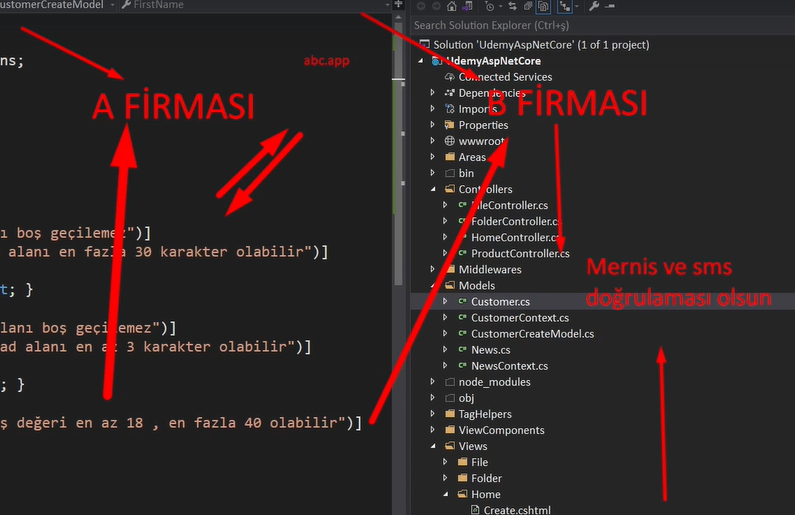
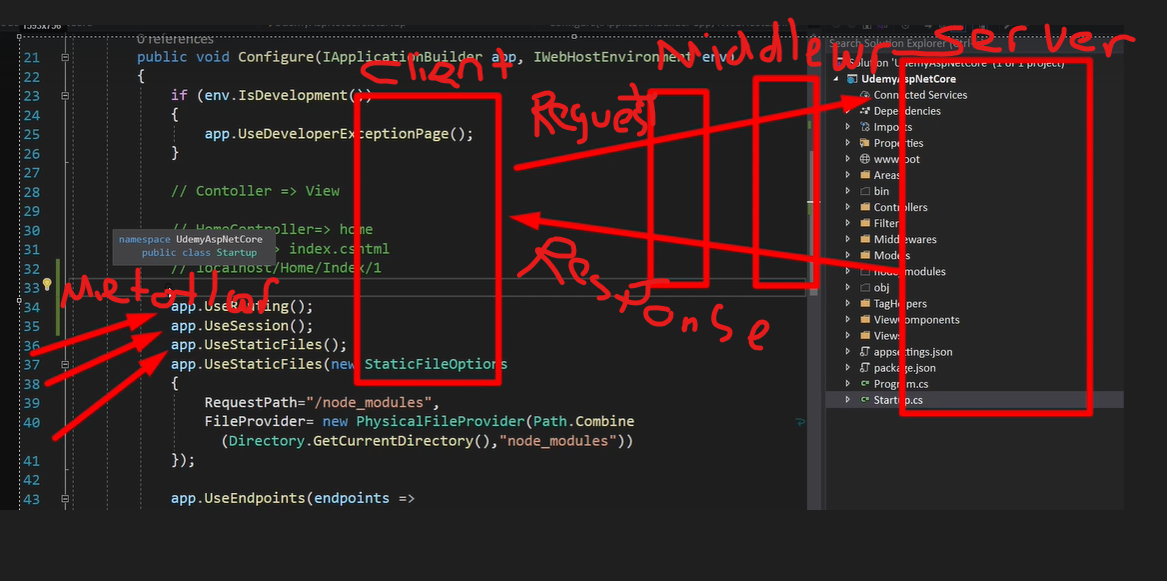
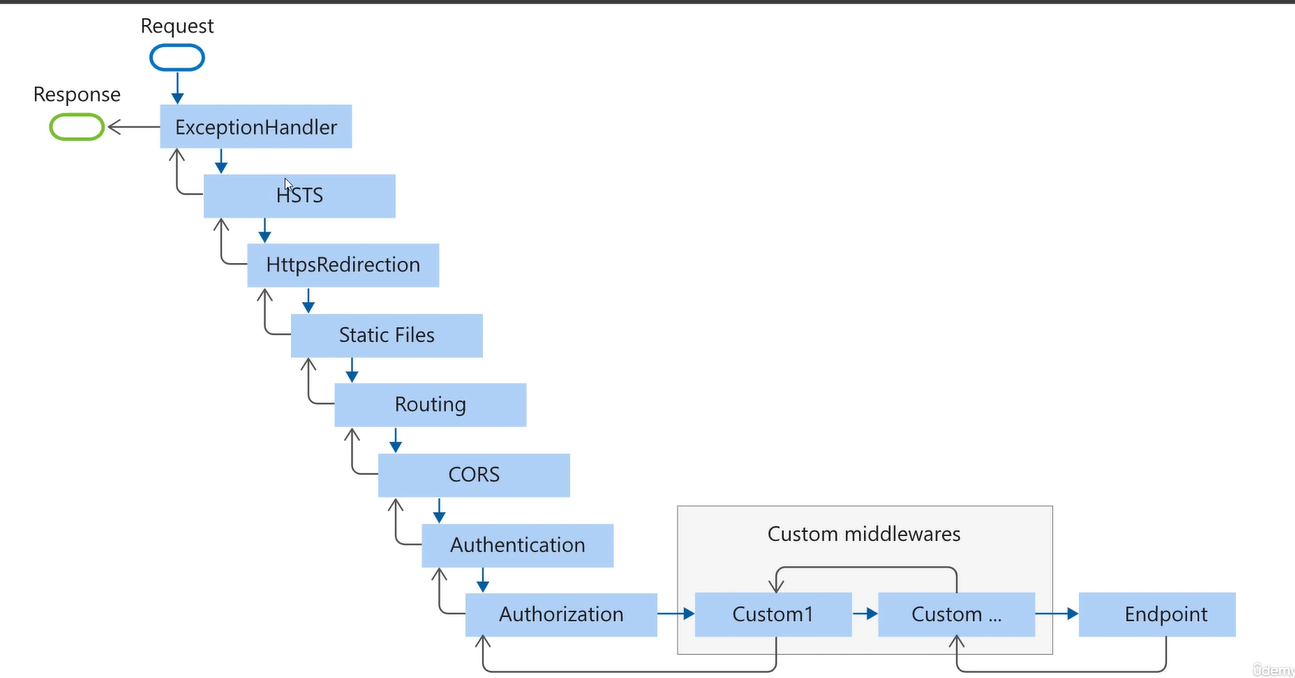
**MVC .NET CORE NOTES**

**TEMEL KAVRAMLAR**

* .Net’ göre daha hızlı çalışır. Nedeni kestrel web server ve pipeline entegrasyonudur. Kestrel web server hafif bir sunucudur.
* .Net Core uygulamaları açık kaynak kodludur. Platformdan bağımsızdır, cross platform.
* Asp.Net Core uygulaması ayağa kalktığında ilk Program.cs dosyasına gider ve Main metotunu çalıştırır. Main metotu default ayarlarla uygulamayı Startup dosyasını kullanarak ayağa kaldırıyor.
* Startup dosyasının içindeki ConfigureServices metotu container’in içerisinde servisleri ekler.
* Startup dosyasının içindeki Configure metotu aracılığıyla HTTP request pipline’nını Configure edebiliriz.
* Startup dosyasının mantığı ConfigureServices metotu içerisine servisimizi eklersek Configure metotunda’da bu servisi configure edip kullanırız.
* Startup dosyasındaki Configure metotu içindeki if bloğunun mantığı uyguluma Development modda ise sen ExceptionPage’i göster ExceptionPage’te kullanıcıya göstermek istemeyeceğimiz hatayla ilgili kritik bilgiler mevcuttur. Fakat eğer development modda değilse
* Program uygulamanın development modda olup olmadığını launchSettings.json dosyasındaki profiles kısmındaki ASPNETCORE\_ENVIROMENT’TA “Development” yazıyor ve Projede ASPNETCORE\_ENVIRONMENT’te de “Development” yazıyor. Bunları “Production” olarak değiştirirsek hata sayfalarını göremeyiz ExceptionPage’i yani.
* Startup dosyasındaki app.UseRouting() ise url’e anlam kazandırıyor. Eğer bu kullanılmasa sitenin urllerinde bir anlam olmaz.
* Startup dosyasındaki app.UseEndpoinst ise eğer sayfa linkinde hiçbir ek yoksa ysk.com.tr/ gibi direkt açılan sayfadaysa asecron şekilde response’a Hello World yaz diyor yani url geldiğinde nasıl bir olayın gerçekleşeceğini belirten kısımdır.
* Bir .NetCore uygulamasını IIS Express aracılığıyla ayağa kaldırabiliriz.Kestırıl Web Server ile de ayağa kaldırabiliriz. Ya da cmd üzerinden dotnet run diyerek ayağa kaldırabiliriz.
* Uygulama ayağa kalkınca hangi port numarası ile ayağa kalkacağının ayarlamaları launchSettings.json dosyası içeriisnde yapılır. ApplicationUrl ayarıdır.
* Uygulama ayağa kalkarken launchSettings.json dosyasına bakıyor ve oradaki yapılandırmalar ile ayağa kalkıyor.
* Web uygulamaları Client Server Mimarisi’ni kullanır. Client tarafı bir istek yapar mesela bu istek bütün kullanıcıları getirme olabilir.Bunun karşılığı Request’tir.Server tarafı bu isteğe bir cevap döner. Bunun karşılığı ise Response’tur. Hatırlarsak Startup dosyasında Configure metotu içerisinde HttpRequestPipeLine’ı yapılandırabiliyorduk. İşte bu yapılandırdığımız kısım Request ve Response’tur. Bunun karşılığı ise Middleware’dir.Request ve Response olaylarını configure etme durumuna Middleware denir.Bu işlemi gerçekleştirebilmek için Delegeler’e ve Async metotlara ihtiyacımız var.
* Delege birden fazla metotu veya bir metotu tetikleyen yapıdır. Bir delege oluşturup sırasıyla ilgili metotları çalıştırabiliriz. Bir delege belirtirken önce delegate keywordu belirtilmeli. Daha sonra bu delegenin tetikleyeceği metotlar geriye ne dönüyorsa geriye döndüğü değer türünü daha sonra delegenin ismini ve daha sonra () içerisine delegenin tetiklediği metotlar hangi parametreleri alıyorsa sırasıyla ilgili parametreleri yazarız. Daha sonra bunu kullanmak için delegeden bir nesne oluşturmamız gerekiyor. New dedikten sonra () içerisine ilgili metot ismi yazılır bu metot ile başlayacaksın denir. Daha sonra bu metotu çalıştırmak için ise delegeismi.Invoke() metotunu kullanmamız lazım. Yani özetle delegeler sırasıyla metotları tetikleyen yapılardır. 3 Adet delge türü var func, action, predicate.
* Senkronik mimari kişileri vezneye tek tek alır. Ahmetin işi bitmeden sıra Mehmet’e geçemez önce Ahmet’in işi bitmeli. Vezne sayısı arttılırılırsa bu sefer diğer veznede de aynı olay olabilir. Asekronik mimari bize bu işi böyle yapma diyor. İlk vezne sadece faturayı alsın ve fatura ödeme işlemi için Ahmeti ikinci vezneye yönlendirsin. Yani işlemler sırasıyla gerçekleşiyor. Sistem bloklanmıyor Senkron yapının aksine.
* Asenkronik mimari bize birbirini blocklamayan kod kümelerini yazmamıza olanak sağlıyor. Mesela bir çarpma toplama yapan uygulamamız var eğer asenkronik yapı kullanamzsak toplama işlemi yapılırken çarpma işlemi yapamayız. Asenkronik yapı kullanırsak çarpmayı ve toplamayı aynı anda yapabiliriz sistemi bloklamayız önce toplama işlemi bitmeli diye.
* Biz uzun süren işlemleri bloklamamak için Ascync kullanırız. Update ve Delete’nin Async metotları yoktur çünkü işlemleri kısa sürer.
* Asenktronik yapıda tanımalma yaparken sadece Task yazılırsa bu void demek geriye değer döndürmez yani ama ğer Task<> kullanırsak bu geriye bir değer döndürecek anlamına geliyor.
* Task.Run() metotu bizden bir ActionDelege istiyor.
* Action Delege herhangi bir tipten birden fazla değer alıp geriye herhangi bir şey dönmeyen delegelerdir.
* Predicate Delege sadece bir adet değer parametre alan ve geriye bool değer döndüren delegelerdir.
* Func Delege birden fazla parametre alıp geriye herhangi tipten bir şey dönen delegedir.
* async keyword’u ilgili işi async hale getiriyor. Await ise yazıldığı yerdeki buradaki işlem sonuçlanana kadar ondan sonraki yazılan kod bloğuna geçme demek oluyor. Await kullanılınca önce o kod bloğunun işlemi biter sonra peşinden gelen kod bloğuna geçilir.
* Request ve Response’ın Head ve Body alanı vardır. Request şöyle isteklerde bulunur; Get,Post,Put,Delete şeklinde istekler yaparız. Mvc uygulamalarında sadece Get ve Post kullanırız. API tarafında ise Put ve Delete işlemlerini gerçekleştiririz. Bir veriyi bir kullanıcı bilgisi getitme veya tüm kullanıcıların bilgisini getirirken Get kullanırız veya bir kullanıcıyı silerken yine Geti kullanırız. Fakat bir kullancıyı güncellerken veya eklerken Post’u kullanıcaz.
* Ben Servera bütün kullanıcıları listele diye bir istekte bulundum eğer server ilgili kullanıcıları bulup listeyebilirse server bize 200 şeklinde bir cevap verir karşılığı OK’ dir. Bu cevaba Status Code denir. İlgili ürünleri bulamazsa 404 NotFound döner. 404 bir StatusCode’tur NotFound ise o StatusCode’un anlamıdır. 500 Problem server tarafında bir problem oluştuğunu bize söyler veritabanı ile bağlantı yaparken bir problem olabilir, Verileri getirirken bir alanı boş bırakmış olabilirsiniz. Ben buradaki bütün olaya HttpContext adında bir nesne aracılığıyla erişebiliyorum. httpcontext.response diyip response’a httpcontext.request diyip request’e erişebiliyorum. Middlewear ise Response’u ve Request’i şekillendirebilen ara katmandır. Aslında aralara ara yazılımalar ekliyoruz. İlgili Request’imiz Client’ten Server’a giderken ara yazılımlardan geçiyor. Aynı şekilde Response Server’den Client’e giderken ara yazılım Middlewear’lardan geçiyor bu şekilde işi manipüle edebiliyoruz. Birden fazla Middlewear ekleyebiliyoruz. Middlewear’ı server tarafına yönlendiren middlewear içindeki Invoke’tur. Url’deki ysk.com.tr/yavuz buradaki /yavuz benim Path’im oluyor.
* Asp NetCore uygulamalarının HTTP kütüphanesinin içinde bir delege vardır. RequestDelegate adında.
* Middleware oluşturduktan sonra kullanabilmek içni startup dosyasının içine app.UseRouting() altına app.UseMiddleware<middlewareismi>() ekliyor olmamız gerekir.
* Dependisiy İnjection kanstraktır içeirsinde ilgili parametreyi gördüğü zaman örneği alır ve bu örneği yukarıda tanımlamış olduğumuz private değişkenin içerisine atıyoruz.

