**Sepet alt yapısının oluştulması.**

Genelde bir çok e-ticaret uygulaması sepetler ile ilgişi işlemleri yaparken iki farklı yöntem kullanıyor

* İlk yöntem kullanıcılarının sepete etklemiş olduğu ürünlerin local storage altında tutulması .
* ikinci yöntem ise bizim burada backend için kullanacağımız yöntem olan işlem sepete eklenen ürünlerin veri tabanında tutulması.

Şimdi burada nasıl avantajlar nasıl dezavantajlar var bunlara bi göz atalım.

**İlk yöntem**: Kullanıcıların sepete ekledikleri ürünlerin local storage altında tutulması

**Avantajları**:

Performans: Local storage, tarayıcı tarafında çalıştığı için sunucu ile herhangi bir etkileşime gerek duymaz. Bu, kullanıcıların hızlı bir şekilde sepetlerini yönetmelerine olanak tanır ve sunucu yükünü azaltır.

Çevrimdışı Kullanım: Kullanıcılar internet bağlantısı olmadan da sepetlerini görüntüleyebilir ve düzenleyebilir.

Kullanıcı Deneyimi: Sayfa yeniden yüklenmelerinden kaçınılabilir, bu da daha sorunsuz bir alışveriş deneyimi sağlar.

**Dezavantajları**:

Güvenlik: Local storage, tarayıcı tarafında tutulduğu için güvenlik zafiyeti riski taşır. Bu, kullanıcılar ve geliştiriciler için güvenlik endişelerine yol açabilir.

Kullanıcı Bağlı: Kullanıcı, farklı bir cihaz veya tarayıcıda alışveriş yaparsa sepet bilgilerini kaybedebilir.

Veri Tutma Sınırlamaları: Tarayıcılar genellikle belirli bir depolama sınırına sahiptir, bu nedenle büyük sepet verileri depolama sınırlarını aşabilir.

**İkinci yöntem**: Sepete eklenen ürünlerin veritabanında tutulması

**Avantajları**:

Veri Bütünlüğü: Veritabanı, verilerin daha güvenli bir şekilde saklanmasını sağlar ve veri bütünlüğünü korur.

Çoklu Cihaz Desteği: Kullanıcılar, farklı cihazlarda ve tarayıcılarda sepet bilgilerine erişebilirler.

Analiz ve İzleme: Veritabanı üzerinde tutulan veriler, analiz ve raporlama için kullanılabilir, bu da işletmelere kullanıcı davranışları hakkında daha fazla bilgi sağlar.

**Dezavantajları**:

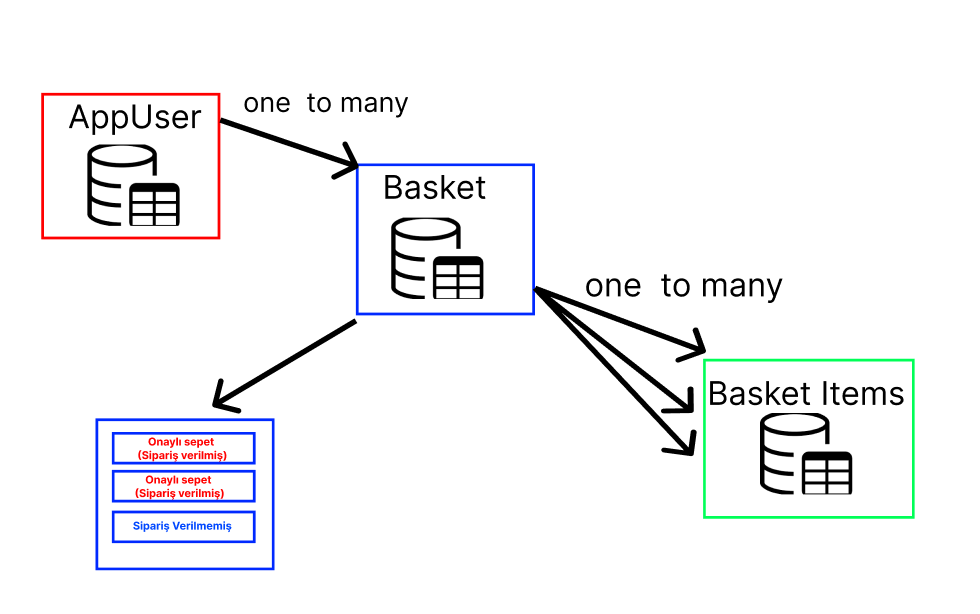
Performans: Her sepet işlemi, sunucu ile etkileşim gerektirir. Yoğun trafik durumlarında veya yavaş internet bağlantılarıyla bu performans sorunlarına neden olabilir.

Çevrimdışı Kullanım: İnternet bağlantısı olmadan sepete erişim mümkün olmayabilir.

Geliştirme Süreci: Bu yöntem, sunucu taraflı bir işlem olduğu için daha fazla geliştirme ve bakım gerektirebilir.

Her iki yöntemin avantajları ve dezavantajları, projenin gereksinimlerine ve kullanıcı deneyimi hedeflerine bağlı olarak değerlendirilmelidir. Örneğin, çevrimdışı kullanım önemliyse veya sunucu yükünü minimize etmek isteniyorsa, ilk yöntem daha uygun olabilir. Veri bütünlüğü ve çoklu cihaz desteği öncelikli ise, ikinci yöntem tercih edilebilir.

Şimdi biz nasıl bir yöntem izleyeceğiz ona karar verelim aslında iki yöntemide kullanmak istiyoruz kullanıcı kayıt olmamış ise ve giriş yapmamış ise o sepete eklemiş olduğu ürünleri local storage içinde tutacağız.   
Daha sonra eğer giriş yapmış ise sepete ekleme işlemlerini veri tabanı üzerinden yapacağız.  
Bu döküman backend dökümanı olduğu için burada sadece backend üzerinde nasıl bir yöntem izleyeceğimizden bahsedeceğim.



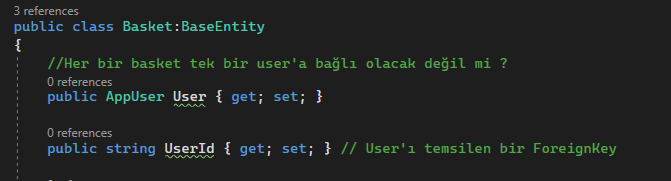
Şimdi burada gerekli düzenlemelere başlamadan önce ilk olarak nasıl bir veri tabanı design sistemi kullanacağız veri tabanımızda kullandığımız bu yöntem bize neler sağlayacak onu inceleyelim.

* Kullanıcıların sepetleri olacak bu sepetler için sipariş verdiyse sipariş verildi olarak gözükmesini sağlayacağız.
* Eğer kullanıcı sepete ürün ekleyip sepet üzerinde ürünler eklemiş 5 ay sonra tekrar sisteme giriş yaparsa bu sepeti tekrar görmesini sağlayacağız.
* Belki kayıtlı sepet oluşturmasını isteyebiliriz bu şekilde bir geliştirme yapabiliriz.

Şimdi burada neden 3. bir tabloya ihtiyacımız var diye düşünelim.   
İlerleyen süreçte kullanıcı farklı bir ürün almak istedi diyelim yeni bir sepet oluşturup ürün eklemek istediği zaman başka sepettede bunlar var bunları eklemek istermisiniz gibi geliştirmeler yapmamız için gerekli.

