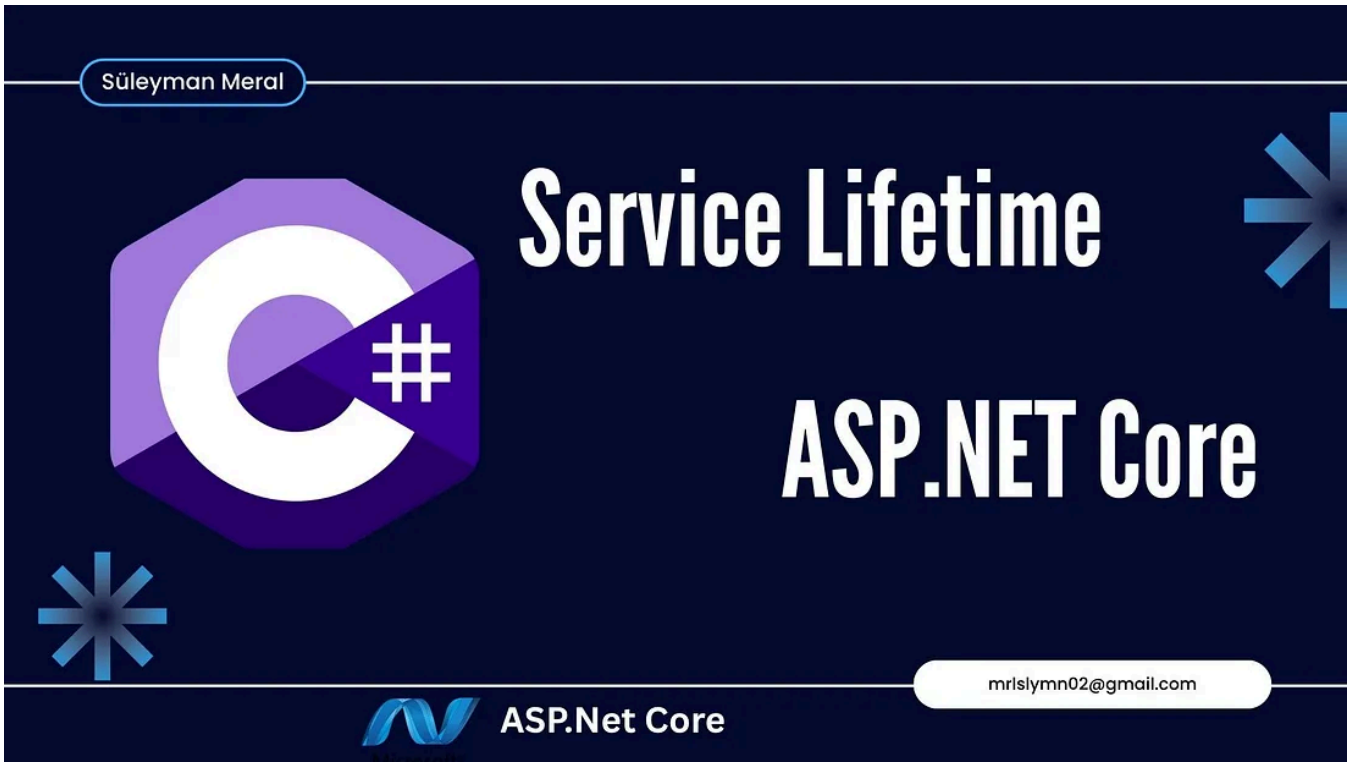
ASP.NET Core Service Lifetime

5 min read · Jun 27, 2025

Süleyman Meral

Bu yazımda, çoğumuzun günlük olarak kullandığı ama arka plandaki detaylarını pek de sorgulamadığı bir konuyu ele alacağım: **Service Lifetime** yani servis yaşam süresi.
Backend geliştiricileri olarak, servisleri `AddScoped`, `AddTransient` veya `AddSingleton` şeklinde kaydetmeye alışığız. Peki bu kavramlar gerçekte ne anlama geliyor? Uygulama içinde nasıl çalışıyorlar, hangi durumda hangisini tercih etmeliyiz? İşte bu yazıda, bu sorulara net ve anlaşılır cevaplar vereceğim.



Service Lifetime Nedir?

