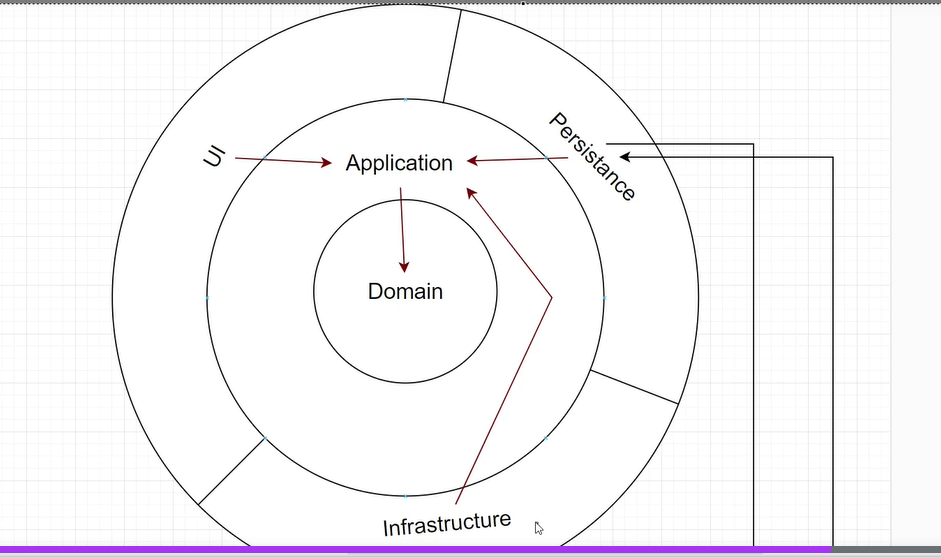
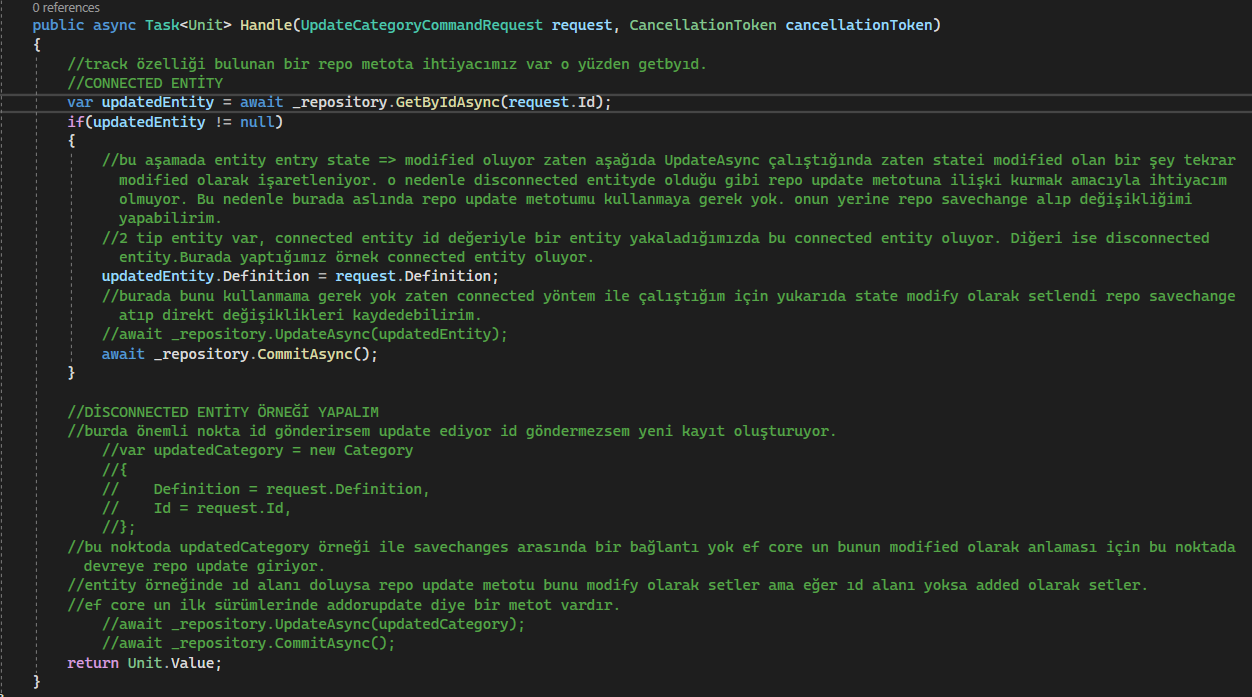
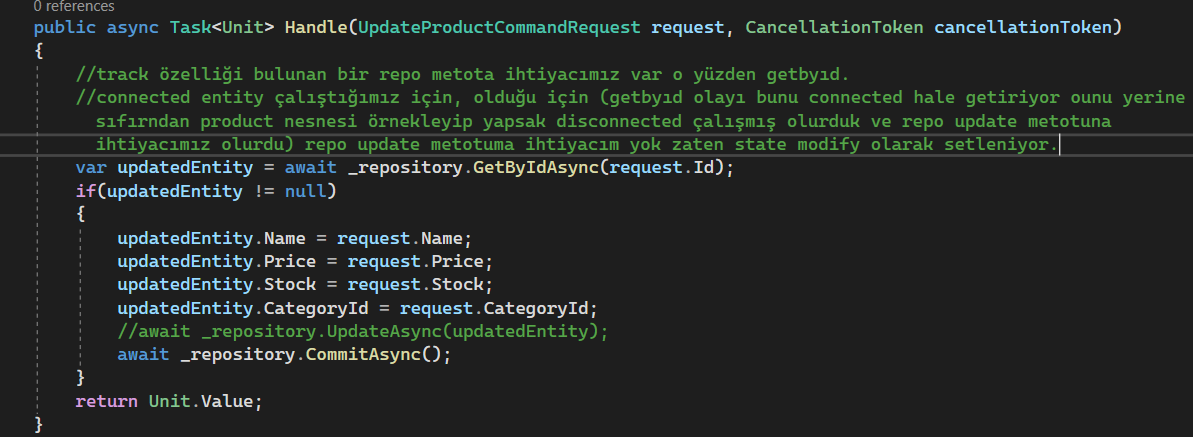
**ONİON ARCH**

