**CLASSLİB:**

**Class Library (classlib)**, .NET platformunda kullanılan bir projedir ve içerisinde tekrar kullanılabilir kodların toplandığı sınıf kütüphanelerini barındırır. Bu kütüphaneler, diğer projeler tarafından referans alınarak kullanılabilir ve yazılımın modüler yapıda olmasına olanak tanır. **.NET class library** projeleri genellikle ortak işlevleri (iş mantığı, veri erişimi, servisler, vs.) bir araya getirip diğer uygulamalarda kullanılabilir hale getirir.

**Class library projeleri kendi başına çalıştırılamaz, bir uygulama tarafından referans alındığında işlevlerini sağlar**. Uygulamalar (konsol, web, masaüstü) bu kütüphanedeki kodları çağırarak iş mantığını kullanabilir.

\*dotnet new classlib -n MyClassLibrary komutu ile bir classlib oluşturulabilir ve ilgili solutiona dahil edilebilir.

**Class Library Kullanım Senaryoları:**

1. **Veri Erişim Katmanı**: Veritabanı işlemleri ve ORM araçlarını kullanarak veri erişim işlemlerini class library olarak tanımlayabilirsiniz. Örneğin, Entity Framework ile bir veri erişim sınıfı yazıp bunu birden fazla uygulamada kullanabilirsiniz.
2. **Servisler**: Web servislerini veya arka plan işlevlerini yöneten kodlarınızı bir class library içerisine alabilirsiniz. Böylece aynı servisleri farklı uygulamalarda (web, masaüstü, mobil) kullanabilirsiniz.
3. **Yardımcı Araçlar (Utilities)**: Sık kullanılan genel işlevler (dosya işlemleri, metin manipülasyonları, güvenlik işlemleri gibi) için class library projeleri oluşturabilirsiniz.

ASP.NET projelerinde **Class Library (classlib)**, genellikle **Repository Pattern** ve diğer iş mantığı, veri erişim katmanları gibi soyutlamalar için kullanılır. Class Library'yi bu senaryoda kullanmanın temel amacı, veri erişim kodunu ve iş mantığını uygulamadan ayırarak kodun daha modüler, test edilebilir ve tekrar kullanılabilir olmasını sağlamaktır.