Service Lifetime, ASP.NET Core'da bir servisin (yani bir nesnenin) uygulama boyunca ne kadar süreyle yaşatılacağını belirleyen bir kavramdır. Üç temel türü vardır: **Transient** servisler her çağrıldığında yeni bir örnek oluşturur; **Scoped** servisler her HTTP isteği için bir kez oluşturulur ve istek süresince aynı örnek kullanılır; **Singleton** servisler ise uygulama çalıştığı sürece yalnızca bir kez oluşturulur ve her yerde aynı örnek paylaşılır. Bu yapı, bellek yönetimi ve uygulama davranışı açısından oldukça kritiktir.

Transient servisler içerisinde veri saklamayan, sadece gelen veriyi işleyip dönen servislerde daha uygundur. Bellek yönetimi performans açısından önemli olmayan servislerde kullanılabilir. Kısa ömürlü işlemlerde kullanılması önerilir. (Hesaplama,Doğrulama vs.) Eğer servis içinde **durum** tutuluyorsa (örneğin bir liste, sayaç vs.), her çağrıda yeniden yaratıldığı için bu veriler kaybolur. Bu durumda kullanılması önerilmez. Ayrıca ağır nesneler kullanılıyorsa bu servis tipini uygulamak performans sorunlarına yol açabilir.

Scoped servisler, her bir HTTP isteği (request) boyunca sadece bir kez oluşturulan ve aynı istek süresince tüm sınıflar arasında paylaşılan servis türüdür. Bu şu anlama gelir: bir kullanıcı sunucuya bir istek yaptığında, o isteğin tamamı boyunca scoped servis sadece bir kez oluşturulur ve o istek içinde bu servisi kullanan tüm sınıflar aynı nesne örneğini kullanır. Ancak başka bir HTTP isteği geldiğinde, bu servis yeniden oluşturulur. Scoped servisler genellikle, istek boyunca aynı veriye ihtiyaç duyan veya bir iş akışını izleyen servisler için uygundur. User işlemleri bu servis tipi için uygun olabilir.

Singleton servisler, uygulama çalıştığı sürece **yalnızca bir kez** oluşturulan ve tüm uygulama genelinde **aynı nesne örneği** ile kullanılan servis türüdür. Yani uygulama başlatıldığında singleton servis belleğe yüklenir ve tüm sınıflar, tüm istekler boyunca bu aynı nesne üzerinden işlem yapar. Bu, hem performans hem de paylaşılan durum yönetimi açısından büyük avantaj sağlar. Ancak dikkat edilmesi gereken nokta, singleton servis içinde tutulan tüm verilerin **uygulama genelinde paylaşıldığı** ve **thread-safe (eş zamanlı kullanıma uygun)** olması gerektiğidir.

Singleton servisler, özellikle **loglama**, **önbellekleme** ve **uygulama konfigürasyonu** gibi tüm sistem tarafından paylaşılan yapılar için en uygun servis yaşam süresidir. Örneğin, bir projede hata kayıtlarını loglamak için her seferinde yeni bir nesne oluşturmak, hem gereksiz bellek kullanımı yaratır hem de yönetimi zorlaştırır. Bunun yerine loglama servisini **singleton** olarak tanımlamak, hem kaynak

tüketimini azaltır hem de merkezi bir loglama mantığı sağlar. Bu yaklaşım, yazılım geliştirme dünyasında uzun süredir bilinen **Singleton Design Pattern** ile de birebir örtüşmektedir ve uygulama genelinde tek bir örneğin kullanılmasını garanti altına alır.

Şimdi bir API projesi oluşturalım ve bu 3 servis tipinin yaşam döngüsününün nasıl çalıştığını gözlemleyelim.

IServicelifetime adında bir interface oluşturuyorum ve içerisine Guid türünde bir Id alanı tanımlıyorum.

```
public interface IServiceLifetime
{
    Guid Id { get; }
}
```

`Guid` (Globally Unique Identifier), her oluşturulduğunda **neredeyse benzersiz** bir değer dönen bir yapıdır.