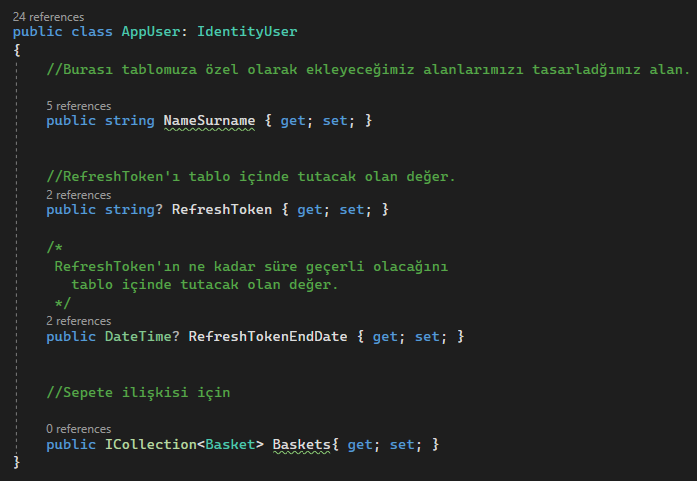
Şimdi öncelikle bir sepet Entity’si oluşturarak işlemlerimize başlayalım.

**Domain katmanında** bulunan **Entities** dosyamızın içine **yeni bir class oluşturalım adını Basket olarak ayarlayalım**.



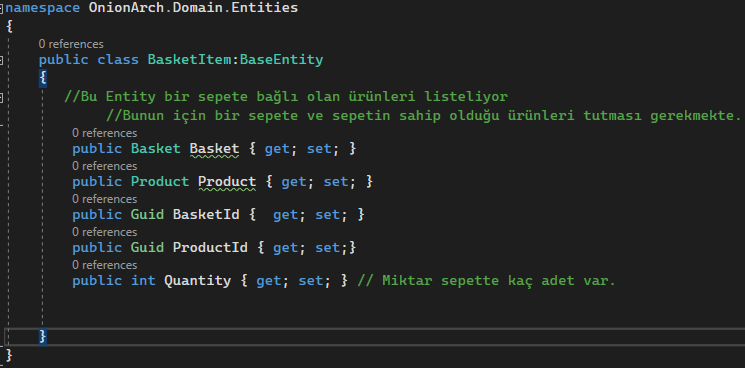
İlişki bütünlüğünü sağlayabilmemiz için ,   
 **User tablosunda ‘da bu oluşturduğumuz Basket tablosuna sahip bilgileri tutmamız gerekir değil mi ?**Şimdi bu işlemi gerçekleştirelim.

| **ICollection** genellikle, bir nesnenin başka bir nesne koleksiyonunu temsil etmek için kullanılır. Özellikle, birincil bir nesnenin başka bir nesne türündeki bir liste, küme veya koleksiyonu içerdiği durumları ifade etmek için kullanılır. Bu durumlar genellikle ilişki adı verilen bir bağlamda gerçekleşir. |
| --- |



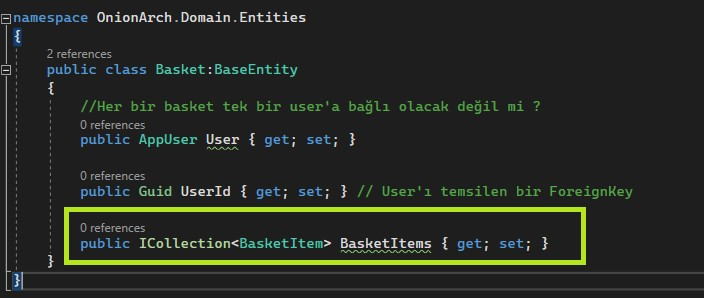
Şimdi oluşturduğumuz tasarıma göre başka neye ihtiyacımız var bizim Basket için **BasketItem** değelerini tutan bir adet başka bir tabloya ihtiyacımız var değil mi ?

Hemen beraber onuda oluşturalım.

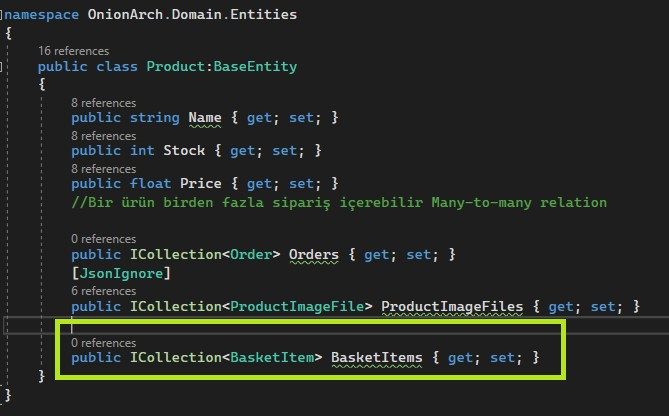


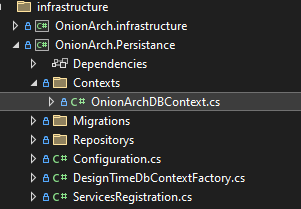
Şimdi burada yapmış olduğumuz eklemelere göre genel bir Entity düzenlemeleri yapalım.  
Öncelikle Basket içinde düzenlemelerimizi oluşturalım.

| Her bir Basket’in birden fazla Basketitem’i olacak. |
| --- |



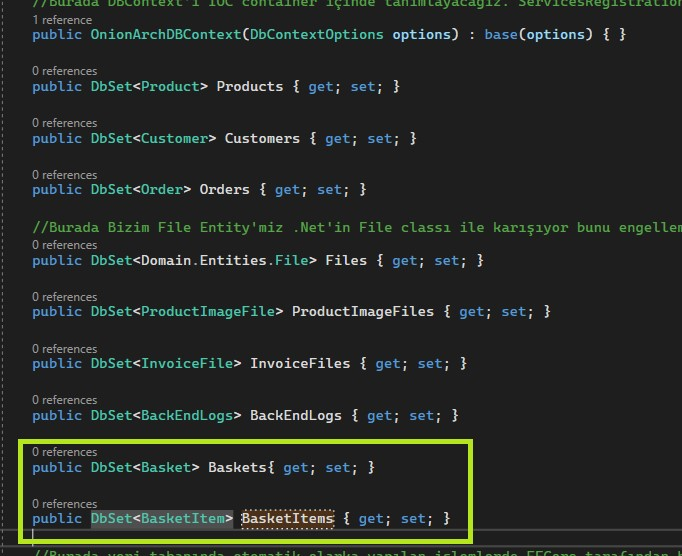
| Her bir Ürün birden fazla BasketItem olabilecek. |
| --- |