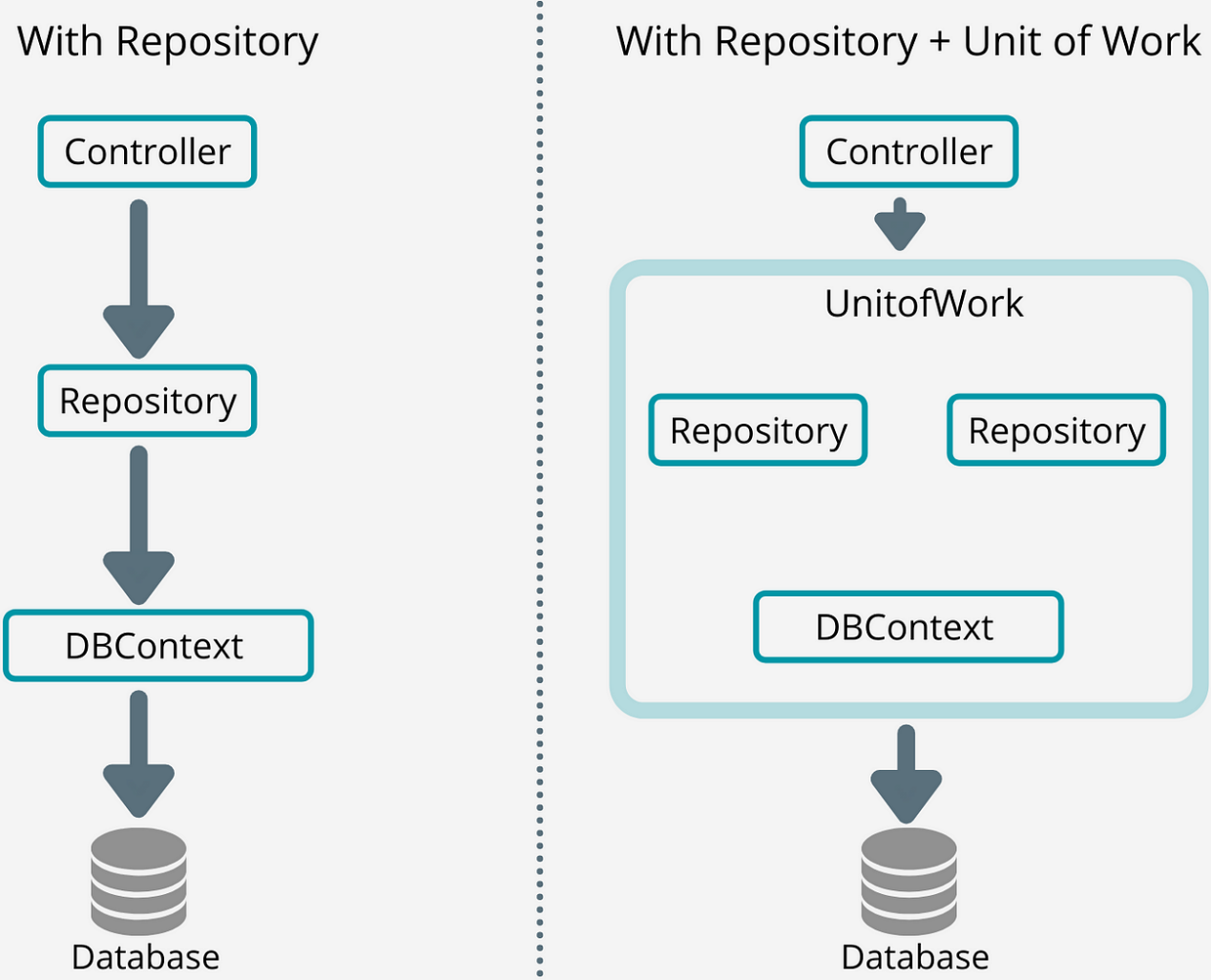
Veri Erişim Katmanının Soyutlanması: Repository’ler genellikle veri erişim işlemlerini yöneten sınıflardır. Bu sınıflar, veritabanı sorgularını yönetir ve veri modeliyle veritabanı arasındaki köprüyü oluşturur. Class Library, bu Repository sınıflarını barındırarak uygulamayı veri erişim katmanından ayırır.

\*Oluşturulan classlib ana projeye referans vermelidir. Uygulamamız Storeapp için ,

dotnet add .\Storeapp\ reference .\Entitites\ şeklide Entitites referansı oluşturulabilir.

**REPOSITORY Kavramı ve UoW:**

Repository Pattern, yazılım geliştirme süreçlerinde veri erişim katmanını yönetmek (repository context) ve kodun daha modüler ve test edilebilir olmasını sağlamak için kullanılan bir tasarım desenidir. ASP.NET projelerinde veri işlemlerini veri kaynağından(database) soyutlayarak, bu işlemleri gerçekleştiren kodları bir repository (depo) içinde toplar.



**Unit of Work** tasarım deseni, bir yazılım uygulamasında birden fazla veri işlemini (CRUD işlemleri) bir arada yönetmek ve işlem bütünlüğünü sağlamak için kullanılan bir desendir.

Unit of Work, bir veritabanı işlemini başlatır, bir veya birden fazla repository ile işlem yapar ve işlem tamamlandığında bu değişiklikleri veritabanına kaydeder. Eğer işlem sırasında bir hata meydana gelirse, tüm değişiklikler geri alınır.

