

LEVEL 3 – EnableInterfaceInterceptors

Kütüphaneler

- Autofac
- Autofac.Extras.DynamicProxy

Bu dökümanda interceptorları aktif ederken kullandığımız `EnableClassInterceptors` ve `EnableInterfaceInterceptors` metodlarını inceleyeceğiz. Ve servislerde interceptorları kullanmak için gerekli olan virtual metod şartını bu sayede aşarak virtual olmayan metodda interceptorları nasıl kullanırız bunu göreceğiz.

Öncelikle interceptor yapılarımızı oluşturmakla başlayalım. Burada base bir interceptor oluşturacağım. Ve oluşturduğum diğer interceptor nesnelere bu yapıyı inherit ederek Intercept metoduyla ortak bir aksiyona bağladım. Fakat istediğiniz bir interceptor yapısının kendine özgü bir aksiyonu olacaksa Intercept metodunu override ederek içini doldurabilirsiniz (LogAspect’de olduğu gibi).

```
public abstract class InterceptorAspect : Attribute, IInterceptor
{
    public virtual void Intercept(IInvocation invocation)
    {
        Console.WriteLine("Common intercept method is run");
        invocation.Proceed();
    }
}

public class CacheAspect : InterceptorAspect
{
}

public class ValidationAspect : InterceptorAspect
{
}

public class LogAspect:InterceptorAspect
{
    public override void Intercept(IInvocation invocation)
    {
        Console.WriteLine("Private LogAspect is run");
        invocation.Proceed();
    }
}
```

Şimdi de interceptor seçici sınıfımızı oluşturalım. Bu sefer interceptor seçicimiz hazır bir interceptor dizisi değil, servisimizdeki interceptorları okuyup bunları bir diziye çevirerek döndürmesini sağlayacağız.

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

Servisimizdeki metod çalıştığı zaman interceptor seçici devreye girecek ve metodun üstündeki `InterceptorAspect` türündeki Attribute yapılarını okuyarak diziye döndürecek. Ve bu diziyi return ederek çalıştırdığımız metod için bu dizideki interceptorları çalıştıracaktır.

Şimdi de servisimizde bu interceptorları kullanmak için yerleştirelim. Burada farkı görmek amacıyla `Delete` metodu oluşturarak virtual olarak işaretledim. `Add` metodu ise virtual değil.

```
public class ProductManager : IProductService
{
    [LogAspect]
    [CacheAspect]
    [ValidationAspect]
    public void Add()
    {
        Console.WriteLine("Product is added!");
    }

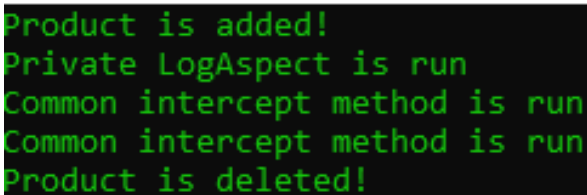
    [LogAspect]
    [CacheAspect]
    [ValidationAspect]
    public virtual void Delete()
    {
        Console.WriteLine("Product is deleted!");
    }
}
```

Sırada `ContainerBuilder` sınıfına yapmamız gereken register işlemleri var. İlk olarak `EnableClassInterceptors()` metodunu ele alarak başlayalım.

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

    var build = container.Build().BeginLifetimeScope();
    var item = build.Resolve<IProductService>();
    item.Add();
    item.Delete();
}
```

Bu halde uygulamayı çalıştırdığımız zaman servis (somut sınıf) ve bu sınıfın içindeki `virtual` metodlardaki interceptorlar çalışacaktır. Yani `Add` metodu interceptorları çalışmayacak fakat `virtual` olan `Delete` metodu üzerindeki interceptorlar çalışacaktır. Ekran çıktısı şu şekilde olacaktır.



```
-----
Product is added!
Private LogAspect is run
Common intercept method is run
Common intercept method is run
Product is deleted!
```

Birde `EnableInterfaceInterceptors()` metodunu kullanalım.

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

    var build = container.Build().BeginLifetimeScope();
    var item = build.Resolve<IProductService>();
    item.Add();
    item.Delete();
}
```

Bu halde uygulamayı çalıştırdığımız zaman sadece servise implement edilen interface'den gelen metodlara (virtual olup olmadığına bakmaksızın) interceptor uygular. Bunun da ekran çıktısı da aşağıdaki şekilde olacaktır.

```
Microsoft Visual Studio Debug Console
Private LogAspect is run
Common intercept method is run
Common intercept method is run
Product is added!
Private LogAspect is run
Common intercept method is run
Common intercept method is run
Product is deleted!
```

`Add()` metodu virtual olmadığı halde interceptorları çalışacaktır.

Fakat burada değinmemiz gereken önemli bir nokta mevcut.

`EnableInterfaceInterceptors()` metodu sadece interface ile gelen metotlara interceptor uygular. Bunu şu şekilde göstermek gerekirse.

```
public class ProductManager : IProductService
{
    [LogAspect]
    [CacheAspect]
    [ValidationAspect]
    public void Add()
    {
        TestDemo();
        Console.WriteLine("Product is added!");
    }

    [LogAspect]
    [CacheAspect]
    [ValidationAspect]
    public virtual void Delete()
    {
        Console.WriteLine("Product is deleted!");
    }

    [LogAspect]
    [CacheAspect]
    [ValidationAspect]
    public virtual void TestDemo()
    {
        Console.WriteLine("TestDemo is run");
    }
}
```

Burada uygulamayı çalıştırdığımız zaman `Add` metodu çalıştığında interceptorlar çalışacak fakat `TestDemo` metodu `IProductService` ile birlikte gelmediği için bu metodun interceptorları çalışmayacaktır.

Eğer EnableClassInterceptors() metodunu kullansaydık TestDemo üzerindeki interceptorlarda çalışacaktı

--EnableInterfaceInterceptors kullanıldığında

```
Microsoft Visual Studio Debug Console  
Private LogAspect is run  
Common intercept method is run  
Common intercept method is run  
TestDemo is run  
Product is added!  
Private LogAspect is run  
Common intercept method is run  
Common intercept method is run  
Product is deleted!
```

1-İlk önce Add metodunun interceptorları çalışır

2-Add metodu çalışır

3-Delete metodunun interceptorları çalışır.

4-Delete metodu çalışır

--EnableClassInterceptors kullanıldığında

```
Microsoft Visual Studio Debug Console  
Private LogAspect is run  
Common intercept method is run  
Common intercept method is run  
TestDemo is run  
Product is added!  
Private LogAspect is run  
Common intercept method is run  
Common intercept method is run  
Product is deleted!
```

1-Add metodu çalışır

2-TestDemo metodunun interceptorları çalışır

3-Test demo metodu çalışır

4-Delete metodunun interceptorları çalışır

5-Delete metodu çalışır

EnableClassInterceptors metodu

- Class seviyesinde interceptor uygular
- EnableClassInterceptors metodu ile işaretlenen sınıftaki sanal (virtual) metodlara interceptor uygular.
- Servise implement edilen interface'den gelen metodlara da interceptor uygular (virtual olma koşuluyla)
- Yani genel itibariyle sınıf ve sınıf içindeki tüm sanat metotlara interceptor uygular

EnableInterfaceInterceptors metodu

- Interface seviyesinde interceptor uygular
- EnableInterfaceInterceptors metodu ile işaretlenen sınıftaki sanal ve sanal olmayan metodlara interceptor uygular
- Yani genel itibariyle interface ve interface içindeki metodlara interceptor uygular.Sınıfın içindeki interface dışında oluşturulan metodlara interceptor uygulamaz.