Inyección de dependencia

La tecnología que vamos a estar usando, la cual es **ASP.NET Core** soporta el patron de diseño inyección de dependencia. La cual es una **técnica** para lograr la i**nversion de control** o el **principio de inversion de dependencia** entre clases y sus dependencias.

Qué es una dependencia?

Una dependencia es un ***objeto*** que otro **objeto** necesita, depende de el. Cuando hablamos de objeto nos podemos referir a componentes, librerías, módulos, clases, funciones, entre otras.

El **concepto** de dependencia esta arraigado a cuando **usas.** En la notación UML es una relación de uso, por ejemplo cliente-servidor.

A nivel de clase se da cuando una clase **A** utiliza otra clase **B,** sin que esta forme parte del estado de **A**. Esto lo podemos ver cuando la clase **A** necesita realizar una lógica que es propia de la clase **B** entonces se da el uso de **B** en **A.**

Se puede dar cuando:

- **A** recibe por parámetro un objeto de tipo **B.**
- Un método de **A** devuelve un objeto de tipo **B.**
- Un método de **A** menciona un objeto de tipo **B.**

Cuando de decimos ***objeto de tipo B*** nos estamos refiriendo a una ***instancia de tipo B.***

Analicemos esto con algunos diagramas:

Diagrama de clases

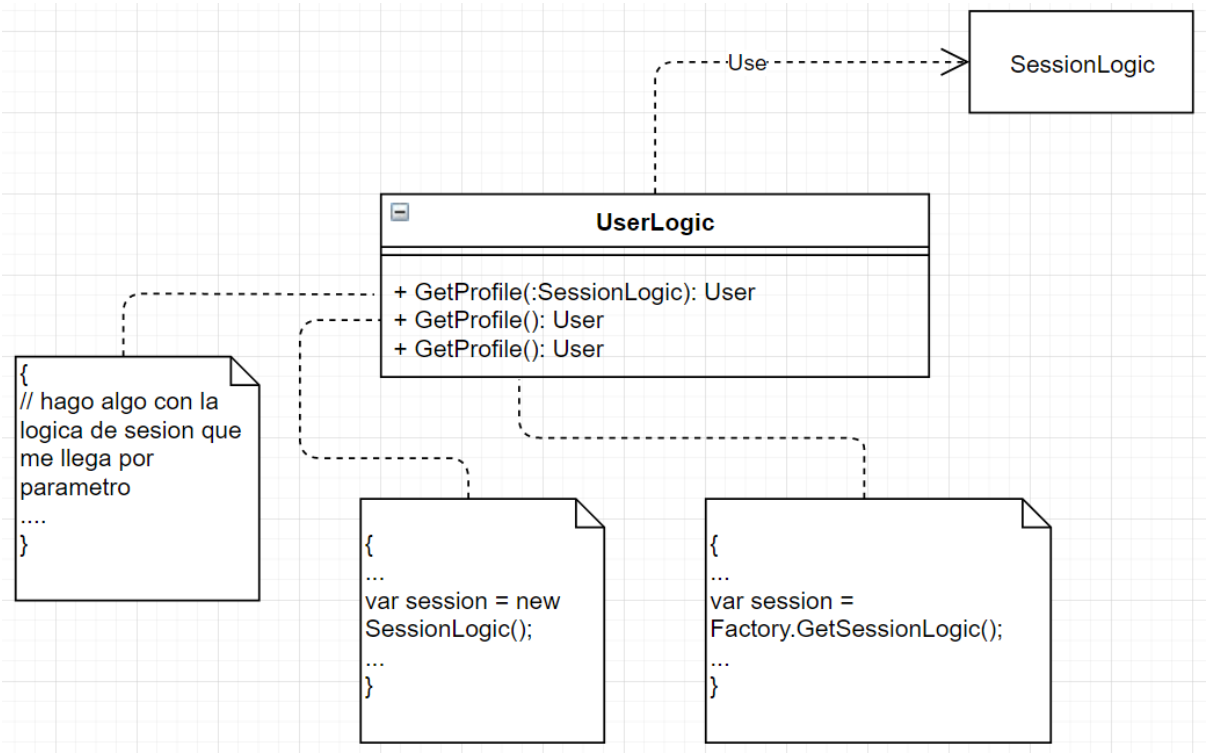
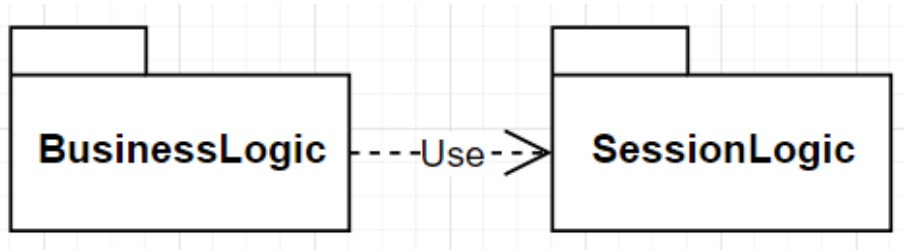


