

Guia inyección de dependencias

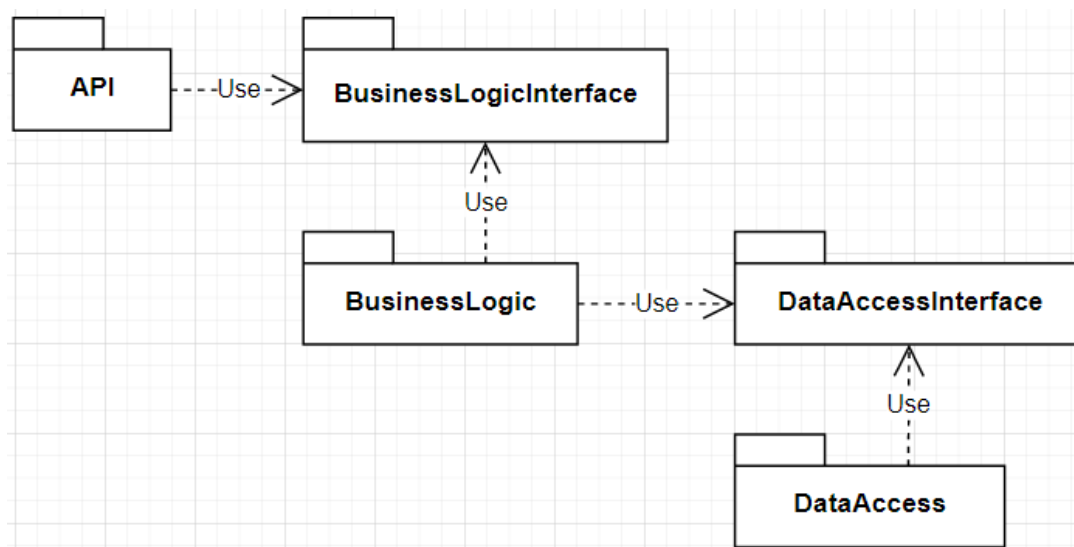
Vamos a continuar con el ejemplo de la lectura pasada, la API Vidly.

Sin querer entrar mucho en detalles sobre las capas de nuestra aplicación, rápidamente podemos distinguir tres grandes capas (pueden ser mas dependiendo de nuestra solución)

1. Nuestra API.
2. BusinessLogic, nuestra lógica de negocio.
3. DataAccess/Repository, capa de acceso a nuestro datos.

Teniendo esto en mente podríamos diseñar nuestra aplicación de la siguiente manera:

Diagrama de paquetes

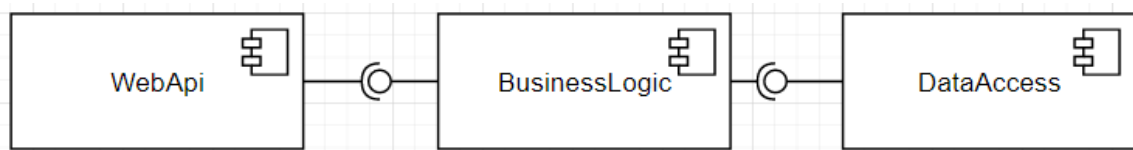


Este diagrama es un diagrama estático, nos muestra que paquete usa a quien, no nos dice como pero sabemos que es por medio de una dependencia. Analicemos las dependencias:

1. **API → BusinessLogicInterface:** esta dependencia es de interfaz lo cual nos asegura que el acoplamiento es el correcto porque no es de implementacion.

2. **BusinessLogic** → **BusinessLogicInterface**: esta dependencia es porque **BusinessLogic** es el paquete encargado de implementar las interfaces que se encuentran en **BusinessLogicInterface**, se esta dependiendo de algo mas abstracto lo cual indica que el acoplamiento es el correcto.
3. **BusinessLogic** → **DataAccessInterface**: esta dependencia va de un paquete concreto a uno abstracto el cual es el correcto porque es una dependencia de interfaz y no de implementacion.
4. **DataAccess** → **DataAccessInterface**: esta dependencia es porque **DataAccess** es el paquete encargado de implementar las interfaces que se encuentran en **DataAccessInterface**, se esta dependiendo de algo mas abstracto lo cual indica que el acoplamiento es el correcto.

Vista de componentes y conectores



Este diagrama es un diagrama dinámico, nos muestra como los diferentes proyectos que viven en la RAM, en tiempo de procesador, se comunican entre ellos por medio de interfaces.

Para nuestro ejemplo, tomemos una clase de cada paquete de manera de ilustrar como funcionaria la inyección de dependencias a través de nuestro flujo.