**MVC**

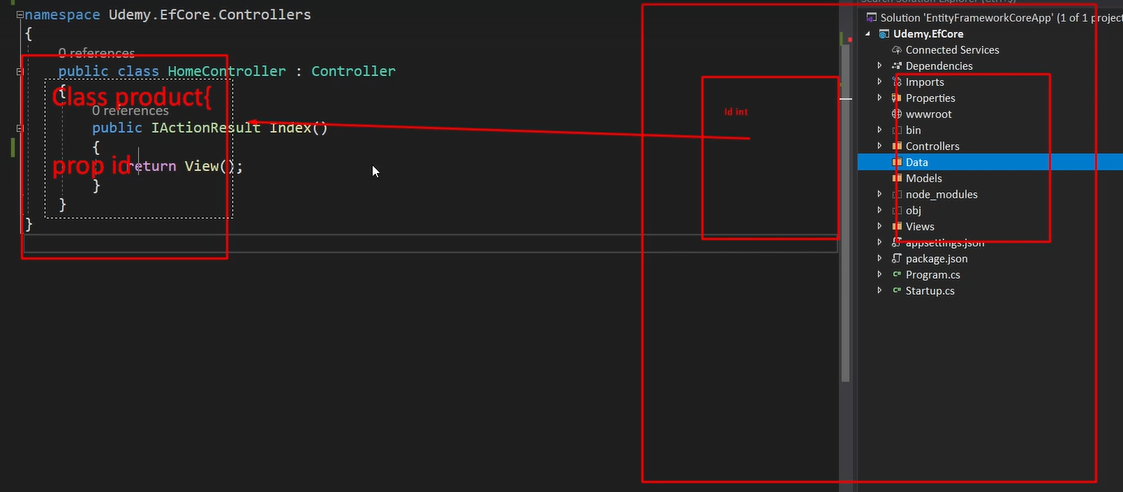
* Views: İlgili kullanıcının gördüğü kısmı temsil eder.
* Controller: View’leri kontrol eden yapıdır.
* Models: İlgili nesnelerdir.
* Bir veri kaynağından bir veriyi çektiğim zaman bu veriyi bir modele bind ederim ve controller içerisindeki action metotlar aracılığıyla ilgili view’e gönderirim. Mvc mimarisi bu şekilde çalışır.
* Projeye MVC ekleyebilmek için projemize bir şey eklerken StartUp dosyasındaki ConfigureServices metotumuzun içersine service ekliyorduk. Oraya gelip services.AddContollersWithViews() diyoruz. Daha sonra Configure metotunun içerisine ise app.UseEndPoints(endpoints=> {}) diyip içine bir delege geçiyoruz. Buranın içerisinde endpoints.MapControllerRoute() metotunu kullanırız. Bu metot içerisine bir routing şeması tanımlıyoruz istediğimiz gibi bir şema. Name kısmına routun ismini veriyoruz pattern kısmına şemayı veriyoruz defaults kısmına ise varsayılan bir ayarlama yapıyoruz. Yani Controller bilgisi gelmezse sen otomatik şu controller’a git action bilgisi gelemzse sen otomatik şu aciton’a git gibi bir söylem yaparız.
* CSHTML dosyasında c# kodu yazmak istiyorsak @ kullanıyoruz tabi eğer tek satır kod yazacaksak bunu kullanırız ama birden fazla satırda c# kodu yazacaksak @{} kullanırız. Buna Razor Syntax denir.
* CSHTML dosyasında bir şart yazacaksak @() kullanabiliriz. Buna Razor Syntax Statement denir.
* Bir CSHTML dosyası içerisinde metot yazıp kullanabiliriz. Buna Razor Syntax Function denir.
* Bir CSHTML dosyası içerisinde bir liste oluşturup daha sonra bu liste içeirsinde dolaşabiliriz. Buna Razor Syntax Loops denir.
* Action metotundan View’e veri taşıyabiliriz. Bunun için 4 adet yöntem bulunur; Viewbag,Viewdata,Tempdata,Model. Öncelikle ViewBag ile data taşıyabiliyoruz. ViewBag.isim = data; şeklinde controller action’da tanımlama yapabiliyoruz. Daha sonra CSHTML içerisinden @Viewbag.isim şeklinde datayı view’a alabiliyoruz.
* ViewData’nın çalışma mantığıda ViewBag’e çok benzerdir. Önce action metotunda VieData[“isim”] = data şeklinde bir tanımlama yaparız. Daha sonra CSHTML dosyasında @ViewData[“isim”] şeklinde bunu çağırmış oluruz.
* TempData ViewBag ve ViewData’dan farklıdır. Farkı ise TempData sayesinde actionlar arasında veri taşıyabiliyoruz. Öncelikle ilgili action metotu içinde TempData[“isim”] = data diyerek tanımlama yaparız. Daha sonra CSHTML dosyasında @TempData[“isim”] şeklinde bunu çağırmış oluruz.
* Model ile veri taşımak için bir modele ihtiyacımız var. Daha sonra ilgili action metotu içerisinde ilgili modelden bir nesne üretilir. Daha sonra bu nesnenin içindeki bilgiler doldurulacaksa doldurulur ve View() içerisine bu nesnenin ismi verilir. Bu sayede ilgili bilgiler CSHTML dosyasına gönderilmiş olur fakat CSHTML dosyası bu bilgiyi bekliyor olmalı onu karşılamalı dolayısıyla CSHTML dosyasının en üstün @model modelin namespacesi.modelismi şeklinde yazarız. Artık CSHTML dosyamızın bir modeli olmuş oluyor. @Model. şeklinde modelin özelliklerine ulaşabiliyorum.
* Controller içeriinsdeki Action metotlarının içindeki View() metotları ilgili CSHTML dosyasını gelir bize gösterir. View metotunun içerisine bir Model gönderebililiriz. Bir ViewName gönderebiliriz. ViewName ile veraber bir Model gönderebiliriz. Eğer bir View ismi girersek View(“Ahmet”) şeklinde ilgili controller dosyası içeirsinde Ahmet adında bir View arar ya da Shared dosyasının içerisinde de arar. Ancak bu şekilde bir kullanım yaparsak direkt aciton metotu içerisindekiler çalışmadan ilgili view’a gidilme yapılır. Mesela ben action metotu içinde bir veritabanı bağlantısı işlemi yaptım diyelim ve böyle bir View kullanımı yaptım. Uygulama çalışınca o veritabanı bağlantısını görmezden gelerek sadece ilgili CSHTML dosyasını çalıştırır. Bu yüzden bu gibi durumlarda RedirecToAction() metotunu kullanmamız lazım.
* Shared klasörü bütün Controller’lar için ortaktır.
* RedirecToAction() metotu doğrudan ilgili Actionu çalıştırıp verilen Actionu çalıştırır. View’den başka bir farkı da bu CSHTML değil verilen Action a gidiyor. Bu metot ile başka bir controller’a da geçiş yapabiliyoruz. Bunu yapmak istiyorsak önce ilgili Action adını daha sonra ise ikinci parametre olarak Controller adını verebiliyoruz. Başka bir parametre olarak ise mesela ilgi action metodunun parametresinde int id bilgisi var biz bu parametreye new{@id=1} id ye bu şekilde değer koyup gönderebiliyoruz.
* Bir interface örneklenemez ancak bir interface bir nesne örneği taşıyabilir. Kendisinden kalıtılan nesne örneğini taşıyabilir.
* Bir ürün detay sayfası için route şemasına üçüncü parametre olarak {id?} ekleyebiliriz ancak ? koymamız gerek opsiyonel yapmış oluruz ve bu sayade üçüncü parametreyi vermek zorunda kalmayız sadece lazım olduğunda devreye girer bu sayede hata almayız ayrıca id:int diyerek bunun sadece int değer olabileceğini sağlayabiliriz yada id:alpha diyerek bunun sadece alfabeten bir harf olmasını sağlayabiliriz.
* Özel route şemaları da oluşturabilriz. Bunun için ana route’un üst kısmına endpoints.MapControllerRoute diyerek yeni şemayı oluşturabiliriz. Daha özel olan route’u her zaman daha yukarıya yazmalıyız. Kontrol yukarıdan aşağıya yapılır. Buna uyuyor mu uymuyorsa alttaki şemaya geç gibi.
* Route’dan data çekmek için biz RouteData.Values diyerek RouteData prop’u sayesinde tüm Route bilgisine ulaşabiliriz bu bilgi bize sözlük yapısıyla gelir. RouteData.Values[“id”] diyerek id bilgisine ulaşabiliriz.
* Route belirtme işini controller içerisinde ilgili action metotunun üstüne [Route(“isim”)] yazdığımızda isim yazıldığında bu action’a erişilir. Bu Attirubte routing startup dosyasındaki route olaylarını ezer. Bu Attirubte routing’i action metotuna kurabileceğimiz gibi bir controller’ın üzerine’de kurabiliyoruz.
* Asp netcore uygulamalarında dosyalar dışarıya kapalıdır. StartUp dosyasında Configrue metotu içerisinde app.UseStaticFiles() dersek wwwroot adında bir klasörü dışarıya açmış oluruz.
* Projeye package.json dosyasını eklemek için solution’a add diyerek new itern diyoruz daha sonra arama kısmınayazarak npm Configuration File ekliyoruz. Paketlerimizi package.json dosyası içerisindeki devDependencies içerisine kuruyoruz. Bootstrap vs. Node\_modules dosyasını kurmak için ise projenin üstüne sağ click atıp open folder diyoruz ve orda cmd çalıştırıp npm i diyoruz. Eğer node\_modules geldikten sonra package.json içerisinde yaptığımız kütüphane tanımlamaları dependencies’te gözükmüyorsa dependencies’te içindeki npm klasörüne sağ tıklayıp restore packages deriz. Eklediğimiz eksik kütüphaneler projeye dahil edilmiş olur.Node\_modules dosyası dışarıya erişime kapalıdır tıpkı wwwroot gibi bunu startup dosyası içerisinde Configure metotunda app.UseStaticFiles ekleyip gerekli birleştirme işlemini yapıyoruz.
* Action metotları varsayılan olarak [HttpGet]’tir.
* Create Formu oluşturuken action’a ilgili post aciton metotunun yolunu veririz ve method özelliğini ise post olarak işaretleriz.
* Request ve Response’a ben HttpContext. Nesnesi üzerinden erişebiliyoruz. Form ile aslında biz action metota bir post isteği gönderiyoruz. Dolayısıyla biz bu request’e ilgili post action metotu içerisinde HttpContext.Request.Form[“ilgili inputun name’ine verilen isim”] şeklinde erişebiliyoruz. Formun içerisindeki ilgili inputa erişmek istiyorsam ilgili inputun name özelliğine verdiğimiz değeri yazmamız gerekir. Bu sayede input içerisine girilen bilgilere ulaşabiliyorum.
* Delete işlemi için Find() metotunu kullanabiliriz fakat find metotu bizden predicate delegate ister. Lambda Exprreson ile bunlar halledilebilir.
* Update metotu için kullanacağımız Where() metotu bizden bir Func Delege istiyor. Bunu Lambda Express ile halledebiliriz. Ancak Where() metotu geriye birden fazla kayıt yani IEnumerable değer döndürür. Where metotu yerine FirsOrDefault metotunu kullanabiliriz. FirstOrDefault geriye ilk bulduğu değeri döndürür sadece. Eğer değer bulamazsa null döndürür. FirsOrDefault metotu da bizden func bir delegate ister. Bu işi Lambda Expresson ile halledebiliyoruz.
* Update işlemi yaparken CSHTML dosyasında inputa girilecek bilgileri backend tarafında yakalamak için bir NAME özelliği veriyoruz. Kullanıcı ilk Update sayfasına girdiğinde inputların içi ilgili ürünün eski bilgileriyle dolu halde gelmesi için bir VALUE özelliği veriyoruz VALUE değerlerini setliyoruz bu set etme işlemi ise ilgili Controller’da ilgili HTTPGET action metotundan tüm dataseti model olarak sayfaya gönderdiğimiz için oluyor yani VALUE inputun içine dolduracağı bilgiyi controller’da ilgili HTTPGET action metotundan gönderilen model’den çekiyor. Benim bu CSHTML sayfası içeirsinde Id’ye de ihityacım var. Hangi ürünün bilgilerini güncelliyorsam o ürünün ıd bilgisini de bu form içerisinde tutmam gerekiyor. Bu sebeple form altına en başa yeni bir input açılır ve TYPE özelliğine hidden denilir çünkü ben bu ıd değerini burada tutucam ama bu değeri kullanıcılar göremeyecek. Bu inputunda NAME özelliğine ıd veriyoruz server tarafında bu bilgiyi yaklamak için ve value değerine de ilgili @Model.Id veriyoruz. Ve kullanıcının seçtiği ürün bu HİDDEN inputun içerisindeki value bilgisine 1,2,3 şeklinde düşüyor ilgili ürün seçilmiş oluyor . Daha sonra bu HİDDEN inputtaki NAME özelliğine verdiğim değer bilgisi sayesinde VALUE bilgisine düşen sayısal değeri ilgili Controller içerisindeki HTTPOST action metotunda yakalayıp bu id değeri ile veritabanındaki ürnün id bilgisini karşılaştırıp ilgini ürünü bulup ilgili güncelleme işlemlerini yapabiliyorum.
* İnputların value bilgilerini ilgili name özellikleri ile alabiliyoruz.
* CSHTML dosyasına TagHelper eklemek için sayfanın en başına @addTagHelper \*,Microsoft.AspNetCore.Mvc.TagHelpers yazarsak tüm TagHelper’ları CSHTML dosyasına eklemiş oluruz.
* TagHelper ekledikten sonra HREF=[/Home/Update/@item.Id](mailto:/Home/Update/@item.Id) şeklinde yol tanımlamaları yapmak yerine, asp-aciton diyerek hangi action metotuna gideceğini söyleyebiliriz. Asp-controller diyerek hangi controller’a gideceğini söyleyebiliriz. Bir id datasını taşıyabilmek için ise asp-route-id(taşıyacağınız data) şeklinde tanımlama yapabiliriz. Formun inputlarının TYPE ve NAME alanını belirtmek zorunda değiliz, asp-for tag helper’ıyla biz bu işi gerçekleştirebiliyoruz. Asp-for bizden bir model bekliyor bu inputu ben ne için oluştarayım diyor. Asp-for helper’ı eğer ilgili property doluysa onu otomatik olarak setler. Yani update yaparken value değerini yazmamıza bile gerek yok kendisi setler.
* Model binding ile action metotlarının parametre içlerine mesela bir customer ekleme olayı yapacaksak parametre içine customer ile çalışacağımızı söylüyoruz Customer customer şekilinde ve model binding tüm ayarlamaları otomatik olarak setliyor. Asp-for sayesinde inputun TYPE ve NAME alanı otomatik ayarlanıyor VALUE bilgisi boş bırakılıyor. Bizde Controller içindeki aciton metotuna parametre olarak o modeli verdiğimizde kullanıcının inputa girdiği VALUE değeri controller’a ulaşmış oluyor ve o parametre sayesinde veritabanı üzerine ekleme yapabiliyoruz. Yani VALUE bilgisini bir bakıma database’e eklemiş oluyoruz biz sadece gerisini asp-for ve model binding ayarlıyor otomatik olarak. CSHTML dosyasındaki inputlardan asp-for sayesinde kullanıcının inputa girdiği VALUE bilgisi buraya ulaşıyor o bilgiyi Customer customer sayesinde alabiliyoruz. Aslında tüm bilgiler ulaşıyor ama asp-for bu bilgileri otoamtik dolduruyor TYPE ve NAME bilgisini otomatik dolduruyor bizi uğraştırmıyor bizde sadece bu şekilde bu bilgileri veritbanına ekliyoruz.
* ÖZETLE MODEL BİNDİNG İLE FORMDAN GELEN KULLANICININ GİRDİĞİ İNPUT BİLGİLERİNİ (NAME,TYPE,VALUE) OTOMATİK VE EŞZAMANLI OLARAK ACTİON METOTUNUN PARAMETRESİNE TANIMLAYACAĞIMIZ MODEL İLE SERVER TARAFINDA ALABİLİRİZ. OTOMATİK OLMASINI SAĞLAYAN İNPUTLARDAKİ ASP-FOR HELPER’IDIR. ASP-FOR BİZDEN BİR MODEL BEKLER @Model.FirstName ŞEKLİNDE BU AYARLAMALARI MODELİN HANGİ ALANI İÇİN YAPACAĞINI SORAR.
* Layout ortak olarak tüm CSHTML sayfalarında kullanılacak (navbar-footer gibi) alanları tanımladığımız kısımdır. Layout bütün View’lerin ortak olarak gördüğü nokta olan Shared içerisine oluşturulur. @RenderBody() ile değişecek olan kısımı işaretleyebiliyoruz. CSHTML sayfalarının kullancağı Layoutları belirtmek için sayfa başına @{Layout = “~/layout yolu”} yazılır eğer sayfanın layout kullanmasını istemiyorsak buraya null yazılır.
* Viewİmports dosyası sayesinde CSHTML dosyalarının üstlerinde model tanımlaması yaparken tek tek uzun uzun dosya yolu tanımlaması yapmak yerine Viewİmports dosyasının içerisine bir kere bu tanımalmarı yaparsak CSHTML dosyalarında model tanımlarken kısa bir tanımlama yapabilriiz sadece modelin ismini yazarak. Viewİmports tüm CSHTML dosyaları için ve Shared klasörü içindeki dosyalar için ortak olacağı için Views klasörünün ana dizininde oluşturulmalıdır.
* Klasör ile ilgili işlem yapmak için Controller içeirsinde FolderController ekleyebiliriz. Burada wwwroot dosyamızın içerisindeki dosyalar ilgili işlemler yapabiliriz. Örneğin içindeki dosyaların oluşturma tarihi, Path’i veya dosya ismi gibi bilgileri bir CSHTML sayfasında listeyebiliyiriz, bir Action metot aracılığıyla. Veya bir create acitonu ile bir cshtml üzerinden wwwroot dosyamıza bir dosya ekleme işlemi yapabiliriz.
* DirectoryInfo classı ile bir klasör listesini listeyebilriz veya create delete metotlarını kullanabiliriz . Nesne benden bir Path bilgisi bekler. Wwwroot dosyası dışarıya açık olduğu için sadece buradaki dosyaları yönetebiliyorum.Dolyasıyla Path.Combine(Directory.GetCurrentDirectory()(uygulamanınçalıştığıyer), “wwwroot”)); bu ikisini combine edip birleştirdiğim zaman artık burayla ilgili directory bilgilerine ulaşabiliyorum. .GetDirectories() metotuyla ilgili klasörlere ulaşabilirim.
* Dosya işlemleri için Controller’a FileController ekleyebiliriz. Ben bir klasörün altındaki dosyaları listelemek istiyorsam FileInfo classından faydalanabilriz. Clasın creade delete gibi metotları bulunmakta.
* Eğer biz form aracılığıyla data taşıyorsak upload işlemi gibi form etiketine enctype=”multipart/form-data” eklememiz gerekiyor. IformFile interfacesi ile upload işlemini yapabiliyoruz. Bu interfacenin içerisinde .CopyTo() metotu var bununla upload gerçekleştiriyoruz. Bu metot bizden bir Stream nesnesi istiyor. Stream Abstract olduğun için örneklenemez. Dolayısıyla Biz FileStream kullanırız bu class abstract olan Streamdan kalıtılarak oluşturulmuştur.
* Guid.NewGuid() diyerek benzersiz isimler oluşturabilirim. Upload yaparken resimler farklı fakat isimleri aynı olabilir. Bu durumda benzersiz isimler oluşturmam gerekiyor. Path.GetExtension metotu ise dosyanın uzantısını geri dönüyor.
* Navbar veya footer veya bir ürün listesini getirmek gibi CSHTML ‘ yi ilgilendiren bazı kısımları parçalara bölebiliriz. Buna PartialWiev denir. Shared’ın içerisinde oluşturulmalı ki tüm CSHTML dosyaları istediği zaman bu parçalardan faydalanabilsin. CSHMTL sayfalarında partialWiew kllanmak için <partial name=”partialname”> şeklinde tanımlama yapıyoruz. Eğer Partial View bir model kullanıyorsa o modelide partial view’i çağırırken model=”model doya yolu ismi” olarak çağırmamız gerekiyor. Bu ModelView’in kötü bir yanıdır. Bu modele bağımlılık sürekli devam eder. İlgili PartialView’i kullandığımız her defasında ilgili model tanımlamasını da yapmamız gerekiyor. Bu noktada ViewComponent’ler bize yardımcı oluyor.
* View Component’in ViewPartial’dan farkı sanki bir controller gibi kendi kendine çalışır. Buna ek olarak asekronik bir şekilde çalışırlar. İlgili classı ViewComponent claasından kalıtırız. Bir ViewComponenti Invoke() metotu ile ayağa kaldırırız. Bir ViewComponent oluşturabilmek için IViewComponentResult diyoruz ve metotun ismini Invoke olarak belirtiyoruz.Daha sonra aynı bir action metot gibi geriye View dönüyoruz. CSHTML sayfalarında bir ViewComponenti kullanabilmek için @await Component.InvokeAsync(“componentismi”) şeklinde tanımalama yapıyoruz. Bu ViewComponenti oluşturacağımız dosya yolu ise Shared’ın altında Components klasörünün altında ilgili komponentin ismiyle oluşturulmuş bir klasör ve onun içerisinde Default.cshtml dosyası oluşturulur ki tüm Viewlar bundan faydalanabilsin. Ayrıca PartialView gibi illa model belirtme zorunluluğu yok sadece Componentin ismini cshtml dosyası içinde belirtmek yeterli performans açısından bu çok iyi bir şey. Kendi kendine ilgili datayı modeli çekiyor. VeiwComponentler parametreli olarakta çalışabilir bunun için Invoke metotu içine bir parametre geçememiz lazım. Daha sonra bu paramatreyi CSHTML de kullanabilmek için Component.InvokeAsync(“componentismi”,new{@parametreismi = “istediğniiz değer”}).
* Microsoftun bize sunmuş olduğu taghalper’lar gibi kendimiz de taghalper oluşturabiliyoruz Bunun için projenin ana dizinine TagHalpers isimli klasör oluştururz. Daha sonra o klasör için taghalperin isminde bir class ekleyip classı TagHalper classından kalıtmamız gerekiyor. Daha sonra Process metotunu override ederiz ve metotun içine output.Content.SetHtmlContent(“butaghalperneyapacak”) parametre içerisine yapacağı işi yazarız mesela <b>selam</b> yazsın gibi. Ve class üstüne [HtmlTargetElement(“taghelperhangielementleçalışacak(paragraph)”)]yani <paragraph> etiketiyle çalısşın demiş oluruz. Daha sonra viewimports’a @addTagHelper UdemyAspNetCore.TagHelpers.\*,UdemyAspNetCore ekleyip uygulamayı bundan haberdar ediyoruz. Daha sonra ilgili cshtml’de < paragraph></ paragraph> şeklinde bunu kullanabiliriz. Eğer işi parametreli hale getirmek istiyorsak. Taghelper classı içerisinde bir prop oluştururuz. Ardından metot içine bunu $ {} sayesinde veririz. Daha sonra cshtml de < paragraph parametreismi=”istenilendeğer”></ paragraph> şeklinde kullanım yapabiliriz.
* Area’ları kullanmazsak controller kısımları gibi kısımlar sürekli şişer karışık bir hal alır dosyalar eklendikce karmaşıklaşmaya başlar iş. Mvc kullanıyorsak Area kullanmakta fayda var. Uygulamanın ana dizinine Areas adında bir klasör oluşturuyorum. Daha sonra Area kalsörlerimi içine ekliyorum. Mesela benim Admin isminde veya Member isminde bir aream olabilir. Admin ve Members Klasörünün içinde de Controllers ve Views klasörü oluşturabilriim. Bu sayede adminlerin ve membersların controller ve viewleri birbirine karışmamış olur karmaşıklıktan kurtuluruz. Ancak yapmam gereken bir şey daha var. Admin controllerıne gidip class üzerine [Area("Admin")] ve members controller’ının class üstüne ise [Area("Member")] diye belirtmem gerekiyor. Daha sorna startup dosyasında bu areanın endpoints ayarlamasını yapmam gerekir.
* Validation yani Doğrulama ile örneğin bir müşteri ekleme formu sırasında belli kurallar getirebiliriz. ModelState.IsValid kavramı ile ilgili validation kurallarına uyulup uyulmadığını anlayabiliyoruz bunun testi ilgili controller’ın action metotu içeirsinde yapılır. Bu değer true ise validation kurallarımı başaryıla yerine getirmiş demektir. Validation kurallarını ilgili entity ve model entity ‘de tanımladığımız propların yani kolonların üzerinde [] açarak attirubutlar’la yazabiliriz. Buna DataAnnotation yöntemi denir.
* TagHalper’lar aracılığyla Validation kurallarına uyulmadığında hata mesajlarını sayfamızda gösterebiliyoruz.İlgili cshtml dosyasına gideriz. Asp-Validation-for helper’ını hatanın gözükmedisini istediğimiz yere ekleriz. Bu validation kuralı kimin için diye bize sorar. Bunun cevabını prop bazında vermeliyiz @Model.FirstName alanı için deriz. Controller içerisinde ModelState.AddModelError diyerek kendi hatalarımızı oluşturabiliyoruz. Bu hataların cshtml dosyasında gözükmesi için ise div ile bir etki alanı oluşturup dive asp-validation-summary=”” eğer buraya All dersek bütün oluşacak hatarlar burda gözükür ama eğer ModelOnly dersek herhangi bir propa setlenmemiş hatalar gözükecektir. Yani sadece özel olarak oluşturduğumuz hatalar gözükür. Yani sadece o modelin o propuna özel bir validation yazmış oluyorum.
* Aslında Validation kurallarını direkt datakaynağındaki modele yani model entity’sine yapmamak lazım. Mesela bir create işlemi yapıcaz bunun için farklı validationlar uygulayabilirirz ve aynı ürün bilgisiyle update için de ayrı validationlar uygulayabiliriz veya ileride validation kurallarımız değişiklik gösterebilir. Ya da diyelim 2 adet firmaya bir yazılım sattık ve bu 2 firmada aynı yazılımı kullanıyor. Fakat firmalardan birisi ürün ekleme işlemi yapıldığında 20 karakterden fazlasını kabul etmek istemeyebilir. Bu gibi durumlarda 0 dan bir controller oluşturmaktansa CustomerCreateModel gibi bir model oluşturarak validation kurallarını direkt orda düzenleyebiliriz. Yani o saf temiz olan data kaynağını bozmamış oluruz. Model klasörünü bu şekilde daha esnek kullanmalıyız işin doğrusu böyledir.
* 
* Bu zamana kadar yaptığımız validation işlemleri SERVER SİDE VALİDATİON olarak geçmektedir. Yani ilgili uygulamaya sunucu tarafına gelinir. Controller’daki ilgili action metotu çalıştırıp daha sonra bir validation kontrol işlemi gerçekleştirilir. Eğer kontrolden geçerse ilgili create işlemini yapıp indexview’a gönderiyordu. Bu validation işlemini CLİENT SİDE ‘ta JAVASCRİPT aracılığıyla kontrol edilebiliyor. Yani sunucu tarafına gitmeden javascript sayesinde kontrol yapaibliyoruz. Server tarafında olunca her create butonuna tıkaldığımızda sayfa yenileniyor yani sunucuya gitme işlemi oluyor. Java script ile sayfanın yenilenmesini sunucuya gidip kontrol yapılmasının önüne geçebiliriz bu sayede daha hızlı bir sonuç alır kullanıcı. Bunun için 2 adet kütüphane lazım. Birisi jquery-validation diğeri ise jquery-validation-unobtrusive’dir. Bunları kurmak için package.json dosyasında ilgili kütüphane isimlerinin eklenmesini yapıyoruz. Daha sonra npm klasörüne restore packages diyoruz. Daha sonra kullanmak için layout sayfasında kullanmak istediğimiz yere @RenderSection açarız. @RenderSection ile biz bir alan oluşturabiliyoruz. @RenderSection(“Scripts”,false) ismini verebiliriz mesela. Ve ikinci parametre olarak false verelim ki bu sectionu sadece istediğimiz belli cshtml’lerde kullanalım her cshtml sayfasında kullanma zorunluluğu olmasın. Sonra bu sectionu kullanmak istediğimiz cshtml dosayına giderek @seciton Scripts(section ismi){ jquery min scriptlerini script tagi ile ekliyorum
* }
* UYGULAMADA BİR DEĞİŞİKLİK OLUNCA SÜREKLİ TEKRAR BAŞTAN BAŞLAT ÇALIŞTIR YAPMAK YERİNE UYGULAMA DİZİNİNDE OPEN FOLDER İN EXPLORER DİYEREK CMD’YE dotnet watch run YAZARSAK YAPTIĞIMIZ DEĞİŞİKLİK SAYFAYI YENİLEYİNCE GÖZÜKÜR.
* Cookie yani çerezler kullanıcıların tarayıcılarıda veri tutmamızı sağlayan yapılardır. Bir cookie’yi setleyebilmek için HttpContext’in üstünde Response’larla hareket ederiz. Bir cookie’yi getirebilmek için ise Request kullanırız. Append() metotu ile bir Cookie’yi setleyebiliyoruz bir key bir de value değeri bekler üçüncü parametre olarak bir takım optionlar belirtebilirsiniz burda önemli 3 property var Expires propu bu cookienin ne kadar süre ilgili kullanıcının ilgili client’ın tarayıcısında ne kadar süre tutulacağını belirtir HttpOnly ise ilgili kişi document.cookie yazdığı zaman yani javascript ile bu cookieyi çekmeye çalıştığı zaman bu özelliği true yaparsak javascripte ilgiyi cookie’yi kapatırız ve bu kişi bizim cookie’mize ulaşamaz SameSite özelliği ise burda bir enum’dan faydalanıyoruz bu enum arcılığıyla SameSite durumunu setleyebiliyoruz eğer burada .Lax dersek bu cookie’Yi dışardıdaki sayfalarada açarız Strict dersek bu cookie’yi sadece sizin ilgili oluşturmuş olduğunuz web sayfası kullanabilir. Bu şekilde set metotu ile oluşturduğumuz cookie’ye ulaşmak için ise get metotunda TryGetValue() metotunu kullanabiliriz. Metota yukarıdaki set metotunda girmiş olduğumuz ilk parametre olan cookie bilgisini veriyoruz. İkinci parametre olarak ise cookie bilgisini string bir değerin içine atıcam diyor. O string değişkenini verip return olarak döndürüyoruz.
* Session kavramı Cookie’nin aksine sunucu tarafında verilerimizi taşıyabildiğmiiz yapılardır. Sunucu tarafında veri taşımamıza olanak sağlar. Controller’a bir SessionController ekleyebilriz. Bir SetSession metotu oluşturup HttpContext.Session.SetSring(“key”,”value”) şeklinde bir session oluşturabilir ve bir GetSession metotu oluşturup retrun HttpContext.Session.GetString(“session ismi”) diyerek bu sessiona ulaşabiliriz. Uygulama session kullanmak istiyorsak startup dosyasında ConfigureServices kısmına services.AddSession() ve daha sonra app.UseRouting() altına app.UseSession() dememiz gerekiyor.
* Bir action metotu çalışmadan önce vaya çalıştıktan sorna bir şeyler yapabiliyoruz. Bir şeyler yapmamızı sağlayan yapı ise Filter’lar dır. Bunun için projenin ana dizinine Filters isminde bir klasör açıyoruz. Classı ekliyoruz ve classı AcitonFilterAttribute ‘tan kalıtıyoruz. Bu abstract classın içinde bizi ilgilendiren 2 metot var OnActionExecuted bu metot action çalıştıktan sorna işe yarar ve OnActionExecuting bu metot action çalışmadan önce işe yarar. Bu metotları class içerisinde override ediyorum. ActionArguments ile action metotundaki parametrelere ulaşabiliyorum. Daha sonra iligli action metotunun ismine [filterismi] şeklinde tanımlama yaparak bu fitler’ı kullanabiliyorum.
* Aslında bizim burada Configure metotunda yazmış olduğumuz metotlarda birer MİDDLEWEARDIR.
* 
* Bu middlewearları kafamıza göre koyamayız Microsoft bunun sırasını belirtmiştir.
* 
* Eğer kendi hata sayfamı oluşturmak istiyorsam sturtup dosyasında en başa yani app.UseStaticFiles ın başına koyuyorum hemen üstüne app.UseStatusCodePagesWithReExecute metotunu kullanıyorum benden bir path format istiyor ve ?code ile ilgili değeri yaklamam gerekiyor. Daha sonra ilgili controller altında Status adında bir action oluşturuyorum. Parametre olarak nullable int code olarak yakalıyorum. Daha sonra hata sayfasını tasarlamak için View hazırlayıp oraya yönlendiriyorum.
* Biz uygulamanın içerisinde gerçekleşeen hatalarıda yakalayabilirz. Yine startup dosyasında en başa Configure metotunda en başa ekliyoruz. app.UseExceptionHandler("/Home/Error"); Bir hata oluştuğu zaman Home Controller altında Error Action metotuna git diyoruz. Action metotunda HttpContext.Features.Get<IExceptionHandlerPathFeature >(); ile ilgili hataya ulaşabiliyoruz.
* Eğer kullanıcı bir hatayla karşı karşıaya kaldıysa ve bu hata sistemsel ise bizim bu hatayı mutlaka loglamamız gerekiyor. Uygulamada oluşan hataları takip edebilmemiz için loglama yapmamız şart. Bu işlemde serilog nlog gibi kütüphaneleri kullanabilriz veya hatayı hazır txt dosyasına yazdırabiliriz klasör ve dosya oluşturma işlemleri ile.
* Appsettings.json dosyası içerisinde bir veri tutup tutmuş olduğumuz bu verileri çekebiliyoruz. Bunun için AllowedHosts un üstüne Person adında bir dictionary tanımlasak içine firstname ve lastname veriyoruz ve daha sonra bu dataya bir interface aracılığı ile ulaşabiliyoruz. Startup dosyası içeirsinde Configure metotunda parametre içine IConfiguration configuration diyerek bir tanımlama yapsak configuration nesnesi aracılığıyla bu datayı çekebilirim. Hataların üstünde bu işlemi yaparım. Aşşağıdaki şekilde bu bilgilere tek tek ulaşabilirim VarfirstName=configuration.GetSection("Person:FirstName").Value;
* var lastName = configuration.GetSection("Person:Lastname").Value;

