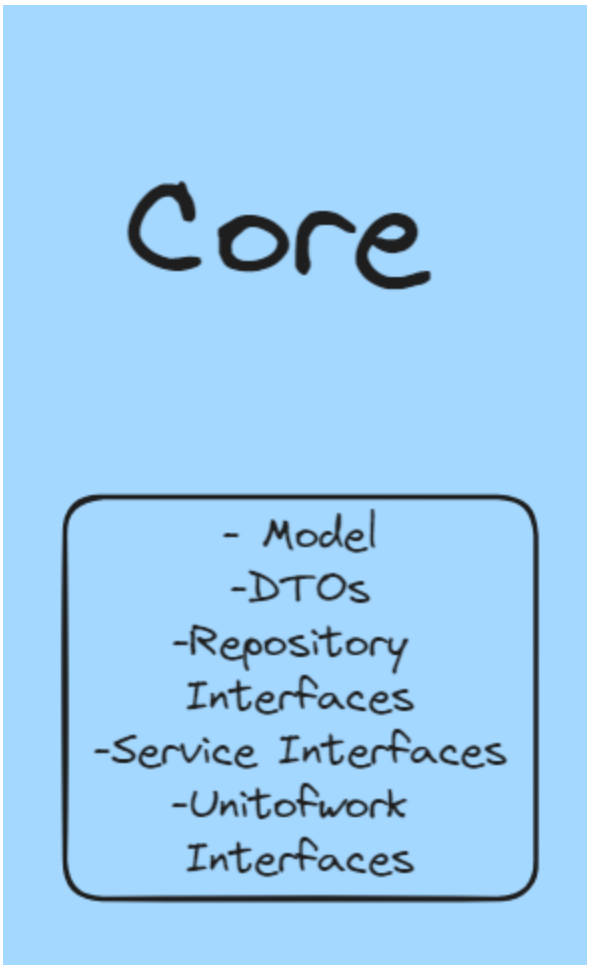
Core Katmanı İle İş Mantığını İnşa Etmek

Yazılım dünyası, kendi içinde bir şehir gibidir diye bir önceki yazımda bahsetmiştim hatırlarsanız :). Bu şehir, farklı katmanlardan oluşan karmaşık bir yapıya sahiptir. İşte bu şehrin ana arterlerinden biri de **"Core Katmanı"**dır. Core Katmanı, yazılım şehrinin kalbidir; burada iş mantığı atar ve sistem canlılığını sürdürür. Bir yazılım projesini düşünün; o proje, bir şehrin sokakları gibi karmaşık ve birbirine bağlı bir ağdır. Her katman, bu şehirdeki bir mahalleyi temsil eder. Ancak, en temel ve önemli mahalle, şehrin merkezindedir — işte burası, yazılım şehrinin Core Katmanıdır. Core Katmanı, yazılımın belkemiğini oluşturur desek abartmış olmayız. Burada, projenin temel iş mantığı ve veri yönetimi yatar. Bu katman, diğer katmanlarla iletişim kurar ve onları bir araya getirir. İşte tam da bu nedenle, Core Katmanı, yazılım şehrindeki en önemli arterlerden biridir. Bu makalede, yazılım şehrinin kalbindeki bu önemli katmanı keşfedeceğiz. Core Katmanı'nın geliştirmelerini adım adım ele alacak ve nasıl iş mantığını inşa ettiğimizi göreceğiz. Yolculuğumuza başlamadan önce, bu katmanın ne kadar kritik olduğunu anlamamız gerekiyor. Core Katmanı olmadan, yazılım şehri yarıda kalır ve diğer katmanlar birbiriyle uyumsuz hale gelir.

Hazır mısınız? O zaman, yazılım şehrindeki ana arterlere doğru bir yolculuğa çıkalım ve Core Katmanı ile iş mantığını inşa etmenin sırlarını keşfedelim.

Aşağıdaki oluşturduğumuz mimariye istinaden aşama aşama bu alanların neden ve niçin oluşturulduğunun detaylarını hepbirlikte göreceğiz!



