LEVEL 3 – Register Assembly

Kütüphaneler

-Autofac
-Autofac.Extras.DynamicProxy

Bu dökümanda projemizdeki yapıları tek tek register etmek yerine proje çözümünün içindeki yapıları ve bu yapılar hakkındaki bilgileri içeren Assembly sınıfını kullancağız.

İlk başta interceptor yapılarımızı oluşturalım.Bu sefer AOP mimarisinin nihai amacını gerçekleştirecek şekilde base interceptor sınıfımızdaki ortak bölücü metodumuza try-catch-finally yapısını ekliyoruz.Ayrıca interceptorlarımızın hangi durumda nasıl çalışacağını sağlamak için çalışan metodun, çalışmadan önce (OnBefore), çalışırken hata verdiğinde (OnException), başarılı bir şekilde çalıştığında(OnSuccess) ve çalışmasını bitirdiğinde(OnAfter) metodlarını virtual olarak tanımlıyoruz.

```
public abstract class InterceptorAspect : Attribute, IInterceptor
{
    protected virtual void OnBefore(IInvocation invocation) { }
    protected virtual void OnAfter(IInvocation invocation) { }
    protected virtual void OnException(IInvocation invocation, Exception e) { }
    protected virtual void OnSuccess(IInvocation invocation) { }
    public virtual void Intercept(IInvocation invocation)
    {
        bool state = true;
        OnBefore(invocation);
        try
        {
            invocation.Proceed();
        }
        catch (Exception e)
        {
            state = false;
            OnException(invocation, e);

            throw;
        }
        finally
        {
            if (state)
                OnSuccess(invocation);
        }
        OnAfter(invocation);
    }
}
```

Interceptor yapılarımız ise şu şekilde:

```
public class CacheAspect : InterceptorAspect
{
    protected override void OnAfter(IInvocation invocation)
    {
        Console.WriteLine($"{invocation.Method.Name} metodu cachee eklendi");
    }
}
```

Cache işlemini metod çalışması bittiğinde yapmak istediğim için OnAfter metodunu override ederek içerisine ilgili kodları yazdım.Fakat siz cache işlemini OnSuccess ve ya herhangi bir durumda çalışmasını istiyorsanız o metodu override ederek yapabilirsiniz.

```
public class LogAspect:InterceptorAspect
{
    protected override void OnBefore(IInvocation invocation)
    {
        Console.WriteLine($"{invocation.Method.Name} metodu çalıştı");
    }

    protected override void OnException(IInvocation invocation, Exception e)
    {
        Console.WriteLine($"{invocation.Method.Name} metodu hata verdi");
    }

    protected override void OnSuccess(IInvocation invocation)
    {
        Console.WriteLine($"{invocation.Method.Name} metodu başarıyla çalıştı")
;
    }

    protected override void OnAfter(IInvocation invocation)
    {
        Console.WriteLine($"{invocation.Method.Name} metodu sona erdi");
    }
}
```

Log işlemi yapılan her işlemi kayıt etme işlemi olduğu için her durumda loglamasını sağlamak için tüm metodları override ederek ilgili kodları yazdım.Siz log işlemini hangi durumlarda yapmak isterseniz kodunuzu ona göre değiştirebilirsiniz.

```
public class ValidationAspect : InterceptorAspect
{
    protected override void OnBefore(IInvocation invocation)
    {
        Console.WriteLine($"{invocation.Method.Name} metodu için validation
                                                    çalıştı");
    }
}
```

Validasyon işlemleri genellikle metod çalışmadan önce yapılır.Bir ürün eklerken ürün ismini veri tabanına eklemeden önce validasyon kurallarına uygunmu diye kontrol edersiniz.Bu yüzden metod çalışmadan önce validasyon kontrolü yapmak için OnBefore metodunu kullandık ve ilgili kodları yazdık.

Interceptor seçicimiz ise aynı şekilde oluşturuyoruz.Burda farklı olarak her metod için ortak çalışacak interceptorları metod için tek tek tanımlamak yerine burda bir dizide tanımlayarak bu işlemi gerçekleştirebiliriz.

```
public class InterceptorSelector : IInterceptorSelector
{
    public IInterceptor[] SelectInterceptors(Type type, MethodInfo method,
                                             IInterceptor[] interceptors)
    {
        var methodInterceptors = type.GetMethod(method.Name)
                    .GetCustomAttributes<InterceptorAspect>(true).ToList();
        var classInterceptors = type
                    .GetCustomAttributes<InterceptorAspect>(true).ToList();
        classInterceptors.AddRange(methodInterceptors);

        classInterceptors.AddRange(new List<InterceptorAspect>
        {
            new LogAspect()
        });

        return classInterceptors.ToArray();
    }
}
```

Ve son olarak register işlemlerini gerçekleştirelim.Burda Assembly statik sınıfını kullanarak proje çözümündeki yapılara ulaşabiliriz.

```
static void Main(string[] args)
{
    var container = new ContainerBuilder();
    var assembly=Assembly.GetExecutingAssembly();
    container.RegisterType<ProductManager>()
                            .As<IProductService>()
                            .SingleInstance();
    container.RegisterAssemblyTypes(assembly)
        .AsImplementedInterfaces()
        .EnableInterfaceInterceptors(new ProxyGenerationOptions
        {
            Selector=new InterceptorSelector()
        })
        .SingleInstance();

    var build = container.Build().BeginLifetimeScope();
    var item = build.Resolve<IProductService>();

    item.Add();
    Console.WriteLine("***************");
    item.Delete();
 }
```

GetExecutingAssembly() metodu ile projemizdeki yapıları assembly değişkenimize atıyoruz.Artık tüm çözümdeki yapılarımız ile ilgili bilgiler assembly değişkenimizin içinde.ContainerBuilder yapımıza RegisterAssemblyTypes metodu ile assemblyi register ediyoruz.EnableInterfaceInterceptors ile yapılarımızın interceptor özelliklerini aktif ediyoruz.

Assembly yapımızın içinde servis sınıflarımızda bulunduğu için aslında ProductManager sınıfımızı ayrıca register etmemize gerek yoktu.Fakat servislerimizin referans ve injection gibi durumlarını ayrı ayrı yönetmek ve kod okunulabilirliğini sağlamak adına ayrıca register etmenin faydalı olacağını düşünmekteyim.