Daha sonra Scoped servis türünü tutması amacı ile IScopedService adında bir interface daha oluşturuyorum. Bu interface IServicelifetime interface'inden inherit alacak.

```
public interface IScopedService : IServiceLifetime { }
```

Transient ve Singleton servis tipleri içinde aynı yapıyı uyguluyorum.

```
public interface ISingletonService : IServiceLifetime { }
```

```
public interface ITransientService : IServiceLifetime { }
```

Bu interface'leri program.cs dosyamda ilgili servis tipleri ile kayıt edeceğim. Fakat bu interface'lerin implement edileceği LifeTimeService classını yazmamız gerek.

```
public class LifeTimeService : IScopedService, ITransientService, ISingletonSer
{
    public Guid Id { get; }

    public LifeTimeService()
    {
        Id = Guid.NewGuid();
    }

}
```

`Guid.NewGuid()` her çağrıldığında farklı bir değer üretir. Ve bu değer Id propuna atanır. Bu sayede biz bir sınıfın **ne zaman oluşturulduğunu ya da kaç farklı örnek üretildiğini** anlayabiliriz.

Şimdi ise bu servislerin kaydını gerçekleştirelim.

```
builder.Services.AddTransient<ITransientService,LifeTimeService>();
builder.Services.AddScoped<IScopedService,LifeTimeService>();
builder.Services.AddSingleton<ISingletonService,LifeTimeService>();
```

Görüldüğü üzere kayıtları ilgili servis isimlerinin eşleştiği tiplerle gerçekleştirdik. Şimdi bir controller oluşturalım ve servisleri controller classımıza enjekte edelim.

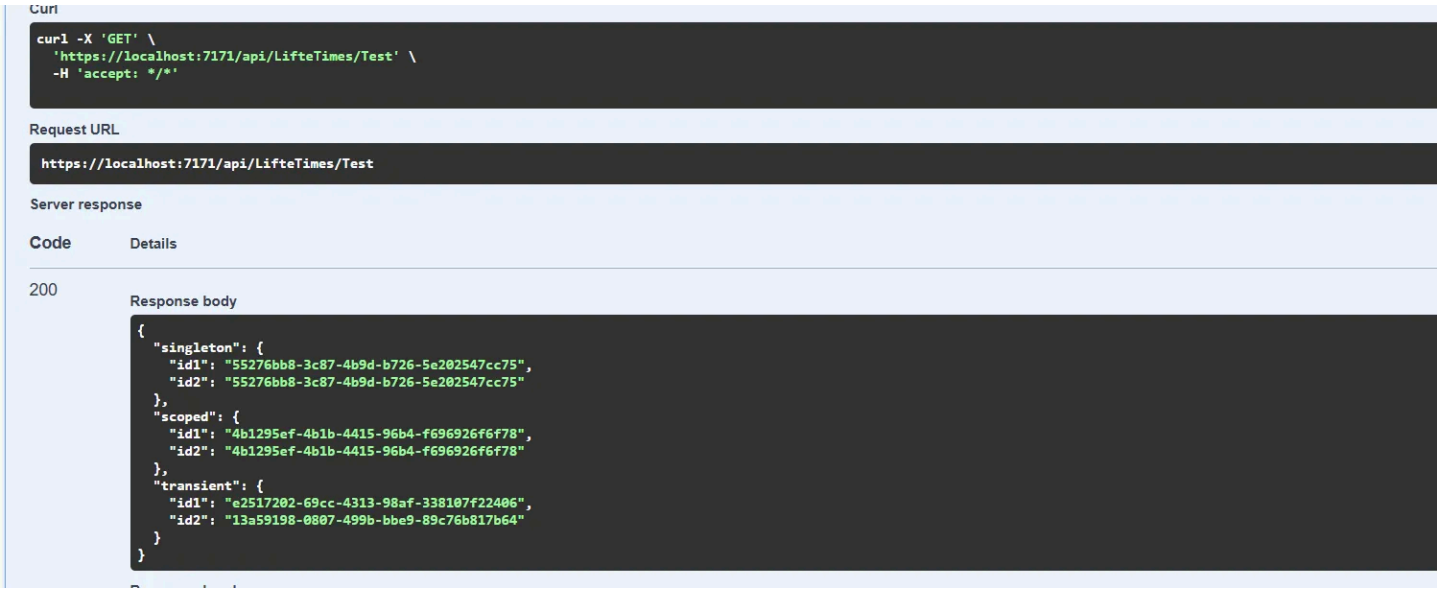
```
    public class LifteTimesController : ControllerBase
    {

        private readonly ISingletonService _singleton1;
        private readonly ISingletonService _singleton2;

        private readonly ITransientService _transientService1;
        private readonly ITransientService _transientService2;

        private readonly IScopedService _scopedService1;
```

```
        private readonly IScopedService _scopedService2;

        public LifteTimesController(ISingletonService singleton1,
            ISingletonService singleton2,
            ITransientService transientService1,
            ITransientService transientService2, IScopedService scopedService1,
            IScopedService scopedService2)
        {
            _singleton1 = singleton1;
            _singleton2 = singleton2;
            _transientService1 = transientService1;
            _transientService2 = transientService2;
            _scopedService1 = scopedService1;
            _scopedService2 = scopedService2;
        }


    }
}
```

