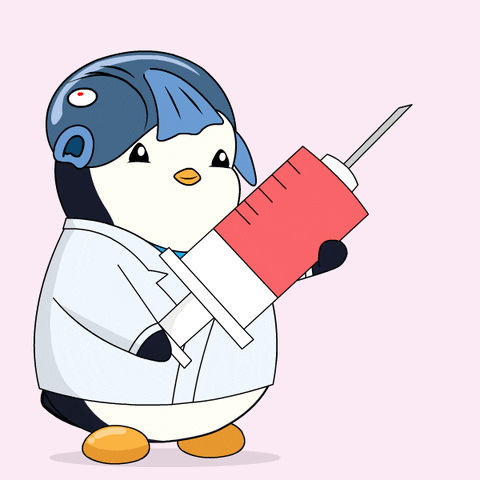
**Dependency Injection Yaşam Döngüleri: Singleton, Scoped ve Transient Nedir?**

Bu yazımda, Dependency Injection yaşam döngüleri olan **Singleton**, **Scoped** ve **Transient** arasındaki farklardan bahsedeceğim.

Dependency Injection, yazılım geliştirme süreçlerinde uygulama bileşenlerinin bağımlılıklarını yönetmek için kullanılan önemli bir yöntemdir. Bir bileşenin ihtiyacı olan bağımlılığını doğrudan oluşturmak yerine, bu bağımlılığı dışarıdan almasını sağlar.

Örneğin, bir projede mail gönderme işlevine ihtiyacımız olduğunu düşünelim. Bu iş için bir ‘EmailService’ sınıfımız var ve bu sınıf, mail göndermek için bir mail protokolü (SMTP) kullanıyor. Eğer projede Dependency Injection kullanmazsak, EmailService sınıfının içinde doğrudan bir ‘SmtpClient’ oluşturabiliriz. Ancak bu yaklaşım, sınıfımızın doğrudan SmtpClient'a bağımlı olmasına neden olur. Bu da, farklı bir mail gönderme yöntemi kullanmak istediğimizde veya birim testleri yazarken bu bağımlılığı değiştirmemiz gerektiğinde bizi zor durumda bırakabilir.

İşte bu tür sorunlardan kaçınmak için Dependency Injection kullanmak daha doğru bir yaklaşımdır. Dependency Injection kullanarak mail gönderme işlemini gerçekleştirmek için önce bir ‘IEmailSender’ interface’i oluştururuz. Ardından, ‘SmtpEmailSender’ adında bir sınıf oluşturur ve bu arayüzü bu sınıfın içine implemente ederiz. Artık EMailService sınıfımız, doğrudan bir SmtpClient örneği oluşturmak yerine IEmailSender arayüzüne bağımlı hale gelir ve bu bağımlılığı dışarıdan alır. Bu şekilde, daha esnek, test edilebilir ve sürdürülebilir bir yapı kurmuş oluruz.



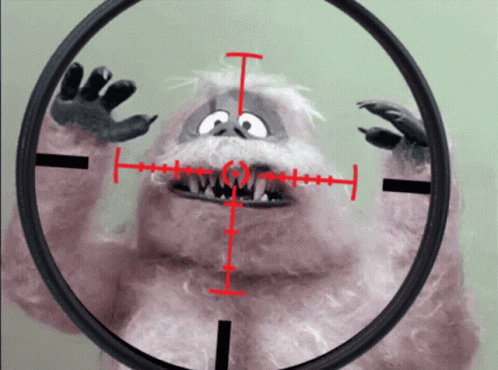
Dependency Injection’ı anladıysak sıra yaşam döngülerine geldi. Yaşam döngüleri denilince aklımıza 3 kavram geliyor: Singleton, Scoped ve Transient.

**Singleton:**Eğer bir servisi singleton olarak tanımlarsak uygulama ayağa kalktığında bir kez oluşur ve uygulamanın ömrü boyunca aynı referans üzerinden kullanılır. Peki servislerimizi hangi durumlarda singleton yapmalıyız, biraz da bunun üzerinde duralım:

* Eğer bir servis ilk oluşturulduğunda çok fazla kaynak (bellek, CPU, vb.) kullanıyorsa, bu servisi yalnızca bir kez oluşturup uygulama boyunca yeniden kullanmak performans açısından faydalı olacaktır. Örnek olarak büyük konfigürasyon dosyalarını yükleyen servisler verilebilir.
* Uygulama genelinde bir kez ayarlanması ve her yerden erişilmesi gereken yapılandırmalar veya ayarlar varsa, bu durumlar için de singleton tanımı idealdir. Örneğin, uygulama genelinde kullanılan konfigürasyonları tutan bir servis singleton tanımlanabilir.
* Uygulama boyunca paylaşılması ve tek bir örnekte saklanması gereken önbellek yapıları için singleton kullanmak en doğrusudur. Örneğin, bir ‘MemoryCache’ veya ‘RedisCache’ gibi bir önbellek yöneticisi, uygulamanın farklı bölümlerinde tekrar tekrar kullanılabilir ve tek bir örnek ile yönetilebilir.
* Eğer bir servis veya nesne, çalışmadan önce bir yapılandırma veya ilk ayar gerektiriyorsa ve bu ayar bir kez yapıldıktan sonra değişmeyecekse, singleton kullanmak mantıklıdır. Örneğin, bir şifreleme (encryption) servisi genellikle ilk yapılandırma sonrasında değişmez.

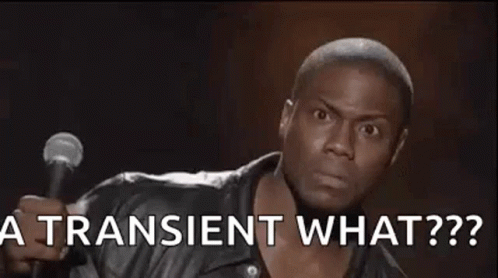
**Scoped:**Eğer bir servisten HTTP isteği boyunca bir kez oluşturmak istiyorsak scoped yaşam döngüsünü kullanırız. HTTP isteği sonlandığında ise oluşturulan servis örneğinin de ömrünün sonu gelmiş olur. Kısacası her istekte servis örneği değişir. Scoped yaşam döngüsünün de kullanıldığı bazı durumlar vardır, işte bunlara birkaç örnek:

* Her HTTP isteği için ayrı bir örneğe ihtiyaç duyan servislerden biri ‘DbContext’ olabilir. Scoped yaşam döngüsü, aynı istekte yapılan tüm veritabanı işlemlerinin aynı DbContext örneğini kullanması ve veritabanı işlemlerinin düzgün bir şekilde yönetilmesi için uygundur.
* Yetkilendirme ve oturum yönetimi için kullanılan servisler scoped tanımlanmalıdır.
* Eğer bir servisin, her istek için ayrı bir logging veya rate limiting işlemi yapması gerekiyorsa, scoped servisler kullanılabilir.



**Transient:**Bu yaşam döngüsüyle tanımlanan servisler ne zaman talep edilirse edilsin her koşulda yeniden oluşturulur. Bu nedenle hafif işlemler için idealdir. Performansı arttırır. Ancak kaynak tüketimi yüksek olan servisler için uygulamayı cevap veremez hale getirebilir.

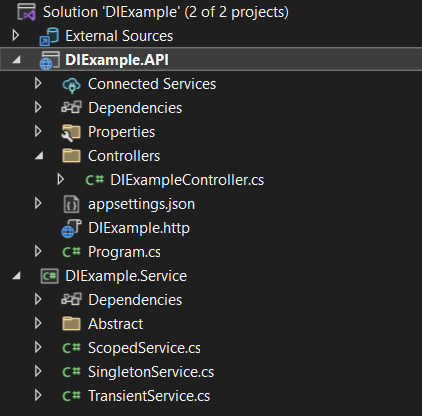
* Örnek olarak basit hesaplamalar yapan servisler, rastgele kod üreten servisler verilebilir.



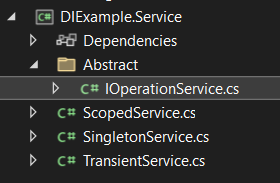
Bence yaşam döngülerinin ne işe yaradığını teoride anladık. Ama biraz da koda dökelim ki iki güne uçup gitmesin biraz kalıcı olsun 🤓

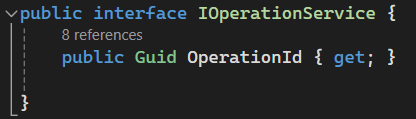
**.Net 8 ile Yaşam Döngüleri Projesi**

İlk olarak .Net 8.0 ile bir API projesi ve servisler için ayrı bir katman oluşturuyoruz. Servis katmanındaki servislerimizi API projesinde kullanabilmemiz için servis katmanını API projemize dahil etmeliyiz.

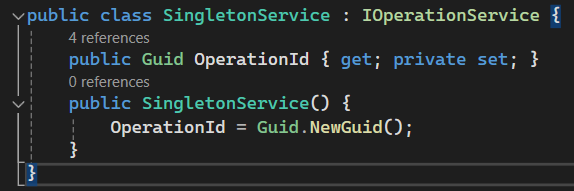


Proje mimarimizi oluşturduktan sonra servislerimizi ve interfaceimizi oluşturuyoruz.

