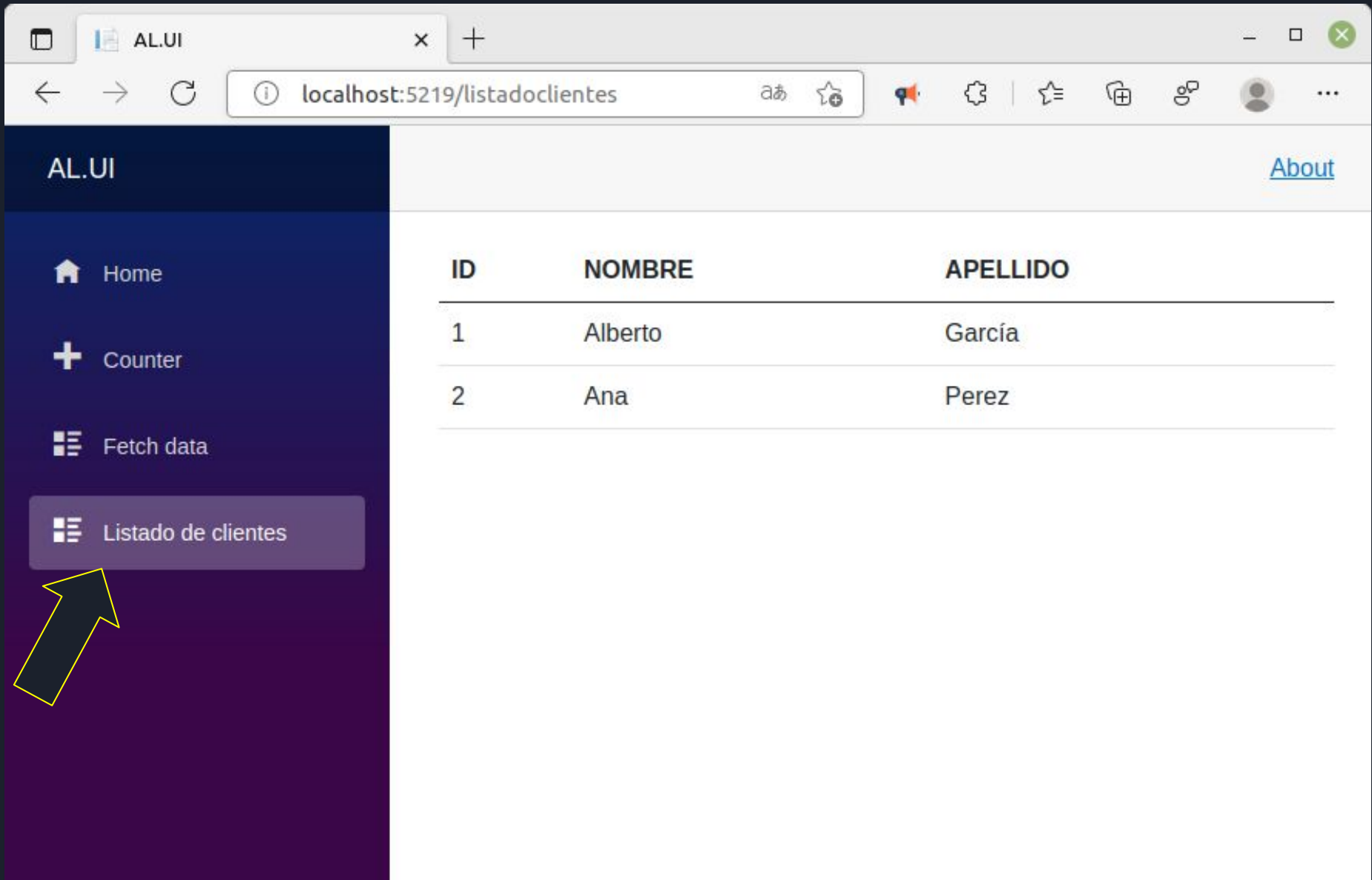


```
<div class="nav-item px-3">  
  <NavLink class="nav-link" href="/listadoclientes">  
    <span class="oi oi-list-rich" aria-hidden="true"></span> Listado de clientes  
  </NavLink>  
</div>
```

. . .

