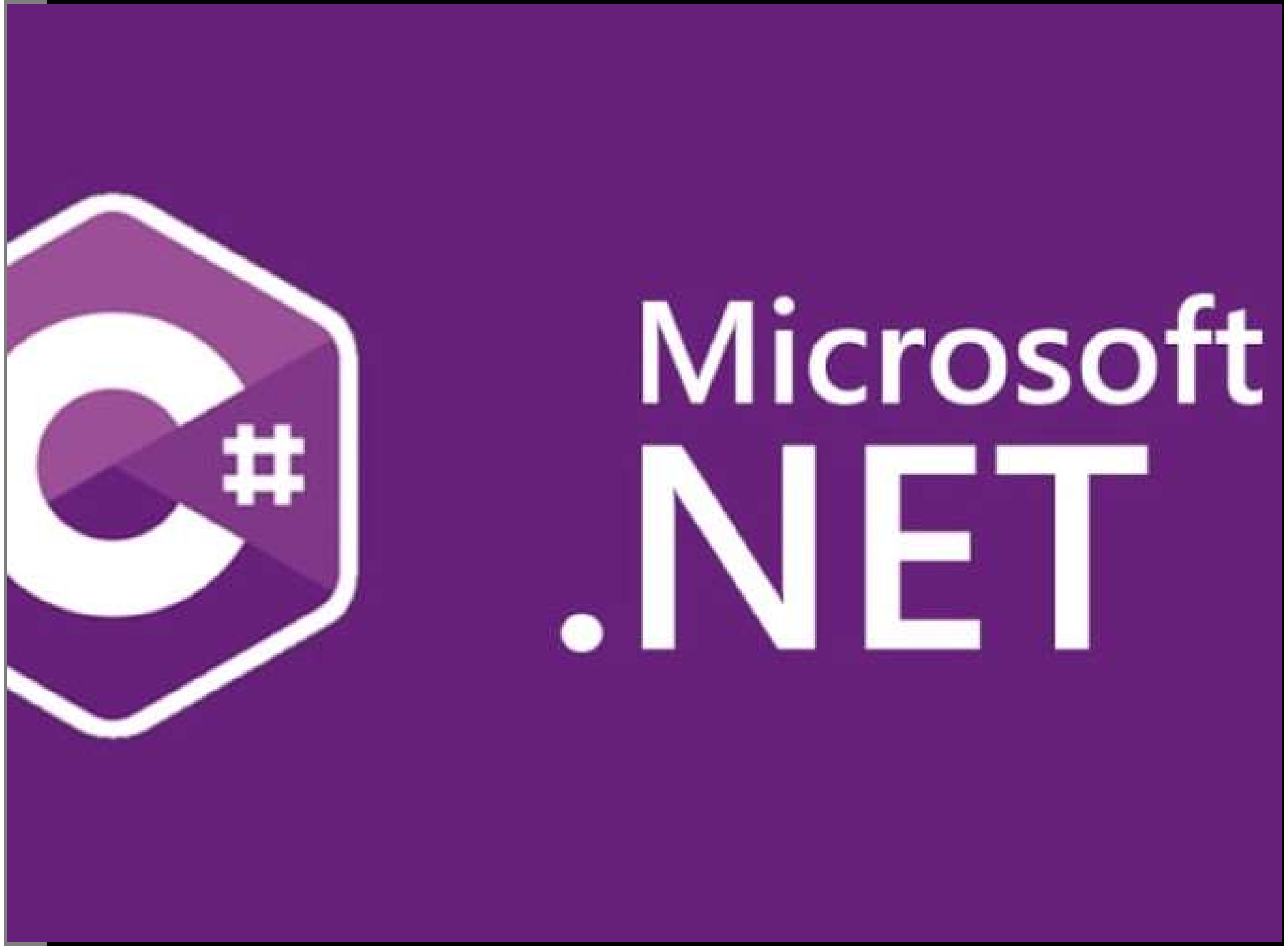


Dependency Injection

ASP .NET Core

Yash Lathiya

Table of Contents

Built-in IoC Container

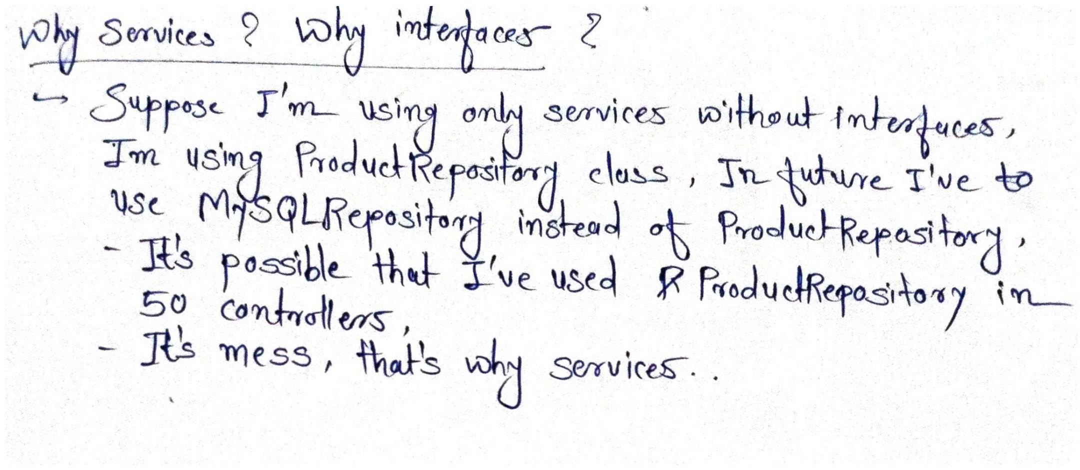
DI (Dependency Injection) container is a container which contains interfaces and implementation of that interfaces.