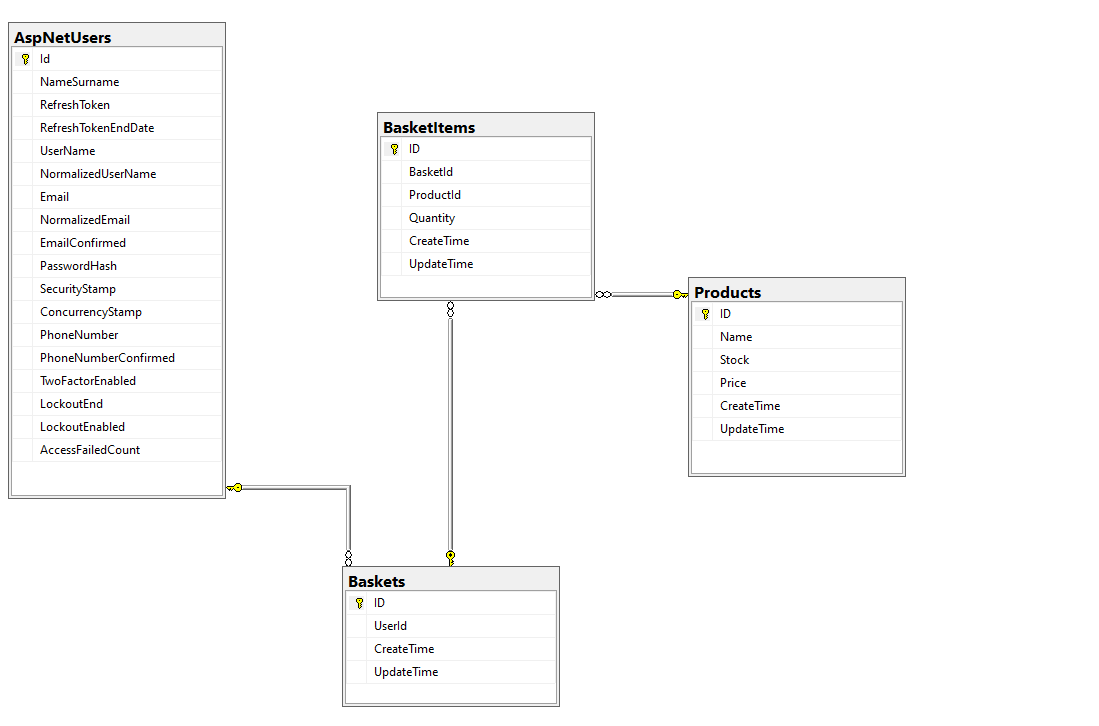
Şimdi **gerekli ilişkileri kurduk** değil mi artık **DBContex** üzerinden **güncelleme yapmamız** gerekmekte.

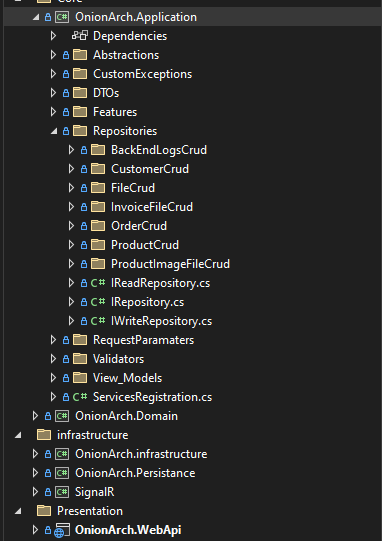
Bunun için bizim **infrastructure** altında bulunan **Context dosyamıza erişip gerekli tanımlamayı yapmamız gerekmekte.**

****

**Tüm gerekli güncellemeleri yaptık hemen şimdi bir adet migration almamız ve veri tabanında update işlemini gerçekleştirmemiz gerekmekte.**

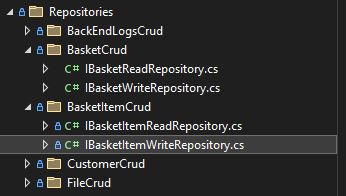
Veri tabanımızı oluşturduk. Şimdi bir kontrol edelim bakalım istediğimiz gibi sonuçlar elde edebildik mi .



Evet bütün sonuçlar istediğimiz gibi.  
  


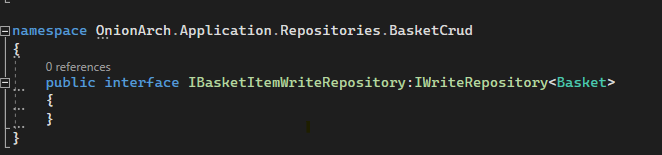
Şimdi **Repository Pattern** ‘ a uyacak şekilde uygulamamızda **eklemiş olduğumuz bu Entityler için Abstrack ve Concreate** nesnelerimizi oluşturalım.

**Application katmanı** içinde **Repositories** adında oluşturduğumuz klasör içine **Basket ve BasketItem** olacak şekilde **2 farklı klasör oluşturalım.**

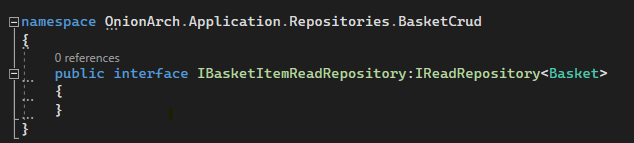
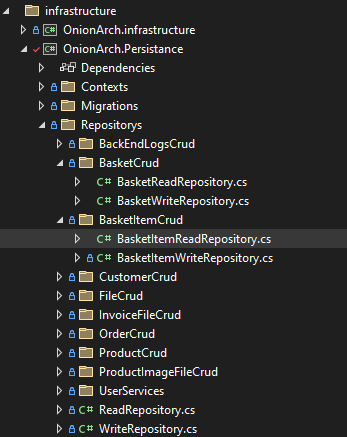
Daha sonra bu klasörlerin hemen altına ekteki gibi dosyalarımızı oluşturalım.  


Oluşturduğumuz bu iki dosyanın içeriği ekteki gibi olacak.

**IBasketItemWriteRepository**



**IBasketItemReadRepository**

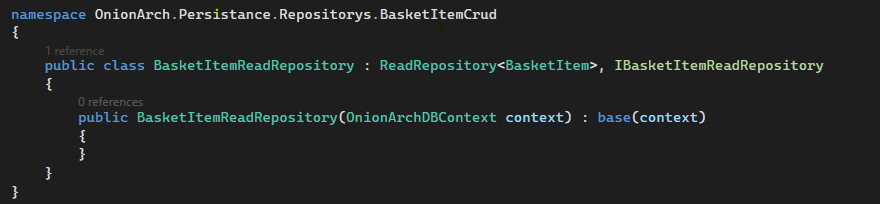
Şimdi bu oluşturduğumuz Repository intefacelerimizin **Concreate** nesnelerini oluşturalım.

Bunun için **infrastructure klasörümüzün** altında bulunan **Persistance katmanına geçiş yapıyoruz.**

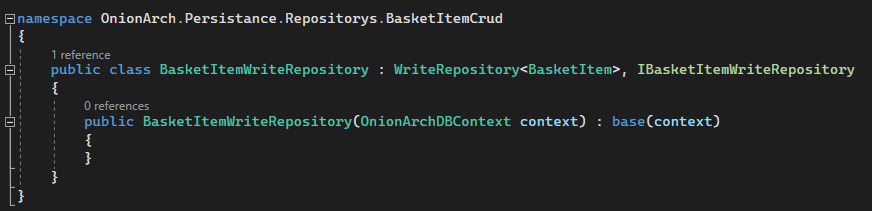
Sağ tarafta ekran görüntüsünde görüldüğü gibi örnek dosyalarımızı oluşturuyoruz.

Daha sonra Generic olarak oluşturduğumuz yapımıza dikkat ederek bu dosyalarımızı düzenliyoruz.

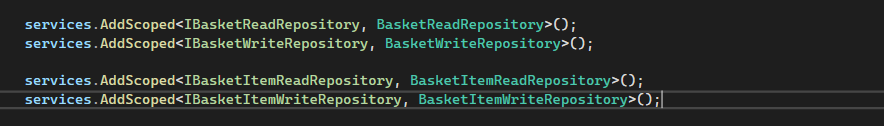
**BasketItemReadRepository**



**BasketItemWriteRepository**

****

Son olarak oluşturmuş olduğumuz bu dosyaları IOC katmanı içinde tanımlamamız gerekmekte.