* Gelenekselleşmiş N katmanlı mimarinin en kötü yanı Data katmanına sıkı sıkıya bağlıyız. Datayı değiştirmeye kalktığımızda bu datayla ilgili datayı referans alan ne kadar nokta varsa o kadar noktayı değiştirme durumunda kalabiliriz.
* Diğer bir kötü yön katmanlar sıkı sıkıya birbirine bağlılardır. Akış doğrusaldır. Onion ile bu katmanalar arasındaki bağlılığı düşürmeye çalışıyoruz aslında.
* Onion da aslında totalde yine 4 adet katman bulunuyor. UI, Persistence, Infrastructure, Core olmak üzere 4 katman var.
* Onionda katmanlar içe doğru birbirine bağımlılardır. Persistence applicationu görüyor application domaini, UI applicationu, application domaini görüyor. Infrastructure applicationu görüyor, Application da domaini. UI, Persistence, Infrastructure katmanları eş seviyedeler. Bunlar birbirlerini görmemeliler.
* Persistence dediğimiz katman, aslında n katmanlı mimarideki data katmanıdır. En dışarıdadır.
* Onionda Domain merkezdir. Domain bizim veri tabanındaki entitylerimizdir. Veya value objeler.
* Infrastructure siz dışarıdan bir servis tüketmek istediniz mesela bu payment service veya hava durumu API veya bir mail göndermek isteyebiliriz bu gibi işleri bu katmanda yapıyoruz.
* Application aslında n katmanlı mimarideki business katmanıdır. Ancak olayı bu değil. Infrastructure ve Persistence ın interfaceleri de bu katmanda tanımlanır. Bu nedenle aslında kritik nokta burası. Persistence Applicationu görüyor, Infrastructure da Applicationu görüyor ikisinin interfaceleri de burada o zaman ben Infrastructure dan Persistence ın interfacesini kullanabilirim işte mesele de burada. İnterfaceleri Application da DI CONTAİNER denilen bir konteynır içinde bulundururuz tüm bunlar bu sayede gerçekleşir.