Entitylerin Oluşturulması

Yazılım şehrinin Core Katmanı, projenin temelini oluşturan entitylerle dolup taşar. Ancak, her entity'nin bir veri tabanındaki karşılığı olup olmadığını anlamak, projenin sağlıklı bir şekilde büyümesi için kritik bir adımdır. Entity, bir sınıfı temsil eder ve genellikle veri tabanında depolanan bir nesnedir. Ancak, her sınıfın bir veri tabanı tablosu oluşturmak zorunda olmadığı unutulmamalıdır. Entity'nin, veri tabanındaki bir tabloya karşılık gelip gelmediğini belirlemek için, sınıfın veri tabanındaki varlığına dair bazı kriterleri göz önünde bulundurmalıyız. Eğer bir class'ın veri tabanında karşılığı varsa, buna "**entity**" adını veririz. Bu, genellikle bu sınıfın bir veri tabanı tablosu ile ilişkilendirildiği ve verilerin bu tabloya kaydedildiği anlamına gelir. Bu durumda, Entity'nin, veritabanındaki yapı ile senkronize bir şekilde çalışması beklenir. Ancak, eğer bir class'ın veri tabanında karşılığı yoksa,entity olmadığını söyleyebiliriz. Bu durumda, sınıfın veritabanıyla doğrudan bir ilişkisi yoktur ve bu sınıf sadece iş mantığına veya geçici verilere hizmet edebilir. Entity oluştururken, sınıflar arasındaki ilişkileri ve her birinin veri tabanındaki rolünü dikkate almalıyız. Bu, projenin veri yönetimi açısından **daha etkili** ve **sürdürülebilir** olmasını sağlar. Ayrıca, veri tabanında her bir entity'nin nasıl temsil edileceği konusundaki kararlar, projenin performansını da

etkileyebilir. Bu nedenle, Core Katmanı'nda entity oluştururken, sınıflar arasındaki ilişkileri anlamak, her bir entity'nin veri tabanındaki durumunu değerlendirmek ve projenin gereksinimlerine uygun bir şekilde yapılandırmak önemlidir. Bu sayede, yazılım şehrinin ana arterlerinden biri olan Core Katmanı, sağlıklı bir şekilde işlemeye devam edebilir.

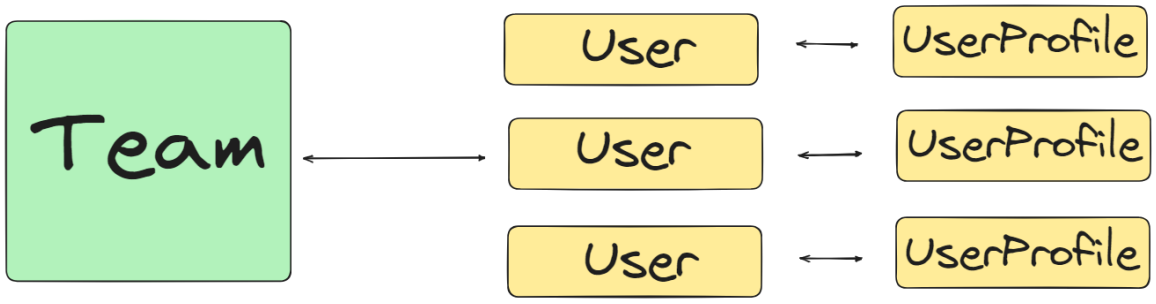
Base Entity Oluşturma

Base Entity, bir yazılım projesindeki diğer entity'lerin temelini oluşturan bir ana sınıf veya arayüzdür. Bu temel sınıf veya arayüz, projede bulunan birçok entity'nin paylaştığı ortak özellikleri içerir. Örneğin, entityler arasında ortak olan oluşturma tarihi veya son değişiklik tarihi gibi özellikler, **Base Entity** içinde tanımlanır. Bu sayede, projedeki diğer entityler, Base Entity'den türetilerek bu ortak özelliklere sahip olur. Base Entity'nin oluşturulmasının temel nedeni, **kodun daha düzenli** ve **sürdürülebilir olmasını sağlamak**, **veritabanı tasarımını kolaylaştırmak**, **kod tekrarını azaltmak** ve **projedeki genel tutarlılığı artırmak**tır. Ayrıca, gelecekteki genişlemelere de uygun bir yapı sunarak projenin esnekliğini artırır. Örneğin, bir proje kapsamında **Id**,**CreatedDate**,**UpdatedDate** gibi alanlar tüm tablolarda ortak olarak kullanılacaksa o zamana aşağıdaki gibi bir yapı kurulabilir.

```
public abstract class BaseEntity
{
    public int Id { get; set; }
    public DateTime CreatedDate { get; set; }
    public DateTime UpdatedDate { get; set; }
}
```