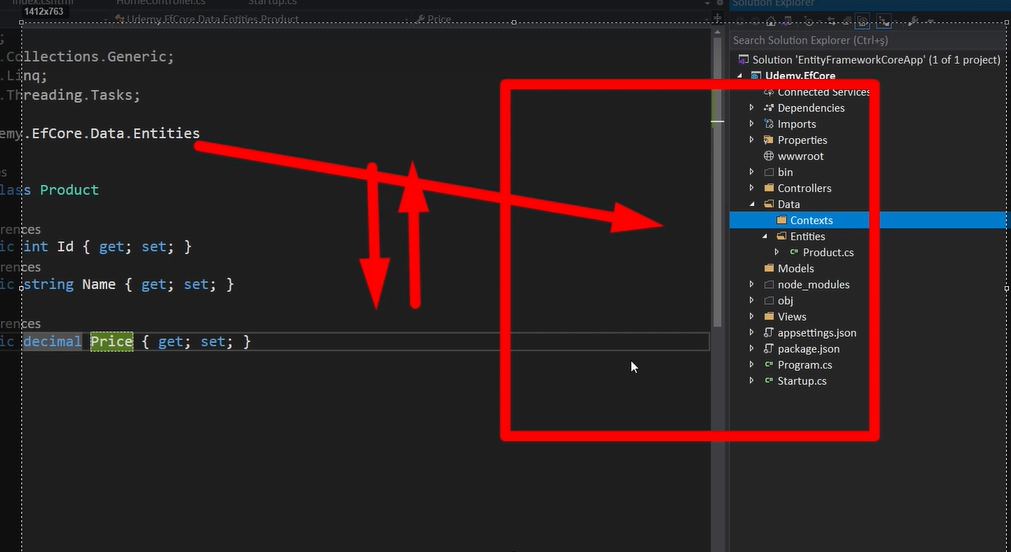
**SOLID PRENSIPLERI**

**AMACI:** Kolay değiştirilebilir, sürdürebilir sistemler inşa edebilmektir.

* **S- Single Responsiblity:** Türkçe karşılığı tek sorumluluktur. Yani bir classımızın birden fazla iş kümesini yapmaması gerekiyor. Yaptığı işlere göre classlarımızı veya metotlarımızı konumlandırmamız lazım.Her bir classımın veya metotumun tek bir sorumluluğu olması lazım.
* **O-Open Closed:** Nesneler değişime kapalı fakat genişlemeye açık olmalıdır.
* **L-Liskov Substitution:** S eğer T nin bir alt türü ise S nin T nin yerine geçebilmesi gerekir.
* **I-Interface Segration:** Alakalı interfaceleri parçalamalısın. Yani bir interface’de alakasız bir şey olmamalı. Zaten Segration ayırmak demek.
* **D-Dependency Inversion:** Bağımlılıkları tersine çevir.
* **Readonly** dersek eğer bir şey sadece kanstraktır içinde setlenebilir. Bunu başka bir yerde setleyemeyiz.
* **Dependency Injection** yapısı gelir container’da belirtmiş olduğnuz şeye göre ilgili örneği alır fırlatır. Siz bu örneği toplayıcı bir interface’nin üzerinde taşıyabilirisinz. Mimari böyle çalışıyor. Arkasındaki mantık Depndency Inversion’da belirtmiş olduğumuz container mantığı aslında.
* ****

**ENTİTY FRAMEWORK CORE**

* Açık kaynak kodlu ve platformdan bağımsızdır cross platform’dur yani. Bir ORM aracıdır. Veritabanındaki verilere erişmek veya bunları değiştirmek eklemek güncellemek silmek gibi işleri kolaylaştıran bir mekanizma sağlar. Entity Framework Core bir ADO.NET geliştirmesidir.
* Entity Framework Core ‘ da veriye 2 çeşit şekilde yaklaşabiliyoruz. Bunlardan birisi DBFirst bir diğeri ise CodeFirst. EntityFrameWork’u uygulamaya kurmak için web.ui dosyasına sağ tıklanıp Manage Nuget Packages kısmı açılır. Browse kısmından Microsoft.EntityFrameworkCore paketi kurulur. Bunun yanında hangi database aracı ile çalışıyorsam örneğin sqlserver ile çalışacağım diyelim Microsoft.EntityFrameworkCore.SqlServer kütüphanesinide projeme kurmam gerekiyor. Daha sonra ise projeme Microsoft.EntityFrameworkCore.Tools paketini kurmam gerekiyor. Bu paket sayesinde package manager console ile migration kodlarımı vs yazabiliyorum. Bunun dışında bir de Microsoft.EntityFrameworkCore.Design paketine ihtiyacımız var. Uygulama içinde Dependencies’te Packages kısmıdna kurduğumuz paketleri kontrol edebilririz.
* Veritabanını oluşturmak için web.uı ‘ a sağ tıklayıp bir klasör açıp ismini Data yapıyorum. Bu Data katmanında ben veritabanındaki tablolara karşılık gelen Entity’lerimi ve Context’imi tutacağım. Data klasörünün içine Entity Classlarımın içinde olacağı bir Entities Klasörü oluşturuyorum. Context’lerimi içinde tutacak bir Contexts isminda klasör oluşturuyorum data katmanına aynı zamanda.
* Veritabanındaki tabloları karşılayan programatik classlara entity yani varlık denmektedir. Tablonun içeriisndeki bulunan kolon isimlerini ise PROP olarak belirtebiliyoruz.
* Context ise o veri tabanındaki session’dır diyebiliriz. Aynı zamanda o veritabanındaki varlıkların entity’lerin bilgisidir diyebiliriz. Contexte aslında Uygulamam ve veritabanım arasında bir trafik polisidir diyebilmek mümkündür.

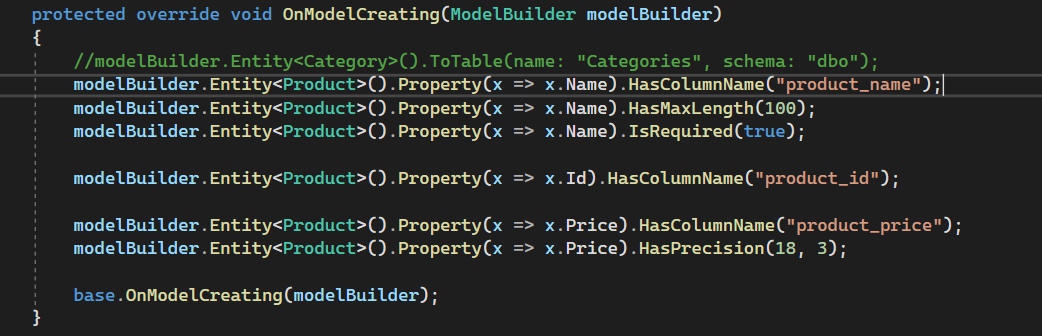


**Context** classımızı oluşturduktan sonra o classı DBContext classından kalıtmamız gerekiyor bu classın bir context olabilmesi adına. Ve claasın içinde benim bu entity’lerimin varlığından contexti’min haberdar olabilmesi için proplar oluşturup propların türlerini DBSet<entityismi> entityçoğulismi şeklinde tanımlayarak contextimi entitylerimin varlığından haberdar ediyorum. Burada çoğul olarak verdiğimiz entity ismi default olarak veritabanımızda bir tablo ismidir aynı zamanda. Daha sonra benim bu context classı içerisinde bir metotu override etmem gerekiyor. Bu metot OnConfiguring metotudur bu metot ilgili contextimi configure ayarlamalar yapıp gönderebiliyorum. Bu metot içerisinde optionsBuilder.UseSqlServer(“”) ilk parametre olarak benden bir conneciton string istiyor. Burada bunun sayesinde database aracı ile yani sqlserver ile bir bağlantı kuruyorum. Bu noktadan sonra geriye database’i oluşturmak kalıyor. Database oluşturabilmek için öncelikle bir MİGRATİON oluşturamamız gerekiyor. Daha sonra o MİGRATİONU UPDATE DATABASE komutuyla database’e göndermemiz gerekyor. Bunun için önce Package Manager Console’u açıyoruz. add-migration InitialCreate diyerek ilk defa bir database oluşturuyorsak InitialCreate ismi default bir isimdir bu şekilde ilk migrationumuzu oluştururuz. Daha sonra proje içerisine Migrations adında bir klasör oluşur içinde verdiğimiz isimde bir migration dosyası vardır bu dosyanın içinde ilgili migrationun neler yapacağı hakkında bilgiler var ve bir de Snapshot dosyası oluşur. Snapshot dosyasının içinde son oluşturduğumuz migrationun bilgisi burda tutuluyor.Daha sonra migration oluştuktan sonra update-database diyerek ilgil migration database tarafına gider. Projede yaplıan değişikler database tarafına aktarılmış olur. SQLSERVER BAĞLANTI KODU : protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