In ASP .NET DI container is already provided which is also known as built in IoC container.

It means we can now take refence of the interfaces in application & that interface will take reference of implementation which is mapped in DI container.

Built in IoC container is used for providing support of dependency injection in application to smooth the process.

Dependency injection also adds future scope for updation within application without major changes.



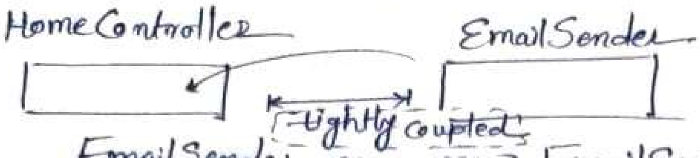
Why Services ? Why interfaces ?

↪ Suppose I'm using only services without interfaces, I'm using ProductRepository class, In future I've to use MySQLRepository instead of ProductRepository,

- It's possible that I've used ~~R~~ ProductRepository in 50 controllers,
- It's mess, that's why services..

Dependency Injection ← .NET Core Web API

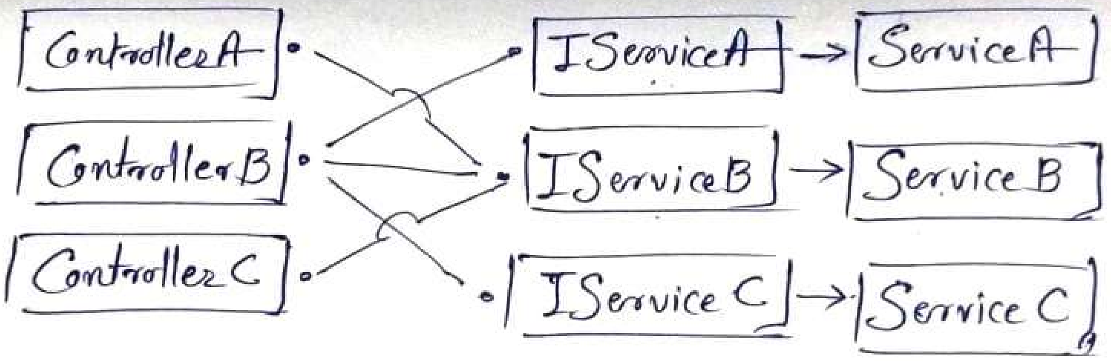
↳ Suppose we are sending email from controller



EmailSender es = new EmailSender();

Problems : - what if I want to modify EmailSender service ?
- What if I want to use third party service in future ?
- As they are tightly coupled, I've to change (modify) home controller, which is violation of SOLID principle.

Solution :



How to configure DI ?

· Built in container → IServiceProvider
Add registered service in Startup. ConfigureServices.

Lifetime of DI

- Singleton (Only once - When application starting)
- Scoped (instance will not be saved) (Everytime
- Transient (Every time - when requested) (instance will be saved)

Registering Application Service

We can register any service with its implementation in ConfigureService() method of startup.cs file.

```
public void ConfigureServices(IServiceCollection services)
{
            services.AddControllers();
            services.AddSwaggerGen();

            //services.AddSingleton<IProductService, ProductService>();

            //services.AddTransient<IProductService, ProductService>();

            //services.AddScoped<IProductService, ProductService>();

            services.AddSingleton<IProductService, MySqlProductService>();
}
```

Here, bold text demonstrates, how can we register any service in DI / IoC container.

Understanding Service Lifetime

Transient

```
            // Instance of service is created every time that
their injection into class is required..
            // Will print different time on console as object of
services are created more than once..
            services.AddTransient<IDateTime,
DI_2_Understanding_Lifetime.Services.DateTime>();
```

Singleton

```
            // Instane of service is created only once
            // Will print same time on console as object of
services are created only once..
            services.AddSingleton<IDateTime,
DI_2_Understanding_Lifetime.Services.DateTime>();
```

Scoped

```
            // Instane of service is created every time when
http request is fired
            // Will throw an error as constructor injection is
used..
            //services.AddScoped<IDateTime,
DI_2_Understanding_Lifetime.Services.DateTime>();
```

Extension Methods for Registration

We can add extension method which registers all services and implementation..

Startup.cs :

```
public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllers();
            services.AddSwaggerGen();

            // Add service through extension method
            services.AddMyService();
        }
```

Code for Extension Method..

```
/// <summary>
/// Static class contails all members which are in-frequent use
/// </summary>
public static class ServiceExtensions
{
    #region Static Methods

    /// <summary>
    /// extension methos which register student service in DI container
    /// </summary>
    /// <param name="services"> collection of services </param>
    /// <returns> services </returns>
    public static IServiceCollection AddMyService(this IServiceCollection
services)
    {
        // Register MyService with the dependency injection container
        services.AddSingleton<IStudentService, StudentService>();

        return services; // This allows chaining of registrations
     }

     #endregion

 }
```

Constructor Injection

Constructor injection is a phynonyma in which dependency injection is provided through constructor.

When instance of class is created it also gives support of required components or dependencies.

Here is a example of service is demonstrated where almost all types of constructor injection is implemented.

```
/// <summary>
/// Used different constructor injection approaches
/// </summary>
/// <param name="dogSoundService"> From another interface
</param>
/// <param name="configuration"> From configuration interface
</param>
/// <param name="options"> From options interface </param>
/// <param name="sheepSound"> without any interface </param>

public AnimalSoundService(IDogSoundService dogSoundService,
                          IConfiguration configuration,
                          IOptions<CowOptions> options,
                          string sheepSound)
{
     lstAnimalSounds = new List<String>()
       {
                dogSoundService.GetSound(),
                configuration["CatSound"],
                options.Value.CowSound,
                sheepSound
     };
}
```