Abstract bir base class kullanmanın temel nedeni, bu sınıfın kendi başına bir örneğinin oluşturulamaması ve yalnızca türetilmiş sınıflar tarafından genişletilerek kullanılabilmesidir. Bu durum, **BaseEntity** sınıfının bir soyut sınıf olduğu anlamına gelir. Soyut sınıflar, ortak özellikleri içerir ancak tam bir implementasyona sahip olmazlar, bu nedenle türetilmiş sınıflar, bu ortak özellikleri kendi ihtiyaçlarına göre özelleştirebilirler. **BaseEntity**'in abstract olması, ***bu sınıfın kendisi üzerinden doğrudan örnek oluşturulamayacağı için, sadece türetilmiş sınıfların bu ortak özelliklere erişebileceği ve kullanabileceği*** anlamına gelir.

Veri tabanı Entitylerinin Oluşturulması



Yukarıdaki örnekte **Team** ve **User** entityleri arasında bire çok ilişki, **User** ve **UserProfile** entityleri arasında ise birbirine karşılık gelen 1–1 ilişki bulunmaktadır. Yani, bir **Team**'e birden fazla **User** bağlı olabilir, ancak her **User** sadece bir **UserProfile**'e sahip olacak. Bu yapıda, **User** tablosundaki kolon sayısı çok artarsa ve yeni özellikler eklemek karmaşık hale gelirse, bu durumu yönetmek için ayrı bir **UserProfile** tablosu eklemeyi düşünlmüştür. Böylece, her bir **User** için detaylı profil bilgilerini ayrı bir tabloda saklamak, veritabanını daha düzenli tutmanıza ve gelecekteki değişiklikleri daha kolay yönetmenize olanak tanır.

Team Entity Oluşturulması

Aşağıdaki entity tasarımına göre Team'e bağlı Userlar olacak ve burada **bire çok ilişki** kullanımı gerçekleşecektir. Team tablosu BaseEntity Class'ından miras alacaktır ve aslında içerisinde **Id**,**CreatedDate**, **UpdatedDate** gibi kolonları da barındıracaktır. Bire çok ilişkiyi belirtmek için, Team entity'sinde **ICollection<User>** türünde bir property kullanılır. Bu property, Team'in birden fazla User'a sahip olabileceğini ifade eden bir **"navigation property"**dir. Yani, bir Team'e ait olan Users koleksiyonu, Team ve User arasındaki bire çok ilişkisini temsil eder ve bu koleksiyon aracılığıyla Team'e bağlı olan tüm kullanıcılara erişim sağlanabilir. Bu, yazılım geliştiricilerin **Team** ve **User** arasındaki ilişkiyi kolayca yönetmelerini ve verilere ulaşmalarını sağlar.

```
public class Team:BaseEntity
{
   public string TeamName { get; set; }

   // Bire-çok ilişki
   public ICollection<User> Users { get; set; }
}
```

User Entity Oluşturulması

Bu **User** sınıfı, **BaseEntity** sınıfından miras alarak temel kimlik ve tarih özelliklerini içerir. **UserName**, **Email** ve **Password** özellikleri, bir kullanıcının temel bilgilerini temsil eder. **TeamId** özelliği, **User** ve **Team** arasında bire çok ilişki kurmak için kullanılır ve User'ın hangi takıma ait olduğunu belirten bir **Foreign Key'**dir. **Team** özelliği, **User** ve **Team** arasındaki bu ilişkiyi temsil eden **navigasyon** özelliğidir. Bu sayede, bir kullanıcının hangi takıma ait olduğunu belirlemek için Team entity'sine erişim sağlanabilir. Böylece, **User** ve **Team** arasındaki ilişkiyi yönetmek ve verilere kolayca erişim sağlamak mümkün olur.

```
public class User:BaseEntity
{
   public string UserName { get; set; }
   public string Email { get; set; }
   public string Password { get; set; }

   // İlişkilendirme
   //Foreign Key
   public int TeamId { get; set; }
   public Team Team { get; set; } // Bire-çok ilişkiyi temsil eden navigasyon özelliği
}
```