- API - MovieController
- BusinessLogicInterface - IMovieLogic
- BussinesLogic - MovieBussinessLogic
- DataAccessInterface - IMovieRepository
- DataAccess - MovieRepository

Empecemos a programar

1. Crearemos dos proyectos de clase a la solucion, uno que se va a llamar **BusinessLogicInterface** y el otro **BusinessLogic**. Primero vamos a hacer la inyeccion de dependencia para la logica de negocio y luego lo haremos igual con **DataAccess**.

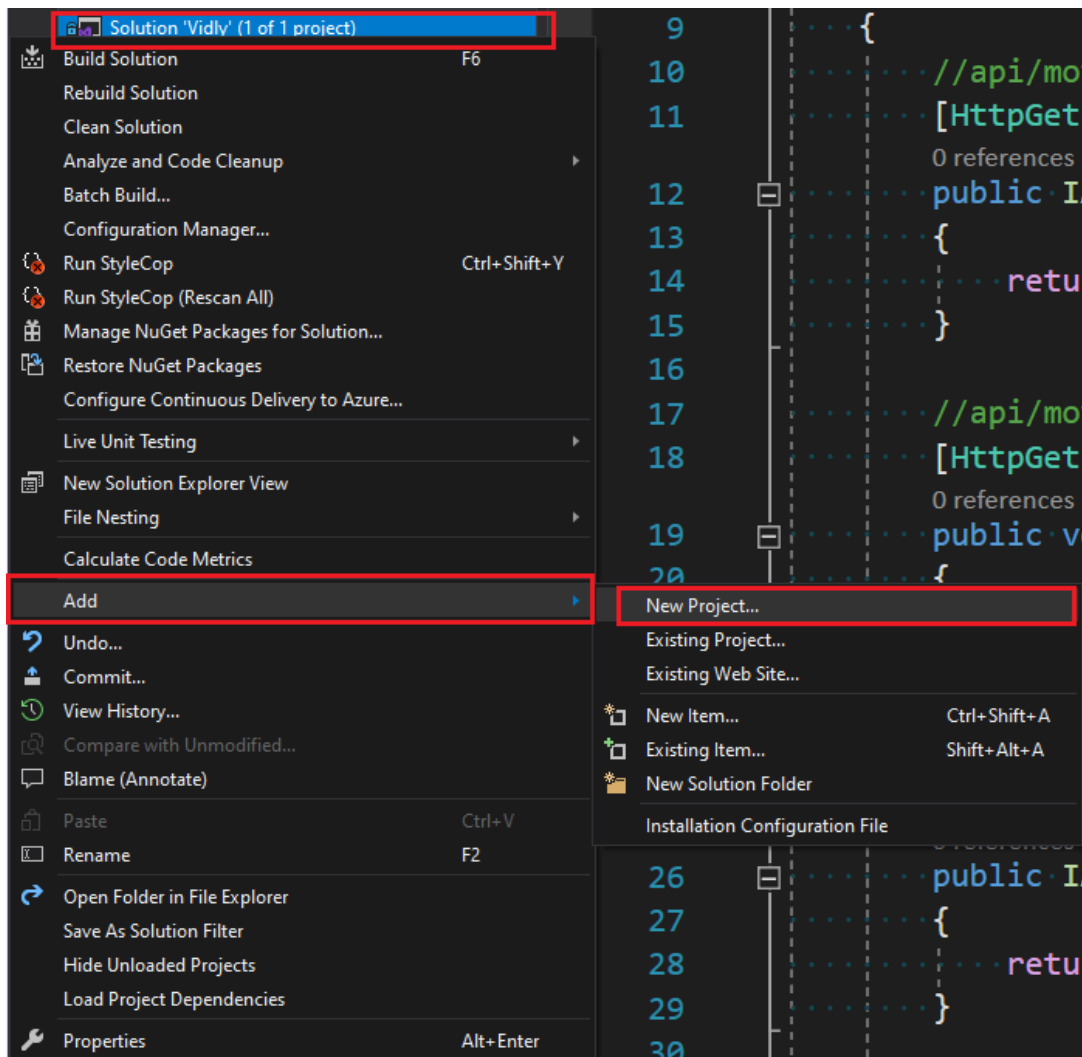
1. Para crear un **proyecto de clase** en **VSC** se tiene que usar los siguientes comando:

```
dotnet new classlib -n BusinessLogicInterface
dotnet sln add BusinessLogicInterface
dotnet new classlib -n BusinessLogic
dotnet sln add BusinessLogic
dotnet add BusinessLogic reference BusinessLogicInterface
dotnet add WebApi reference BusinessLogicInterface
```