Diagrama de paquetes



Chequeemos el siguiente ejemplo mas genérico:

```
public class MyDependencyRepository
{
```

```
    public void GetData(int identifier)
    {
        Console.WriteLine($"MyDependency.GetData called. Message: {identifier}");
    }
}
```

```
public class BusinessLogic
{
    private readonly MyDependencyRepository _dependency = new MyDependencyRepository();

    public void OnGet()
    {
        _dependency.GetData(5);
    }
}
```

`BusinessLogic` al momento de crear una instancia de `MyDependencyRepository`, esta depende de la misma.

Qué problema tenemos acá?

Nuestras capas de **lógica de negocio** y el **acceso a datos** no deberían estar tan fuertemente acoplados, ademas de que no deberían tener qué preocuparse sobre la creación de sus dependencias.

Este tipo de **dependencias**, el uso de la palabra reservada **new,** son problemáticos y deberían de evitarse por las siguientes razones.

1. Si quisiéramos llegar a cambiar `MyDependencyRepository`, entonces nuestra clase `BusinessLogic` se vería afectada por este cambio.
2. Si `MyDependencyRepository` tiene sus propias dependencias, **debemos configurarlas dentro de** `BusinessLogic`. Para un proyecto grande con muchas dependencias y clases, el código de la configuración empieza a esparcirse a lo largo de toda la solución.
3. **Es muy difícil de testear, ya que las dependencias 'están hardcodeadas'.** Nuestro controller siempre llama a la misma lógica de negocio, y nuestra lógica de negocio siempre llama al mismo repositorio para interactuar con la base de datos. En una prueba unitaria, se necesitaría realizar un mock/stub de las clases dependientes, para evitar probar las

dependencias. Por ejemplo: si queremos probar la lógica de `BusinessLogic` sin tener que depender de la lógica de la base de datos, podemos hacer un mock de `MyDependencyRepository`. Sin embargo, con nuestro diseño actual, al estar las dependencias 'hardcodeadas', esto no es posible.

Solución: Inyección de dependencias

La inyección de dependencias es un patron de diseño que nos permite romper el acoplamiento de entre diferentes componentes de nuestro software. Por lo general se utiliza para romper el acoplamiento de implementacion, el cual es el que sucede entre dos clases concretas.

El acoplamiento es la relación de uso que puede suceder entre dos clases lo que lleva a que se den entre dos paquetes, esta relación de uso también da a saber el sentido de la propagación de cambio de alguno modulo o clase, el cual es sentido inverso a la flecha. Llevándolo a los diagramas creados mas arriba, si `SessionLogic` cambia entonces `UserLogic` se ve afectada por ese cambio.