Copiar y pegar las 5 líneas que corresponden al menú “Fetch data” y luego cambiar el texto y el valor del atributo href

Arquitectura Limpia - UI con Blazor



AL.UI

[About](#)

ID	NOMBRE	APELLIDO
1	Alberto	García
2	Ana	Perez



Agregar un nuevo item en el componente NavMenu.razor



```
. . .  
<div class="nav-item px-3">  
    <NavLink class="nav-link" href="/listadoclientes">  
        <span class="oi oi-list-rich" aria-hidden="true"></span> Listado de clientes  
    </NavLink>  
</div>  
<div class="nav-item px-3">  
    <NavLink class="nav-link" href="cliente">  
        <span class="oi oi-plus" aria-hidden="true"></span> Agregar Cliente  
    </NavLink>  
</div>  
. . .
```



Codificar un componente razor llamado EditarCliente.razor. Ejecutar



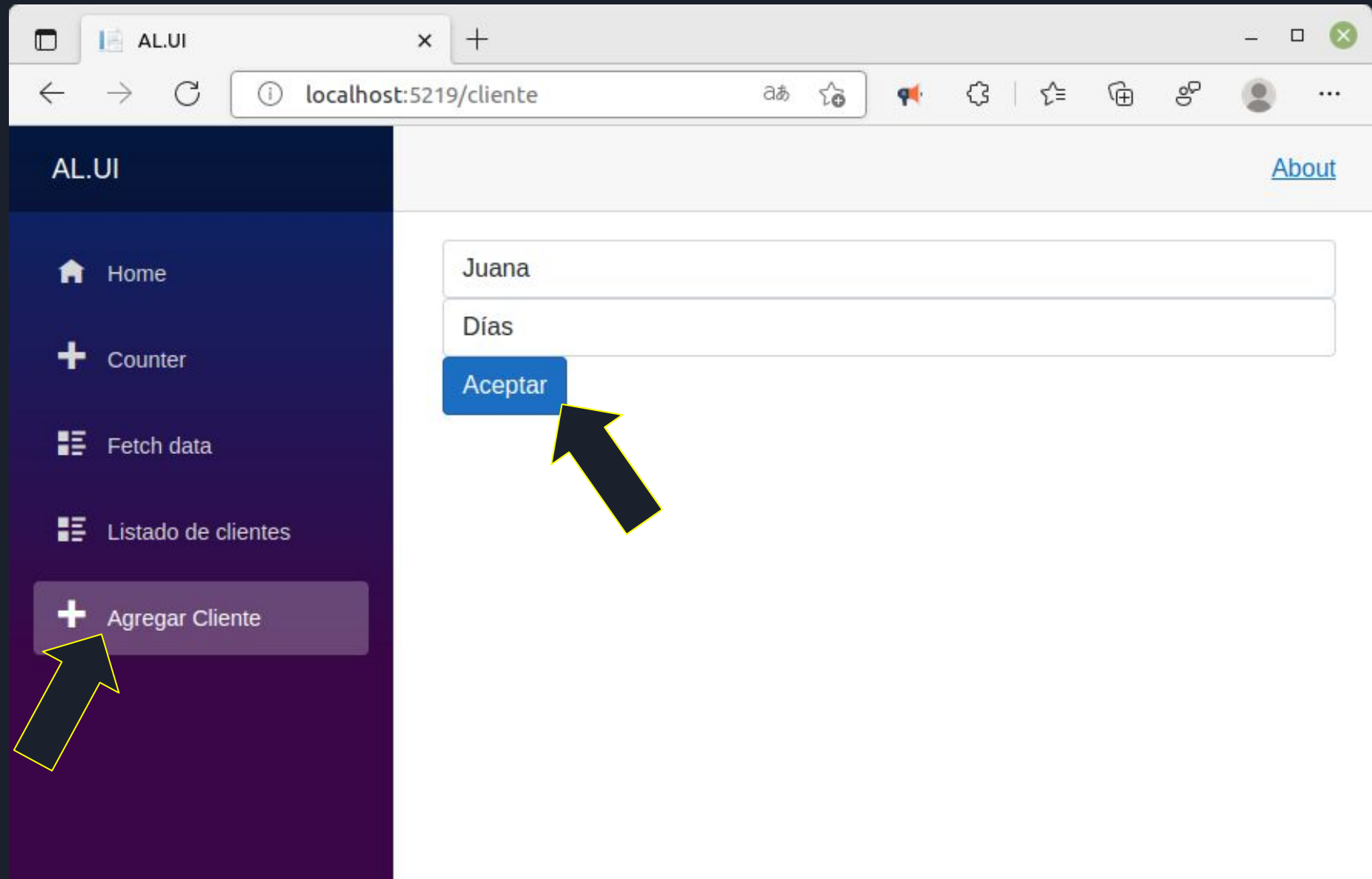
```
@page "/cliente"
@inject NavigationManager Navegador;
@inject AgregarClienteUseCase AgregarClienteUseCase

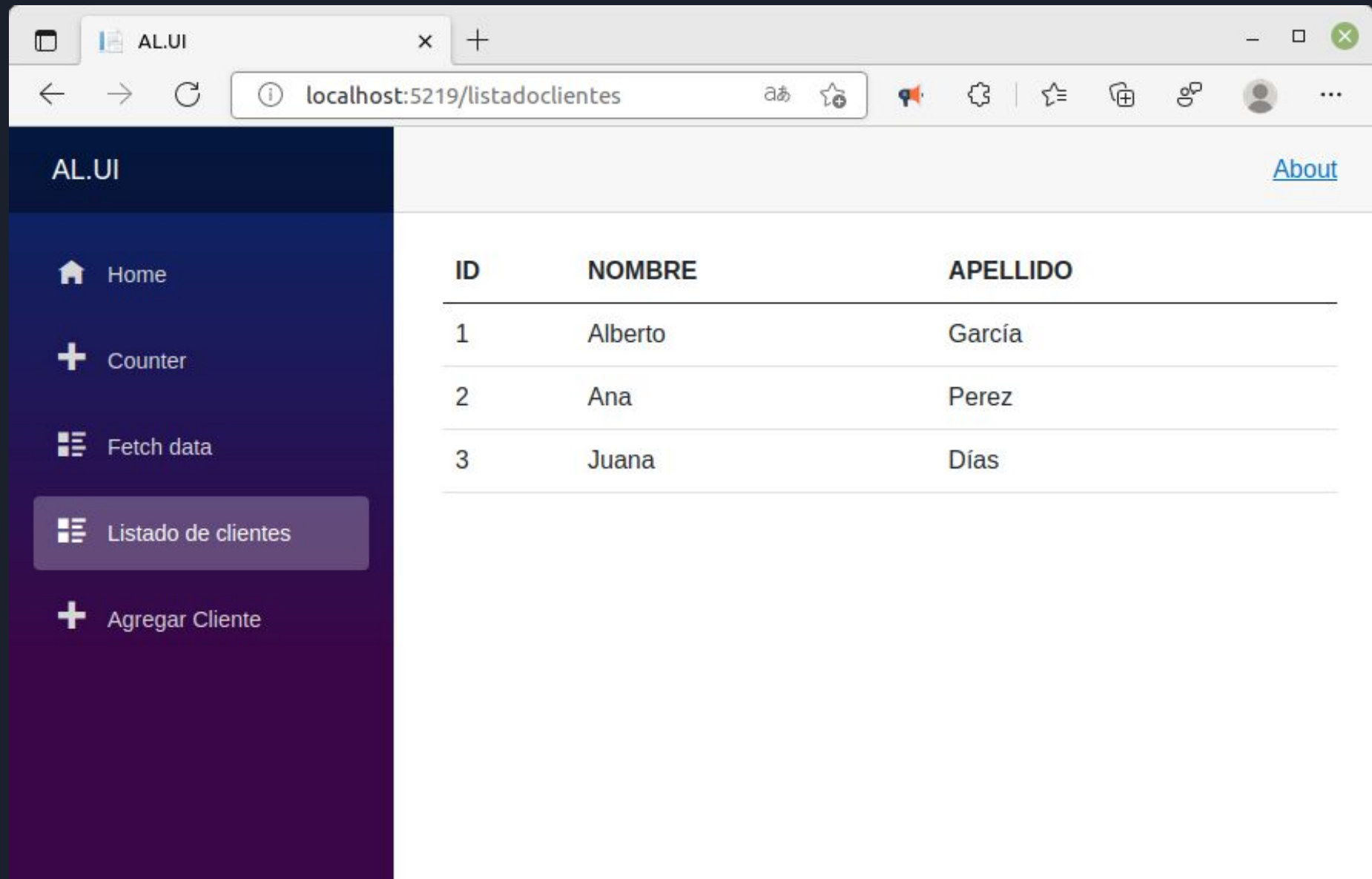
<input placeholder="Nombre" @bind="_cliente.Nombre" class="form-control">
<input placeholder="Apellido" @bind="_cliente.Apellido" class="form-control">
<button class="btn btn-primary" @onclick="Aceptar">Aceptar</button>

@code {
    Cliente _cliente = new Cliente();
    void Aceptar()
    {
        AgregarClienteUseCase.Ejecutar(_cliente);
        _cliente = new Cliente();
        Navegador.NavigateTo("listadoclientes");
    }
}
```