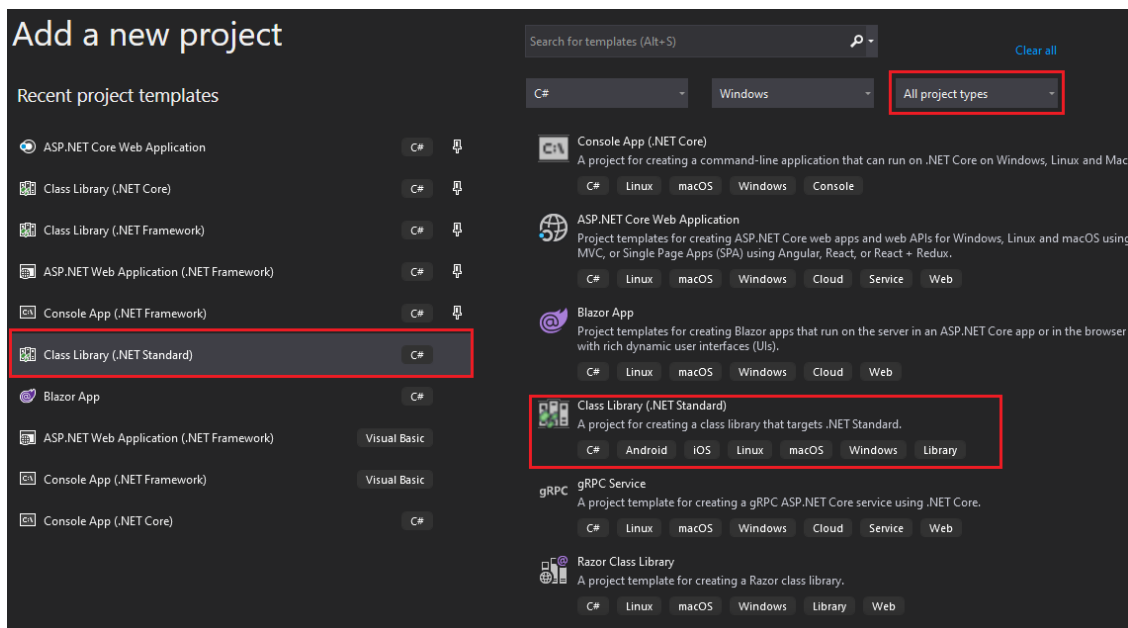
Lo que se hace en los comandos es crear los proyectos luego agregarlos a la solución, una vez agregados a la solución se empieza a agregar las referencias entre ellos como se había mostrado en el diagrama de paquetes.

2. Para crear un **proyecto de clase** en **VS** se tienen que seguir los siguientes pasos:

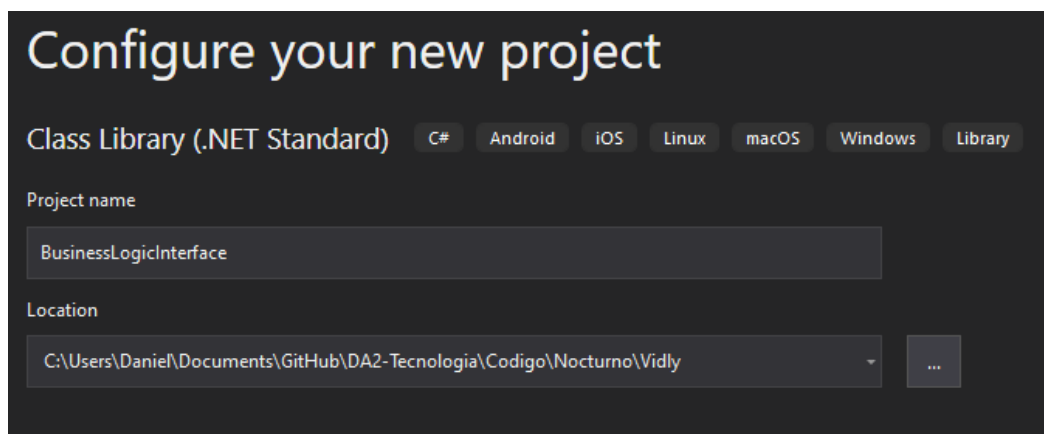
1. Click derecho en la solución → Add → New Project