**Persistance** katmanında bulunan **ServicesRegistration** altında tanımlamalarımızı yapıyoruz.

Şimdi oluşturmuş olduğumuz bu sınıflar için gerekli operasyonların neler olacağına karar verelim.

Basket Görüntüleme

Basket içine item ekleme

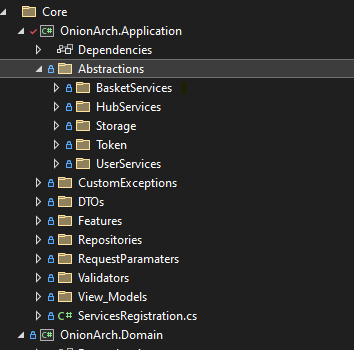
Basketi içinde bulunan elemanın sayısını düzenleme

Basketi kaldırma …

Gibi gibi bir çok işlem yapabiliriz şimdilik bunları referans alarak işlemlerimize başlayalım.

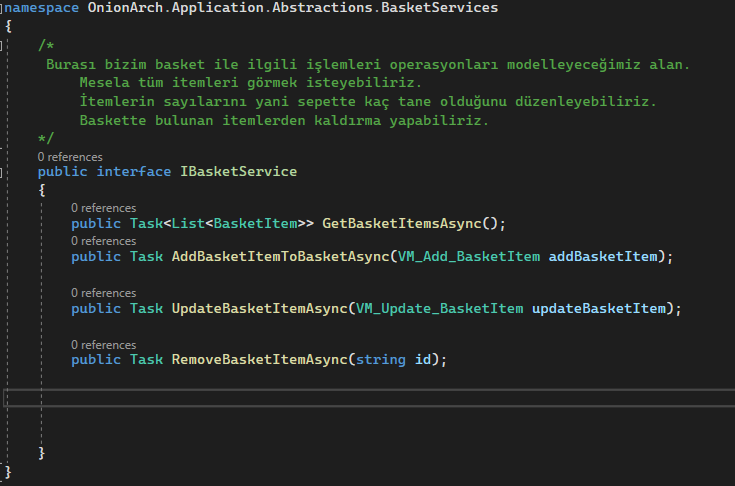
İlk olarak ,

**Application katmanı içinde ,**  daha önceden yaptığımız gibi oluşturacağımız bu servisin **interface tanımlarını Abstractions isimli klasörü**  içinde tanımlayarak işlemlerimize başlayalım.



Burada **Abstractions** isimli klasör içinde biz yapacağımız işlemleri öncelikle bir tamamlıyorduk burada amacımız şuydu biz bir işleme başladığımızda neler yapacağız neler olması gerekiyor sonradan eklememiz gereken şeyleri nerelere eklememiz gerekiyor karmaşası altına girmemek için bu işlemi gerçekleştiriyorduk.

Şimdi burada **BasketServices** adında bir klasör oluşturmak ve bu klasörün altında **BasketServices** adında başka bir interface oluşturarak hangi işlemleri yapacağız buradaya eklememiz gerekiyor.



Şimdi bizim öncelikle kullanıcıya karşılık bir sepet oluşturmamız gerekiyor değil mi ? Ama kullanıcının sepeti oladabilir olmayadabilir ? Yada bu sepeti sipariş verdikten sonra kaldırmamız gerekebilir buna daha sonrada erişmemiz gerekebilir vs vs değil mi ?

Hmm ne yapmamız gerekiyor ….

**Bir State elimizde olmalı değil mi ?**

Peki bu state için nasıl bir yöntem izleyebiliriz   
=>Eğer Sipariş verilmiş ise Durumu Sipariş Verildi

=>Eğer Sipariş verilmemiş ise Durumu Bekleyen olabilir.

=>Eğer sepet silinmiş ise silinmiş olabilir.

=>Belki varsa müşteri o an sepetini kaydetmiş olmak isteyebilir.

Başka bir yöntem ne olabilir ?

Siparişe bağlı şekilde sepet yönetimi gerçekleştirebiliriz.

Peki bu ne demek ?

Şimdi kullanıcı tablomuz var bizim değil mi kullanıcı tablosuna göre sepetlerimiz var bu sepetlerin hangisinin açık hangisinin kapalı olacağını bilemeyeceğiz değil mi ? Yani söyle bir yapımız olacak her bir sepet bir order’a bağlı olacak.Eğer order var ise herhangi bir sepette o sepetin çalışması sonlandırılır.

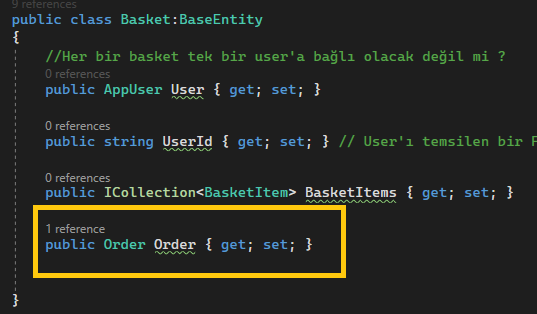
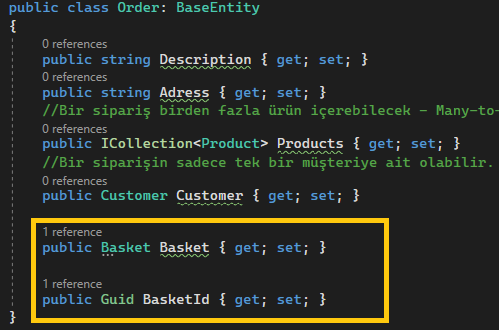
O zaman eğer kullanıcının order’ ı olmayan bir siparişi varsa yeni bir sepet açmış olacağız.

Önce bu bahsettiğimiz 2. yöntemi deneyelim.

Şimdi mantığımıza göre bir sepetin sadece tek bir order’ı olacak değil mi?

Aynı zamanda bir order’da tek bir sepete sahip olacak.

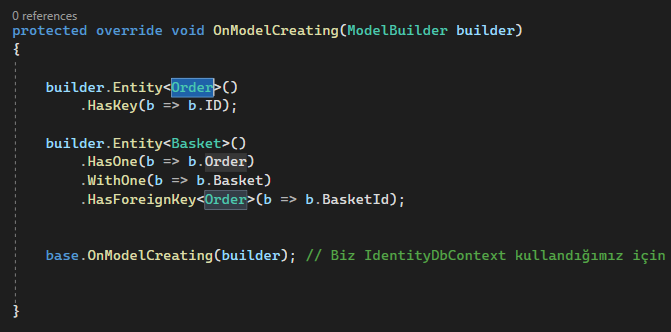
O zaman bizim Tablolarımız arasında bir ilişki olması gerekmekte. Yani Basket ile Order arasında bire bir bir ilişki olacak.



Peki bizim bu kurduğumuz ilişkiyi kim yönetecek ?