**Örnek Bir Senaryo:**

Diyelim ki bir e-ticaret uygulamasında bir sipariş oluşturuluyor. Bu sipariş işlemi sırasında birkaç farklı işlem gerçekleşiyor:

1. Sipariş bilgisi veritabanına kaydedilir.
2. Stok miktarı güncellenir.
3. Ödeme işlemi yapılır.

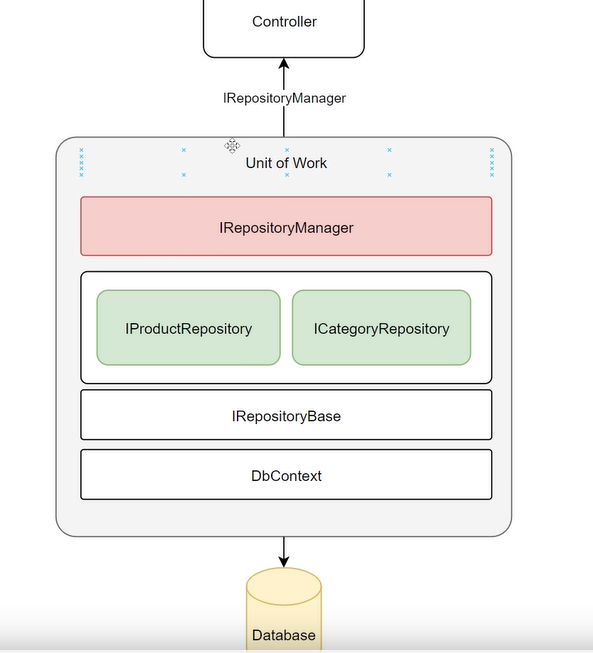
Eğer bu işlemlerden biri başarısız olursa, diğerlerinin de geri alınması gerekir. **Unit of Work** bu senaryoda tüm bu işlemleri tek bir birim içinde yönetir.

DbContext, bir sınıfın doğrudan veri tabanına bağlı çalışmasını sağladığı için küçük projelerde performans açısından genellikle yeterli olur. Ancak, birden fazla DbContext veya işlemler arasında karmaşıklık artarsa yönetim zorlaşabilir.

Unıt of work(UoW) ise birden fazla veritabanı işlemi tek bir iş birimi içinde gerçekleştirilir ve bu işlemler bir bütün olarak başarılı olursa commit edilir, aksi halde hepsi geri alınır (rollback). Bu, veritabanı işlemlerinde tutarlılık sağlar.

**Transaction Yönetimi**: Unit of Work, tüm işlemleri tek bir transaction (işlem) içinde yürütür. Bu, özellikle birden fazla repository kullanıldığında önemli hale gelir. Eğer işlemlerden biri başarısız olursa, tüm işlemler geri alınır.

**IRepositoryBase** genellikle Repository Pattern’in bir parçası olarak kullanılan bir arayüzdür ve **her tür veri varlığı (entity) ile çalışan** temel CRUD (Create, Read, Update, Delete) işlemlerini tanımlar. Unit of Work (UoW) deseni ile bir arada kullanıldığında, veri erişimini daha modüler ve yönetilebilir hale getirir.



IRepositoryBase, tipik olarak bir veri kaynağına (genellikle bir veritabanına) erişim için gereken temel işlevleri sağlar. Bu işlevler, Entity Framework gibi ORM (Object-Relational Mapping) araçlarını kullanarak belirli bir türdeki veri varlıkları üzerinde işlem yapmayı (CRUD)sağlar.

public interface IRepositoryBase<T> where T : class

{

void Add(T entity);

void Update(T entity);

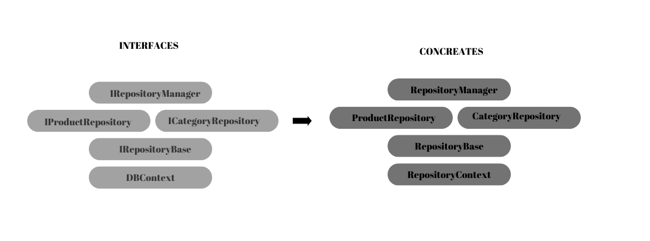
void Delete(T entity);

T GetById(int id);

IEnumerable<T> GetAll();

}

Bu şekildeki bir arayüz, bir veritabanı tablosunu temsil eden varlık sınıfları için ortak veri erişim operasyonlarını soyutlar. Böylece, her bir varlık için (örneğin, Product, Customer, Order) ayrı repository’ler oluşturarak kodun tekrarlanmasını önler ve tekrar kullanılabilir hale getirir.



**EK BİLGİ:**

Veritabanı sadece tablolar ve ilişkiler barındırır.