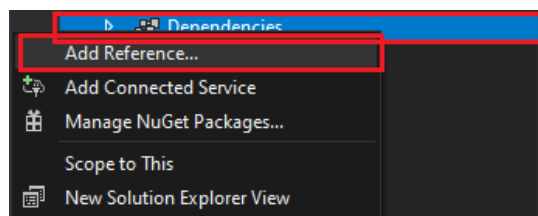
2. Hacer doble click en **Class Library (.NET Standard)** que se encuentra a la izquierda o seleccionar **All project types** en el combo de arriba a la derecha y luego seleccionar la opción **Class Library (.NET Standard)**



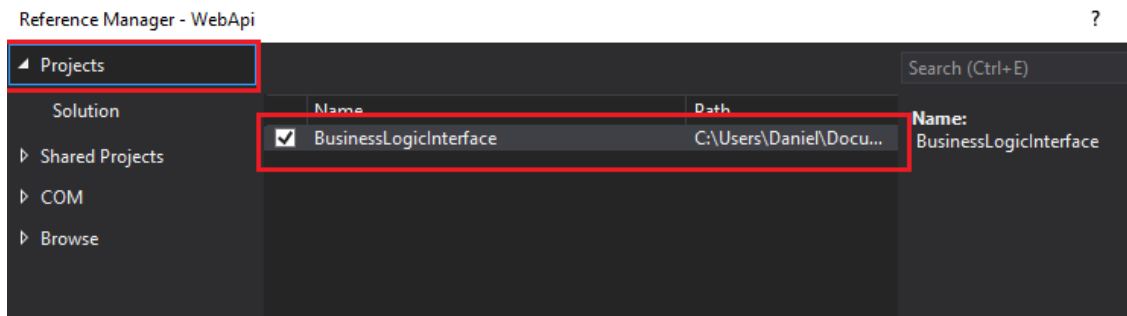
3. Escribir el nombre **BusinessLogicInterface** y darle **Create**.



4. Ahora para agregar la referencia de **BusinessLogicInterface** en **WebApi** hacemos click derecho en **Dependencies** de **WebApi** y seleccionamos **AddReference**.



5. En la parte de **Projects** seleccionamos el proyecto que recién creamos (**BusinessLogicInterface**) y le damos a **Ok**

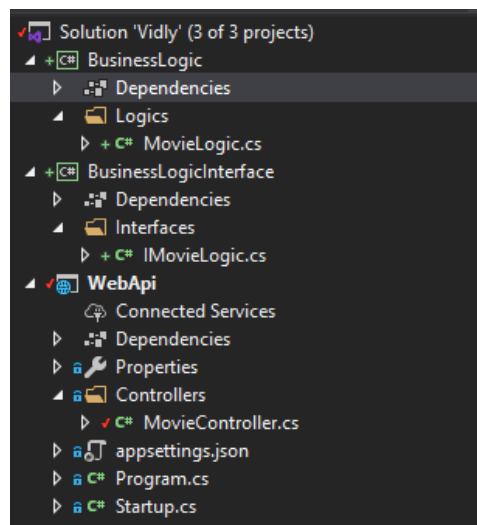


6. Para crear el proyecto **BusinessLogic** hay que repetir los pasos desde el **1** hasta el **5**

💡 La creación de los proyectos se puede hacer por comando por mas que se este usando **VS**, ya que estos se agregan a la solución entonces se podrán ver.

Luego de que ya tienen los dos proyectos creados y referenciados correctamente pasaremos a crear una interfaz llamada **IMovieLogic** en el proyecto **BusinessLogicInterface** y una clase **MovieLogic** en el proyecto **BusinessLogic** la cual implementara la interfaz **IMovieLogic**.

El explorador de la solución quedaría algo así:



Inyección de BusinessLogic

Siguiendo el diagrama vemos que la **WebApi** solo conoce **BusinessLogicInterface** y tomando en cuenta la introducción, la inyección de **IMovieLogic** en **MovieController** sería así:

WebApi

```
namespace WebApi.Controller
{
    [Route("api/movies")]
    public class MovieController : VidlyControllerBase
    {
        private readonly IMovieLogic moviesLogic;

        public MovieController(IMovieLogic moviesLogic)
```

```

    {
        this.moviesLogic = moviesLogic;
    }

    //...API CODE
}

```

BusinessLogicInterface

```

namespace BusinessLogicInterface
{
    public interface IMovieLogic
    {
        //...MOVIE OPERATIONS
    }
}

```

BusinessLogic

```

namespace BusinessLogic
{
    public class MovieLogic : IMovieLogic
    {
        // ...BUSINESS LOGIC CODE
    }
}

```

Por ahora lo que hicimos fue definir una interfaz, una clase concreta que la implementa y una clase que le pasan una interfaz y esta la utiliza.

A esta altura se deberían de estar preguntando:

Quién es el encargado de crear estas clases?