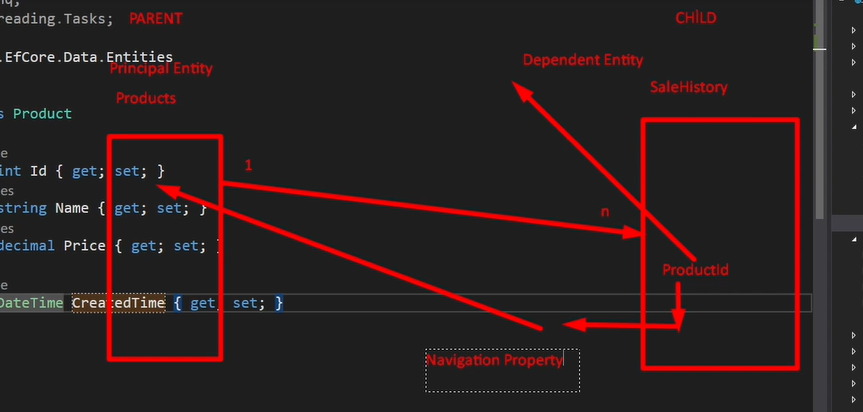
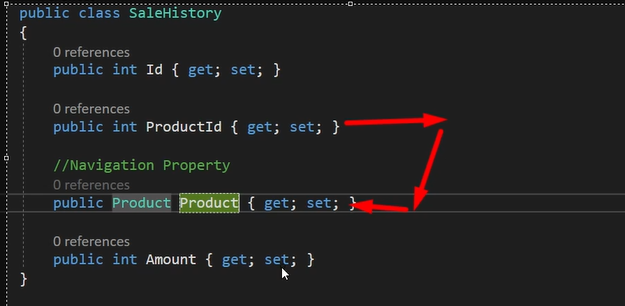
optionsBuilder.UseSqlServer("server=DESKTOP-4UF23CE; database=UdemyEfCore; integrated security=true;");

}

* **DBContext** nesnesi üzerinden database ile ilgili kurucu işlemlerimizi gerçekleştirebiliyoruz. Yani ekleme silme güncelleme ya da database’e bir query atma bir veri getirme. Bunu kullanabilme için ilgili controller’ın action metotu içinde DbContext örneği almamız gerekiyor. Oluşturduğumuz Context isminden bir örnek oluştururuz. Daha sonra ben context.Products.Add() metotuyla Products tablosuna bir veriyi ekleyebilirim. Ekleme işlemi yapılırken Id bilgisine ekleme yapılmaz çünkü ID bilgisi Identity’dir yani boş bile bıraksam o database tarafında kendi kendine kayıt geldikçe artmaya devam eder. Bu durumda ben add update gibi metotlarla entityframework üstünden bir şey belirtmiş oluyorum ben bu datayı ekleyeceğim ya da güncelleyeceğim gibi ama işlemi gerçekleştrimiyorum bu durumda add metotu state durumuna added düşürüyor yani ekleme bekleniyor diyor update metotunda ise statement kısmında modied yazar yani güncelleme bekleniyor işaretleme yapıldı diyor. İşlemi context.SaveChanges() dediğim zaman gerçekleştirmiş oluyorum. Yani ilgili değişiklikleri database’ye gönder demiş oluyoruz bu sayede işlem gerçekleşmiş oluyor.
* Veritabanında bir kayıt güncellerken ise Update() metotunu kullanacağız. Find metotu bizden bir primary key istiyor id istiyor aslında ilgili ıd değerini buraya vermiş olmamız gerekiyor. Ayrıca find metotunu kullanmadan update metotu içeirsinde id tanımalamsı yaparsak sadece bu sefer diğer girmediğimiz güncellemediğimiz kolonların bilgileri default bilgilerine döner meseela string bilgiler null olur int bilgiler 0 olur find metotunu kullanarak diğer verileride kaybetmemiş oluyoruz. Güncelleme yapılırken Primary Key olan Id değerine ihtiyaç duyulur ekleme işleminin aksine. Güncellemek istediğim kayıdın ıd değerini alıyor olmam lazım. Dolayısıyla güncelleme işlemi yapmadan önce güncelleyeceğim kaydı Find metotu ile database’den almam çekmem gerekiyor. Güncelleme işlemi bittikten sonra yine SaveChanges() metotu ile işlemi gerçekleştirmeyi unutmuyorum.
* Bir kaydı silmek için Remove () metotunu kullanırız. Silme işlemi yaparken aynı update işleminde olduğu gibi önce find metotu sayesinde silmek istediğim veriyi veritabanından çekeceğim sonra silme işlemini gerçekleştireceğim. Bu işi find metotuyla yapabildiğimiz gibi FirstOrDefault metotu ile de yapabiliriz bu metot bizden func bir delegate ister. (x=> x.Id == 1) gibi bir lambda expression aracılığıyla x derken aslında ilgili tablo ismini vermiş oluyoruz x in içinde yani tüm tablonun içinde ıd alanı 1 ‘ eşit olan ilk kayıdı seç demiş oluyoruz. Metot kayıtı bulursa seçme işlemini yapacak ve geriye o ilgili kayıdı döndürecek ama bulamazsa geriye null bir değer döndürecek yani defaultunu getirecek ilgili değişken türünün.
* Biz sonradan bir entity eklemesi daha yaparsak bunu önce context içerisinde dbset olarak tanımlamamız gerekiyor sonra database’ye aktarmak için yeni bir migration oluşturup update-database yapmamız gerekiyor. Migration ismini yaptığımız işin ismini yazmak mantıklı olur AddingCategoryEntity gibi.
* Tablolarımızda koyulan özellikleri yani mesela name bilgisi nvarcharmax olarak işaretlenir ben böyle istemiyorumda nvarchar100 girmek istiyorum veya name özelliği null olarak işaretlenir ama ben istiyorum ki notnull olsun bu gibi olayları yapabilmenin 2 yolu mevcut bunlardan birisi DATA ANNOTATİONS bir diğeri ise FLUENT APİ’dir.
* Bu işi DATA ANNOTATİONS aracılığıyla gerçekleştirmek için ilgili entity classımıza gideriz ve kolonlarımızın yani proplarımızın üzerine [] açarak ilgili özellikleleri orada tanımlarız. Normalda database tarafında tablo isimleri context içinde entityleri contexte tanımlarken verdiğimiz isim orada neyse o isimle databaseye kaydolur tablolar. Biz bunu değiştirebiliriz. Classın üzerine gelip [Table(name:”yenisim”, Schema=”yenischemadı”)] yazarak bunu yapabiliriz. Ayrıca farkettiysek , koyup schema yazarak schema bilgisine değiştirebiliyoruz default schema ismi dbo.tablo ismidir schemaya c verirsek c.yenitablo ismi şeklinde bir değişiklilk olmuş olur. Yada böyle yapmayızda [Column(“yenikolon\_ismi”)] şeklinde de yapabiliriz bu daha sağlıklı olur. İlgili kolon propunun üstüne [Required] dersek artık o alan doldurulması zorunlu yani notnull bir alan olumş olur. Veya iligli kolon propunun nvarcaharmax değilde [MaxLenght(50)]şeklinde yaparsak nvarchar(50) olarak değiştirmiş oluruz bu özelliği. Yada böyle yapmayızda [Column(TypeName=”nvarchar(100)”)] olarak yapmak daah sağlıklı olur.Bu değişikliğin database tarafına yansıması için mutlaka bir migration oluşturmalıyız.
* Bu işi FLUENT APİ ile yapmak istersek eğer, bu noktoda ilgili context classımıza gideriz. Ve burada OnModelCreating() metotunu context içerisine override etmemiz gerekiyor. Daha sonra modelBuilder.Entity<ilgilientityismi>().ToTable(name:”değişecek tablo ismi”,schema:”değişecek şema ismi”); şeklinde bir tanımalama yaparız. FLUENT APİ HER ZAMAN DATA ANNOTATİONS ‘I EZER YANİ PROJE İÇİNDE AYNI TABLODA BU İKİSİNİ AYNI ANDA KULLANIRSAK FLUENT APİNİN KODLARI ÇALIŞIR. Bu işlemi yaptıktan sonra değişiklerin database’ye aktarılması için migration oluşturup update-database demeyi untumuyorum.
* FLUENT APİ İLE YAPABİLECEĞİMİZ BAZI BAŞKA ÖZELLİKLER: kolon ismi değiştirme, özellik değiştirme vs. vs. BU TÜR KOLON İSİMLERİNE SNEAK CASE DENİYOR. Kolon\_ismi şeklinde olan kolon isimlerine. DECİMAL default olarak notnull bir alandır başına ? koyarak nullable yapabiliriz. Decimal alana HasPrecision ile bir hassasiyet verebiliyoruz.
* 
* Projede DATA ANNOTATİONS kullandığımız zaman Solid prensiplerinden olan SİNGLE RESPONSİBİLİTY’i çiğnemiş oluyoruz bu yüzden FLUENTAPİ kullanmak çok daha sağlıklıdır.
* EntityFrameWorkCore eğer kolonumun adı Id ise bunu gidiyor Primary Key olarak setliyor. CustomerId gibi bir kullanım olursada içinde Id olduğu için Primary Key olarak setlenir. Eğer tablomun içinde herhangi bir primary key alan olsun istemiyorsam context içerisinde OnModelCreating metotunun içerisine modelBuilder.Entity<entityismi>.HasNoKey() dememiz gerekiyor. Veya ben primary key olarak işaretleyeceğim alanın içeirinde Id geçmesini istemiyorsam ilgili entity ve ilgili kolon propun üsünte [Key] attirubetu’nu kullanabiliriz. O prop kolonu primary key olarak işaretlemiş oluruz. Bunun kötü yanı sadece tek bir kolonu key olarak setler bizim bazı durumlarda 2 kolonu birden primary key olarak setlememiz gerekebiliyor. Çoka çok ilişki kurarsak bunu yapmamız gerekir mesela. Bunu bu şekilde DATA ANNOTATİONS olarak yapabildiğimiz gibi FLUENTAPİ olarak yapmak için context classımızın içerisinde OnModelCreating metotunun içeirisinde modelBuilder.Entity<Customer>().HasKey(x => x.Number); yazarak istediğimiz seçtiğimiz kolonu primary key olarak işaretleyebiliriz. Eğer birden fazla kolonu primary key olarak belirtmek istiyorsak ki burda şu önemli bir tabloda bir tane primary key olabilir sadece 2 kolon birden oluyor bir primary key olmuş oluyor birden fazla primary key belirtmek için isimsiz bir obje rotasyonu geçmemiz lazım modelBuilder.Entity<Customer>().HasKey(x => new {x.Number,x.Name}); şeklinde bu işi gerçekleştirebiliriz. Daha sonra bu işlemlerden sonra migration oluşturup update-database demeyi unutmuyorum.
* Ben database’ye kayıt yaparken sadece bir kolonun bilgisini gireyim diğer kalan kolonlar default olarak benim verdiğim değerleri alsın istiyorsam mesela kayıt oluşturulurken otomatik olarak o oluşturulma tarihini ilgili tablonun kolonuna atabilirim böyle bir şey için default değerler atamalıyım. Bunun için FLUENT APİ kullanabiliriz. İlgili context’in OnModelCreating metotunun içine gidelim. Aşağıdaki gibi kodlar ile bu işi yapabiliyoruz.

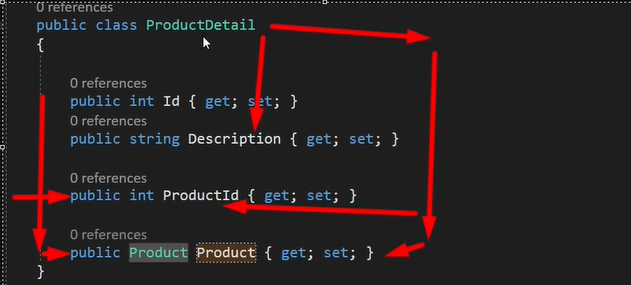
modelBuilder.Entity<Product>().Property(x => x.Name).HasDefaultValueSql("'Urun bilgisi girilmemis'");

modelBuilder.Entity<Product>().Property(x=>x.CreatedTime).HasDefaultValueSql("getdate()");

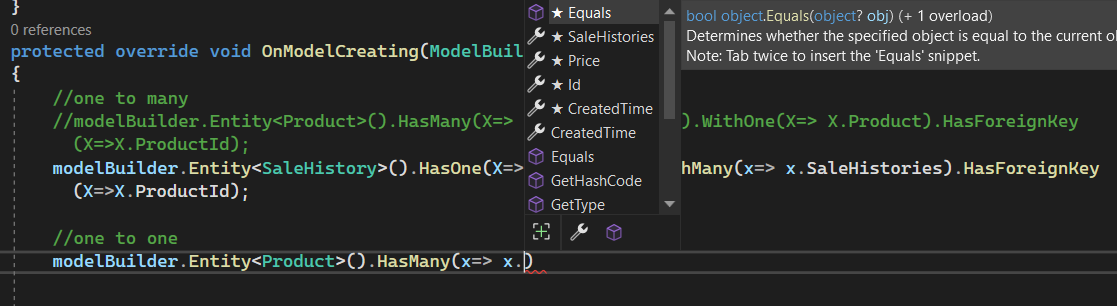
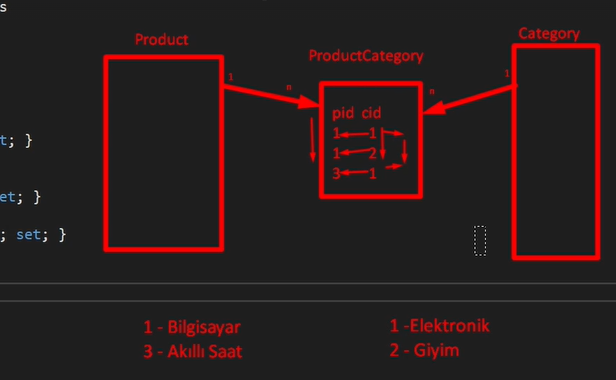
* Ben kolonlarımdaki alanların tekrarlanamaz olmasını isteyebilirim. Bune Unique denir. Aynı isimli bir kayıdın eklenmemesini isteyebilirim mesela. Bunun için FLUENT APİ kullanabiliriz. İlgili context’in OnModelCreating metotunun içine gidelim. Aşağıdaki kodu oraya yazarsak product name alanına telefon dediysem bir kayıdın bir başka kayıdın product name alanına telefon diyemem. Tabi bunu yaptıktan sonra migraiton oluşturmayı ve update-database demeyi unutmuyorum.
* modelBuilder.Entity<Product>().HasIndex(x => x.Name).IsUnique(true);
* Bir product tablom birde satış kayıtlarını ttuttuğum tablom olsun. Bir product birden fazla kez satılmış olabilir. Bu ikisi arasında bire-çok bir ilişki olur. Dolayısıyla Product tablosu Parent satış tablosu ise Child olmuş olur. İçerisinde herhangi bir Foreign Key barındıran(ProductId) Satış tablosunda hangi product’ı sattığımı bilebilmek için bir Foreign Key yani Produc’ın Id’sini gösteren bir kolon tutmam lazım. İçerinde Foregin Key bulunduran alttaki CHILD tabloya DEPENDENT ENTİTY deniyor. Üstteki PARENT tabloya ise PRİNCİPAL ENTİTY diyoruz. Bunların haricinde NAVİGATİON PROPERTY denilen bir kavram vardır.
* 
* **ONE TO MANY İLİŞKİ :**  SaleHistory tablosuna ekleyeceğimiz ProductId kolonu aslında bir Product’ı göstermesi gerekiyor. İşte tam olarak burda NAVİGATİON PROPERTY denilen bir kavramdan faydalanıyoruz. Bir SaleHistory tablo kaydı, bilgisi aslında product bilgisine ihtiyaç duymak zorunda. Bu noktada NAVİGATİON PROPERTY belirtiyoruz. Bunu belirttiğimiz zaman bu ProductId kolonuna karşılık gelen bir product var demiş oluyoruz. Bir salehistory tablosunda bir product olmak zorunadır demiş oluyoruz bunu dediğimiz kksım ile ProductId kolonudur eğer intin sağına bir soru işareti koyarsak bu zorunluluğu kaldırmış oluruz.Ve aynı şekilde bu yapmış olduğumuz işlemi gidip Product tablosundada belirtmemiz gerekiyor yani product tablosunda da bir NAVİGATİON PROPERTY belirtiyoruz fakat bunu bir list olarak tanımlamalıyız çünkü bir product’ın birden fazla satış bilgisi olabilir. Daha sonra bir migration oluşturup update-database dediğimiz zaman entityframeworkcore ilgili tablo ilişkisini database tarafında otomatik olarak oluşturur. SqlManagmentStudio diyagramında anahtar olan kısım 1 olan kısım çok olan kısım ise yuvarlak oalrak belirtilir tablolar arasında.
* 
* Bu bire-çok ilişkisini FLUENTAPİ ile gerçekleştirmek daha sağlıklıdır. Bunu yapmak için ilgili context classının içerisindeki OnMoelCreating metotunun içine gitmemiz lazım. Bu işlemi aşağıdaki kodlar ile 2 ayrı şekilde de yapabiliriz. Tek olan ve çok olan tablomuzu belirtiyoruz ve Foreign Keyimizi belirtiyoruz. Bu kodlardan 2 sinden birini yazmamız yeterli bizim tercihimize kalmış.

modelBuilder.Entity<Product>().HasMany(X=> X.SaleHistories).WithOne(X=> X.Product).HasForeignKey(X=>X.ProductId);

modelBuilder.Entity<SaleHistory>().HasOne(X=>X.Product).WithMany(x=> x.SaleHistories).HasForeignKey(X=>X.ProductId);

* **ONE TO ONE İLİŞKİ:** Bir Product tablomuz ve bu Product’ın Detay bilgilerini tutan bir ProductDetails tablomuz olsun. Bir Product’ın ancak Bir Detail’i olabilir. Bir Detail’in ancak bir Product’ı olabilir. Buna birebir ilişki denir.
* 
* Ben burda şunu söylemiş oldum. ProductDetail’ı bir Product olmalı ve bu bu olmak zorunda. Yani ben bir productdetail kaydı gireceksem bu productdetail kaydında bir product bilgisi girmek zorundayım. Bunu yaptıktan sonra Context’in içindeki OnModelCreating metotunun içine gidelim. Product tablosunun içinde de navigation property oluşturmayı unutmayalım bundan önce yoksa fluent api kısmında products tablosu üzerinden productdetail tablosuna ulaşamayız ve fluent api kısmında ilişkiyi kuramayız. Daha sonra fluent apide aşağıdaki kodu yazıyoruz. Dikkat edilmesi gereken burda onetoone ilişki olduğu için foregin key belirtiyken <> arasına foregin keyimizin hangi tabloda olacağını belirtmemiz gerekiyor. Kodu yazdıktan sonra migration oluşturup updata-database diyerek ilişkinin database tarafında kurulmasını sağlarım. Burada dikkat edilmesi gereken bir product’ı karşılık bir productdetail gelir yani aynı productıd ile bir başka kayıt ekleyemeyiz veya aynı description alanı ile bir başka description alanı ekleyemeyiz tablolardaki kayıtlar ve productıd kolonu farklı kayıtlar içermelidir birebir ilişki olduğu için. Her product’ın benzersiz olarak sadece bir tane product detail’i olabilir.

modelBuilder.Entity<SaleHistory>().HasOne(X=>X.Product).WithMany(x=> x.SaleHistories).HasForeignKey(X=>X.ProductId);