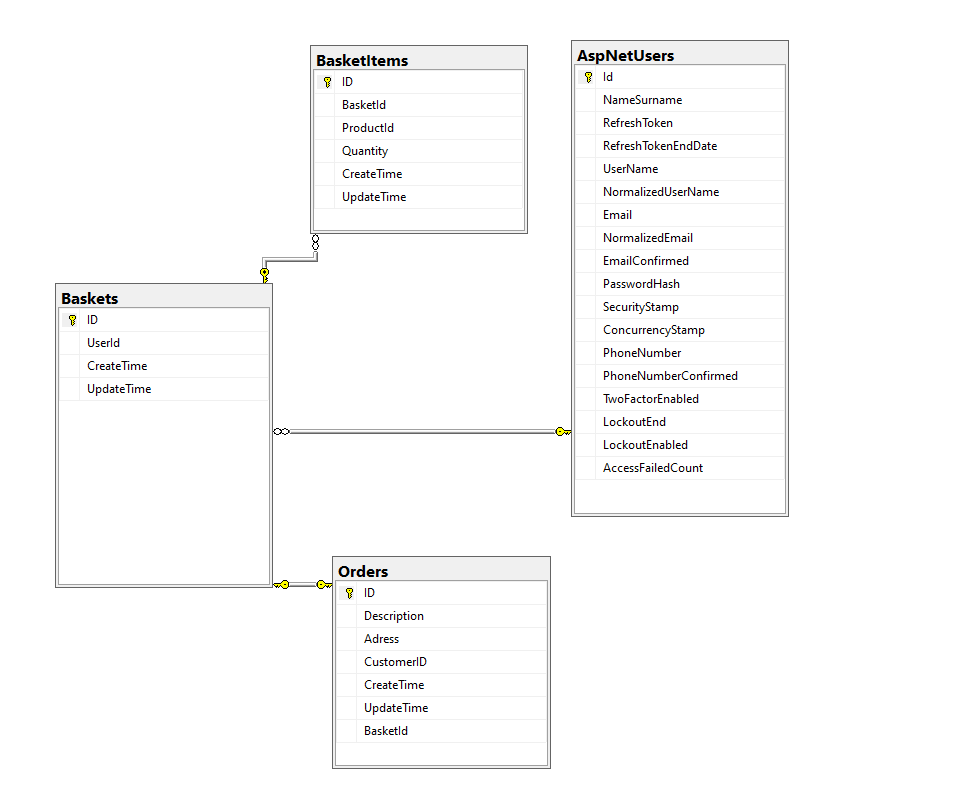
Order yöntecek eğer order varsa sepet olmayacak gibi gibi ..

O zaman buna göre bizim **FluentApi** kullanarak bir düzenleme yapmamız gerekmekte değil mi ?



| **builder.Entity<Order> —>** Order adlı bir varlık (entity) için bir anahtar (primary key) belirler. HasKey yöntemi, Order sınıfının birincil anahtarını belirlemek için kullanılır ve burada b => b.ID ifadesi, Order sınıfındaki ID özelliğini birincil anahtar olarak kullanır.  **builder.Entity<Basket> —>** Basket ve Order arasında bir ilişki kurar. builder.Entity<Basket>() ifadesi, Basket varlığının yapılandırılması için kullanılır. HasOne ve WithOne ifadeleri, Basket ve Order arasındaki ilişkiyi belirtir. HasForeignKey yöntemi, yabancı anahtarın hangi özelliği kullanacağını belirler ve burada b => b.BasketId ifadesi, Order sınıfındaki BasketId özelliğini yabancı anahtar olarak kullanır.  **base.OnModelCreating(builder)** ifadesi, OnModelCreating yöntemini miras aldığı DbContext sınıfının uygulamasını çağırır. Bu, genellikle IdentityDbContext sınıfını kullanıyorsanız, Identity Framework için gerekli konfigürasyonları etkinleştirmek içindir.   * Basket varlığı için bir ilişki belirliyoruz: "Her sepet bir siparişe sahiptir." * Order varlığı için bir ilişki belirliyoruz: "Her sipariş bir sepete sahiptir." * HasForeignKey metodu ile yabancı anahtarı belirtiyoruz: "Siparişin BasketId özelliği, Basket varlığının anahtarına (ID) bağlıdır." |
| --- |

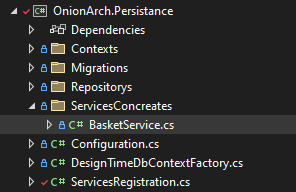
Şimdi bir migration alalım ve veri tabanımızda sonucu inceleyelim.



Şimdi Orders başlangıçta yaptığımız içeriği random olarak doldurduğumuz bir durumdu bunu ilerleyen süreçte düzenleyeceğiz.

**Şimdi tekrar ediyorum buradaki mantığımız şu kullanıcı eğer sepetinde olan ürünler ile ödeme vs hallettikten sonra bir sipariş oluşturuyor ise, buna karşı biz biz order oluşturacağız .**

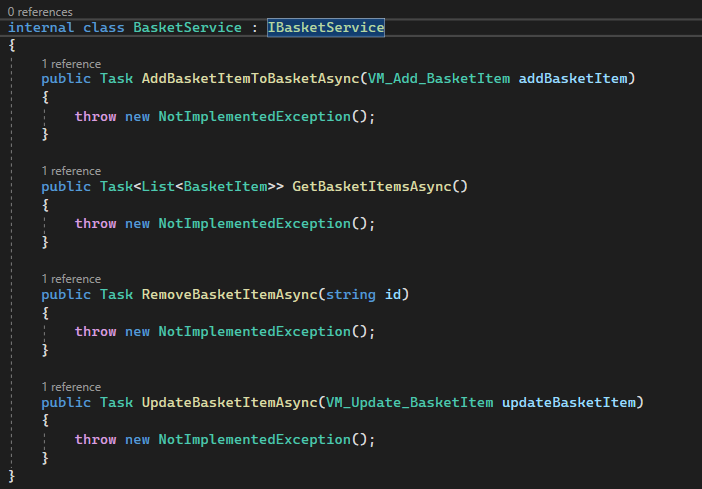
Artık oluşturmuş olduğumuz IBasketService için bir **Concreate** sınıf oluşturup işlemlere başlayalım.



Bunun için **infrastructure klasörümüzün** altında bulunan **Persistance katmanına geçiş yapıyoruz.**

Sağ tarafta ekran görüntüsünde görüldüğü gibi örnek dosyalarımızı oluşturuyoruz.

Daha sonra Generic olarak oluşturduğumuz yapımıza dikkat ederek bu dosyalarımızı düzenliyoruz.



Şimdi artık gerekli düzenlemelere başlayabiliriz.

İlk olarak ekleme işlemi ile başlayalım.

Şimdi ekleme işleminde öncelikle bizim bir şekilde kullanıcıya erişmemiz gerekmekte değil mi ?

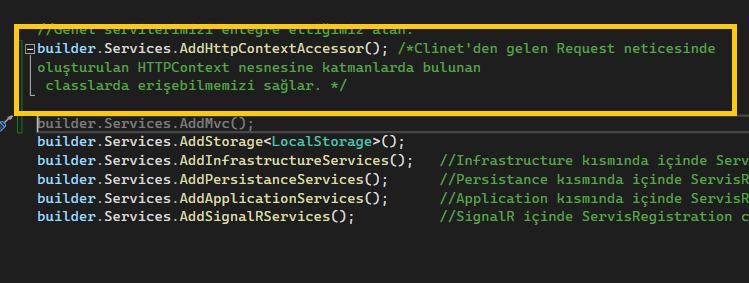
**Bunun için 2 farklı yöntem kullanabiliriz.**

* İlk yöntem istek gelirken kullanıcının bilgileri hakkında **UserId gibi bilgileri talep edebiliriz.**
* İkinci yöntem ise JWT oluştururken kullandığımız gibi **Claim kullanabiliriz.**

Biz Claim üzerinden ilerleyelim.