Inyectamos un objeto `NavigationManager` que nos permite navegar entre las páginas

Arquitectura Limpia - UI con Blazor





The screenshot shows a web browser window with the address bar displaying `localhost:5219/listadoclientes`. The application title is "AL.UI". The sidebar on the left contains the following navigation items:

- Home
- Counter
- Fetch data
- Listado de clientes (highlighted)
- Agregar Cliente

The main content area displays a table with the following data:

ID	NOMBRE	APELLIDO
1	Alberto	García
2	Ana	Perez
3	Juana	Días



Agregar el archivo EditarCliente.razor.css



Para definir estilos específicos de un componente, se debe crear un archivo `.razor.css` cuyo nombre coincida con el del archivo `.razor` del componente en la misma carpeta.


```
-----EditarCliente.razor.css-----
```


```
.form-control {  
    margin-bottom: 10px;  
}
```


Agregamos margen debajo de los elementos que tengan el atributo `class="form-control"`
Esto afecta sólo al componente `EditarCliente.razor`

AL.UI

 Home

 Counter

 Fetch data

 Listado de clientes

Nombre

Apellido

Aceptar







Modificar el componente ListadoClientes y ejecutar



```
<table class="table">
  <thead>
    <tr>
      <th>ID</th>
      <th>NOMBRE</th>
      <th>APELLIDO</th>
      <th>ACCIÓN</th>
    </tr>
  </thead>
  <tbody>
    @foreach (var cli in _lista)
    {
      <tr>
        <td>@cli.Id</td>
        <td>@cli.Nombre</td>
        <td>@cli.Apellido</td>
        <td>
          <button class="btn btn-primary">
            <i class="oi oi-pencil"></i>
          </button>
          <button class="btn btn-danger">
            <i class="oi oi-trash"></i>
          </button>
        </td>
      </tr>
    }
  </tbody>
</table>
```

AL.UI

[About](#)

ID	NOMBRE	APELLIDO	ACCIÓN
1	Alberto	García	 
2	Ana	Perez	 

Falta agregarle comportamiento a estos botones



En el componente ListadoClientes inyectar estos servicios



```
@page "/listadoclientes"  
@inject ListarClientesUseCase ListarClientesUseCase  
@inject IJSRuntime JsRuntime;  
@inject EliminarClienteUseCase EliminarClienteUseCase
```

```
<table class="table">  
  <thead>  
    <tr>  
      <th>ID</th>  
      <th>NOMBRE</th>  
      <th>APELLIDO</th>  
      <th>ACCIÓN</th>  
    </tr>  
  </thead>  
  <tbody>
```