* **ÖNEMLİ NOT: EĞER NAVİGATİON PROPERY BELİRTMEZSEK BİR TABLO ÜZERİNDEN DİĞER TABLOYA UYGULAMA TARAFINDA GEÇİŞ YAPAMAYIZ MESELA BURDA GÖRDÜLDÜĞÜ ÜZERE PRODUCTDETAİLS TABLOSUNA PRODUCT TABLOSU ÜZERİNDEN GEÇİŞ YAPAMADIM. BU SEBEPLE ASLINDA NAVİGATİON PROPERTY UYGULAMA TARAFINDA BİR TABLONUN ÜZERİNDEN DİĞER TABLOYA GEÇİŞ YAPMAMIZI SAĞLAR.**
* ****
* **MANY TO MANY İLİŞKİ:** Elimizde bir Product birde Category tablosu var diyelim. Bir kategoride birden fazla ürün olabilir ve bir ürünün de birden fazla kategorisi olabilir. Bu duruma bu ilişkiyi gerçekleştirmek için araya başka bir tablo koyuyoruz. ProductCategory gibi isimlerle tablosu denir buna genelde. Bu ProductCategory tablonun içinde bir ProductId ve birde CategoryId adında 2 tane kolonu olur.
* 
* Bu işlemi gerçekleştrimek için Entities klasörüme ProductCategory Entitiy’ini ekliyorum. Daha sonra Product entity’ine yeni bir nav prop oluştururm.
* public List<ProductCategory> ProductCategories { get; set; }
* Daha sonra aynı şekilde Category entity’ine gidip bir nav prop ekliyorum.
* public List<ProductCategory> ProductCategories { get; set; }
* Bu sayede her iki entitiy’minde navigation property’si ProductCategories olmuş oldu. Daha sonra oluşturduğumuz ProductCategory entity’sine de bu propları ekliyorum:

public int ProductId { get; set; }

public Product Product { get; set; }

public int CategoryId { get; set; }

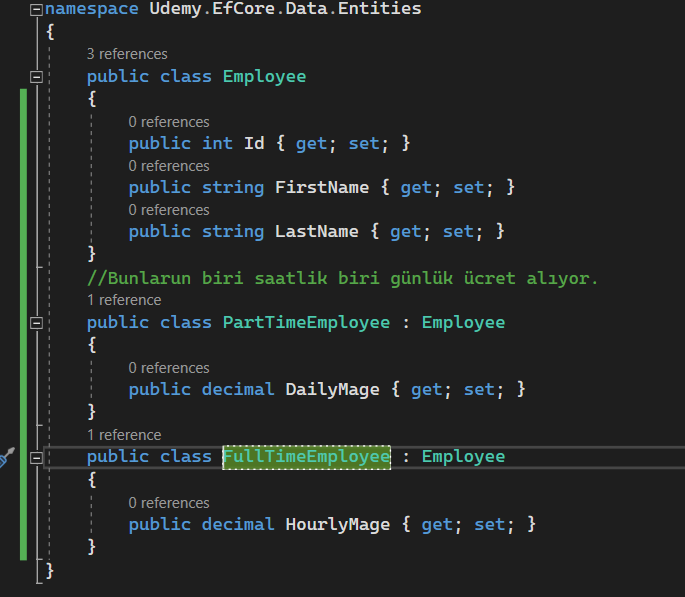
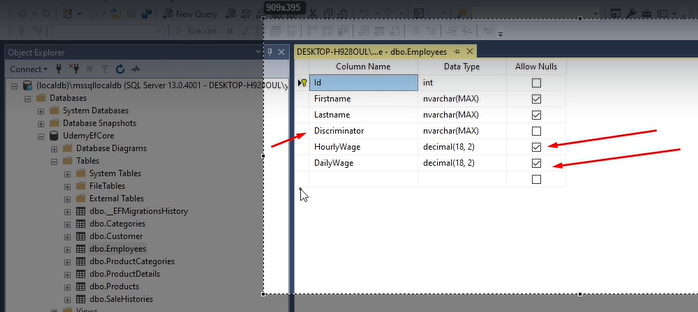
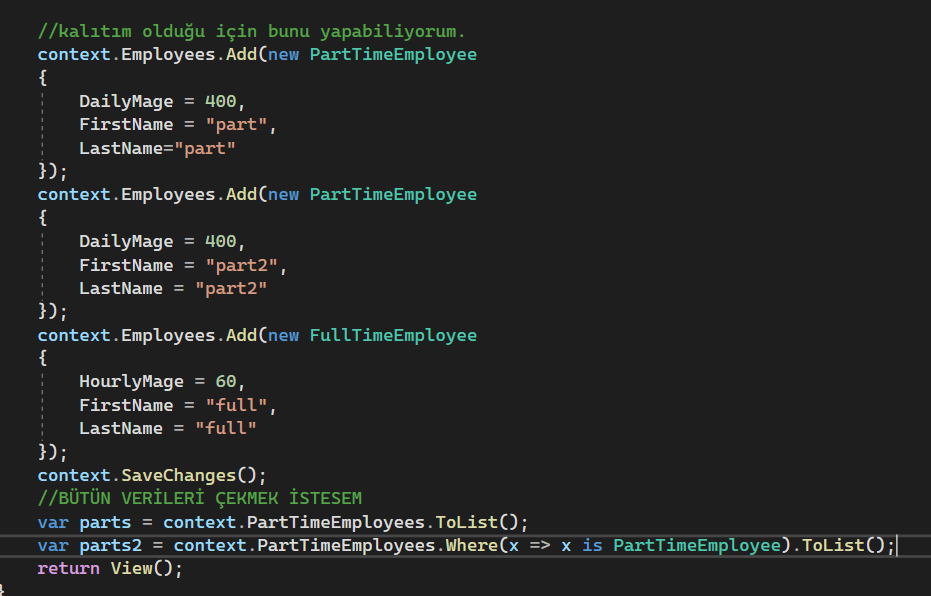
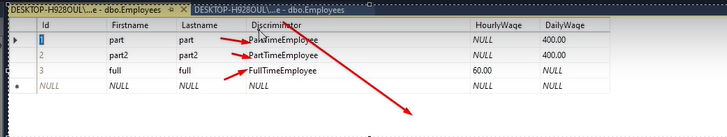
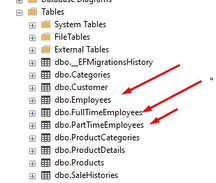
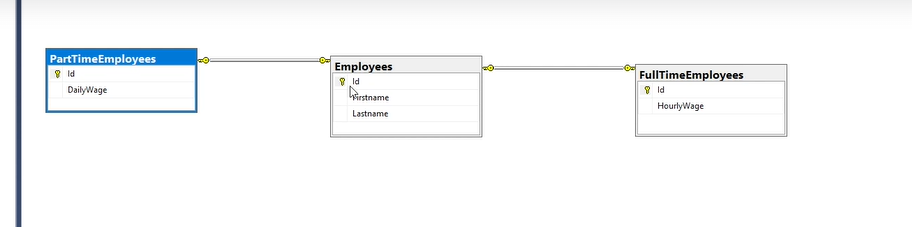
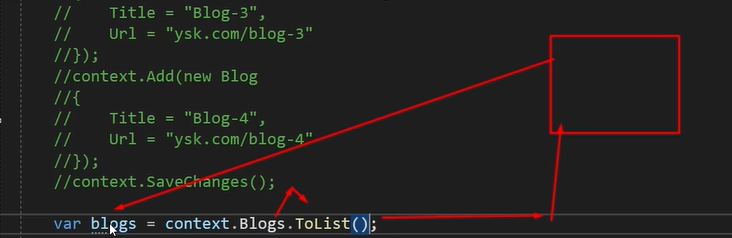
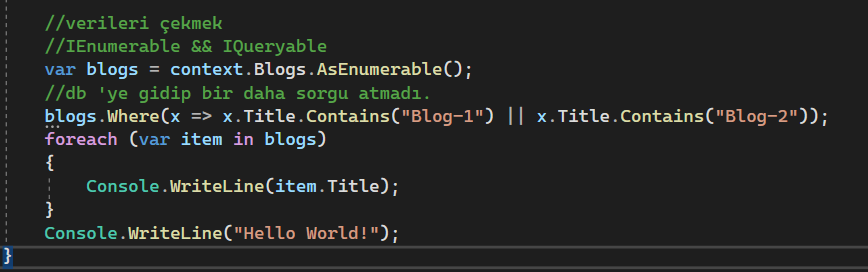
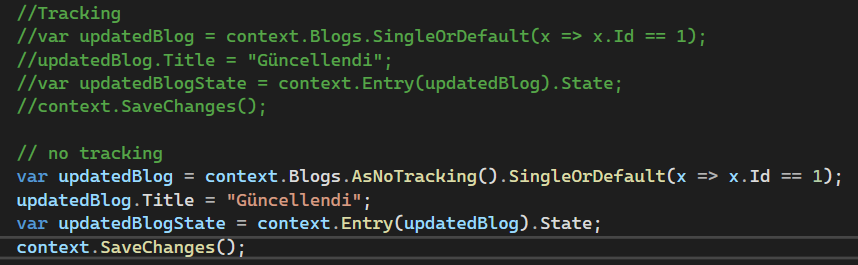
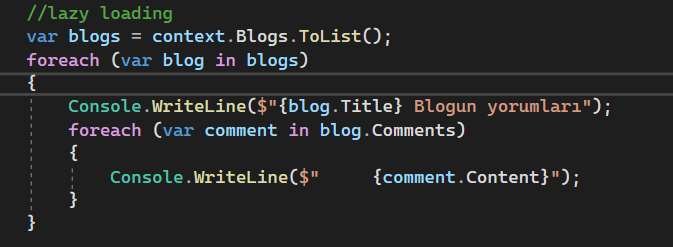
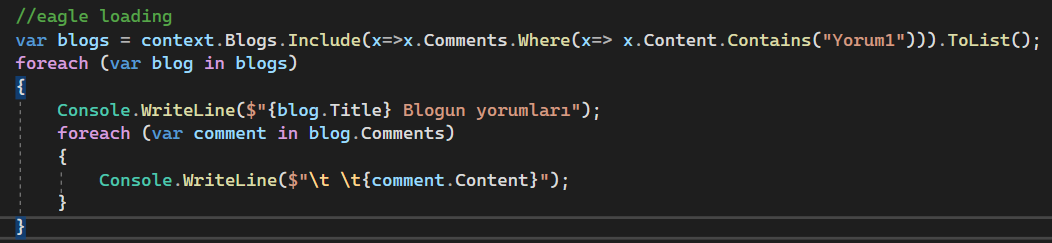
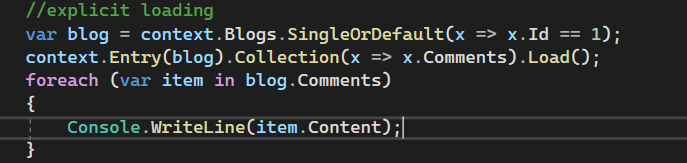
public Category Category { get; set; }

* Daha sonra fluent api kısmında bunu yazmak için OnModelCreating metotu içinde aşağıdaki kodlar yazılır bu ilk kodu yazma sebebimiz ise sürekli 1,1 1,1 şeklinde aynı kayıtlar girlmesin tekrarlanamsın diye PrdouctCategort tablosundaki 2 foreign key olan Id kolonlarını primary key olarak setliyoruz ki sürekli birbirini tekrar eden kayıtlar tablo içerisine girilmesin. Daha sonra migration oluşturup update-database diyerek bu özelliği database tarafına geçirmeyi unutmuyorum.

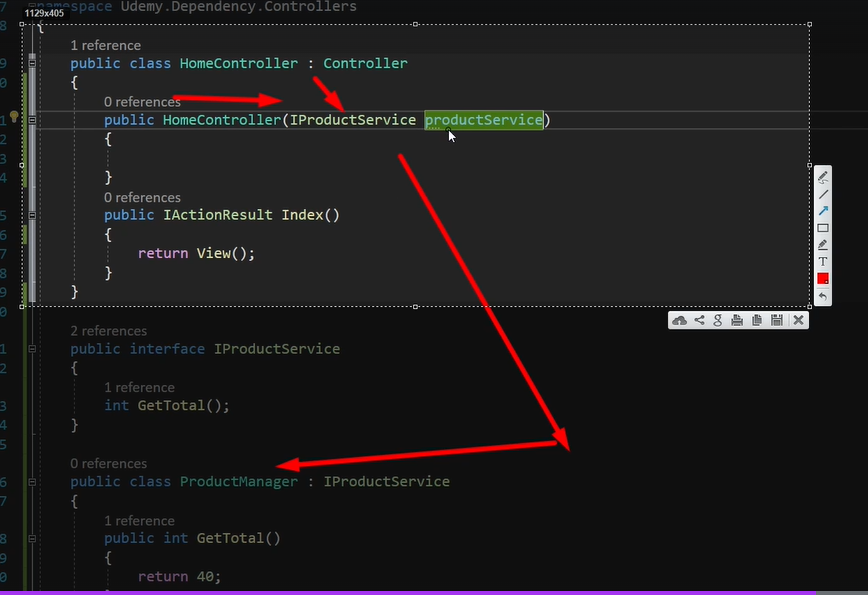
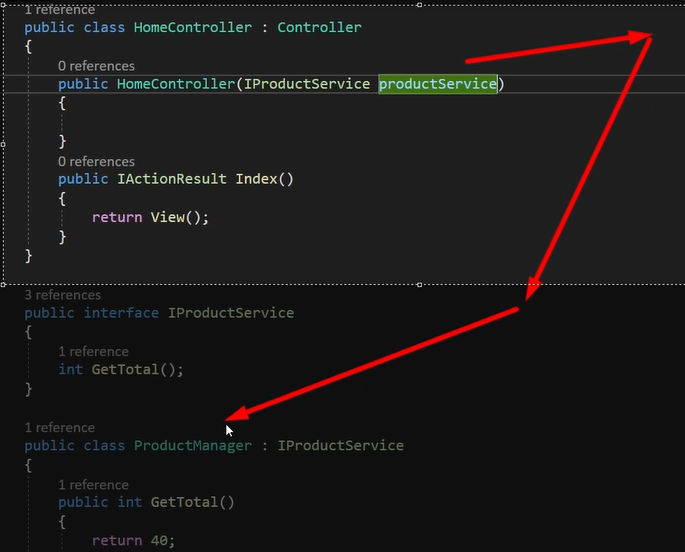
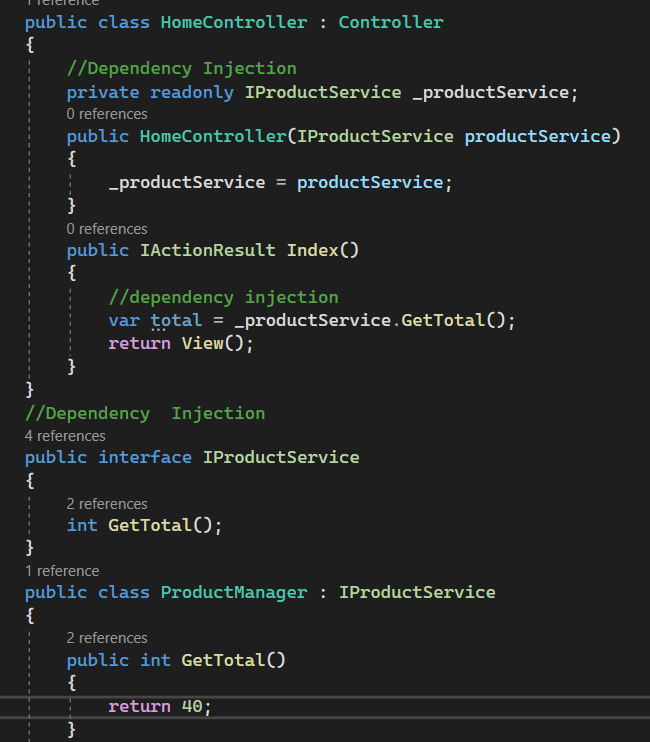
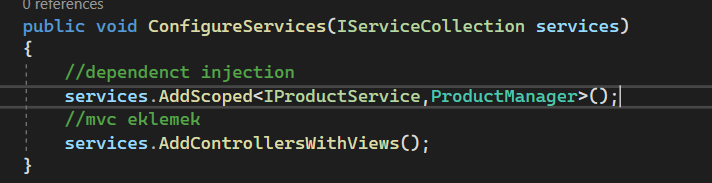
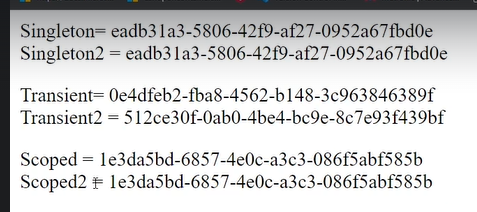
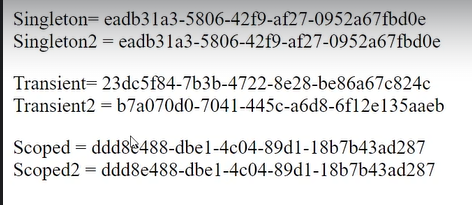
- modelBuilder.Entity<ProductCategory>().HasKey(x => new { x.CategoryId,x.ProductId });

- modelBuilder.Entity<Product>().HasMany(x => x.ProductCategories).WithOne(x => x.Product).HasForeignKey(x => x.ProductId);