***Entity Framework (EF) Core** kullanırken, veri tabanı tablolarını ve ilişkilerini tanımlarken isimlendirmeye dikkat etmek önemlidir. **Primary key** olarak algılanacak bir özellik için **"Id"** ismini kullanmak, EF Core'un varsayılan standartlarına uygun hareket etmeyi sağlar. Aynı şekilde, bir özelliği **Foreign Key** olarak belirtmek için **"TeamId"** gibi isimlendirmeler kullanabilirsiniz. Ancak, alt çizgi gibi özel karakterler eklememek önemlidir; çünkü EF Core, bu tür isimlendirmeleri otomatik olarak Foreign Key olarak algılamaz. Eğer özel bir isimlendirme kullanmak istiyorsanız ve EF Core'un varsayılanlarını tercih etmiyorsanız, "**ForeignKey**" attribute'ünü kullanarak **explicit** olarak ilişkileri belirtmelisiniz. Bu sayede, EF Core'un anlayabileceği isimlendirmeleri koruyabilir ve migration süreçlerinde hataların önüne geçebilirsiniz. Özetle, EF Core'un standartlarına uygun isimlendirmeler kullanmak ve gerektiğinde özel isimlendirmeleri açıkça belirtmek, veri tabanı tasarımını daha tutarlı ve hatasız hale getirir.*

UserProfile Entity Oluşturulması

```
public class UserProfile
{
```

```
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    //Foreign Key
    public int UserId { get; set; }
    // Bire-bir ilişki
    public User User { get; set; }
}
```

Bu kod parçasında, **UserProfile** adlı sınıf BaseEntity'den miras almamıştır çünkü zaten **User** oluştuğunda **UserProfile** da oluşacağı için **CreatedDate** ve **UpdatedDate** alanları ortaktır. O nedenle içerisinde bir kullanıcının profil bilgilerini temsil eden özellikler (**Id**, **FirstName**, **LastName**) bulunmaktadır. **UserProfile** sınıfı, bir kullanıcı ile **bire-bir** ilişki içinde olduğu için **User** özelliğiyle bu ilişkiyi temsil eder. Ayrıca, kullanıcının oluşturulma tarihini tutan **CreatedDate** özelliği, **User** sınıfından alındığı için burada da ayrıca belirtilmemiştir. Bu şekilde, **UserProfile** sınıfı, bir kullanıcının profili ve ilgili kullanıcının temel bilgileri arasında bir bağlantı sağlanır. **User** tablosu ile bağlantısı için **UserId** Foreign Key'i de unutulmamalıdır.

IGenericRepository Geliştirmeleri

Core katmanında R**epository Design Pattern**'inin uygulanması için ilk adım olarak, bir arayüz (interface) oluşturulacaktır. **Repository Design Pattern**, kodumuz ile veri tabanı arasına bir ara katman yerleştirerek, her bir entity için temel **CRUD (Create, Read, Update, Delete)** operasyonlarını uygulamamıza olanak tanır. Bu ara katman, veri tabanı işlemlerini soyutlar ve her bir entity için genel bir yapı sağlar. Generic bir yapı kullanmak, her bir entity için temel **CRUD** operasyonlarını daha genel ve tekrar kullanılabilir bir şekilde uygulamamıza imkan tanır. Yani, bu pattern sayesinde herhangi bir entity için **Create**, **Update**, **GetAll**, **GetById** gibi CRUD operasyonları kolayca uygulanabilir hale gelir, ve bu operasyonlar Core katmanındaki repository aracılığıyla veri tabanına iletilir.

```
public interface IGenericRepository<T>
    where T: class
{
Task<T> GetByIdAsync(int id);

//user.GetAll(s => s.UserName == "kardel"); ( Filtreleme yapmak için aşağıdaki expression parametresini function delege olarak tanımlamanız gerekir. )
IQueryable<T> GetAll(Expression<Func<T,bool>> expression);
```

```
//user.Where(s=> s.userName == "kardel").ToList().OrderBy(); ( Veri tabanına gider sonra sıralama yapar. )
//user.Where(s=> s.userName == "kardel").OrderBy(); ( Veri tabanına gitmeden ön bellekten sıralama yapar. )
IQueryable<T> Where(Expression<Func<T,bool>> expression);
Task AddAsync(T entity);
Task AddRangeAsync(IEnumerable<T> entities);
void Update(T entity);
void Remove(T entity);
void RemoveRange(IEnumerable<T> entities);
}
```