• • •

JSRuntime nos va a permitir
invocar funciones de Javascript



En el componente ListadoClientes agregar el método EliminarCliente



```
. . .
@code {
    List<Cliente> _lista = new List<Cliente>();
    protected override void OnInitialized()
    {
        _lista = ListarClientesUseCase.Ejecutar();
    }

    async Task EliminarCliente(int id)
    {
        bool confirmado = await JsRuntime.InvokeAsync<bool>("confirm",
            $"Desea eliminar el cliente {id}?");

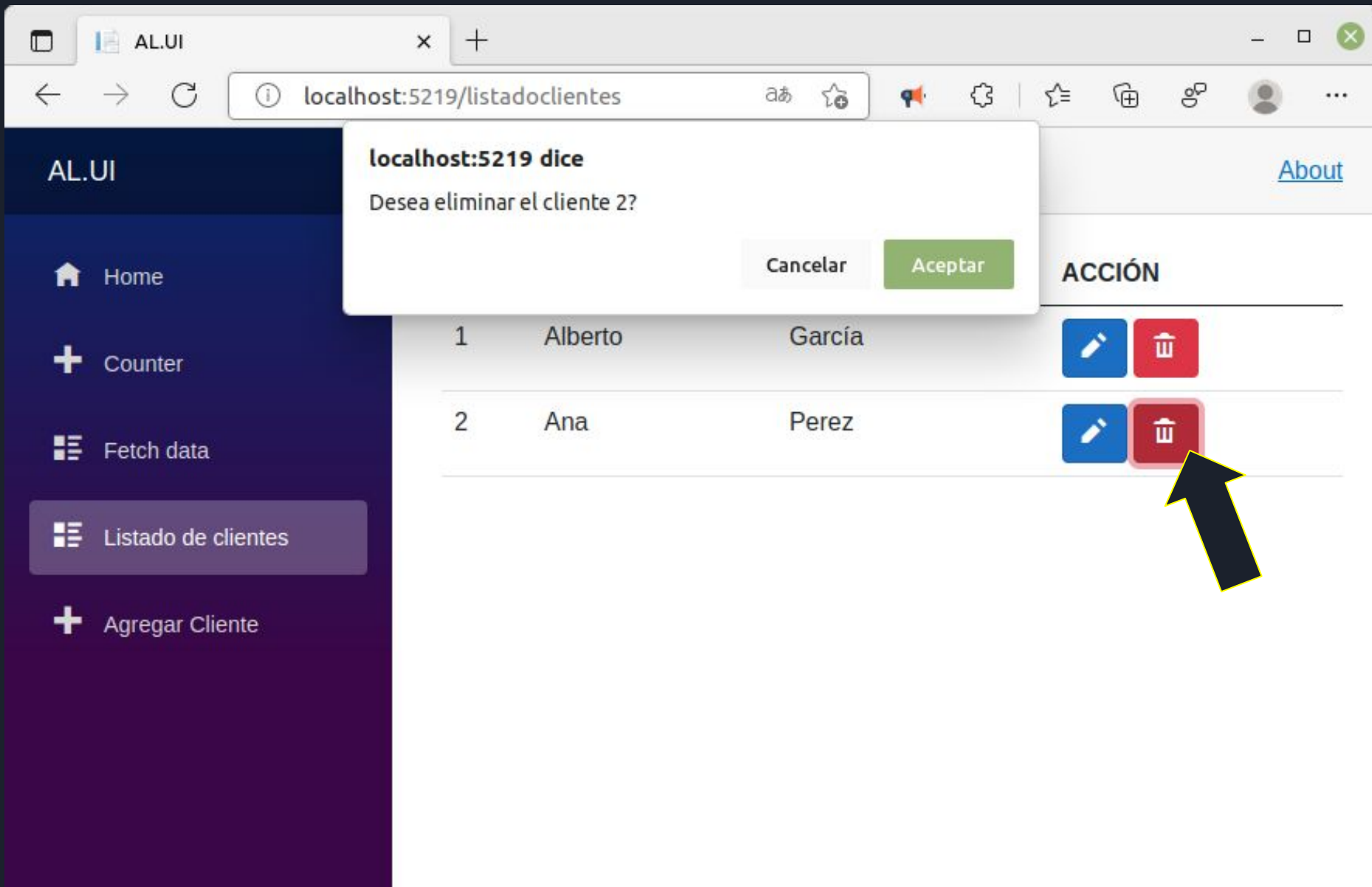
        if (confirmado)
        {
            EliminarClienteUseCase.Ejecutar(id);
            _lista = ListarClientesUseCase.Ejecutar();
        }
    }
}
```



Invocar el método EliminarCliente con una expresión lambda y ejecutar



```
. . .  
<tbody>  
    @foreach (var cli in _lista)  
    {  
        <tr>  
            <td>@cli.Id</td>  
            <td>@cli.Nombre</td>  
            <td>@cli.Apellido</td>  
            <td>  
                <button class="btn btn-primary">  
                    <i class="oi oi-pencil"></i>  
                </button>  
                <button class="btn btn-danger"  
                    @onclick="()=>EliminarCliente(cli.Id)">  
                    <i class="oi oi-trash"></i>  
                </button>  
            </td>  
        </tr>  
    }  
</tbody>  
. . .
```



¿Cómo modificar un cliente?

El componente `EditarCliente.razor` lo utilizamos para agregar nuevos clientes. Vamos a reutilizarlo también para modificar un cliente existente.