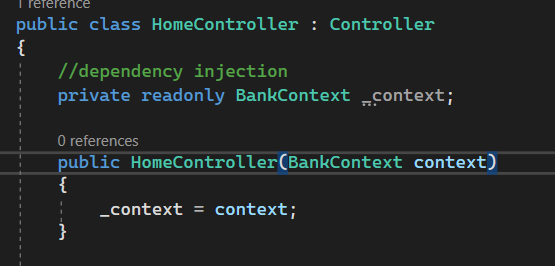
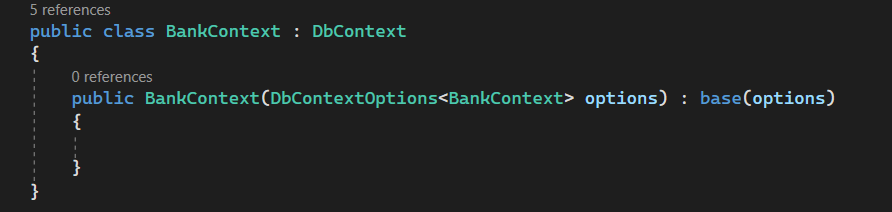
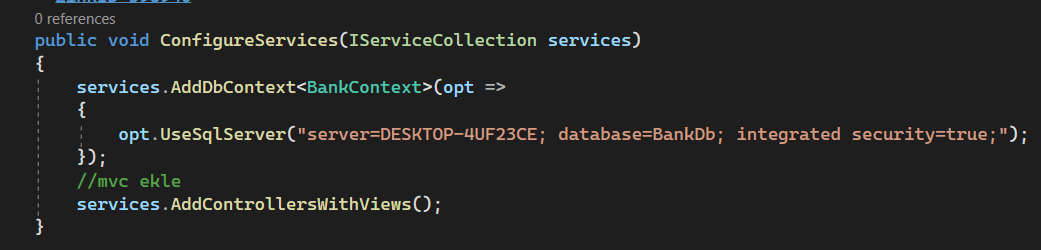
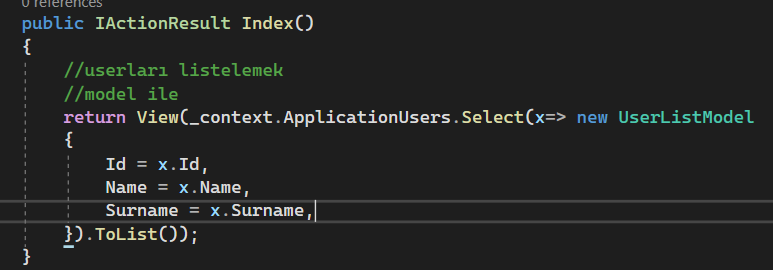
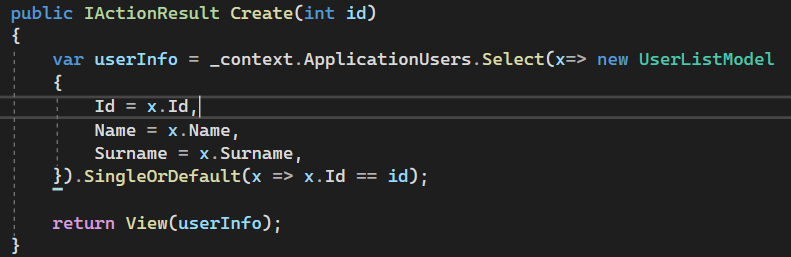
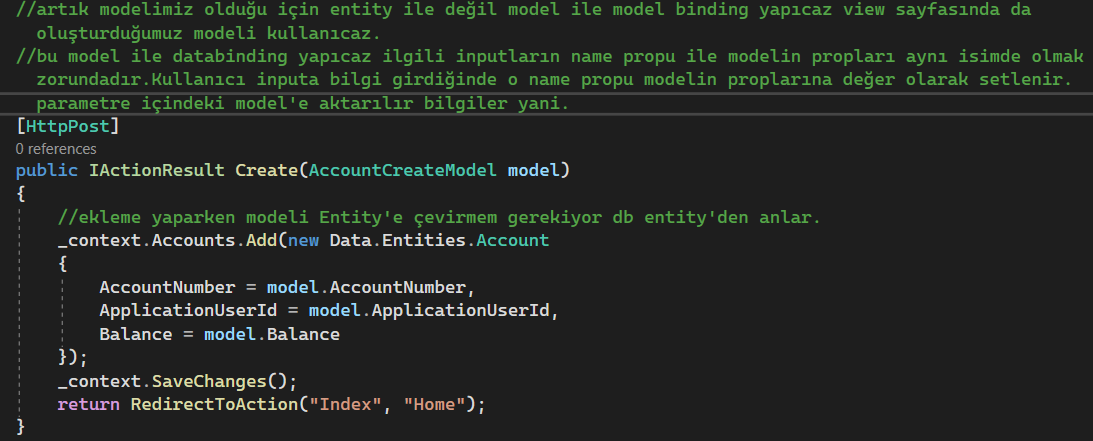
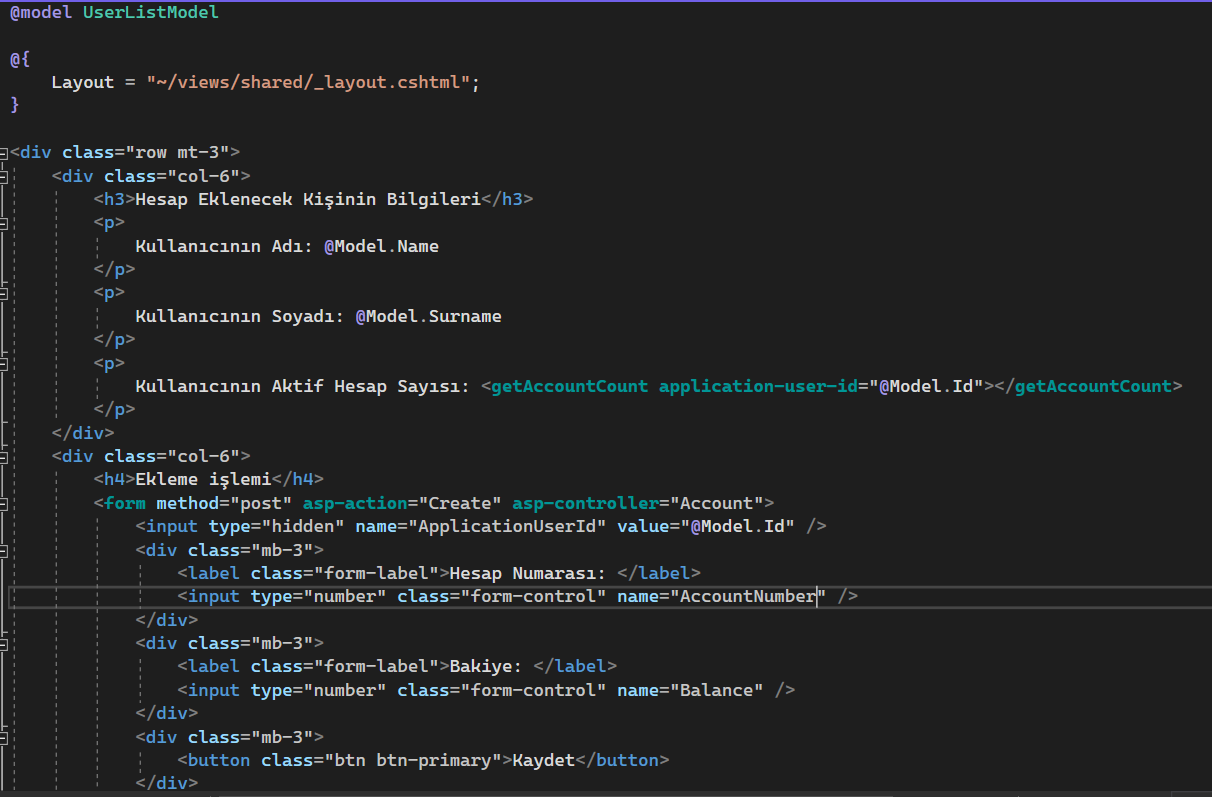
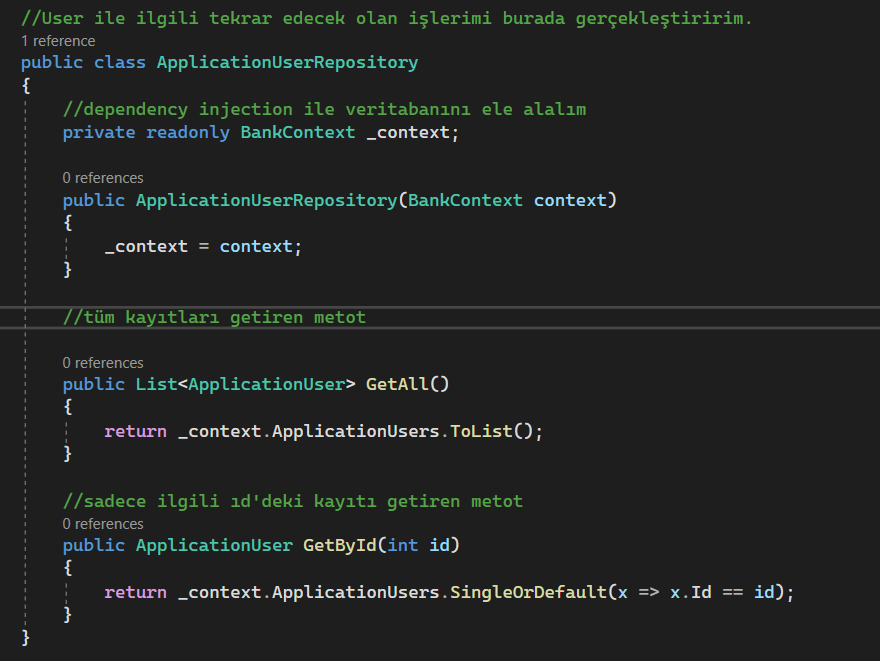
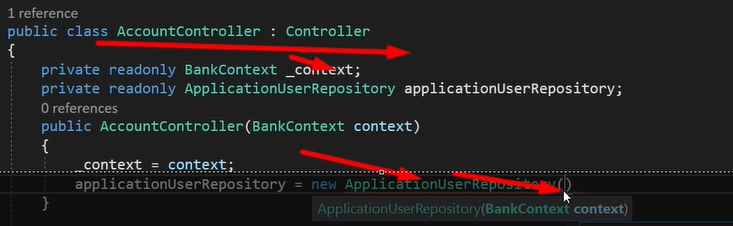
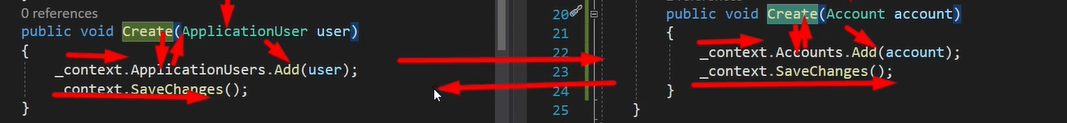
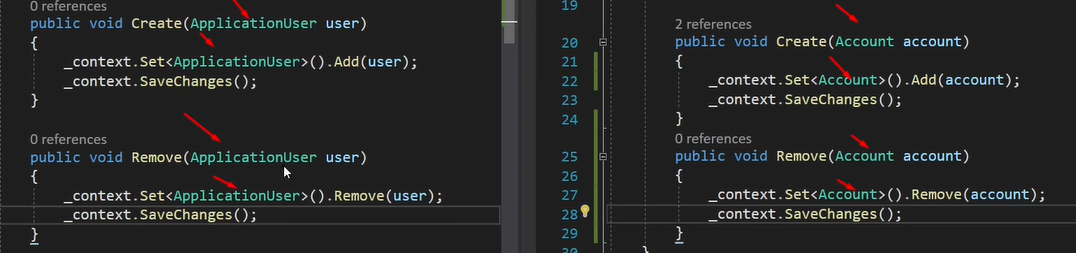
- modelBuilder.Entity<Category>().HasMany(x => x.ProductCategories).WithOne(x => x.Category).HasForeignKey(x => x.CategoryId);

* Database tablolarında TABLE PER HİERARCHY denen bir kavram vardır. Bir entity içerisinde başka entityler oluşturup o entitiyleri ana entitityden kalıtıp bunları database’e yollarsak eğer ayrı ayrı tabll oluşmak yerine sadece ana entity’den tek bir tablo oluştuğunu ve diğer alt entitylerin bu tablo içerisine kolon olarak geldiğini görebilririz.
* 
* ****
* İşin database kısmına yansıması bu şekilde olacaktır ve biz bu kolonlara veri eklersek eğer. Görünüm aşağıdaki şekilde olacaktır
* 
* ****
* Burada Discriminator ayırıcı anlamı taşımaktadır. Kolonların nullable olduğuna dikkat edelim. Eğer çalışan Parttime ise HourlyMage boş oluyor yani saatlik ücreti boş geliyor. Yani özetle TABLE PER HİERARCHY aynı tablo içerisinde bize böyle bir kullanım imkanı sunuyor.
* TABLE PER TYPE kavramında ise TABLE PER HİERARCHY ile tamamen aynı mantık var sadece discriminator adında bir kolon yok onun yerine ayrı tablolarla bakıyor bu işe. Bu kavram için yapmamız gereken küçük bir configurasyon var. Bunun için ilgili context’in OnModelCreating metotunun içine aşağıdaki kodu ekliyoruz. Bu kodlar sayesinde bunların bir tablo olacağını entityframeworkcore’a söylemiş oluyoruz. Artık bunları tablo olarak ele alıyor ve tph’e göre değil tpt’ye göre maplame işlemini gerçekleştiriyor. Artık database tarafında bunların ayrı tablo olarak geldiğini görebiliriz.
* modelBuilder.Entity<Employee>().ToTable("Employees");
* modelBuilder.Entity<FullTimeEmployee>().ToTable("FullTimeEmployees");
* modelBuilder.Entity<PartTimeEmployee>().ToTable("PartTimeEmployees");
* 
* 
* Bir Emplooyes tablomuz var ve birebir olarak 2 tabloya da bağlı olduğunu görebiliriz. Bir emplooye ya PartTimeEmployee olabilir ya da FullTimeEmployee olabilir. Benşm fulltime yada parttime kaydım varsa bir emplooye kaydımında olmak zorunda. Sonuç olarak TABLE PER HİERARCHY ile aynı işi yapmış oluruz fakat farklı olarak bu sefer bunları ayrı ayrı tablolarla yapmış oluyoruz.
* Eğer yanlış bir MİGRATİON oluşturursak bundan kurtulmak için öncelikle get-migration yazarak hangi migration’ların uygulanıp uygulanmadığı kaç tane migrationunuzun olup olmadığını burda gözlemleyebiliyoruz. Applied kısmı true ise migration uygulanmış demektir. Ben son olan yanlış oluşturduğum migrationu remove-migration demem yeterli son migration kaydı silinecektir. Yanlış bir uygulama sonucu ben bir önceki migrationuma geri dönmek istersem veya herhangi bir migrationuma geri döneyim istiyorsam update-database migrationismi şeklinde kullanım yapabiliyorum. Bu durumda son eklenen migration applied durumu false olarak işaretlenir ben bu false migrationdan kurtulmak istersem remove-migration yazdığımızda artık son migration silinmiş olur tamamen. Diyelim ki ben migrationların farklı bir konumda oluşmasını istiyorsam önce tüm migrations verilerini ve databasemi silerek istediğim konuma Migrations adında bir klasör oluşturmam gereklidir. Daha sonra add-migration migrationismi -outputDir nereyekonumlandırıcaz(Data/Migrations) şeklinde kullanım yapabiliriz. Default Proje kısmındaki seçime dikkat etmemiz lazım bunun için. Artık yeni bir migration eklerken bu kullanımı kullanmaya dikkat etmem lazım.
* Bu zamana kafar FluentApi işlemlerimizi context içerisindeki OnModelCreating metotu içinde yaptık ancak çok basit bir uygulamamız olsa bile bir çok ayar yapıldığı için o metotun içi şişiyor çok karmaşık bir hal alıyor ben projelerimde böyle olsun istemem. Bu yüzden Data klasörüme gidiyorum. İçerisine Configurations isminde bir klasör ekliyorum bu klasör ismi Mappings olarakta kullanılır tercih meselesi. Bu klasör içine yeni bir class ekliyorum class ismine hangi entitiy ile çalışma yapacaksam onun ismini veriyorum örnek olarak BlogConfiguration ismini veriyorum. Daha sonra bu classı IEntityTypeConfiguration İnterfacesinden kalıtım yapıyorum. Bu interface bizden bir entity istiyor <> içerisine entity ismimi veriyorum. Daha sonra Configure metotunu implemente ediyorum. Ben artık bu blog entity’i için ayarlamalarımı context içinde yapmaktansa burada sade ve anlaşılır bir şekilde yapmış olurum. Ben bunu yaptım ancak benim contextimin bundan haberi yok onu haberdar etmem gerekiyor. Contextimin OnModelCreating metotunun içine **modelBuilder.ApplyConfiguration(new BlogConfiguration());** yazmamız yeterli oluyor.
* Eğer ben mesela controller sınıfımda kodfirst ile database tabloma kayıt ekleme işlemi yapıyorum fakat kodu çalıştırdığımda database’e nasıl bir sorgu gittiğini görmek istiyorsam yapmam gereken ilgili contextimin içinde OnConfiguring metotunun içinde aşağıdaki kodu eklemem gerekiyor. Bunun için entityframeworkcore 5.0 gereklidir. **optionsBuilder.LogTo(Console.Writeline);**
* Eğer aşağıdaki kodu yazarsakta sadece sql soruglarını görürüz.
* optionsBuilder.LogTo(Console.WriteLine,Microsoft.Extensions.Logging.LogLevel.Information);
* EntityFrameWorkCore bir şeyin Enumerable olduğunu gördüğünde onu gelir size tekrar geri verir. ToList’i gördüğü an ilgili tracking’i bitirir ve datayı gösterir. Yani ToList kullanıldığında ne kadar Blog tablosunda diyelim data varsa bunu gelir bir sqlserver’ın anlayacağı bir sorguya çevirir ilgili sorguyu database tarafında işletir ve işlettikten sonra tekrar geri döner. ToList dediğimizde gerile bir List tipi döner ancak Ben ToList yerine .AsEnumerable() dersem geriye IEnumarable<Blog> dönüyor.
* 
* Bizim veriyi çekerken bilmemiz gereken 2 adet kavram var. IEnumerable ve IQueryabledır. Bir şeyin türü IEnumerable, Enumerable ise biz buna sayılabilir yani foreach ile dönülebilir deriz. IQueryable ise sorgulanabilir gibi düşünebiliriz. Eğer bir Where sorugusu koyarsak geriye IQueryable bir değer döner. Ancak biz başta .AsEnumrable() olarak belirtirsek bu arkadaşın database ile işi bitti yani db ‘ye gidip bir daha sorgu atmadı where kullansak bile db’ye giden sorguda where parametresi olmaz buradaki sorgu programatik olarak gerçekleşir. Ancak eğer bir şey IQueryable ise db’ye where sorgusu varsa where sorugusu’da gidecektir. Yani özetler bir şey eğer .AsQueryable(), IQueryable ise bunun daha database ile işi bitmedi. Yani yazmış olacağınız diğer sorgular veya işler database tarafında yorumlanır. Ancak bir şey eğer .AsEnumrable(), IEnumerable ise ya da .ToList() geçtikten sonra artık bunun database ile işi biter. Yapılacak olan işler programatik olarak gerçekleşir gidip database tarafında sorguya dönüşmez.
* 
* Bir Context oluşturduktan sonra yani DbContext’ten bir örnek aldığımız zaman Context üzerinde yaptığımız hareketleri entityframeworkcore izlemektedir. Bu izleme olayına genel çatı altında ChangeTracking deniyor. Varsayalım bir db miz olsun seneryomuz şu olsun ben gidicem db tarafından bir tane kayıt çekicem ıd si falanca filanca olan kaydın singleordefault’unu ya da firstordefault’unu bana getir. İlgili kayıt bana geldiği zaman EfCore bu kaydı izlemeye başlar. Siz eğer bu kaydın üstünde herhangi bir değişiklik yaparsanız bu kayıdın entriystatmant’ı modified olarak setlenir. Ve sizin çağırmış olduğunuz SaveChanges() metotu çalıştığı anda efcore ilgili şeyin state’ine bakar state’i eğer modified ise günceller, eğer added ise ekler, eğer deleted ise siler. Seneryomuz veriyi çekmek üzerine kuruluydu peki ben bir veride güncelleme işlemi gerçekleştirmeyeceksem Tracking’in çalışması benim için bir maliyettir. Ben sadece ilgili veriyi çekicem ve son kullanıcıya göstericem Tracking’in çalışmasını istemiyorum. FirstOrDefault metotu birden fazla kayıt gelirse ilk kaydı seçmektir. SingleOrDefault ise gelen bir tek kayıdı seçmektir. Tracking kapatmak için .AsNoTracking() metotunu kullanırız bu metotu koyduğumuz zaman bu kaydı artık izleme diyoruz. Eğer bu metotan sonra güncelleme işlemi yaparsak işlem gerçekleşmeyecek. Sebebi ilgili state modified değil DETACHED olarak işaretlenir. Bu yüzden Ef hiç bir iş yapmayacaktır. EfCore artık bu kaydı izlememiş oluyor.
* 
* **LAZY LOADİNG:** Bloglarımız olsun ve bu bloglara yapılan yorumlar olsun. Ve ben istiyorum ki ilgili blogla beraber yorumlarımda gelsin. Böyle bir seneryoda farklı veri yükleme yöntemlerimiz var. Bunlar LAZY, EAGER, EXPLİCİT Bunlardan harhangi birini kullanarak veri yükleme işlemini gerçekleştirebiliriz. En çok yaygın olarak kullanılan EAGER yöntemidir. Tabi bu seneryonun gerçekleşmesi için tablolarımda önce ilgili ilişkileri ve nav proplarını kurmam gereklidir. Lazy yöntemi Efcore’da default olarak kapalıdır. Bizim kullanmak içiz Lazy loading’i aktif hale getirmemiz gereklidir. Lazy Loading’i kullanabilmek için Microsoft.EntityFrameworkcore.Proxies paketine ihtiyaç duyarız. Lazy Loadingi devreye sokabilmek için Context içindeki OnConfiguring metotunun içinde UseSqlServer metotunun önüne UseLazyLoadingProxies() metotunu eklemeliyiz. Ayrıca bir ayar daha yapmalyız. İlgili Comment ve Blog entity’i içinde nav propları virtual olarak işaretlemeliyiz. Ardından farkederiz ki her olay gerçekleştiğinde ayrı bir sorgu database’e gidiyor yani ben ne zaman ilgili commenti çekmek istesem db’ye bu sorgu gidiyor. Ben her bir yorum datasını çekmek istediğimde her bir blogum için gidiyor ilgili select sorgusunu atıyor. Ben istiyorum ki bu sorgu bir kere çalışsın bu data bana bir kere gelsin. Lazy loadiingin davranışı her bir isteğinize karşılık gider db’de ilgili sorgusunu çalıştırır geri döner. Yani her bir istek gerçekleştirildiğinde db tarafına gidip ayrı bir sorgu çalıştırıyor. Yani burada tüm blog kayıtlarını ve bloglara gelmiş olan commint kayıtlarını aynı zamanda getirebiliyoruz.
* ****
* Bu resimde farkettiysek Blog tablosu üzerinden Comments tablosuna gidebildik bunu nav prop sağladı.
* **EAGLE LOADİNG:** Yapmamız gereken tek şey blogları çekerken .Include() metotunu kullanmamız. Daha sonra neyi işin içine dahil edeceğimizi söyleriz (Comments’ı). Böyle yaparsak eğer artık ilgili Bloglarla beraber Commentlar’da yükleneceği için dolayısıyla doğrudan tek bir sorguda commentları çekmiş oluruz. Lazy’in aksine db’ye sadece bir tane sorgu gider. Ve bu sorguda ilgili commentları’da işin içine dahil etmiş oluruz. Ayrıca efcore 5.0 ‘dan sonra Include sorgusunun içinde sonunda .Where() kullanarak hangi komutları dahil edip etmeyeceğimiz hakkında da konuşabiliyoruz. Tabi bu eagle loadingi kullanırken nav propların önemini de görmüş oluyoruz onlar sayesinde iki tablo kaydını birden getirebiliyoruz. Yani burada tüm blogların içinden commentsi sadece Yorum1 olan blog kayıtlarını ve comment kayıtlarını getiriyoruz.
* ****
* **EXPLİCİT LOADİNG:** Explicit Loading kullanabilmek için elimizde bir data olmalıdır. Ve biz bu datanın nav propuna ulaşıp o propu load etmemiz gerekiyor. Explicit Loading’te Load() metotunu ne zaman gözükürse kullanılırsa o zaman loading işlemini gerçekleştirir. Explicit Loading’te Doğrudan ilgili entity üzerinden hareket edemiyoruz dolayısıyla ilgili context’in Entry’si üzerinden hareket ediyoruz. Collection içine neyi yüklemek istiyorum ben bunun Comments’larını yüklemek istiyorum .Load kullanarak’ta o zaman yüklenir. Yani 1. Blogun yorumlarını bu şekilde getirebiliyoruz.
* ****