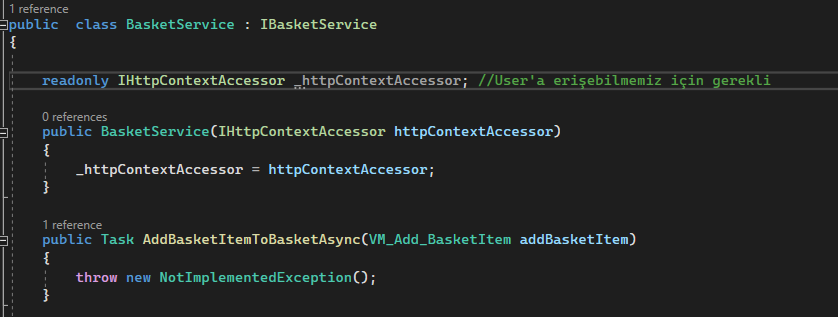
İlk olarak bu yöntemi kullanabilmemiz için bizim **HttpContextAccessor sınıfını kullanmamız gerekiyor.**

Bu servisi kullanabilmemiz için , ilk olarak program.cs içinde bu sınıfın servisini tanımlamamız eklememiz gerekmekte.



Gerekli tanımlama işlemini yaptıktan hemen sonra **BasketService** içinde işlemlerimize başlayabiliriz.

İlk olarak **IHttpContextAccessor** import edelim ve IOC kaydını oluşturalım.



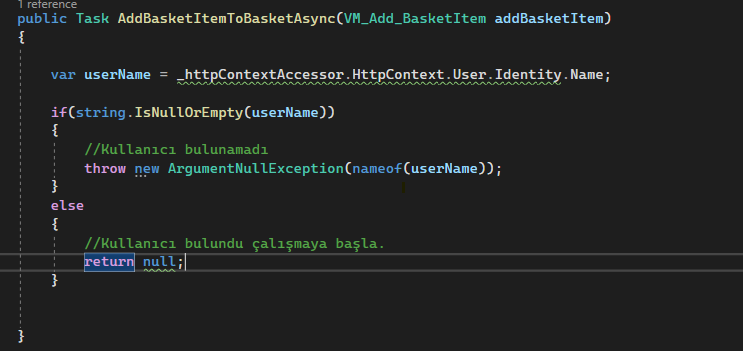
Şimdi kullanıcı bilgilerine nasıl erişeceğiz diye sorabilirsiniz

| **var userName = \_httpContextAccessor.HttpContext.User.Identity.Name;** |
| --- |

Şimdi **AddBasketItemToBasketAsync** üzerinden çalışmalara başlayalım.

Öncelikle burada bir konu hakkında düşünmemiz gerekiyor.   
Bizim birçok sepet ile ilgili işlemimiz söz konusu değil mi ? Ekleme Silme Ürün Düzenleme vs vs …

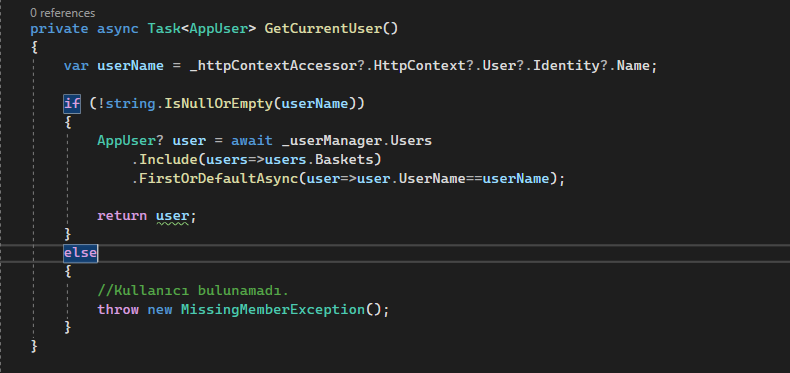
Her birinde kullanıcı varmı yokmu diye bakmamız gerekecek.



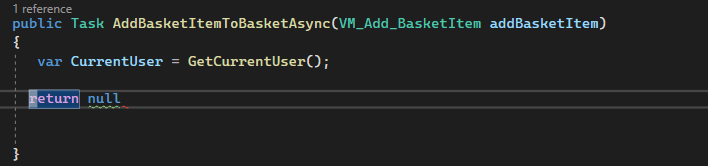
Burada bulunan username değerine her fonksiyon method içinde tek tek yazmak kontrol etmek varsa işleme başlamak gibi gibi bir işlem bizim Solid mantığımıza ve temiz kod mantığımıza çok ama çok ters değil mi …

Bunu her işlemde yapan private bir fonksiyon yapmak daha mantıklı değil mi sizcede?

O zaman şöyle bir yöntem izleyelim sadece bu dosyaya özel private bir fonksiyonumuz olsun bu fonksiyon bizim interface üzerinde ihtiyacımız olan bütün işlemleri gerçekleştirecek bir operasyon olsun.



Yukarıda görüldüğü gibi artık bizim user global olarak kod tekrarı yapmadan ilgili .cs dosyası içinde istediğimiz şekilde çağırabileceğimiz bir fonksiyon oldu.



artık istediğimiz yerde bu şekilde çağırıp kullanabileceğiz ve user’a ulaşabileceğiz.

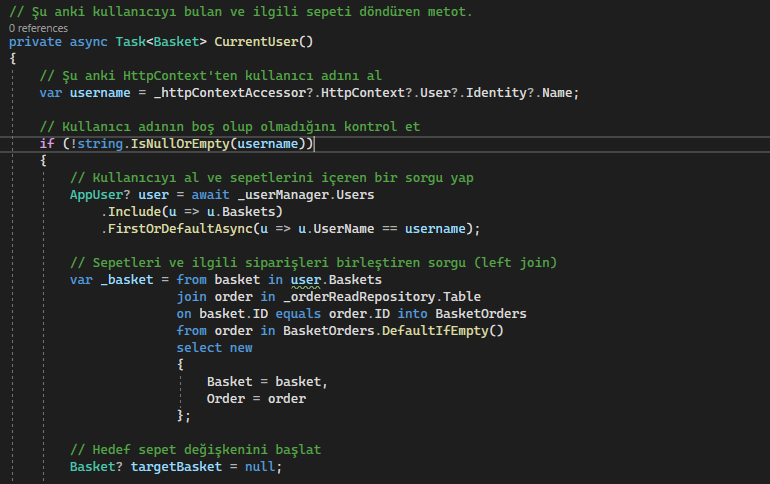
Şimdi bizim asıl amacımız ne ?

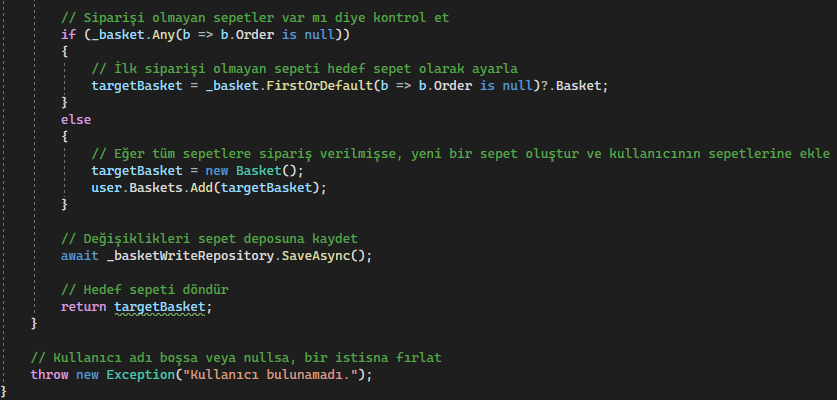
User üzerinden CurrentUser’ın aktif olan basketine ulaşmak ,CurrentUser eğer hiçbir şekilde baskete sahip değilse yani bir basketi yoksa yeni bir sepete sahip olmasını sağlamak değil mi?