GetAll Neden Queryable?

- **GetAll** metodunu çağırdıktan sonra filtreleme işlemi yapmak için **IQueryable** kullanılır. Örneğin, **GetAll** çağrısı sonrasında id'si 3'ten büyük olan verileri getirmek amacıyla kullanılabilir. Bu tür sorgular, **ToList()** veya **ToListAsync()** çağrıldığında veri tabanına iletilir.

Where Neden IQueryable?

- **IQueryable** kullanılmasının temel nedeni, sorguların veri tabanına direkt gitmemesini sağlamaktır. Bu sayede, **Where** gibi sorgular yazıldığında, sorgular veri tabanına gitmeden önce belirli bir yapıya sahip olur. Örneğin, **Where** kullanıldıktan sonra **ToList** çağrıldığında sorgu performanslı bir şekilde çalışabilir.

Function Delegesi ve Where Sorguları

- Bir function delegesi kullanarak Entity Framework Core'daki sorguları ifade eden **Expression** türünde sorgular oluşturulabilir. Function delegesi, bir entity'i alıp true veya false dönen bir yapıdır. Bu, her bir satır için geçerli olan bir kontrol yapısını temsil eder.

Asenkron Metotlarda Ek Bilgi

- Asenkron metotlar, var olan thread'leri bloklamamak amacıyla kullanılır. Asenkron metotlar genellikle **Async** ekini alır. Bu tür metotlar, uzun süren işlemleri gerçekleştirmek için kullanılır. Ancak, **Update** veya **Remove** gibi metotlar, genellikle uzun süreli işlemler içermez ve dolayısıyla asenkron olarak işaretlenmez. **Add** gibi işlemler, uzun sürebileceği için asenkron olabilir.

Add metotu asenkron ve geri dönüş değeri alıyorken Update ve Remove neden asenkron değil ve geri dönüş değeri almıyor ?

Asenkron metotlar, genellikle uzun süren işlemleri bekletmeden ve bloke etmeden arka planda gerçekleştirmek için kullanılır. **AddAsync** metodu asenkron olarak tasarlanabilir çünkü veritabanına **yeni bir öğe eklemek** zaman alabilir ve bu süreç, diğer işlemlerin devam etmesini sağlamak için asenkron olarak yönetilebilir. Diğer taraftan, **Update** ve **Remove** işlemleri genellikle **mevcut bir öğenin durumunu güncellemek veya kaldırmak gibi** daha hızlı işlemler olduğu için asenkron olmayabilirler. Ayrıca, bu işlemler genellikle sadece veritabanı üzerinde değişiklik yapar ve geri dönüş değerine ihtiyaç duymazlar çünkü işlemin başarılı olup olmadığını belirlemek için istemci kodunda genellikle geri dönüş değerine gerek duyulmaz. Asenkron olarak tasarlanan metotlar, genellikle uzun süren I/O işlemleri veya ağ çağrıları gibi durumlarda daha büyük bir fayda sağlar.

IEnumarable yerine List kullansam ne olurdu ki ?

IEnumerable Kullanımı
Genel Koleksiyon Temsilcisi: **IEnumerable** genellikle koleksiyonlara genel bir şekilde erişim sağlamak için kullanılır. Bu, bir koleksiyonun üzerinde sıralı bir şekilde gezinme yeteneği anlamına gelir.