Entity (Varlık), bu tabloların kod tarafındaki nesne karşılığıdır.

Veritabanındaki Entity’yi doğrudan dış dünyaya açmak genellikle kötü bir pratiktir. Sonrasında DTO ları işleyeceğiz bu yüzden.

Repository genellikle doğrudan Entity ile çalışır.

**SERVİCE(Servis) KATMANI:**

Service katmanı, iş mantığına odaklanır. Repositories'i kullanarak uygulamanın iş akışlarını yöneten ve daha karmaşık iş süreçlerini içeren katmandır. Servis katmanında, birden fazla repository kullanılabilir ve bu repository'ler arasındaki iş mantığını yönetir.

**Service Katmanının Sorumlulukları:**

* **İş Mantığı Yönetimi**: Servisler, uygulamanın iş mantığını kapsar. Örneğin, bir ürün ekleme işlemi yaparken, stok kontrolü veya fiyatlandırma gibi işlemler bu katmanda yönetilir.
* **Birden Fazla Repository Kullanımı**: Service katmanı, birden fazla repository ile çalışabilir. Örneğin, bir sipariş oluşturma işlemi, hem OrderRepository hem de ProductRepository kullanabilir.
* **Karmaşık İş Akışları**: Service katmanı, repository'leri kullanarak daha karmaşık iş akışlarını yönetir. Yani, veriyi yalnızca veritabanına kaydetmek değil, bu süreçteki iş mantığını uygulamaktır.

**Controller > Services > Repositories > DataBase**

Servis katmanı, iş akışlarını yöneterek veritabanı ile doğrudan iletişim kuran **repository** katmanı ve uygulamanın diğer bileşenleri (UI, API) arasında bir köprü görevi görür.

**Servis Katmanının Ana Görevleri ve Faydaları**

**1. İş Mantığını Yönetmek**

Servis katmanı, uygulamanın temel iş mantığını barındırır. Bu, bir işlemin nasıl gerçekleştirileceği ile ilgili tüm adımları içerir. Örneğin, bir kullanıcı siparişi oluşturduğunda, siparişin doğrulanması, envanter kontrolü, fiyat hesaplaması gibi tüm iş mantığı servis katmanında yürütülür.

* **Örnek**: Sipariş oluşturulurken stok durumu kontrol edilir, varsa sipariş onaylanır, yoksa hata döndürülür.

**2. Repository'ler Arasındaki İşlemleri Yönlendirmek**

Bir servis katmanı, birden fazla repository kullanarak daha karmaşık iş akışlarını yönetebilir. Örneğin, bir ProductService hem ürün bilgilerini günceller hem de ürünün ait olduğu kategori hakkında işlemler yapabilir.

* **Örnek**: ProductService, hem ProductRepository hem de CategoryRepository kullanarak bir ürünün kaydını yaparken, aynı zamanda kategoriye göre işlemler yürütebilir.

**3. Karmaşık İş Akışlarını Basitleştirmek**

Bazı iş süreçleri birden fazla adımı veya veritabanı işlemini içerir. Servis katmanı, bu karmaşık iş akışlarını mantıksal olarak organize eder ve bu sayede bu işlemleri çağıran kod (örneğin, controller veya UI) bu detaylarla ilgilenmez.

* **Örnek**: Bir servis, bir ürün eklerken hem veritabanına ekleme işlemi yapar, hem de stok bilgilerini güncelleyebilir.