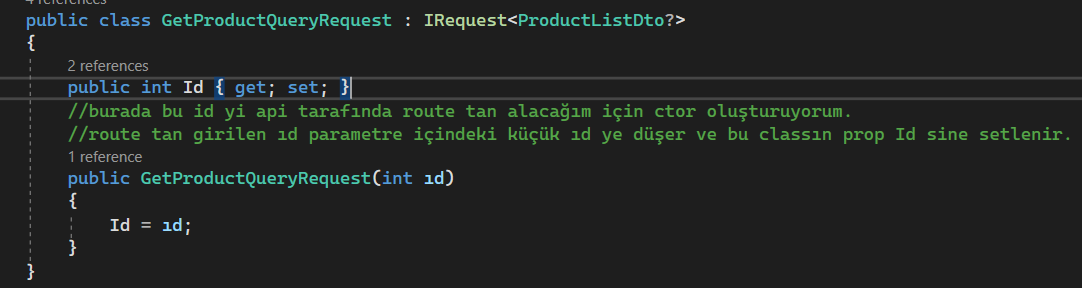


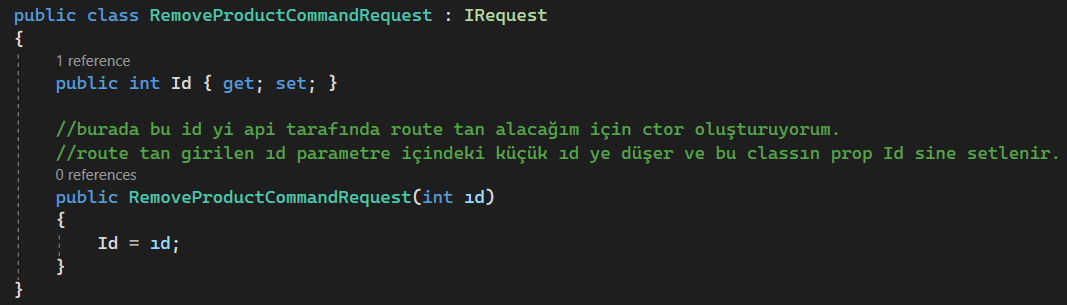
**CONNECTED-DİSCONNECTED ENTİTY**

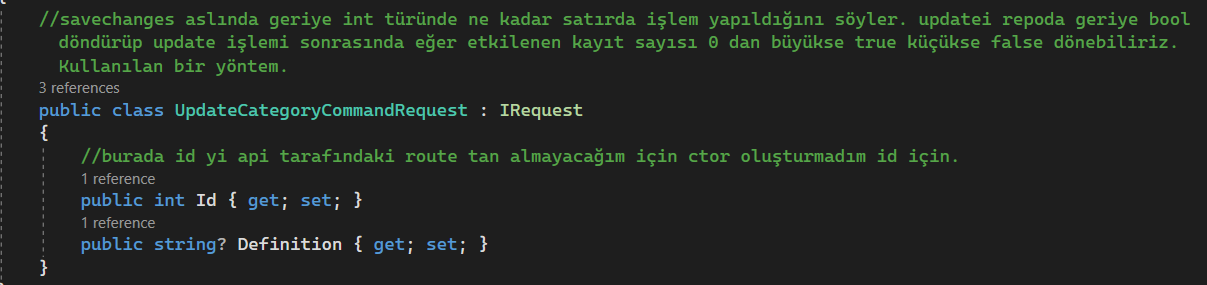




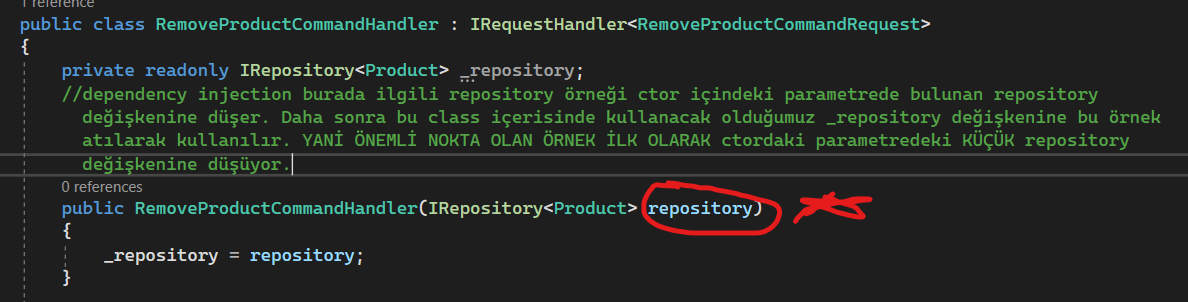
**KAFA KARIŞTIRAN ID CTOR OLUŞTUMA OLAYI NEYE GÖRE**