Lazy Evaluation (Gecikmeli Değerlendirme): **IEnumerable** genellikle gecikmeli değerlendirme sağlar. Yani, koleksiyonun elemanlarına ihtiyaç duyulduğunda elemanlar üretilir.

```
IEnumerable<int> numbers = GetNumbers();
foreach (var number in numbers)
{
  // Her elemana erişim sırasında elemanlar üretilir
  Console.WriteLine(number);
}
```

List Kullanımı
Dinamik Büyüme: List, bir koleksiyonu dinamik olarak genişletebilen bir türdür. Bellekte daha fazla yer talep edebilir ve performans avantajı sunabilir.
Daha Zengin API: List, bir dizi (array) üzerindeki işlemleri genişleten ek bir API sağlar. Eleman eklemek, çıkarmak, sıralamak vb. için daha fazla yöntem içerir.

```
List<int> numbersList = new List<int> { 1, 2, 3, 4, 5 };
numbersList.Add(6); // Listeye eleman eklemek
numbersList.Remove(3); // Listeden eleman çıkarmak
```

Eğer sadece koleksiyon üzerinde sıralı bir şekilde gezinme ihtiyacınız varsa ve bellek meseleleri önemliyse, **IEnumerable** kullanmak daha mantıklı olabilir. Ancak, koleksiyonu sık sık değiştirmeniz veya dinamik olarak büyütmeniz gerekiyorsa, **List** daha uygun bir seçenek olabilir.

Unutulmaması gereken önemli bir nokta, kodunuzu mümkün olduğunca genel ve esnek tutmaktır. Bu nedenle, kodunuzda mümkünse, **IEnumerable** veya **ICollection** gibi daha genel türleri kullanmak iyi bir uygulama tasarımıdır. Biz de o nedenle örneklerimizde **IEnumerable** kullanımı gerçekleştirdik. Ancak, belirli bir durumda **List** kullanmak daha uygunsa, bu türü kullanabilirsiniz.

IService Geliştirmeleri

IService arayüzü, yazılım mimarisinde servis katmanını temsil eder ve iş mantığının bulunduğu yer olarak öne çıkar. Generic bir metottur, genellikle repositorylerde "**Generic**" ifadesi kullanılırken servislerde pek kullanılmamaktadır. Bu katman, veritabanından alınan veriler üzerinde ek işlemler gerçekleştirir ve çeşitli iş katmanlarını koordine eder. Ayrıca, repository'den alınan verileri dönüştürme ve mapping işlemlerini içerir, böylece aynı dönüşüm mantığı tekrar kullanılabilir ve metotlar esnek bir yapıya sahip olabilir. **IService**, iş mantığı kodlarını düzenleyerek, repository'ye göre dönüş türlerini belirleyerek ve veri dönüşümü sağlayarak servis katmanının temelini oluşturur. Bu sayede, kodun tekrarlanmasını önler ve iş mantığının sağlıklı bir şekilde yönetilmesine olanak tanır.

```
public interface IService<T>
   where T: class
{
   Task<T> GetByIdAsync(int id);
   Task<IEnumarable<T>> GetAllAsync();
   IQueryable<T> Where(Expression<Func<T,bool>> expression);
   Task AddAsync(T entity);
   Task AddRangeAsync(IEnumerable<T> entities);
   Task UpdateAsync(T entity);
   Task RemoveAsync (T entity);
   Task RemoveRangeAsync(IEnumarable<T> entities);
}
```

Repository'de **void** olarak belirtilen **Update** ve **Remove** işlemleri; **Servis Katmanı**'nda kullanılıp veri tabanına yansıtma işlemi gerçekleşeceği için **IService** içerisinde **Task** döndürmeli ve asenkron şekilde yazılmalıdır. Bu nedenle, **Update** ve **Remove** metotları **UpdateAsync** ve **RemoveAsync** olarak

güncellenmelidir. Bu şekilde, asenkron bir yapıda çalışarak işlemleri daha verimli bir şekilde gerçekleştirilebilir.

GenericRepository ile hemen hemen aynı metotları kullandık, ee o zaman bunu neden oluşturduk ?