Porque alguien tiene que hacerlo en algún momento. Vayamos a ver la clase **WebApi.Startup**.

Startup

Esta clase es la encargada de registrar los servicios, que son dependencias, en un contenedor, y dependiendo del ciclo de vida que se les marque a estos servicios, va a ser el momento en el cual se cree una instancia.

Siguiendo con Vidly, se verá de la siguiente manera:

```

namespace WebApi
{
    public class Startup
    {
        //...STARTUP CODE


        public void ConfigureServices(IServiceCollection services)
        {
            //...CONFIGURE_SERVICES CODE

            //Registrando el servicio IMovieLogic
            services.AddScoped<IMovieLogic, MovieLogic>();
        }
    }
}

```

```
//...MORE STARTUP CODE  
}
```



Si quieren profundizar mas sobre como funciona internamente `AddScoped` y porque lo usamos para nuestra WebAPI toquen el librito → 

Seguramente les este lanzando un error de que no encuentra una referencia de **MovieLogic** y también deben estar analizando que **Startup** para inyectar dependencias tiene que conocer no solamente lo abstracto si no que la implementacion por ende tiene que haber una dependencia a la implementacion.

Gracias a este análisis que hicieron deberían de llegar a la conclusion de que alguien tiene que conocer ambas partes de la dependencia, lo abstracto y la implementacion para poder inyectarles esa implementacion aquellos proyectos que lo necesiten.

Dicho esto podemos sacar la conclusion de que **WebApi** no es ese experto de conocer ambas partes si no que es responsabilidad de otro. Ese otro va a ser un proyecto llamado `Factory`.

Como resultado de este **Factory** tendremos:

- Una clase **Startup** mas limpia y con pocas lineas de código ya que estamos delegando la responsabilidad de registrar los servicios en `IServiceCollection` a un experto en conocer ambas partes de la dependencia.
- Se respetara el acoplamiento a paquetes mas abstractos y estables que uno
- Bajo acoplamiento
- Alta cohesion

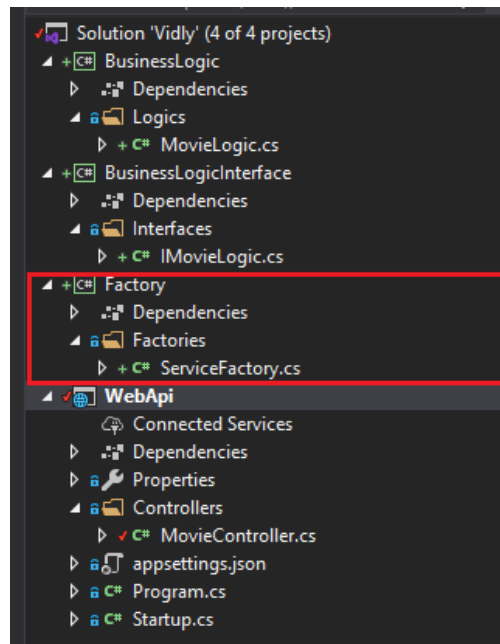
Factory

Factory va a ser un proyecto encargado de registrar los servicios en `IServiceCollection` de forma que tiene que conocer tanto la interfaz como la implementacion.



Mas adelante vamos a ver con **Reflection** que con conocer solamente la interfaz alcanza para averiguar la implementacion en una .dll conocida.

Este proyecto por ahora va a tener una única clase que llamaremos **ServiceFactory**. El explorador de soluciones quedaría de la siguiente manera:



Esta clase la podemos implementar de varias formas diferentes. Algunas de ellas son:

- Que sea **static** para poder usar los métodos en **WebApi.Startup** de forma sencilla
- Que sea una extensión del servicio **IServiceCollection**, esto significa que a esa interfaz le podemos agregar comportamientos que nosotros definamos
- Que sea una clase normal y en el método **WebApi.Startup.ConfigureServices(IServiceCollection services)** realicemos una instancia de este tipo y usemos los métodos.

Por motivos de minimizar la extensibilidad del documento y la complejidad vamos a realizar la implementación de una clase normal.

Factory

```
namespace Factory
{
    public class ServiceFactory
    {
        private readonly IServiceCollection services;

        public ServiceFactory(IServiceCollection services)
        {
            this.services = services;
        }

        public void AddCustomServices()
        {
            services.AddScoped<IMovieLogic, MovieLogic>();
        }
    }
}
```

Para que este código compile van a tener que agregar las siguientes referencias a este proyecto:

1. **BusinessLogicInterface**
2. **BusinessLogic**

Ahora veamos como queda **WebApi.Startup**:

WebApi

```
namespace WebApi
{
    public class Startup
    {
        //...STARTUP CODE
        public void ConfigureServices(IServiceCollection services)
        {
            //...CONFIGURE_SERVICES CODE

            ServiceFactory factory = new ServiceFactory(services);
            factory.AddCustomServices();
        }
    }
}
```

Nuevamente para que este código compile hay que agregar la referencia del proyecto **Factory** a este.