**DEPENDENCY İNJECTİONUN EN CAN ALICI NOKTASI**

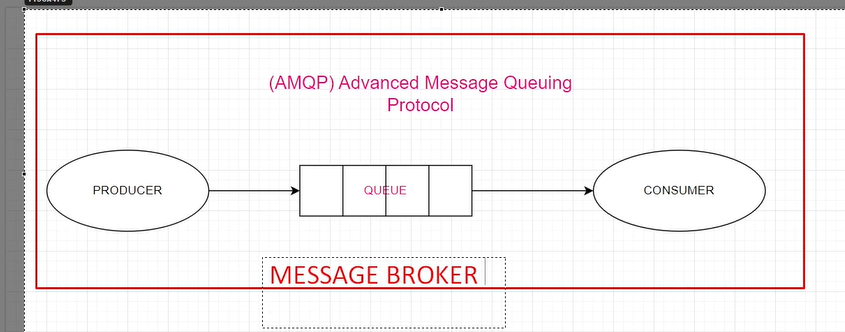


**BAZI ÖNEMLİ BİLGİLER ONİON BÖLÜMÜNDEN**

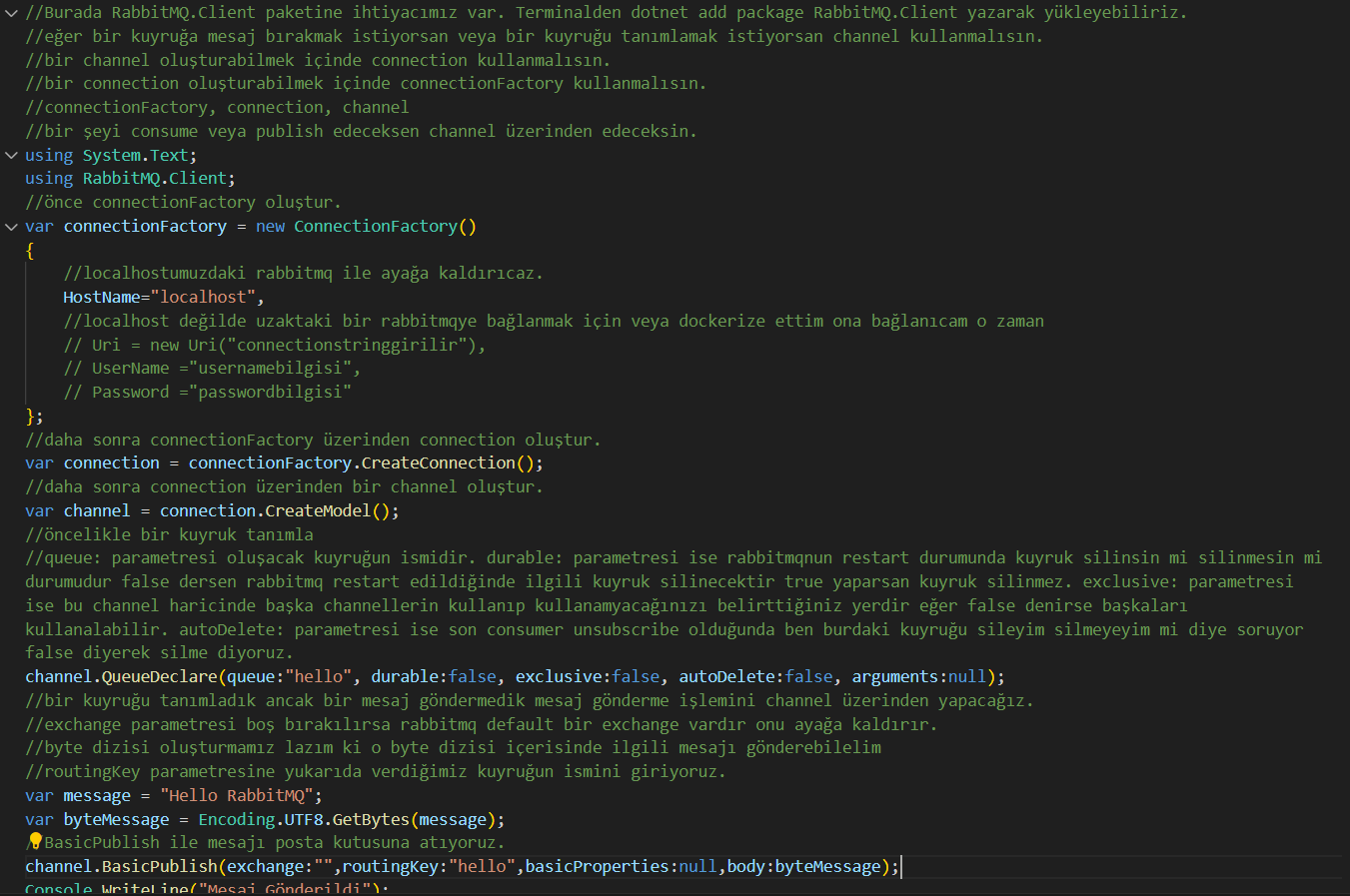
* //remove işleminde direkt olarak bir kaydı silmek mantıklı değildir. Yapılması gereken aslında delete işlemi sırasında bir update yaparak bool olacak olan isdeleted prop alanını kolonunu false şeklinde değiştirmektir. Çünkü veriler çok önemlidir direkt olarak silmek mantıklı değildir. Bu şekilde bir yaklaşım daha faydalıdır.
* //savechanges aslında geriye int türünde ne kadar satırda işlem yapıldığını söyler. updatei repoda geriye bool döndürüp update işlemi sonrasında eğer etkilenen kayıt sayısı 0 dan büyükse true küçükse false dönebiliriz. Kullanılan bir yöntem.
* //create işlemi sırasında genellikle client tarafından create olan kayıdın idsi veya başka bir bilgisi istenir biz id sini geriye döneceğiz. Bu zorunlu değildir geriye bir şeyde döndürmeyebilirsin isteğe göre şekillenen bir durum
* //MediatR pattern hangi querye karşılık hangi handler ı çalıştıracağımızı interfaceler aracılığıyla belirleyen patterndır.
* //bir rolün birden fazla userı olabilir. builder.HasOne(x => x.AppRole).WithMany(x => x.AppUsers).HasForeignKey(x => x.AppRoleId).OnDelete(DeleteBehavior.NoAction); //OnDelete(DeleteBehavior.NoAction) yaparak bir rol silinirse o role sahip tüm kullanıcıların silinmesinden kurtuluruz. Veriler değerlidir silinmesini istemeyiz bu çok önemli o yüzden.
* PROGRAM CS: //bunu buraya ekleme sebebeimiz AddPersistenceServicesı DI Conteinera register edebilmek. //tüm ServiceRegistrationlarımın register işlemlerini burada yapmalıyım. // ctrl + k + s ile böyle bir region attım karışmasın diye bunun etkisi yok tamamen görüntü. #region Service Registration builder.Services.AddApplicationServices(); builder.Services.AddPersistenceServices(builder.Configuration); #endregion
* //track özelliği bulunan bir repo metota ihtiyacımız var o yüzden getbyıd.
* //burda entityin statei added olarak setliyor. Savechangesi çağırdığında da stateine bakıyor eğer added ise gidiyor ilgili ekleme sql sorgusunu çalıştırıyor.
* //ef core un tracking özelliğini sadece update ve remove işleminde isteriz burası önemli.
* //update veya remove işlemi için bunu kullanıcaz. bu yüzden tracking özelliğini bırakmamız önemli getbyıd repo.
* MediatR ın görevi query yada commend requeste karşılık hangi handlerın çağıracağını belirlemek tabi interfaceler aracılığıyla amacı bu.