Veamos como la inyeccion de dependencia puede solucionar los problemas antes mencionados:

- El uso de una interfaz o una clase abstracta base es una buena metodologia para abstraer las dependencias de implementacion
- Registrar las dependencias en un contenedor de servicios. ASP.Net Core provee un contenedor de servicios, `IServiceProvider`. Tipicamente estos servicios son registrados en el método `Startup.ConfigureServices`.
- Inyección del servicio en el constructor de la clase que lo utiliza. El framework toma la responsabilidad de crear una instancia de la dependencia y de eliminarla cuando no se necesita mas (le hace el dispose).

Nuevos diagramas:

Diagrama de clases

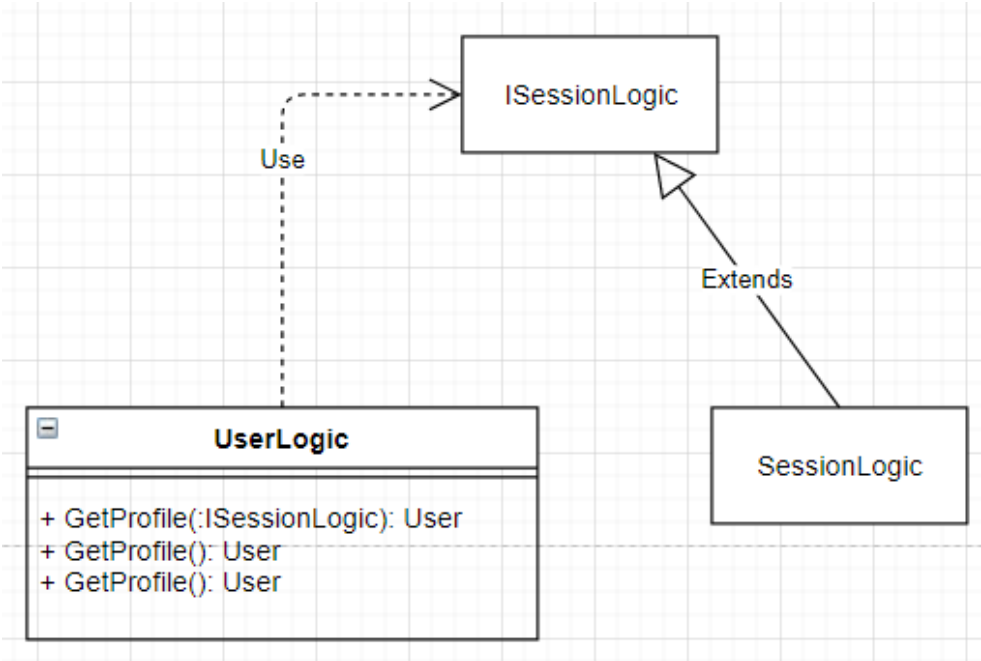
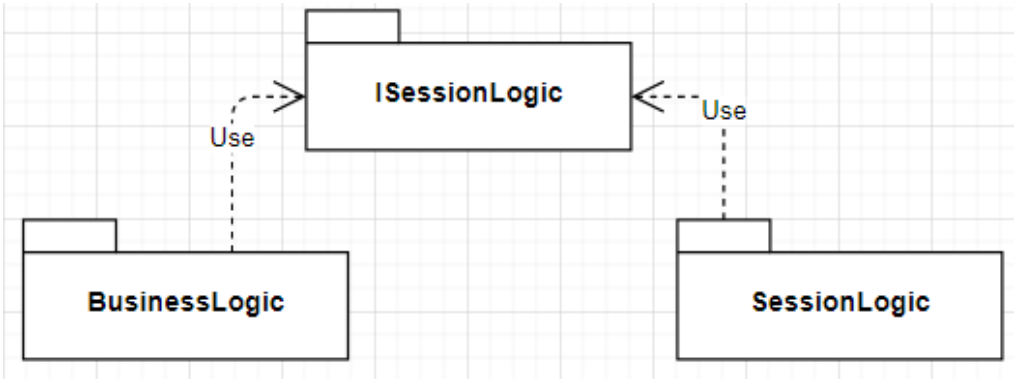