Peki bizim bu işlemleri de tek tek her fonksiyon içinde yapmamız ne kadar kod okunurluğu açısından doğru ?

İşte burada **az önce yaptığımız fonksiyonu** **sepet objesi yollayacak şekilde düzenlemiş olsak** ve burada amacımız ilgili sepet üzerinden manipülasyonlar sağlamak olduğu için ilgili **sepete doğrudan müdahale etmemizi sağlayacak bir şekilde güncellesek** daha kolay olmayacak mı bizim için ?

**Şimdi kodumuzu bu şekilde güncelleyelim.**



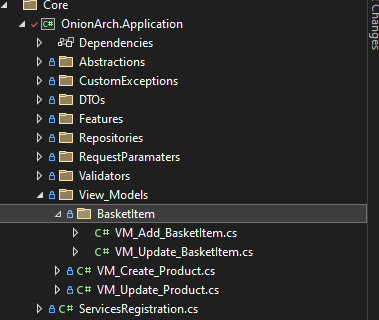


| **from basket in user.Baskets**: Kullanıcının sepet koleksiyonundaki her bir sepet öğesini seçer.  **join order in \_orderReadRepository.Table on basket.ID equals order.ID into BasketOrders**: Kullanıcının sepetini, \_orderReadRepository.Table’dan aldığı Orders adlı sipariş tablosu ile sepetin ID'sine göre birleştirir. Bu, sepetin bir siparişi olup olmadığını kontrol etmemizi sağlar.  **from order in BasketOrders.DefaultIfEmpty()**: Left join yaptık, bu nedenle bir sepetin siparişi yoksa, bu kısmın çalışmasını sağlayarak varsayılan bir değer döndürür. Böylece her sepetin yanında ilgili siparişi içeren bir nesne elde ederiz.  **select new { Basket = basket, Order = order }**: Her sepet ve ilgili siparişi içeren yeni bir anonim nesne oluşturur. Bu nesne, Basket özelliği aracılığıyla sepeti ve Order özelliği aracılığıyla ilgili siparişi içerir.  Sonuç olarak, \_basket adlı değişken, kullanıcının sepetleri ile bunlara ait siparişleri içeren bir koleksiyonu temsil eder. Bu koleksiyonu kullanarak, kullanıcının sipariş vermiş veya vermeye hazır olduğu sepetleri belirleyebilirsiniz.   | **SELECT Baskets.\*, Orders.\***  **FROM Baskets**  **LEFT JOIN Orders ON Baskets.ID = Orders.ID;** | | --- | |
| --- | --- |

**Artık kodumuz bir basket döndürecek biz bu basketin içinde gerekli manipülasyonlar yapabileceğiz örnek olarak ürün ekleyebileceğiz ürün silebileceğiz gibi gibi .**

Şimdi ilgili kullanıcının basketine başarılı bir şekilde ulaştık.  
Artık bize burada kullanıcının basketi ile iligli ekleme çıkarma işlemlerini yapma kaldı.

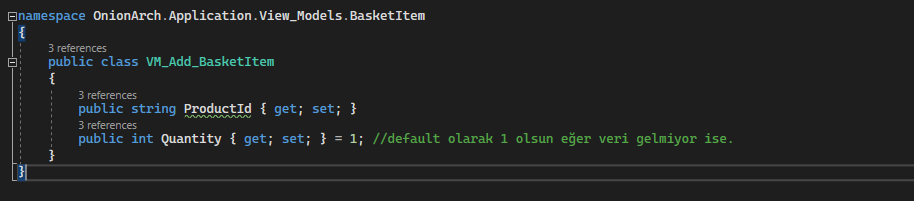
İlk olarak **Baskete ekleme**  işlemine başlayalım.

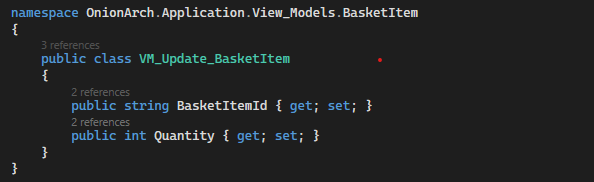


**Başlamadan önce ;**

Öncelikle ekleme işleminde bize neler gerekiyor buna karar verelim bize ürün ekleme işlemi için ilgili ürünün Id değeri ve bu üründen kaç adet ekleneceğini kullanıcıdan almamız bizim için yeterli değil mi ??

**O zaman ekleme işlemi için oluşturduğumuz VievModellerimizin içlerini dolduralım.**



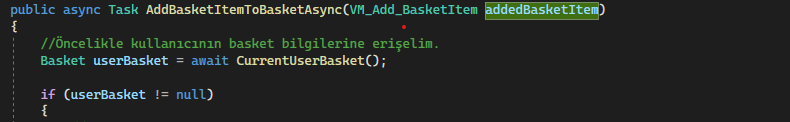


Artık tamamen hazırız şimdi geliştirmelerimize başlayabiliriz. 🙂

**İlk olarak sepete ürün ekleme işleminden başlayalım.**

**Öncelikle kullanıcının anlık olarak kullandığı baskete erişmemiz gerekmekte.**

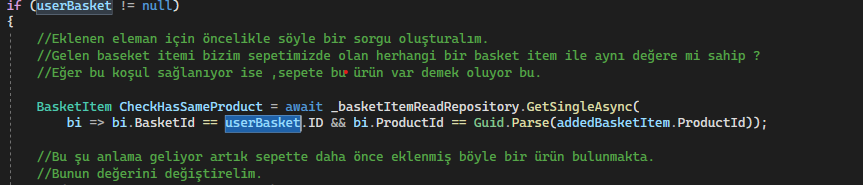
Bunun için biraz önce oluşturduğumuz **CurrentUserBasket()** methodu ile kullanıcının anlık olarak basketine ulaşalım.



Şimdi baskete eriştik burada şöyle bir kontrol sağlamamız gerekiyor eğer **kullanıcının sepetinde eklenen ürün daha önce eklenmiş işe burada kullanıcıya ürünleri listelerken aynı üründen 2 tane olduğunu listelememiz saçma olacaktır değil mi .**

Burada şöyle bir yöntem üzerinde çalışma yapacağız, **kullanıcının sepetini önce kontrol edeceğiz eklenen bu ürün kullanıcının sepeti içinde daha önce eklenmiş mi eklenmiş ise tekrar eklendiği kadar üstüne ekleme yapacağız** yani ürün sepette bir adet iken biz bir kaç kez daha butona tıkladığımızda sepete bulunan bu sayı artacak.