De esta forma logramos tener una clase **WebApi.Startup** bastante reducida, legible y con responsabilidades concretas que le corresponden. Hasta ahora hicimos la inyección de dependencia de la lógica de negocio, pasemos a inyectar la dependencia **DataAccess**.

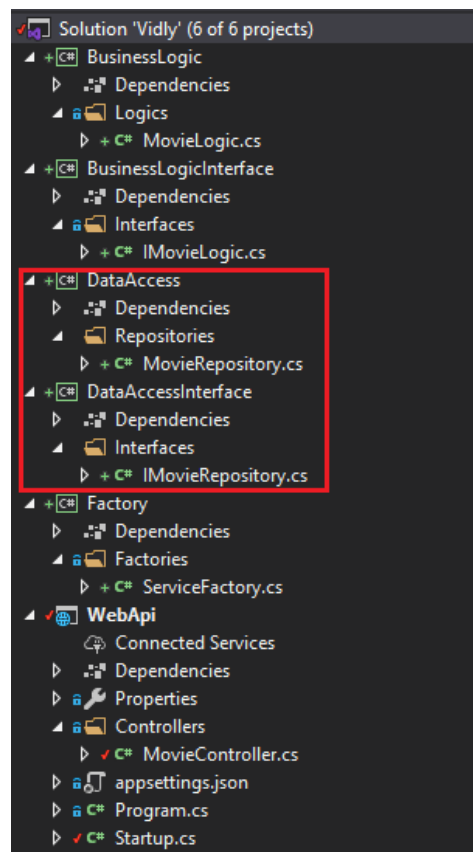
Inyección de DataAccess

Para realizar esta inyección de dependencias tendremos que crear primero los proyectos correspondientes. Los cuales son:

1. **DataAccessInterface**. En donde se va a tener una interface **IMovieRepository**
2. **DataAccess**. En donde se va a tener una clase concreta **MovieRepository**

▲ La creación de estos proyectos se desprende de los pasos que se realizaron de la creación de los proyectos **BusinessLogic** y **BusinessLogicInterface**.

Con respecto a las referencias, se debería de seguir el diagrama mostrado mas arriba. Mostremos como quedaría el explorador de soluciones:



Una vez creados estos proyectos y realizados las referencias pasemos a ver como queda **MovieLogic** el cual necesita de **IMovieRepository**:

BusinessLogic

```
namespace BusinessLogic
{
    public class MovieLogic : IMovieLogic
    {
        private readonly IMovieRepository moviesRepository;

        public MovieLogic(IMovieRepository moviesRepository)
        {
            this.moviesRepository = moviesRepository;
        }

        //...MOVIE_LOGIC CODE
    }
}
```

Veamos como registrar servicio en el contenedor de servicios:

Factory

```
namespace Factory
{
    public class ServiceFactory
    {
        private readonly IServiceCollection services;

        public ServiceFactory(IServiceCollection services)
        {

```

```

        this.services = services;
    }

    public void AddCustomServices()
    {
        services.AddScoped<IMovieRepository, MovieRepository>();
        services.AddScoped<IMovieLogic, MovieLogic>();
    }
}

```

Por supuesto para que esto compile van a tener que agregar las referencias de los proyectos **DataAccess** y **DataAccessInterface** a **Factory**.

Ultima inyección

Ahora realizaremos la ultima inyección, la cual es la del contexto a nuestro repositorio para que este pueda ejecutar queries a la base de datos y pueda persistir las entidades al motor de base de datos SQL Server. Veamos la clase **MovieRepository**:

DataAccess

```

namespace DataAccess
{
    public class MovieRepository : IMovieRepository
    {
        private readonly DbSet<Movie> movies;
        private readonly DbContext vidlyContext;

        public MovieRepository(DbContext context)
        {
            this.vidlyContext = context;
            this.movies = context.Set<Movie>();
        }

        //.../MOVIE_REPOSITORY CODE
    }
}

```

Como vemos en el código, el repositorio le tienen que inyectar la dependencia **DbContext** la cual es una clase abstracta de la librería **Microsoft.EntityFrameworkCore**. También habrán notado la clase **Movie** la cual nos vamos a referir a las de su clase como entidades del negocio y esta debería de estar situada en un proyecto llamado **Domain**, esto lo veremos más adelante.