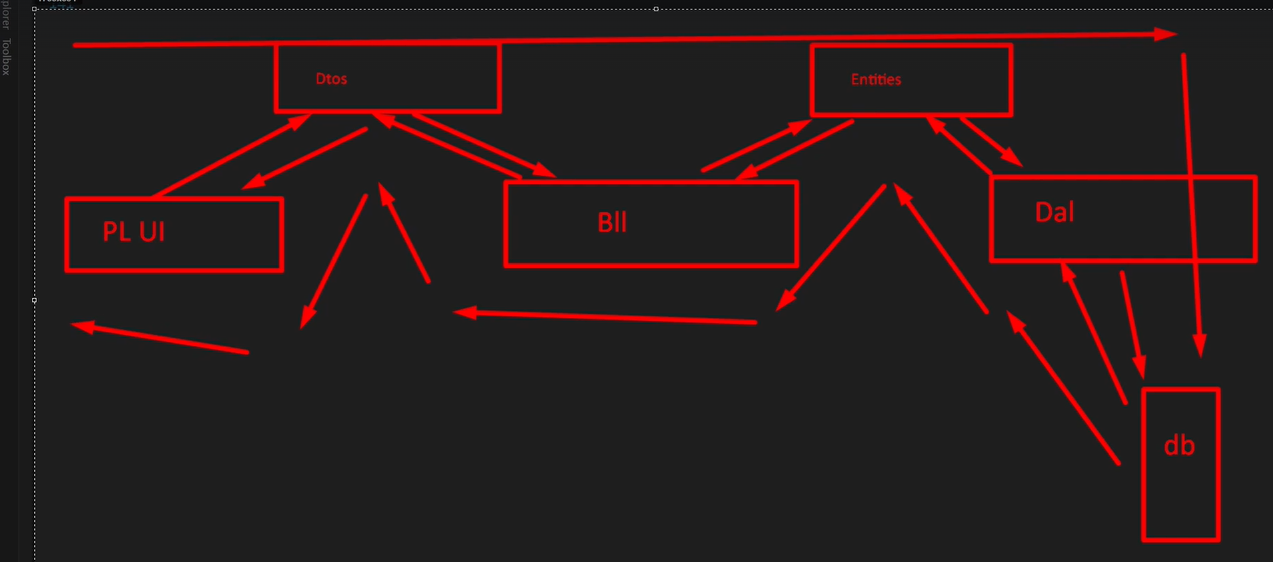
**DEPENDENCY INJECTION**

* App.UseEndpoints metotunun içine endpoints.MapDefaultControllerRoute() metotunu eklersek bu metot varsayılan olarak route’ı {Controller=Home}/{action/Index}/{id?} olarak ayarlayacaktır.
* EN BAŞTA ŞUNU UNUTMAYALIM İNTERFACELER’İN ÖRNEĞİ OLUŞTURULAMAZ DEPENDENCY İNJECTİON’DA İNTERFACELERE NORMAL CLASSLARIN ÖRNEĞİNİ BAĞLAMA SEBEBİMİZ BU. ÖRNEK OLUŞTURMADAN NEW DEMEDEN İNTERFACE DEĞİŞKENİNİN İÇERİSİNE CLASSIN ÖRNEĞİNİ GETİREBİLİYORUZ.
* Dependency Inversion şununla alakalıdır. Ben bir nesneyi doğrudan New’lememeliyim. Çünkü ben burda ProductManger’da bir değişikliğe gidersem artık ben bunu kaç yerde böyle new diyerek nesne üreterek çağırdıysam her yerde bu değişikliği tek tek yapmam gerekiyor. Dolayısıyla bizim bu bağımlılıktan kurtulmamız gerekiyor. Yani newleme olayından kurtulmamız lazım. Bunun için bir interface ile hareket edicez. Dependency Injection ise kanstraktır ile alakalı bir durumdur. Biz kanstraktırın içerisinde parametre olarak IProductService productService geçtik. Biz bunu bir configurasyon yaparak şu hale getirebiliyoruz. Sen bir kanstraktırın içerisinde bu interfaceyi gördüğün zaman git şurdaki classı örnekle diyebiliyoruz. Şurdaki class derken normal olan classımız oluyor interface değil. İşte tam olarak bu yapıya Dependency Injection deniyor. Bu sayede bir metotu kullanmak için her seferinde new diyerek örnekleme yapmaktan kurtuluruz ve ayrıca yarın öbür gün bu metotta veya classta bir değişiklik olacaksa tek tek tüm new dediğimiz nesnelerden değişiklik yapmaktansa tek bir yerden değişiklik yapmış oluruz.
* 
* Biz bunu aslında startup dosyasında bulunan ConfigureServices metotunun parametresi olan IserviceCollection ile söylüyoruz. Yani bu metot içerisine gidip services.AddScoped dediğim zaman tabi bunu AddSingleton, AddTransient adında 3 adet metotla belirtebilriiz şuanlık AddScoped hakkında konuşalım. AddScoped dedikten sonra sen <IproductService,ProductManager> IproductService’yi gördüğün zaman ProductManager’i örnekle diyebiliyoruz. Nerde gördüğün zaman kanstraktır’da gördüğün zaman. Eğer sen kanstraktırda interface olarak IproductService görürsen gidip ProductManager’i örnekle diyoruz. Kanstraktır parametresi içinden bahsediyoruz. Kanstraktır parametresinde IproductService gördüğü için uygulama çalıştığı zaman parametre içindeki interface olan productService’nin üzerinde ProductManager classının örneğini taşıyacak.
* 
* Peki daha sonra ben bunu kanstrakatır içerisinde ele aldığım için productService interfacesinden bahsediyorum bu değişkene ulaşabilmem gerekiyor. Yani bu interface’e bir şekilde ulaşmam lazım. Ancak ben bu interface olan productService’e biliyoruz ki sadece kanstraktırın scopu içinde erişebiliriz. Bu noktada kanstraktırdan önce aynı tipte bir değişken tanımlamam lazım. Private readlonly IproductService \_productService; şeklinde bir değişken tanımlıyorum. Private olmasının sebebi sadece ilgili scopun yani HomeController classının içinde kullanulabilmesi. Readonly ise sadece ilgili kasntraktıın içerisinde bu değer setlenebilir. Ve daha sonra kanstraktır scopeu içerisinde this.\_productService = productService; derim. Burada This’deki kasıt bu HomeController classı scopunda bulunan \_productService ‘a dependency injectiondan gelen productService’i ver diyebiliriz. Bu sayede artık kanstraktır dışından da bu dependency değişkenine ulaşabiliriz. Resimde farkettiysek normal classta olan GetTotal metotuna bu interface üzerinden ulaşabiliyoruz.
* 
* **STARTUP DOSYASI (**startup ortak bir dosyadır bunu bir kere tanımlamak yeterlidir tüm controller veya classlar bunu kullanabilir hale gelirler.**)**
* 
* Dependency Injection geçtiğimiz zaman bu interfaceyi gördüğün zaman şu classın örneğini bana ver diyoruz. AddScoped, AddSingleton, AddTransient ‘im tamamıyla örneklemeyle alakalıdır. TransienService şu şekilde davranır eğer bir request içerisinde iki farklı Transient isteği varsa ayrı ayrı 2 nesne örneği gelir size. Bir request içerisinde birden fazla Scoped durumu söz konusuysa bu sefer size o request için aynı örnek gelir. Singleton’ta ise kaç defa request atarsanız atın her request için aynı sonuç gelir. Scoped ve Singleton farkı ise sayfa yenilendiğinde yani yeni bir request geldiğinde Singletonlar hiç bid değişim göstermezken hep aynı örneği fırlatırken, Scopedler kendi içlerinde aynı olsalarda değişim gösteriyorlar farklı örnek fırlatıyorlar. Transientler ise hem kendi içlerinde farklı oluyorlar hemde yeni istek gelince yine değişiyorlar farklı örnek fırlatıyorlar.
* 
* **YENİ İSTEKTEN SONRA**
* ****
* CSHTML sayfası içerisinde @inject kullanarak injection geçebiliyoruz. Yani dependency injectionu chstml sayfası içerisinde de geçebiliyoruz bu sayede.

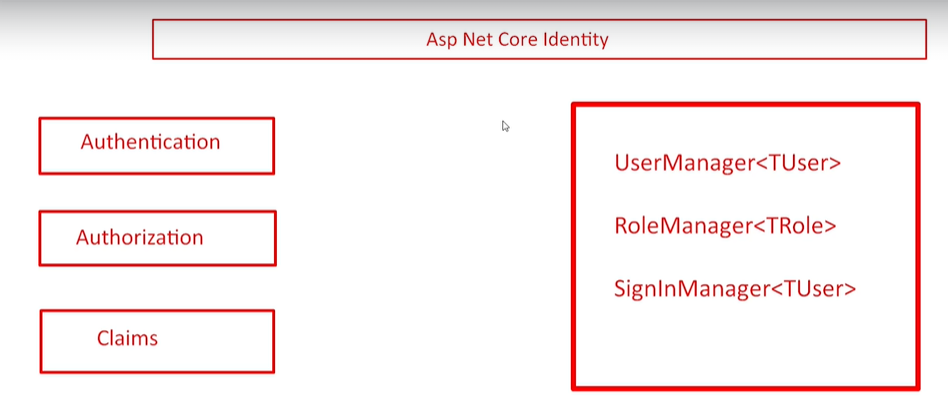
**REPOSİTORY DESİGN PATTERN – UNİT OF WORK**

* Artık Dependency Inhect3ion kavarmını bildiğimiz için. Controller’larımızın içerisinde veri tabanında yani context ile işlem yapmak için her action metotunta ayrı ayrı new deyip contexten yeni bir örnek almak yerine controller’ın en başına sadece bir kere contexti Dependency Injectıon yapmamız yeterli daha sonrasında tüm action metotlarında örnekleme yapmadan kolayca context ile çalışabiliirz.
* 
* Artık veritabanı bağlantısı için context içinde OnConfiguring metotunu override etmicez. Context içerisinde sadece dbset ve configuration sınıfı tanımlamaları kalacak. Bunun yerine Context içerisinde bir Kanstraktır oluşturucaz. : base() ile base’e gidicez. Base() ikinci parameteresinde DbContextOptions<ilgiliContextismi> options’u parametre olarak alıyor. Ben bu Option’u parametrede veriyorum ve o zaten gidip sanki bir OnConfiguring metotu gibi çalışıyor. Daha sonra bağlantıyı yapacağımız yani options’laı vereceğimiz yer ise StartUp dosyası içerisindeki ConfigureServices metotunun içerisinde yapıcaz. Önce DependencyIncetion’u geçebilmek için services.AddDbContext<BankContext(ilgili context ismi)> burada bunu tanımlamazsak uygulamada classların içerisinde BankContext’i dependency injection ile kullanamayız services.AddDbContext dediğimiz zaman artık bu context dependecnt injection ile hareket edebilir hale geliyor yani sen kanstraktırda BankContext’i gördüğün zaman bir BankContext örenği al diyebiliyorum neyle alt opt ile yapacağımız configurasionlarla al diyoruz. Yani burada startup içinde tanımladığımız opt gidecek context sınıfı içindeki kasntraktırdaki parametre içindeki options yerine geçecek ve buda gidecek DbContext sınıfındaki OnConfiguring metotunu benim burda vermil olduğum ayarlamalara göre ayağa kaldıracak.
* 
* 
* Entity’min içerisinde bazı, View sayfalarımda hiç kullanmadığım proplar olabilir. Dolayısıyla bunları son kullanıcıya boş yere göstermenin anlamı yoktur. Dolayısıyla iligli kullandığım propları ilgili view’a gönderecek bir nesneye ihtiyacım var benim. İşte bu noktada model kavramı ortaya çıkıyor. Oluşturduğum modelin içererisinde ilgili View sayfamda hangi propları kullanacaksam sadece o propları tanımlarım bu sayede kullanıcı o view sayfasını ilgilendirmeyen propları hiç görmemiş olur. BENİM DORĞUDAN VİEW’LERE ENTİTY GÖSTERMEM HİÇ DOĞRU BİR HAREKET DEĞİL. Mesela işin içine Validation kuralları girince bunu direkt entity’lere uygularsak veri kaynağımız bozulur işin içinden çıkılmaz bir hale gireriz. Bu yüzden modeller üzerinden ilerlemek en mantıklsıdır. En basitinden mesela bir create metotumuz var o create metotunda tc kimlik numarası zorunludur ama başka bir controller’da bir tane daha create metotumuz vardır ama onda tc kimlik nuamrası zorunlu değildir. Bu durumda tek bir entity üzerinden yani data kaynağından ilerliyor olsak tıkanır kalırız. Fakat 2 ayrı model oluştursak onları ayrı ayrı ilgili create metotlarında kullansak hem data kaynağı bozulmaz hemde istediğimiz create metotuna istediğimiz modeli verip uygun data ortamını oluşturabiliriz. Model oluştrudan sonra bu modeli Viewlere dönme durumumuz söz konusu dolayıısyla ilgili controller’a gideriz. View’a model gönderirken .Tolist(), ten önce \_context.Entity seçtikten sonra araya .Select(x=> new Modelismi {}) diyerek modelin içini doldururuz. Yani burda bir nevi entity yani database içindeki data kaynaklarını modelin proplarına aktarmış oluruz. Ve artık View sayfasında model olarak ilgili modelin ismini bekleriz. Entity’in değil.
* CONTROLLERDAN MODELİ VİEW CSHTML SAYFASINA GÖNDERMEK.
* 
* 
* Ancak şöyle bir şey var. Örneğin biz bir create işlemi yaparken model binding olarak parametre içerisine oluşturduğumuz modeli vericez. Çünkü view sayfamız bu oluşturduğumuz modeli kullanacak entity’i değil. Ancak biz ekleme işlemini yaparken bu modeli databaseye gönderirken geri entity’e çevirmemiz gerekiyor. Database modelden anlamaz entity’den anlar.
* 
* 
* Aynı iş kümelerini birden fazla kez kullanacaksak yani mesela db den tüm kullanıcı bilgilerini çekiyorsam ve bunu tekrar tekrar bir çok yerde aynı şekilde aynı kodlarla uzun uzun çekip kullanıyorsam bu bir yanlıştır.Bu konuda bir prensip vardır DRY prensibi. DRY prensibi bize der ki kendini tekrar etme. Bu şekilde birden fazla kez kullanılan tekrara düşen kodlarımızı yapacak olan bir metot yazabiliriz mesela GetAll metotu gibi bu metot bize tüm kullanıcı bilgilerini verir ve bu metotu global bir noktaya taşımamız gerekir ki tüm uygulama içerisinden bu metota erişebilelim. Aynı zamanda yarın öbür gün bu metotta bir değişiklik olduğunda tek tek her bu metotu tanımladığımız yerde değişiklik yapmaktansa tek bir noktada değişiklik yapmamızı sağlasın. İşte tam olarak bu noktada RepositoryDesingPattern kavramı ortaya çıkıyor. Bizim kendimizi tekrar etmeyeceğimiz bir altyapı kurgulamamzı sağlıyor. Dolayısıyla Data klasörümüze Repositories adında bir klasör oluştururuz. Klasör içerisinde ilgili işlerimizi kümelere bölüyoruz. Yani User ile alakalı işlerimi UserReposityory classı içerisinde yapmam gerekir ki repository’ler arasında bir karışıklık yaşanmasın.
* Burda önemli bir nokta var Data katmanındaki işlerimizi entity’lerle yaparız modellerle değil. Dolayısıyla Repository classlarımızın içiersinde Entity’lerle çalışmamız gerekiyor.
* 
* BUNU NASIL KULLANABİLİRİM UYGULAMAMDA?: Öncelikle kullancağım action metot içerisinde bir dependency injection yapmam gerekiyor ilgili repository ile alakalı olarak çünkü ilgili repository’e bağımlı kalmak istemiyorum her defasında bu repository’i kullanmak için nesne,örnek üretmek istemiyorum. ApplicationUserRepsitory() kanstraktır yani parametresi benden bir context bekiyor. Aslında dependency injection ‘ın çalışma mantığı doğrusaldır. Yani controllerlardan başlar ilgili classlara doğru gider. Yani Controller’ın içerisinde bir context görürse bu örneği alır ve daha sonra siz burda ilgili ApplicationUserRepository’i çalıştırdığınız zaman burda almış oldığunuz context örneğini gelip buraya verirsiniz. Siz bu örneği verdiğiniz zaman ilgili context parametresini benim göndermiş olduğum örnek setlenir. Ve ben bu örnekle yoluma devam edebilirim.
* 
* Ancak böyle yapığımızda new diyeceğimiz için bir bağımlılık oluşmuş oluyor ve yaptığımız işe ters kalıyor bu bağımlılıktan kurtulmak için ilgili repository’imin inteface halini oluşturmam gerekiyor. Ve ilgili reposityoryimi bu interface’ten kalıtmam gerekiyor. Daha sonra ben artık dicemki sen IApplicationUserRepository interfacesini gördüğün zaman git dependency injection aracılığıyla ApplicationUserRepository’in bana bir örneğini ver. Ben böyle dediğim zaman bu BanckContext’te dependency injection ile ele alınacak. Çünkü ben bir interface üzerinden hareket ediyorum ve kasntraktır’da bunu gördüğü zaman gidecek ilgili inejction’a geçecek. Yani ben artık dependency içerisinde tanımlama yaparken interfaceler ile tanımlama yapıcam ama intefaceler benim normal classlarımın örneğini taşıyacaklar.Benim bunu startup tarafında ConfigureServices metotunda söylemem gerekiyor. Daha sonra ben artık controller içinde dependency inejction içindeki bağımlılığımı kaldırabilirim new dememe gerek kalmadı. Ancak farkettiysek controller’da artık sayfaya gönderdiğimiz model entity oldu model değil çünkü repository’lerde entityler ile çalıştık. Bildiğimiz üzere bu yanlış bir şey biz bu entity’i model’e çevrimek amacıyla projeye yeni bir Mapping adında klasör ekleyelim. Ve UserMaping adında class ekleyelim. Bu class içerisinde bir takım metotlar yazıcaz. Bize entity olarak bir liste gelecek gelen listeyi modele çeviricez. Dependecny injection için bunun interface halini oluşturmayı unutmayalım yoksa new demek zorunda kalırız bağımlı kalırız. Bu classı İnterface halinden kalıtalım. Unutmayalım ki interface’nin örneği oluşturulmaz aslında tüm olay buna bağlı. Startup dosyasına bununda dependency injectionunu geçiyoruz addscope olarak.
* Bakınız burada 2 farklı Repository’imde 2 adet create metotum var. Bu metotlar aslında bakıldığında tamamen aynı işi yapıyorlar sadece nesnelerinin değişim gösterdiğini görebiliyoruz işte bu noktada GENERİC REPOSİTORY kavramı ortaya çıkıyor.
* 
* Buradaki 2 var değişkeni de aynı işi yapıyor aslında. Geriye dbSet türünden bir tablo dönüyor. Aralarında hiç bir fark yok.
* 
* Burada görüldüğü üzere değişen kısımlar sadece ilgili nesneler onun dışında 2 Repositoryim’de de metotlarımın yaptığı işler tamamen aynı böyle bir durudma GenericReposiyory kavramını kullanmalıyız.
* 
* Eğer ben GenericReposityory ile bir işlem yapacaksam Entity’lerden bağımsız olmam gerektiğini bu noktada anlamış olmalıyız. Benim 1000 adet entity’im varsa ilgili oluşturduğum GenericRepository’im tüm bu entityler için tıkır tıkır çalışmalıdır. Ben 1000 adet entity’in hepsi için tek tek aynı metotları yazmamalıyım.
* GenericRepository sınıfımı Repository classımın içerisinde oluştururum class isminin yanına<T> eklerim yani bu class generic ve herhangi bir T tipi için çalışıyor diyoruz ve ayrıca hemen yana where ekleyerek bir takım kısıtlamalar koyabilirim. T bir class olmalı ve newlenebilmeli yani bu t için gidipye bir abstract metot koyamasın ya da bir static metot. Bu classın bir interface’den kalıtmayı unutmuyorum yoksa bağımlı kalırız. Daha sorna startup dosyasında addscope olarak bu bağımlılık olayını belirtelim. Ancak bunu normal şekilde değil typeof aracılığıyla belirticez hata alırız yoksa. Herhangi bir tip için çalıştığını belirtiriz bu şekilde.
* services.AddScoped(typeof(IGenericRepository<>, typeof(GenericRepository<>));
* **Unit Of Work** 6 7 tane Repository’im var bunların hepsi aynı context ile çalışacaksa. Bu sınıfı kullanıyoruz. Ayrıca bu sınıf içeriisnde SaveChanges’ı de ayırmış oluruz. Daha sonra startup’ı değiştirmeyi unutmamalyıız. Bu pattern, iş katmanında yapılan her değişikliğin anlık olarak database e yansıması yerine, işlemlerin toplu halde tek bir kanaldan gerçekleşmesini sağlar. Bir e-ticaret sitesini düşünelim.Bir ürün satın alma aşamasında birden fazla aşama vardır.Sepetteki ürünlerinize göre ekstra ürün eklemek isteyeceğiniz bir sayfa, sonra kart bilgilerini gireceğiniz bir sayfa, en son alışverişi tamamlayacağınız bir sayfa olur.Bu aşamaların her birinde database e kayıt atmak yerine, en son satın alma aşamasında bu girilen kayıtların toplu halde database e aktarımı hem performans, hemde sağlıklı data açısından önemli olacaktır.Çünkü son aşamaya gelindiğinde satın almadan vazgeçilmesi halinde önceki aşamalarda oluşan datalar dummy data olarak database’de ekstra yer işgal edecektir. Bu ve bunun gibi, birden fazla işlemi tek bir işlem olarak düşünmemiz gereken yerlerde bu design pattern’i kullanabiliriz. İşlemler tek bir kanaldan(tek bir transaction) toplu halde yapıldığı için performansı artı yönde etkileyecektir.Ayrıca işlemleri geri alma(rollback), hangi tabloda ne işlem yapıldı kaç kayıt eklendi gibi sorulara da cevap verebilir. Unit of work sınıfı her controller için ortaktır. Çünkü DbContext içinde yaptığımız her değişikliğin veritabanına yansıması için bu sınıf içerisindeki SaveChanges() metotunu kullanıyoruz. YANİ UYGULAMIMIZDA BÜTÜN REPOSTİYORY SINIFLARINI AYRI AYRI ELE ALIYORUZ HER BİR REPOSİTORY ÜZERİNDEN BİR İŞLEM YAPTIĞIMIZDA CONTEXT.SAVECHANGE() İ ÇAĞIRIYORUZ BUNA GEREK YOK. BU REPOSİTORY SINIFLARININ HEPSİNİ UNİTOFWORK SININDA GRUPLAYARAK KULLANIMINI OLDUKÇA BASİTLEŞTİREBİLİRİZ. Bu sayede controllerlarda ayrı ayrı bir sürü repository’i inejct etmemiz ve ayrıca startup dosyasında tüm repositoyleri uzun uzun addscope yapmamıza gerek kalmaz.