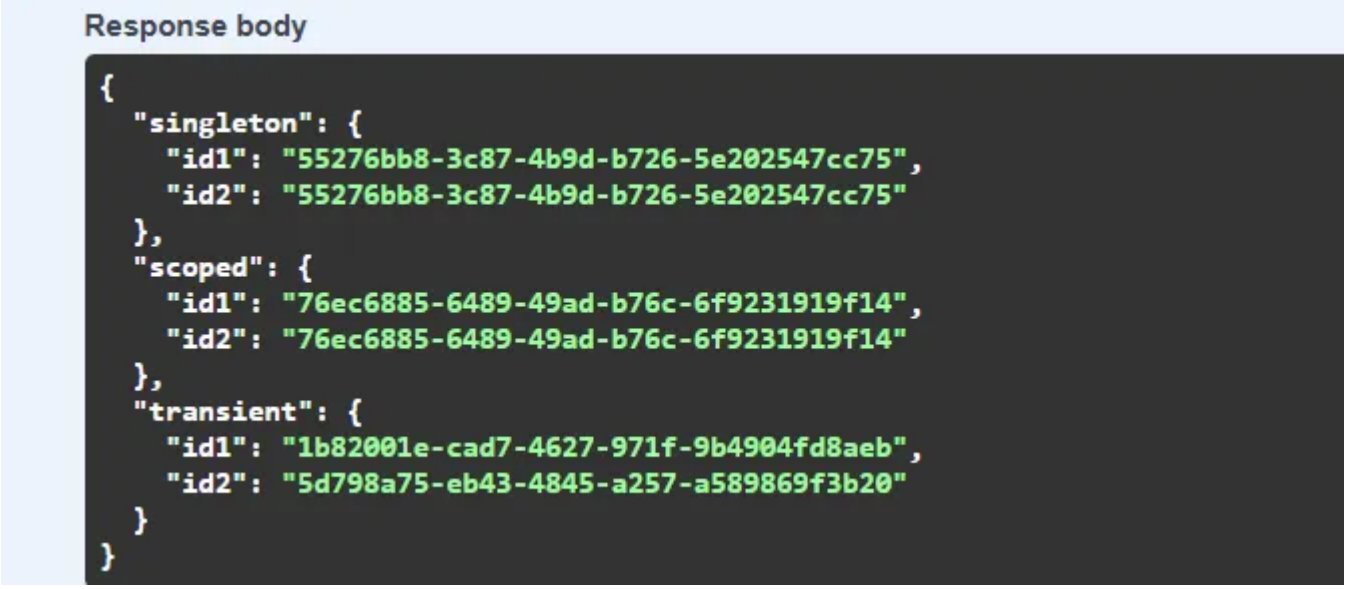
Oluşturulan örnekleri karşılaştırmak amacıyla her servis tipinden 2 örnek oluşturduk. Normal şartlar altında bir classın bu kadar bağımlılık barındırması doğru değildir. Fakat bizim amacımız yapıyı test etmek.

```
[HttpGet("Test")]
public IActionResult Get2()
{
    var result = new
    {
        Singleton = new { Id1 = _singleton1.Id, Id2 = _singleton2.Id },
        Scoped = new { Id1 = _scopedService1.Id, Id2 = _scopedService2.Id },
        Transient = new { Id1 = _transientService1.Id, Id2 = _transientService
    };

    return Ok(result);
}
```

Test etmek için bir result nesnesi oluşturalım ve dönen respons'u inceleyelim.