💡 Para aquellas personas que les interesa saber más sobre esta clase lean la lectura: [Configuring a DbContext](#)

Esa dependencia se tiene que registrar como un servicio como estuvimos haciendo hasta el momento de la siguiente manera:

Factory

```

namespace Factory
{
    public class ServiceFactory
    {
        //...SAME CODE AS ABOVE

        public void AddDbContextService(string connectionString)
        {

```

```

        services.AddDbContext<DbContext, VidlyContext>(options => options.UseSqlServer(connectionString));
    }
}

```

Acá utilizamos otro método para registrar un servicio porque estamos registrando un servicio específico el cual es el contexto y para este ya hay un método.

Este método recibe por parámetro la configuración del contexto que debería de tener cuando se cree una instancia del mismo. Para esto se tendrán que instalar los siguientes paquetes:

1. Microsoft.EntityFrameworkCore
2. Microsoft.EntityFrameworkCore.SqlServer

Veamos la clase **VidlyContext** la cual se encuentra en el proyecto **DataAccess**

DataAccess

```

namespace DataAccess.Context
{
    public class VidlyContext : DbContext
    {
        public DbSet<Movie> Movies { get; set; }

        public VidlyContext() { }
        public VidlyContext(DbContextOptions options) : base(options) { }
    }
}

```

WebApi

```

namespace WebApi
{
    public class Startup
    {
        //...STARTUP CODE
        public void ConfigureServices(IServiceCollection services)
        {
            //...CONFIGURE_SERVICES CODE

            ServiceFactory factory = new ServiceFactory(services);
            factory.AddCustomServices();
            factory.AddDbContextService(this.Configuration.GetConnectionStrings("VidlyDb"));
        }
    }
}

```

El `ConnectionString` se va a extraer de un archivo de configuración de nuestra API llamado `appsettings.json`, lo que nosotros debemos poner arriba en el método `GetConnectionString` es la key donde se encuentra este String.

El archivo `.appsettings.json` sería:

```

{
  "ConnectionStrings": {
    "VidlyDB": "Server=.\SQLEXPRESS;Database=VidlyDB;Trusted_Connection=True;MultipleActiveResultSets=True;"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  }
},

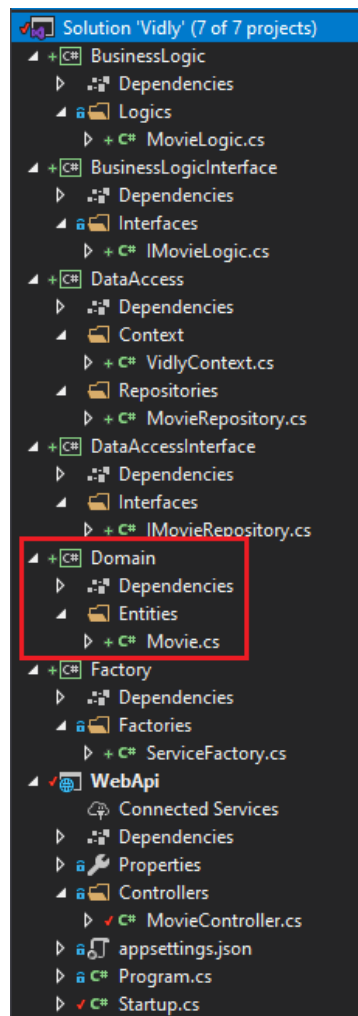
```

```
"AllowedHosts": ""
}
```



Este archivo de configuración se encuentra dentro del proyecto WebApi y es generado por defecto cuando se crea un tipo de proyecto WebApi.

Para terminar queda crear el proyecto **Domain** que contenga la clase **Movie** y que **BusinessLogic**, **BusinessLogicInterface**, **DataAccess** y **DataAccessInterface** agreguen una referencia a este proyecto. El explorador de soluciones quedaría así:



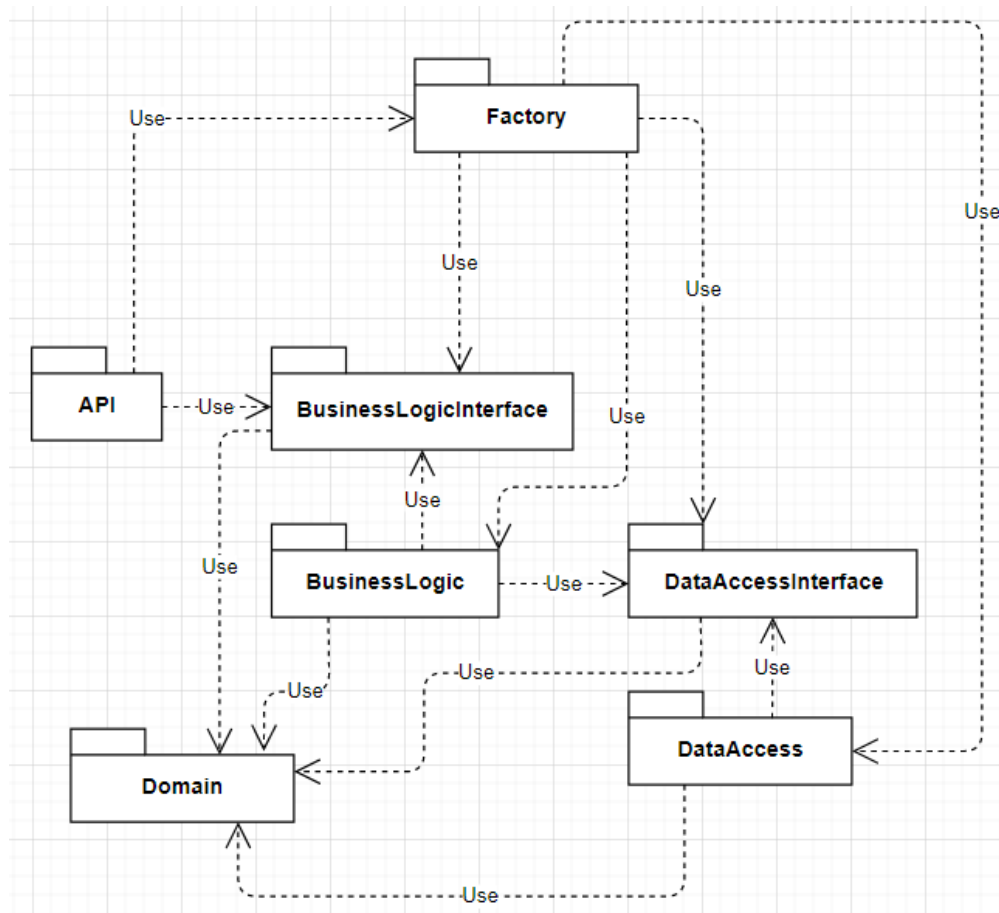
En donde por ahora la clase **Movie** tendrá lo siguiente:

```
namespace Domain
{
    public class Movie
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public int AgeAllowed { get; set; }
    }
}
```

Resumen

Al final del tutorial les debió haber quedado un sistema que utiliza inyección de dependencia por lo tanto es un sistema extensible y que usa acoplamiento de interfaz y no de implementación.

En retrospectiva les tuvo que haber quedado el siguiente diagrama de dependencias:



Algunos paquetes extras que hay que instalar:

- En **Factory** hay que instalar:
 1. Microsoft.EntityFrameWorkCore
 2. Microsoft.EntityFrameWorkCore.SqlServer
 3. Microsoft.Extensions.DependencyInjection.Abstractions
- En **DataAccess** hay que instalar:
 1. Microsoft.EntityFrameWorkCore

Que nos permite que nuestro sistema sea extensible? Nos permite muchas cosas, entre ellas esta que sea flexible a los cambios, cambiar implementacion en tiempo de ejecucion, nos permite respetar los SOLID de diferentes formas, y muchas cosas mas.

La forma de ver esto es entre **BusinessLogic** y **DataAccess**, podriamos crear una clase que se llame **MovieMemoryRepository** la cual va a implementar la interfaz **IMovieRepository** pero va a persistir en memoria en vez de en base de datos como hace **MovieRepository**. Lo unico que habria que hacer es cambiar la inyeccion de dependencia en **Factory** de:

```
public void AddCustomServices()
{
    //....
    services.AddScoped<IRepository, MovieRepository>();
}
```

por:

```
public void AddCustomServices()
{
    //....
    services.AddScoped<IRepository, MovieMemoryRepository>();
}
```

y **MovieLogic** no se entero que le cambiaron la implementacion porque el solamente ve la interfaz, no ve que es lo que ocurre por atrás. Si lo pensamos un poco mas podríamos realizar este cambio de implementacion en tiempo de ejecución, ya sea por un parámetro que me manda el usuario, una cambio de configuración por un administrador, etc. Este cambio ocurriría mediante un estímulo desde un origen donde impactaría en un artefacto que esta dentro de un ambiente y se llevaría a una respuesta.

En pocas palabras se esta mencionando una forma de como implementar el patron **Strategy** con **Inyeccion de dependencia** y **APIs**.