RABBİTMQ

* Rabbitmq bir postane bir posta kutusu ve bir mektup taşıyıcısıdır. Bu üçünün toplamıdır. Bir mektubu bir posta kutusuna bıraktığınızda o mektubun bir şekilde alıcıya ulaşacağını biliyorsunuz ve bu süreçle ilgilenmiyorsunuz işte rabbitmq böyle çalışır aslında. Bu tanımla rabbitmq arasındaki tek fark rabbitmqnun kağıt ve kalemle değil byte dizileriyle ilgileniyor oluşu. Yani posta kutusuna bir mektup, yazılı bir kağıt değil byte dizisi bırakıcaz. Nasıl bir web uygulamasında http gibi protokoller kullanabiliyoruz aynı şekilde rabbitmq nun da kullandığı bir protokol var bu protokol AMQP (Advanced Message Queuing Protocol). Rabbitmq da belirli jargonlar var birisi Producer kavramı, Queue kavramı, Consumer kavramı. Burada Producerdan kasıt üretici anlamında değil, sender yani gönderici anlamında. Consumerdan kasıt ise tüketici değil dinleyici anlamındadır. Queue Kuyruk kavramı ise az önceki örnekteki posta kutusu aslında. Kuyruk kavramında ilk giren ilk çıkar mantığı var. Tüm bu olaya MESSAGE BROKER denmektedir.



* RabbitMQ ile mesaj göndermek



* RabbitMQ ile gönderilen mesajı okumak