Gördüğümüz gibi singleton servisleri ve scoped servisleri tamamen aynı şekilde oluştu. Transient servisi ise nesne her çağrıldığında değiştiği için farklı şekilde oluştu. Aynı get isteğini tekrar çağırıp sonuca bakalım.



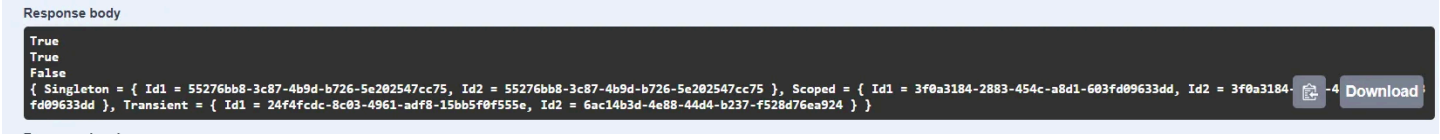
Singleton türü halen aynı. Çünkü uygulama sonlanana kadar bu şekilde tutulacak. Scoped türündeki servis get isteği tekrarlandığı için değişti. Fakat id1 ve id2 halen aynı şekilde oluşuyor. Transient ise yine farklı şekilde oluştu.

Farklı bir get metodu daha yazarak çıktıları gözlemleyelim.

```
        [HttpGet]
        public IActionResult Get()
```

```
{
    var result = new
    {
        Singleton = new { Id1 = _singleton1.Id, Id2 = _singleton2.Id },
        Scoped = new { Id1 = _scopedService1.Id, Id2 = _scopedService2.Id }
        Transient = new { Id1 = _transientService1.Id, Id2 = _transientServ
    };


    bool areSingletonEquals=ReferenceEquals(_singleton1 , _singleton2);
    bool areScopedEquals=ReferenceEquals(_scopedService1 , _scopedService2)
    bool areTransientEquals=ReferenceEquals(_transientService1 , _transient

    StringBuilder stringBuilder= new StringBuilder();
    stringBuilder.AppendLine(areScopedEquals.ToString());
    stringBuilder.AppendLine(areSingletonEquals.ToString());
    stringBuilder.AppendLine(areTransientEquals.ToString());

    stringBuilder.AppendLine(result.ToString());




    return Ok(stringBuilder.ToString());
}
```

Bu metodumuz diğerinden farklı olarak oluşturulan nesnelerin aynı referanstan gelip gelmediğini de karşılaştırıyor. Kısacası aynı nesneler mi çalışıyor bunu kontrol ediyor. Şimdi bunun çıktısını gözlemleyelim.



Scoped ve Singleton servis türlerinin aynı nesneyi kullandığını gördük. Transient ise 2 farklı nesneyi kullanıyor. Çıktı da gördüğümüz gibi singleton servis türünün id'si hala aynı. Uygulama sonlanana kadar bu devam edecek. Projemizi sonlandırıp tekrar ayağa kaldıralım ve sonuca bakalım.

Response body


```
{
  "singleton": {
    "id1": "75d3637a-e05e-446e-a102-856e1d1f68cf",
    "id2": "75d3637a-e05e-446e-a102-856e1d1f68cf"
  },
  "scoped": {
    "id1": "07eac00c-e304-47b9-aa4d-ede79ca23287",
    "id2": "07eac00c-e304-47b9-aa4d-ede79ca23287"
  },
  "transient": {
    "id1": "ec616604-f41d-40b5-a412-3e9ab3267b06",
    "id2": "5994b602-10a6-4b5b-bde0-e561d1dc019b"
  }
}
```


Görüldüğü üzere bu sefer singleton farklı id ile oluştu. Çünkü bu servis türü ancak ve ancak uygulama sonlandığında yeni nesne oluşturur.

Service Lifetime (Servis Yaşam döngüsü) konusunu kendimce ele almaya çalıştım. Umarım faydalı olmuştur. İlgili konu ile alakalı Youtube kanalımda da bir video mevcut dilerseniz onu da izleyebilirsiniz. Linki aşağıda bulabilirsiniz. Herkese Kolay gelsin !

Süleyman Meral

Software Engineering Student / Backend Developer