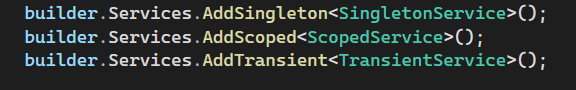




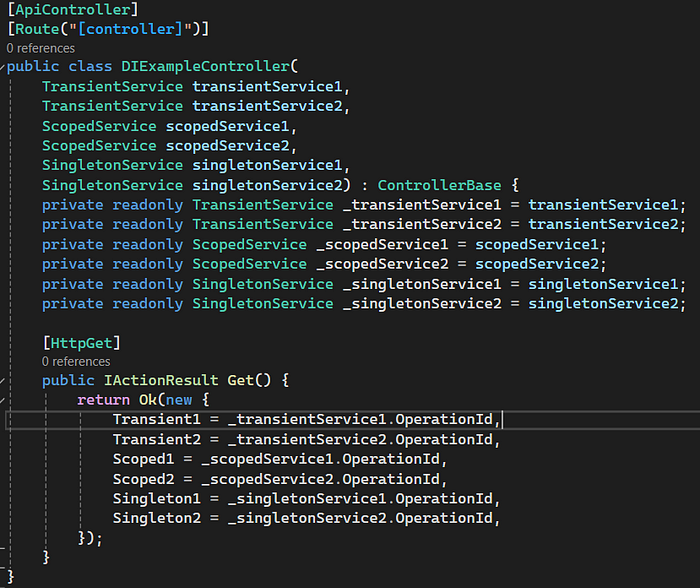
Soyut sınıfa Guid property ekliyoruz. Servislerin içinde constructor oluşturuyoruz ve OperationId’ye değer atama işlemini gerçekleştiriyoruz.



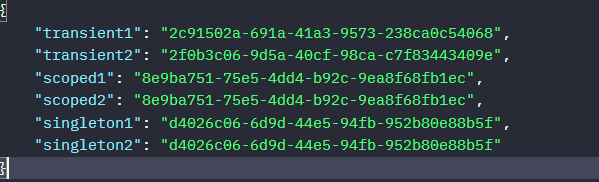
Daha sonra Program.cs dosyasında oluşturulan servislerin tanımlamalarını yapıyoruz.



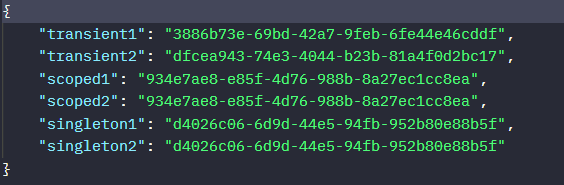
Son olarak da API içinde bir controller oluşturarak istek atacağımız endpointi yazıyoruz.



Şimdi projemizi çalıştırıp endpointe istek atalım. İlk olarak postman üzerinden istek atıyoruz. İstekten dönen cevap:

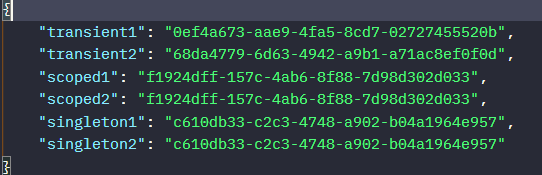


Görselden de anlaşılacağı gibi ilk istekte transient yaşam döngüsüne sahip nesnemizdeki Id’ler farklı iken, scoped ve singleton için oluşturulan nesneler aynı. Uygulama hala ayakta iken bir istek daha atalım:



Görüldüğü üzere, 2. isteğimizde singleton yaşam döngüsüne sahip nesnemiz hala aynı iken, scoped ve transient için oluşturulan nesnelerin Id değerleri değişti. Burdan anlıyoruz ki uygulamay yaşam döngüsü boyunca kaç defa çağırırsak çağıralım singleton nesnesi değişmeyecek. Ancak scoped yaşam döngüsüne sahip olan nesnelerin aynı istekte aynı Id’ye sahipken farklı isteklerde farklı değerlere sahip olduğunu anlıyoruz.

Son olarak da uygulamayı durdurup tekrar başlatalım ve çıkan sonuca bakalım:



Uygulamayı tekrar çalıştırıp istek attığımızda singleton yaşam döngüsüne sahip nesnemizin Id değerinin de değiştiğini gözlemliyoruz. Bu da demek oluyor ki singleton yaşam döngülü nesneler uygulama süresince aynı değere sahip, ancak uygulama durdurulup tekrar başlatıldığında değeri değişir.

Artık yaşam döngülerini anladığımızı düşünüyorum 💣 Umarım sizler için de öğretici bir yazı olmuştur 🤗

Eğer .Net 8 projesiyle oluşturduğum örneğe erişmek isterseniz buyurunuz: <https://github.com/sudeasena/DIExample>