**4. Tekrar Kullanılabilir Kod Sağlamak**

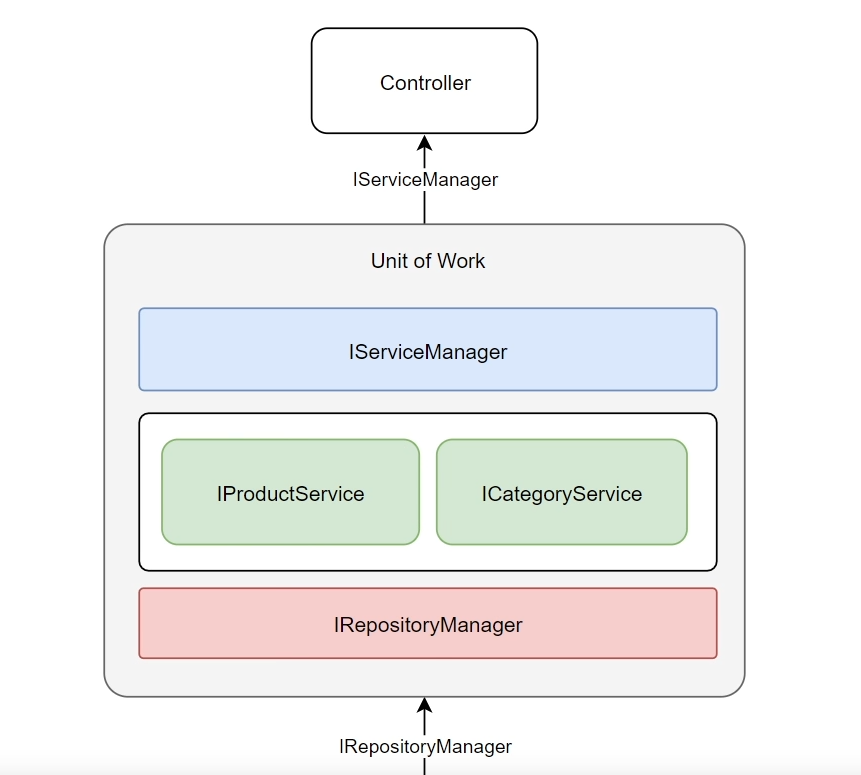
Servis katmanı, iş mantığını merkezi bir noktada topladığı için, aynı iş mantığını uygulamanın farklı yerlerinde kullanmak gerektiğinde aynı servisi tekrar kullanabilirsiniz.Bu da kodun tekrar kullanılabilir olmasını sağlar ve gereksiz kod tekrarından kaçınmanızı sağlar.

* **Örnek**: OrderService, hem web uygulaması üzerinden gelen siparişlerde hem de mobil uygulamadan gelen siparişlerde aynı mantığı uygulayabilir.

dotnet new classlib -f net8.0 -o \Services\ 🡪 Services adlı classlib klasörü oluşturulur.

dotnet sln .\StoreApp.sln add ..\Services\ 🡪 Services, solutiona eklenerek projeye kazandırılır.

Özetle, tasarımı Repositories ile benzer olup repositorieste oluşturulan tanımlamalar iş mantığı ekler.



**VİEWCOMPONENT :**

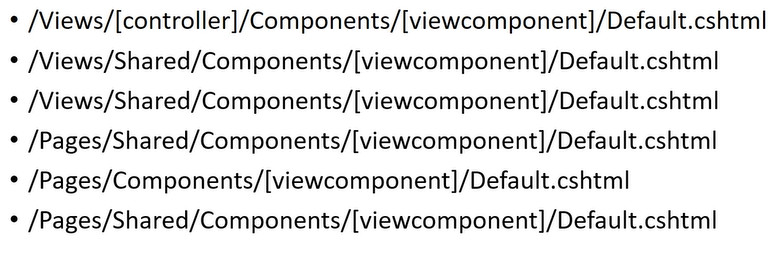
View Component (Görünüm Bileşeni), uygulamanızda yeniden kullanılabilir ve bağımsız bileşenler oluşturmanıza olanak tanır. Bu bileşenler, bir sayfanın belirli bölümlerini dinamik olarak oluşturmak için kullanılabilir. View Component'ler, Razor sayfaları veya MVC görünüm katmanları içerisinde kullanılabilir.

**View Component**

* **Tanım**: View Component, ASP.NET Core uygulamalarında bağımsız ve yeniden kullanılabilir bileşenler oluşturmaya yarar. Her View Component, kendi modelini alır, iş mantığını içerir ve bir görünüm oluşturur.
* **Kapsam**: Genellikle belirli bir işlevselliği temsil eder (örneğin, son gönderiler, kullanıcı bilgileri).
* **Veri Alma**: Veriyi almak ve işlemek için kendi iç mantığını kullanır. Diğer bileşenlere veya verilere bağımlılığı minimize eder.
* **Kullanım**: View Component, Razor sayfalarında veya MVC görünümünde Component.InvokeAsync ile çağrılır.