Agregaremos un `parámetro opcional` a la ruta `/cliente` para identificar al cliente que queremos modificar. En caso de que ese parámetro sea `null`, estaremos agregando un nuevo cliente tal cual lo venimos haciendo hasta ahora



Modificar el componente EditarCliente.razor



```
@page "/cliente/{Id:int?}"  
@inject ObtenerClienteUseCase ObtenerClienteUseCase  
@inject ModificarClienteUseCase ModificarClienteUseCase  
@inject NavigationManager Navegador  
@inject AgregarClienteUseCase AgregarClienteUseCase  
  
. . .
```

Inyectar los casos de uso
ObtenerClienteUseCase y
ModificarClienteUseCase

Agregar en la directiva **@page**
un parámetro de ruta.
Debe coincidir con una
propiedad pública del
componente calificada con
[Parameter]



Modificar el componente EditarCliente.razor



```
...  
@code {  
    Cliente _cliente = new Cliente();  
    [Parameter] public int? Id { get; set; }  
    bool _esNuevoCliente=true;  
    protected override void OnParametersSet()  
    {  
        if (Id != null)  
        {  
            var cli_hallado = ObtenerClienteUseCase.Ejecutar(Id.Value);  
            if (cli_hallado != null)  
            {  
                _cliente = cli_hallado;  
                _esNuevoCliente=false;  
            }  
        }  
    }  
}
```

Debe coincidir con el parámetro de ruta de la directiva `@page`

Este método se invoca cuando el componente ha recibido parámetros y los valores entrantes se han asignado a las propiedades

...



Modificar el componente EditarCliente.razor



```
. . .
@if (_esNuevoCliente)
{
    <h3>Agregando un nuevo Cliente</h3>
}
else
{
    <h3>Modificando al Cliente "@_cliente.Nombre"</h3>
}

<input placeholder="Nombre" @bind="_cliente.Nombre" class="form-control">
<input placeholder="Apellido" @bind="_cliente.Apellido" class="form-control">
<button class="btn btn-primary" @onclick="Aceptar">Aceptar</button>
. . .
```

Agregamos un título
distinto en función de lo
que se esté realizando



Modificar el método Aceptar del componente EditarCliente.razor



```
. . .  
    void Aceptar()  
    {  
        if (_esNuevoCliente)  
        {  
            AgregarClienteUseCase.Ejecutar(_cliente);  
        }  
        else  
        {  
            ModificarClienteUseCase.Ejecutar(_cliente);  
        }  
        _cliente = new Cliente();  
        Navegador.NavigateTo("listadoclientes");  
    }  
. . .
```

AL.UI

Home

Counter

Fetch data

Listado de clientes

Agregar Cliente

About

Modificando al Cliente "Ana"

Ana





Perez

Aceptar

Ya podemos modificar un cliente colocando la ruta correspondiente

AL.UI

[About](#)

ID	NOMBRE	APELLIDO	ACCIÓN
1	Alberto	García	 
2	Ana	Perez	 

Falta incorporar la navegación correspondiente a los botones de edición en el componente ListadoClientes.razor



Modificar el componente ListadoClientes.razor



```
@page "/listadoclientes"  
@inject ListarClientesUseCase ListarClientesUseCase  
@inject IJSRuntime JsRuntime;  
@inject EliminarClienteUseCase EliminarClienteUseCase  
@inject NavigationManager Navegador
```

```
<table class="table">  
  <thead>  
    <tr>  
      <th>ID</th>  
      <th>NOMBRE</th>  
      <th>APELLIDO</th>  
      <th>ACCIÓN</th>  
    </tr>  
  </thead>  
  . . .
```



Modificar el componente ListadoClientes.razor



```
. . .  
@code {  
  
    . . .  
  
    void ModificarCliente(int id)  
    {  
        Navegador.NavigateTo($"cliente/{id}");  
    }  
  
}
```




Modificar el componente ListadoClientes.razor







. . .

```
@foreach (var cli in _lista)
{
    <tr>
        <td>@cli.Id</td>
        <td>@cli.Nombre</td>
        <td>@cli.Apellido</td>
        <td>
            <button class="btn btn-primary"
                @onclick="()=>ModificarCliente(cli.Id)">
                <i class="oi oi-pencil"></i>
            </button>
            <button class="btn btn-danger"
                @onclick="()=>EliminarCliente(cli.Id)">
                <i class="oi oi-trash"></i>
            </button>
        </td>
    </tr>
}
```

. . .

AL.UI

[About](#)

ID	NOMBRE	APELLIDO	ACCIÓN
1	Alberto	García	 
2	Ana	Perez	 

Ya podemos acceder al componente para modificar un cliente desde los botones correspondientes

Fin teoría 13