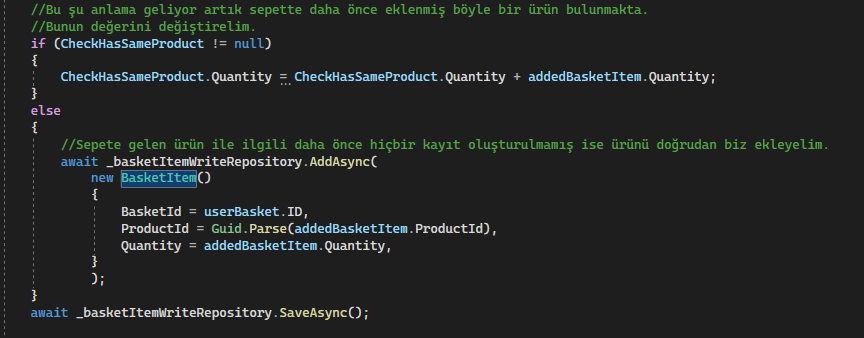
Burada bahsettiğim konu için şöyle bir düzenleme yapacağız,



| **bi => bi.BasketId == userBasket.ID && bi.ProductId == Guid.Parse(addedBasketItem.ProductId)**  Bu lambda ifadesi, BasketItem öğelerini filtrelemek için kullanılır. Burada, sepet ID'si (BasketId) kullanıcı sepetinin ID'sine (userBasket.ID) ve ürün ID'si (ProductId) eklenmiş olan bir addedBasketItem nesnesinin ürün ID'sine eşit olup olmadığını kontrol eder. |
| --- |

Gerekli kontrolu sağladıktan sonra burada şöyle bir yöntem izleyeceğiz.

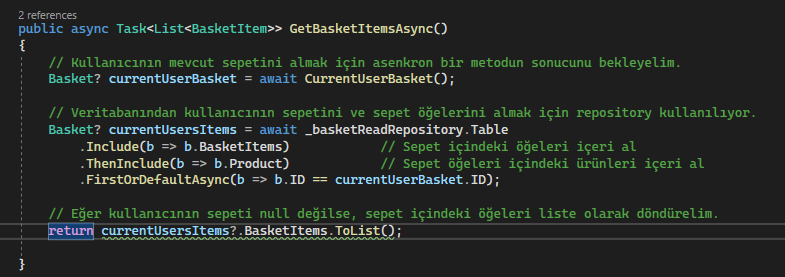
Bu ürün var ise bu ürünün sepet içinde bulunan quantity yani adet değerini artıracağız.



* Eğer sepette böyle bir ürün varsa bu ürünün **quantity değerini artıracağız**
* Eğer sepette böyle bir ürün yoksa sepete biz **yeni bir ürün olacak** şekilde bu ürünü ekleyeceğiz.

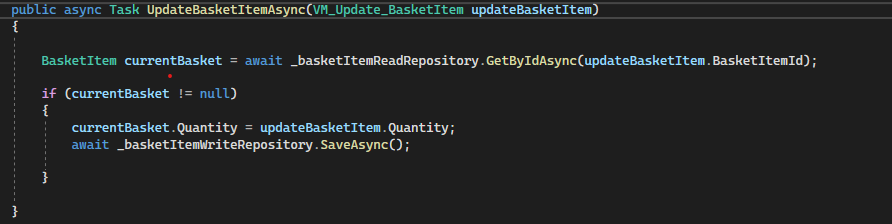
Şimdi ise basket içinde bulunan elemanların nasıl listelendiğine bakalım,

**Şimdi sepet içine ulaşıp ürünleri listeleme işlemini kodlayalım.**

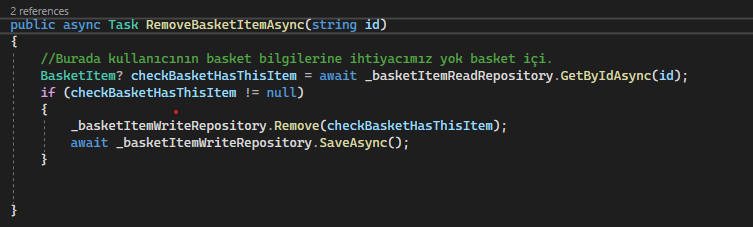


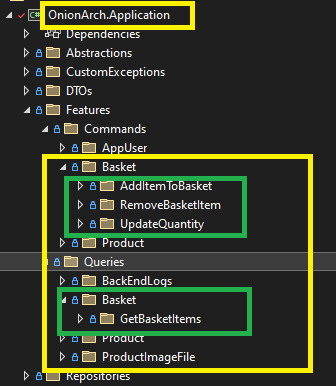
| **await \_basketReadRepository.Table..**. satırı, bir veritabanı işlemi gerçekleştirir. \_basketReadRepository muhtemelen bir veritabanı işlemlerini gerçekleştiren bir repository (veri deposu) sınıfını temsil eder. Include ve ThenInclude metodları, Entity Framework Core'daki ilişkisel veritabanı modellerinde kullanılan "eager loading" (iç içe geçmiş nesnelerin bir arada alınması) işlemlerini ifade eder.  **Include(b => b.BasketItems)** ifadesi, Basket nesnesinin BasketItems ilişkisini içeri almayı sağlar. Bu, sepet nesnesinin içindeki sepet öğelerini temsil eden nesneleri de almayı sağlar.  **ThenInclude(b => b.Product**) ifadesi, BasketItem nesnesinin içindeki Product ilişkisini içeri almayı sağlar. Bu, sepet öğeleri içindeki her bir ürün nesnesini almayı sağlar.  **FirstOrDefaultAsync(b => b.ID == currentUserBasket.ID)**; ifadesi, filtreleme yaparak, belirli bir sepet ID'sine sahip olan sepeti asenkron olarak veritabanından çeker. Bu sepet nesnesi currentUsersItems adındaki değişkene atanır.  **return currentUsersItems?.BasketItems.ToList();** satırı, eğer currentUsersItems değeri null değilse, sepetin içindeki öğeleri temsil eden BasketItems listesini döndürür. Eğer currentUsersItems null ise, null döner. |
| --- |

**Şimdi sepet içine ulaşıp ürünleri update edilme işlemi.**

****

**Şimdi sepet içine ulaşıp ürünleri delete edilme işlemi.**

****



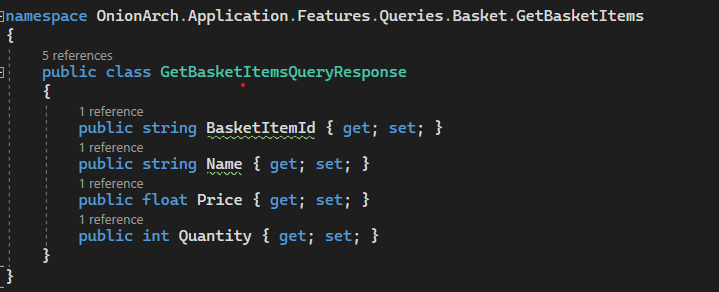
Şimdi gerekli olan tüm düzenlemeleri gerçekleştirdik artık Mediator kullanarak gerekli düzenlemeleri yapmamız gerekmekte.

Örnek olarak bir tane ekleme yapalım,

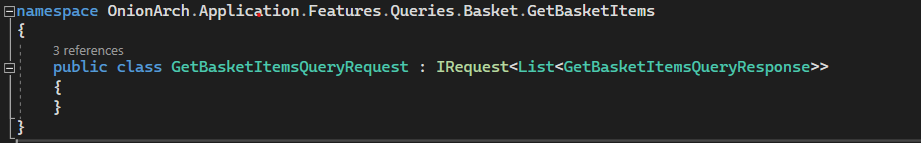
**Handler sınıfımız :**



**Response sınıfımız :**



**Request sınıfımız :**



Şimdi son olarak controller üzerinde bu verilere erişelim.