ViewComponent yapıları PartialView’leri destekleyici bir yapı olarak size bir logic sağlayarak backend tarafında işletebileceğiniz bir fonksiyonalite sağlamış olur. İlgili fonksiyon sayfanın genel görünümünden bağımsız olarak asenkron bir şekilde çağrıldığı için sayfa genelinde bir değişiklik yapmadan sonuç döndürür bu sayede kullanıcı deneyimi artmış olur ve sayfa modern bir hale evrilir.

*ViewComponent* yapıları, View ile ilişkilendirilebilir ve bir *View*'ın parçası olarak çalışabilir. *ViewComponent*, ASP.NET Core'da yeniden kullanılabilir bileşenler oluşturmak için kullanılan bir yapıdır ve bir *Partial View* gibi çalışsa bile dinamik ve bağımsız bir yapıdır. ViewComponent'ler, yukarıda da bahsedildiği gibi çoğunlukla belirli bir iş mantığını veya veriyi işleyip sadece belirli bir kısmı (örneğin bir menü, liste veya küçük bir form gibi) görüntülemek için kullanılır.



ViewComponent yapılarını Coding by Conversion yapısına göre yukarıdaki gibi dosya yollarında tanımlayabiliriz. ViewComponent çağrısı, View dosyasında @await Component.InvokeAsync("MyExample") ile yapılır. Bu çağrı, ViewComponent’in ilişkili View’ını görüntüler.

ViewComponent yapıları sayfalarda kullanılacak küçük bağımsız .cshtml dosyaları döndürebileceği gibi buna ek veri işleme gibi basit işlevsellikler de sağlayabilir. Örneğin, ürünler sekmesinde ürünlerin kaç adet olduğunu parantez içinde dinamik bir şekilde(sürekli güncellenen) görüntüleyebiliriz.

**AREAS:**

ASP.NET Core'da Areas (Alanlar), büyük veya karmaşık uygulamalarda kodu daha düzenli ve modüler hale getirmek için kullanılan bir özellik. "Area" yapısı, uygulamanın belirli bölümlerini kendi başlarına bağımsız birimler gibi organize etmenizi sağlar. Bu yapı özellikle çok sayıda Controller ve View içeren uygulamalarda önemli bir rol oynar.

Areas Yapısının Kullanımı

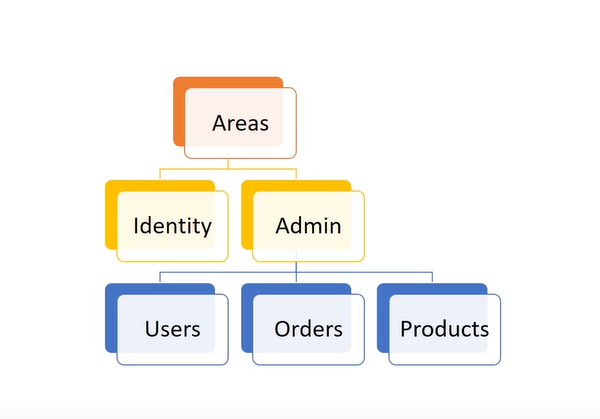
"Areas" genellikle büyük projelerde, özellikler veya bölümler arasında mantıksal bir ayrım sağlamak için kullanılır. Örneğin, bir e-ticaret uygulamasında şu alanlara ihtiyaç duyabilirsiniz:

* Admin: Yönetim paneli işlemleri.
* Customer: Müşteri arayüzü.
* Support: Destek işlemleri.

Her bir "Area" kendi Controllers, Views ve Models dosyalarına sahip olabileceğinden, bu yapı sayesinde uygulama bölümleri birbirinden izole edilebilir.

Örneğin, projemiz kapsamında admin paneli üzerinden çeşitli yönetimler yapılabilir.

NOT:Gerçek projelerimizde bir admin paneline ihtiyaç duyulduğunda “AdminLTE” ile Google’da arama yapabiliriz.