Diagrama de paquetes



Dicho esto, mejoremos el código anteriormente escrito:

1. Pasaremos a crear una interfaz que contenga la firma del metodo que `MyDependencyRepository` implementaba:

```
public interface IRepository
{
  void GetDate(int identifier);
}
```

2. Ahora pasaremos a que `MyDependencyRepository` implemente dicha interfaz:

```
public class MyDependencyRepository : IRepository
{
  public void GetDate(int identifier)
```

```
  {
    Console.WriteLine($"MyDependency.GetData called. Message: {identifier}");
  }
}
```

3. Ahora pasaremos a registrar el servicio `IRepository` con el tipo concreto `MyDependencyRepository.` El metodo **AddScoped** registra el servicio con un ciclo de vida **scoped,** esto significa que vive solamente por la request.

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddScoped<IRepository, MyDependencyRepository>();
}
```

4. Luego de que esta registrado pasaremos a solicitarlo en la clase `BusinessLogic`

```
public class BusinessLogic
{
  private readonly IRepository myDependency;

  public BusinessLogic(IRepository dependency)
  {
    this.myDependency = dependency;
  }

  public void OnGet()
  {
    this.myDependency.GetData(5);
  }
```

Analicemos estos cambios:

- `BusinessLogic` dejo de interactuar con una clase concreta y paso a interactuar con una interfaz, esto logra que sea mas fácil de cambiar la implementacion sin tener que modificar `BusinessLogic`.
- No se crea una instancia de `MyDependencyRepository` , es creada por el contenedor de servicios.

Ventajas de ID

Logramos resolver lo que antes habíamos descrito como desventajas o problemas.

1. Código más limpio. El código es más fácil de leer y de usar.
2. Nuestro software termina siendo más fácil de Testear.

3. Es más fácil de modificar. Nuestros módulos son flexibles a usar otras implementaciones. Desacoplamos nuestras capas.
4. Permite NO Violar SRP. Permite que sea más fácil romper la funcionalidad coherente en cada interfaz. Ahora nuestra lógica de creación de objetos no va a estar relacionada a la lógica de cada módulo. Cada módulo solo usa sus dependencias, no se encarga de inicializarlas ni conocer cada una de forma particular.
5. Permite NO Violar OCP. Por todo lo anterior, nuestro código es abierto a la extensión y cerrado a la modificación. El acoplamiento entre módulos o clases es siempre a nivel de interfaz.

Ciclos de vida de los servicios

Anteriormente se menciono el ciclo de vida **AddScoped** para registrar el servicio `IRepository`.

Los servicios pueden registrarse con los siguientes ciclos de vida:

1. Transient
2. Scoped
3. Singleton

Transient

Estos servicios son creados cada vez que son solicitados desde el contenedor de servicios. Este ciclo de vida sirve mejor para servidores stateless livianos. Para registrar servicios transient se tiene que utilizar `AddTransient`.

Scoped

Estos servicios son creados por cada request cliente. Para registar estos servicios se hace con `AddSingleton`.

Singleton

Estos servicios son creados la primera vez que son pedidos o por el desarrollador, cuando se provee una instancia de la implementacion directamente en el contenedor.

Esto quiere decir que todas las requests utilizan la misma instancia.

⚠ 1. Los constructores pueden aceptar parámetros que no se proveen por inyección de dependencia, estos tienen que tener tener un valor por defecto.
2. Cuando los servicios son resueltos por `IServiceProvider` sus constructores

tienen que ser **públicos.**

3. Servicios de un ciclo de vida no deberían de usar un contexto de la base de datos con un ciclo de vida mas corto que el de ellos.

Referencias extra

- Dependency injection in ASP.NET Core
- App startup in ASP.NET Core