Buradaki amaç Repository Katmanı'ndan almış olduğunuz veriyi Service Katmanı'ında mapping yapabilmek bir nevi dönüşüm yapabilmektir. Çünkü ***Business*** *burada gerçekleşmektedir.* ***GenericRepository*** *dışında bir repository oluşturduğunuzda veya* ***IService*** *dışında farklı bir servis oluşturduğunuzda o zaman dönüş tipleri farklı olacaktır. Örneğin, oluşturacağınız* ***ITeamService****'in içerisindeki metotların dönüş tipleriyle* ***ITeamRepository*** *içerisindeki metotların dönüş tipleri farklılaşacaktır. O nedenle bu kısımda kodu tekrarlıyoruz gibi düşünmek yanlış bir yaklaşım olacaktır.*

IUnitOfWork Geliştirmeleri

UnitOfWork tasarım deseni, veri tabanına yapılacak işlemleri toplu bir şekilde tek bir **transaction** üzerinden yönetme yeteneği sağlar. Özellikle **Entity Framework Core** kullanılırken, her bir repository üzerinden Remove, Update gibi metotlar çağrıldığında, değişikliklerin veri tabanına yansıtılması **SaveChanges** metodu çağrıldığında gerçekleşir. Ancak, bu işlemleri ne zaman çağıracağımızı kontrol altına almak önemlidir.

UnitOfWork tasarım deseni, farklı repositorylerde yapılan işlemleri tek bir **transaction** bloğunda toplar ve veri tabanına yansıtır.
Örneğin, **UserRepository** ve **TeamRepository** üzerinde yapılan değişiklikler birlikte **SaveChanges** metotu çağırılana kadar Entity Framework Core tarafından Memory'de tutulur,**SaveChanges** çağrıldığında ise veri tabanına yansır. Eğer bu iki repository üzerinde**SaveChanges** metotunu çalıştırdığınızda **UserRepository** üzerinde veri tabanına yansıma işlemi gerçekleşirken **TeamRepository** üzerinde herhangi bir güncelleme olmazsa veri tabanında tutarsız bir durum oluşabilir. **UnitOfWork** ise bu tutarsızlık durumuna karşın şöyle diyor *"Repositoryler üzerinde değişiklikler yapınız ancak ne zaman* ***UnitOfWork*** *interface üzerinden* ***SaveChanges*** *metotu çağırılırsa bu iki repository üzerindeki değişikliklerden ikisi de başarılıysa veri tabanına yansır. Eğer biri başarısız ise diğer repository üzerindeki değişiklikler de* ***rollback*** *yapılarak geri alınır****."*** EF Core otomatik olarak bir **transaction** yapısı kurup **rollback** işlemini de kendisi yapar. **SaveChanges** kontrol altına almak ve tek bir yerden çağırılmasını sağlamak için **UnitOfWork Design pattern** güzel bir seçimdir. Amacımız birden fazla repository üzerinde yapılan değişiklikleri tek bir transaction üzerinde veri tabanına yansıtmaktır. Herhangi bir hata olması durumunda da rollback ile geri almaktır. Bu temeli de bizlere EF Core sağlayabilmektedir.

```
using System.Threading.Tasks;

public interface IUnitOfWork
{
   Task CommitAsync();
   void Commit();
}
```

Bu interface, **CommitAsync** ve **Commit** adında iki metot içermektedir. Bu metotlar, **SaveChangesAsync** ve **SaveChanges** metodlarını çağırmak için kullanılabilir ve **UnitOfWork** tasarım desenini uygulayan sınıflar tarafından implemente edilebilir.

Bugünlük benim anlatacaklarım bu kadar!

Bir sonraki yazıda görüşmek üzere :)

KAYNAKÇA

- https://www.youtube.com/watch?v=r-RUY2caw3s
- https://www.youtube.com/watch?v=xJC7ItRoEbw
- https://www.youtube.com/watch?v=Srp1iyZu-ww&pp=ygUNbi1sYXllciBwcm9qZQ%3D%3D
- https://www.udemy.com/course/asp-net-core-api-web-cok-katmanli-mimari-api-best-practices/
- https://www.gencayyildiz.com/blog/c-ta-n-tier-architecturecokn-katmanli-mimari/