**EK BİLGİ:**

**Route Data** ve **Query String** web uygulamalarında veri taşımak için kullanılan iki farklı yöntemdir.

Route Data : https://example.com/products/12345

* **Route Data**, URL'nin bir parçası olarak belirli bir yapı içinde taşınan verilerdir.
* Genelde bir **web framework'ü** tarafından belirlenen bir rota tanımına uygun şekilde kullanılır.
* Özellikle **RESTful API'ler** ve modern web uygulamalarında çok yaygındır.

Query String: <https://example.com/search?query=shoes&sort=price>

* **Query String**, URL'de "?" işaretinden sonra eklenen ve key=value çiftlerinden oluşan veridir.
* Veriler ampersand (&) ile ayrılır.

**ViewBag Nedir?:**

ViewBag, küçük miktarda veri aktarmak istediğinizde, hızlı ve kolay bir yöntem sunar. Controller'da tanımlanan ViewBag, ilgili View'da doğrudan kullanılabilir. Bu, küçük miktarda veri iletimi için idealdir.

**SelectList**, ASP.NET Core'da kullanılan bir sınıftır ve genellikle bir dropdown listesi (açılır menü) oluşturmak için kullanılır. SelectList, bir veri kaynağından gelen verileri kolayca **<select>** etiketi için uygun formata dönüştürür.

            ViewBag.Category = new SelectList(\_manager.CategoryServices.GetAllCategories(false), "CategoryId", "CategoryName", "1");

**"CategoryId"**: Bu, açılır menüdeki her bir seçeneğin **değeri** olacak olan özellik. Kullanıcı bir kategori seçtiğinde, bu CategoryId değeri formda gönderilecek.

**"CategoryName"**: Bu, açılır menüde gösterilecek olan **görünür metin**. Kullanıcıya hangi kategoriyi seçtiğini göstermek için kullanılacak olan metin. Örneğin, kullanıcıya "Elektronik", "Giyim" gibi kategoriler gösterilecek.

**"1"**: Bu, açılır menüde **varsayılan olarak seçili** olan değeri belirtir. Bu durumda, CategoryId'si "1" olan kategori başlangıçta seçili olacak. Varsayılan1 olan kategori, başlangıçta seçili olur.

**asp-items Nedir?**

* **Amacı:** <select> etiketine, bir **seçenekler listesi (options)** sağlar.
* **Kaynağı:** Seçenekler listesi genellikle bir **SelectList**, **List<SelectListItem>**, veya **IEnumerable<SelectListItem>** nesnesinden alınır.
* **Dinamik Veri:** Bu liste, veritabanından, API'den veya kod içinde oluşturulan bir veri kaynağından doldurulabilir.

ekran görüntüsü içeren bir resim

Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

metin, ekran görüntüsü, yazı tipi içeren bir resim

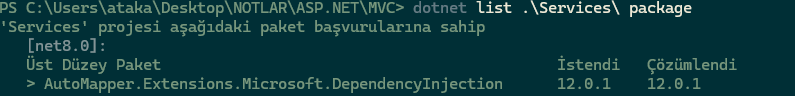
Yapay zeka tarafından oluşturulan içerik yanlış olabilir.

**AUTO MAPPER:**

AutoMapper, .NET platformunda kullanılan bir nesne-nesne eşleme (object-object mapping) kütüphanesidir. AutoMapper, bir nesneden başka bir nesneye **otomatik olarak veri eşleme (mapleme)** işlemi yapan bir kütüphanedir.

AutoMapper, bir kaynak sınıf (source class) ve bir hedef sınıf (destination class) arasındaki eşlemeleri (mapping) tanımlayarak çalışır. Alan adları ve türleri aynı olduğunda eşleme işlemini otomatik olarak yapar.

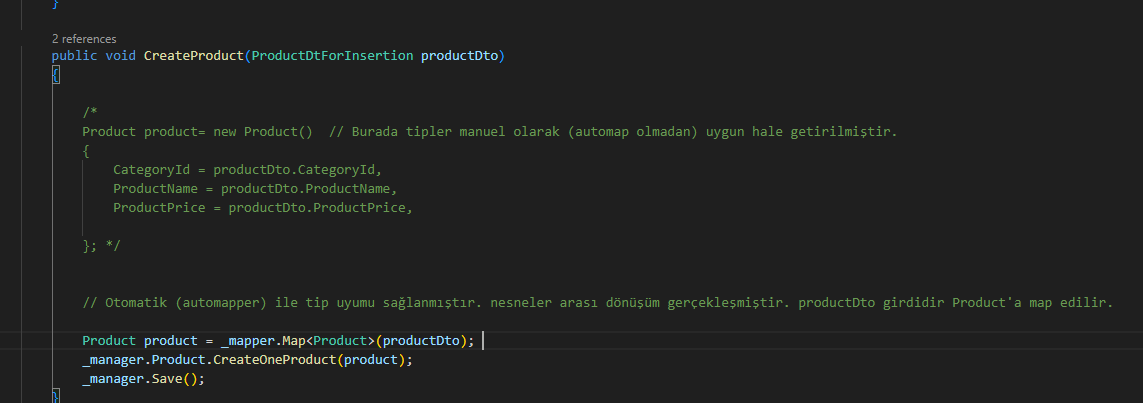
dotnet add package AutoMapper.Extensions.Microsoft.DependencyInjection



Ardından Program.cs dosyasında AutoMapper eklenir. Bu işlem, AutoMapper'ı global olarak uygulamanıza dahil eder.

Ardından bir profile sınıfı tanımlanır. **Profile**, AutoMapper'ın **dönüşüm kurallarını** içeren sınıftır. AutoMapper bu sınıfı kullanarak hangi türlerin birbirine nasıl dönüştürüleceğini öğrenir. Profile sınıfı içinde *CreateMap<TSource, TDestination>()* gibi yöntemlerle dönüşüm kuralları tanımlanır. Uygulama başlatıldığında, MappingProfile sınıfındaki dönüşüm kuralları AutoMapper'a tanıtılır. (IoC kaydı sayesinde)

Projemizde Infrastructure/Mapper/ MappingProfile.cs şeklinde tanımlanmıştır



Yukarıda CreateOneProduct metodu, Product türünde bir nesne istiyor ancak CreateProduct’a gelen productDto bu tür uyuşmazlığını kaldırmak için yorum satırına alınmış koddaki gibi manuel bir mapleme işlemi yapılabileceği gibi AutoMapper ile süreç otomatize edilebilir.

**DTO(Data Transger Object):**

**DTO (Data Transfer Object)**, aslında **veri taşıma aracı** gibidir. Yani, bir yerden başka bir yere veri taşımak için kullanılan bir nesnedir. ASP.NET Core'da ise **katmanlar arasında veri taşımak** için kullanılır. Bu katmanlar genellikle:

* **Veritabanı katmanı** (Data Access)
* **İş mantığı katmanı** (Business Logic)
* **Sunum katmanı** (UI veya API) vb.

**Neden DTO Kullanılır?**

1. **Veri taşımak** için: Diyelim ki veritabanından bir kullanıcıyı çektiniz ve bunu **UI (kullanıcı arabirimi)** veya **API'ye** göndereceksiniz. Ancak, veritabanındaki tüm bilgiler, her zaman UI ya da API için gerekli değildir. **DTO**, sadece gerekli olan verileri taşımak için kullanılır.
2. **Gereksiz verilerden kaçınmak** için: Veritabanındaki tüm veriyi göndermek yerine, sadece ihtiyacınız olan bilgiyi göndermek daha **verimli ve güvenli** olur. Mesela, kullanıcı şifresini UI'ye ya da API'ye göndermek istemezsiniz.
3. **Farklı katmanlar arasında veri iletimi** için: Bir katmandan diğerine veri aktarırken, veri modelinin değiştirilmesini önlemek için DTO kullanılır. Yani, UI ve veritabanı bağımsız olur, bu da uygulamanın daha kolay bakımını sağlar.

DTO, **gereksiz verilerin taşınmasını engelleyerek** sadece **gerekli bilgileri** taşır. Böylece, hem **performansı artırır**, hem de **güvenliği** sağlar.