**N-TİERARCHİTECTURE-AUTOMAPPER-FLUENTVALİDATİON**

* **N-Tier Architecture:** N katmanlı mimaride aslında özünde 3 adet katman vardır. Bunlardan bir tanesi User Interface buna Presentation Layer’da denir.Bir diğeri Businness Logic Layer bir diğeri Data Access’tir. Bu katmanlar arasında iletişim tek yönlüdür. UI BLL’I, BLL ise DAL’I, DAL ise DB’i görür bu akış bu şekildeidir mesela bir veriyi insert ederken akış bu şekilde. Ters yönde de aynı şekilde ama tam tersi işler veriyi çekerkende akış bu şekildedir mesela. Tabi Entities gibi ara katmanlar oluşturmakta bizim elimizdedir. Mesela biliyoruz ki son kullanıcı doğrudan entityleri görmemelidir işte bu noktada DTOS Data Transfer Objects anlamına gelir bu katman ile modelleri entity’e veya entity’leri modele çevirebiliriz düz yönlü ve ters yönlü işleyecek şekilde düşünürsek. Birde Common katmanımız vardır bu katmanı tüm katmanlar ortak olarak kullanabiliyorlar ortak bir katmandır. Bu katmanların her birinin bir görevi vardır.
* Data Access Layer in olayı doğrudan database ile haberleşmektir. Yani buradaki kasıt SaveChanges() metotunu kullanması yada Create Update Delete Review gibi metotları doğrudan database’e iletmesi. İlgili database ile alakalı metotların kullanılması diyebiliriz.
* User Interface veya Presentation Layer katmanı son kullanıcın gördüğü katmandır. Bu bir web uygulaması, bir mobil uygulama, API olabilir. Yani Sunum katmanıdır.
* ****
* Data Access katmanına referans olarak Entities katmanımın referansını eklemem gerekir.
* Businnes katmanına referans olarak Data Access katmanının referansını eklemem gerekir. Bu sayede Business’ta hem DataAccess hemde DataAcces’in Entities Katmanını kullanabilirim. Ayrıca Businness katmanına Dto Katmanının referansını da eklemem gerekiyor.
* UI Katmanına ise referans olarak Businnes katmanının referansını eklerim bu saeyde UI hem Business’ın DataAccess ini DataAccess’in Entities’ini Hemde Business’ın Dtos katmanını referans olarak almış olur.
* Bu şekilde katmanlar arasındaki ilişkiyi kurmuş oluruz daha sonra projeyle çalışmaya başlayabiliriz. Projeyle çalışmaya Entities katmanından başlanır. Entityler oluşturulduktan sonra DataAccess katmanında bu işi yavaş yavaş gerçekleştirmem gerekiyor. Data Access’te Contexts klasörümü oluştururum birden fazla contextim olabilir. Context classımı oluştuduktan sonra DataAccess katmına ilgili EntityFrameWork paketlerimi yüklemem gerekiyor. DataAccess katmanına Configurations isminde klasör ekliyorum FluentApı’larım için. Migrationlarım için DataAccess katmanına Migrations isminde klasör oluştururum.
* Normalde contextimin ilgili bağlantılarını Startup dosyası içinde yapardım fakat katmanlı mimaride böyle olmuyor. Çünkü UI Katmanı doğrudan DataAccess katmanını görmemeli onunla çalışmamalı bunun yerine Businness katmanıyla iletişimde olması gerekiyor. Dolayısıyla Business katmanına gideriz ancak burada öncelikle Extension Metod yazmayı bilmemiz gerekiyor. Extension metotlarla amaçlanan şey şudur bir classı genişletmek , bir classın hali hazırda mevcut özellikleri olabilir bu özelliklere ek olarak bu classı genişletmemizdir bütün olay. Siz bir Extension Metot yazıyosanız bu metotu yazdığınız class’ın static olması gerekiyor. Ve yeni static classta static metotu oluşturarak parametre içerisine this genişetmekistediğinclassın adı değişken adı şeklinde tanımlama yapılır. Bunu yaparak biz bu classı genişletiyorum ve bu classa iligli metot ismnide bir özellik kazandırıyorum diyoruz. Bu sayede ana metot ismiyle ek classtaki ek metota ulaşabiliyoruz. Classa özellik kazandırmış oluyoruz. Konumuza dönelim bunu gerçekleştirebilmek için Businness katmanına DependencyResolvers adında bir klasör oluşturalım. Bu klasörün içinede biz Dependecny Injection kullandığımız için Microsoft adında bir klasör oluşturalım. Buradaki oluşturacağımız class içerisinde StartUp dosyası içindeki IserviceCollactionu genişleticez. Yani bu sayede contextin ilgili bağlantılarını ayarlamalarını bu class içerisinde yapmış oluruz. Daha sonra startup dosyasındaki ConfigureServices metotu içinde de services.ExtionMetotismi olarak bunu çağırırız. Dolayısıyla sturtup dosyası ilgili context bağlantılarını businnes katmanından almış oluyor. Daha sonra Migrationumu oluştururum. Yalnız burda önemli bir nokta var UI Katmanıma Design paketini kurmuş olmam lazım yoksa Migration oluşturamam. Bu sayede veritabanımı oluşturduktan sonra DataAccess katmanımda repositorylerimi oluştururum. 2 adet klasör eklerim Repositories ve Interfaces adında. Generic bir repository oluştururum ama Asenkronik şekilde. Daha sonra projeme UnitOfWork eklerim. Data Access katmına UnitOfWork adında klasör eklerim bu UnitOfWorkun bağımlılığını addscopunu Extension metotta belirtiyoruz artık startupta değil. Daha sonra artık Businnes katmanına geçiyorum. Services adında ve Interfaces adında bir klasör açıyorum. Daha sonra Dtos katmanında WorkDtos isminde bir klasör oluşturuyorum. Listeleme yapmak için WorkListDto adında bir class oluşturuyorum ve içerisine listeleme yapmak için gerekli olan proplarımı buraya yazıyorum. Daha sonra Businnes katmanında oluşturduğum Service dosylarında bir GetAll metotu yazıyorum ve metotun bilgilerini bu Dto katmanından getirtiyorum ve bu service classında dönüştürme işlemlerini yapıyorum. Entityden gelen bilgileri Dto bilgisine dönüştürüyorum. UI dto bilgisi ile çalışsın diye. Servislerimi Extension classında bağımlılıklarını vermeyi adscope vermeyi unutmayalım. \*\*\*\*\*\* BURADA İŞİN MANTIĞI ŞÖYLE BİZ REPOSİTORY’DE UYGULAMAMIZDA SÜREKLİ TEKRAR EDECEK OLAN GENEL GEÇER METOTLARIMI GENERİC OLARAK TANIMLARIZ YANİ METOTLARIN İÇİNE GELECEK OLAN ENTİTY VEYA DTO BİLGİSİ BELLİ DEĞİLDİR. İŞTE BU METOTLARA ENTİTY VEYA DTO YOLLADIĞIMIZ YER İLGİLİ SERVİCE CLASSLARIMIZDIR. SERVİCE CLASSLARIMIZDA HEM İLGİLİ ENTİTY VEYA DTOLARIMIZI BU GENERİC REPOSİTYORY METOTLARINA GÖNDERİM YAPARIZ HEMDE BU BİLGİLERİ GÖNDERMEDEN ÖNCE İLGİLİ DÖNÜŞÜM İŞLEMLERİYLE GİDECEK OLAN BİLGİNİN METOTA GÖRE DEĞİŞİKLİK GÖSTERECEK OLAN ENTİTY VEYA DTO OLACAĞININ KARARINI VERMİŞ OLURUZ. MESELA DATABASEDEN BİR BİLGİ ÇEKECEKSEK SON KULLANICIYA BİR LİSTE GÖSTERECEKSEK BUNU SERVİCE CLASSIMIZIDA DATABASEDEN ENTİTY OLARAK ÇEKİP DTO BİLGİSİNE DÖNÜŞTÜRÜP REPOSİTORY İÇİNDEKİ GETALL METOTUNA DTO BİLGİSİ GÖNDERİRİZ ÇÜNKÜ SON KULLANICI HER ZAMAN DTOLARIMIZI GÖRMELİ ENTİTYLERİMİZİ DEĞİL. VEYA DATABASEYE CREATE METOTUMUZLA BİR EKLEME YAPACAKSAK İLGİLİ SERVİCE CLASSIMIZIDA KULLANICIDAN BİLGİYİ DTO OLARAK ALIP DTO BİLGİSİNİ ENTİY’E DÖNÜŞTÜRÜR REPOSİTORYDE BULUNAN CREATE METOTUNA ENTİTY BİLGİSİ GÖNDERİRİZ ÇÜNKÜ DATABASE İLGİLİ İŞLEMİ GERÇEKLEŞTİRMESİ İÇİN ENTİTY’DEN ANLAR ENTİTY GELMEZSE EKLEME İŞLEMİNİ YAPMAZ. \*\*\*\*\*
* Artı UI katmanına geçebiliriz. Bir Mvc Eklenmesi yapılır.StartUp tarafında endpoints.MapDefaultControllerRoute() kullanabilirim route olarak. Ve mvc ekleyebilirim. WWWROOT doysamıda burada dışarıya açarım ve node\_modules’ı dışarıya açarım.
* Bir nesneyi bir nesneye çevirirken yani bir entity’i bir dto’ya çevirirken bazı kütüphanelerden faydalanabiliriz. En sık kullanılan 2 tanesi AutoMapper bir diğeri MapSter.AutoMapper aslında bir nesneyi bir başka nesneye Maplemeye yarıyor. Projeye Auto mapper eklemek için Nuget Package Manager’dan AutoMapper paketi install edilir. AutoMapper kullanabilmek için MapperConfiguration classından faydalanıyoruz. Daha sorna lambda exprassion ile bir opt değişkeni veriyourz parametresine ve sonra opt ile bazı ayarlamalar yapabiliyoruz. Opt.CreateMap<Product,ProductListDto> içine önce bir kaynak veririz yani Sen Product’ı PrdouctListDto’ya çevirebil diyoruz. Daha sonra bu ayarlamaları kullanmak için CreateMapper() metotunu kullanırız. Daha sonra .Map metotu ile ilgili çevirme işlemini yaparız. Burda önemli olan nokta bir mapleme işlemi gerçekleştiriyordak prop tiplerimiz ve prop isimlerimiz aynı olmalı entity ve dto ile.Eğer isimde bir faklılık söz konuusuysa .CreateMap yaparken sonda .Formember ve sonra .MapForm metotunu kullanarak bu farklı olan propların maplame işleminide yapabiliyoruz. Ancak aynı işin ters versiyonunun çalışması için sona .ReverseMap metotunuda ekleriz. AutoMapper katmanlı mimaride Businnes katmanına Mapping adında bir klasör oluştururum. İçerisine ilgili entity ismindeProfile şeklinde class oluşturup bu clası Profile classından kalıtırız. Ancak bunun için businness katmanına AutoMapper paketini kurmamız gerekli. Daha sorna bu classımızın ctor’u içinde CreateMap<Work,WorkListDto> ().ReverseMap(); metotunu çağırırz. Bu sayede sen Work entity’sini WorkListDto ya çevirebil ve bunun tersini de yapabil demiş oluruz. Daha sonra Startupextension’mun içerisine gidiyorum. MapperConfiguration classımdan nesne üretiyorum parametre içerisine bir opt adında delege geçiyorum. Opt.AddProfile metotuna new diyerek profile classımın ismimi veriyorum. Daha Sonra CreateMapper metotı ile bir mapper ele alıyorum. Mapper’ı dependency inejction ile ele almak için add.singlaton kullanıyorum. Daha sonra bunu service classımda dependency injection ile kullanabiliyorum. Serice içimde işlemlerini artık .Map<çevirmekistediğimbilgi> (hangibilgidençevireceğim) şeklinde basit bir kullanım yapabiliyorum.
* Validation kurallarımız için FluenValidation paketini kulalnırız.Business katmanına kurarız.Business katmanına ValidationRules adında klasör açarız. İçerisindeki classlarımızı AbstractValidator classından kalıtırız. Daha sonra ctor içerisinde kurallarımızı yazarız. RuleFor metotundan faydalanarız. Bunu projede dependency injection ile kullanmamız lazım bağımlı kalmamak için. Dependency injection yapabilmek için business katmanında FluentValidation.DependencyInjectionExtension paketini kurmamız gerekiyor. Daha sonra Service classımda dependency injection ile validatorları ele alabiliyoruz.

**IDENTİTY**

* **Projemize Identity olayını dahil etmek için nuget packet aracılığıyla Microsoft.AspNetCore.Identity paketini ve Microsoft.AspNetCore.Identity.EntityFramework pkaetini yükleriz. Ve ayrıca projeme SqlServer paketini ve Tools paketini yüklüyorum. EntityFrameWork’u kurmadık çünkü Identity paketinin içinde bu paket var.**
* **Context classımızı bu sefer DbContext yerine IdentityDbContext’ten kalıtıyoruz. IdentityDbContext günün sonunda bir şekilde DbContext sınıfından kalıtılmıştır.**
* **Bir database bağlantısı yapıp migration oluşturduktan sonra Identity paketi sayesinde database’e otomatik olarak gelen bazı tablolar mevcuttur. AppNetUserTokens tablosu token ile ilgili verileri tuttuğumuz tablodur. AppNetUserLogins tablosu ise siz üçüncü parti bir uygulama kullandığınız zaman google microsoft vs bu uygulama ile giriş yapma özelliğini uygulamamıza entegre ettiğimizde bu üçüncü parti uygulamalardan gelen dataların tutulduğu tablodur. AspNetUserClaims tablosu ise AspNetUsers tablosundaki bilgilerden haricen ek bilgileri tuttuğumuz tablodur. AspNetRoles tablosu ise role bilgilerini tuttuğumuz tablodur. AspNetRoleClaims tablosu ise AspNetRoles tablosunda tutulan bilgiler haricinde ek başka bilgiler tutmak için kullandığımız tablodur. Bir rolün birden fazla user’ı olabilir. Bir user’ın birden fazla Rolü olabilir AspNetUsers ile AspNetRoles arasında çoka çok ilişki vardır.**
* **AspNetUsers tablosunda UserName ve Email alanı indekslenmiştir. Bu sayede biz bir user araması yaptığımızda sonuç bu indeks üzerinden geliyor gelme işlemi fazla uzun sürmesin diye. Normalize edilerek indekslenmiş Normalizeden kasıt girilen username’i komple büyük harfe çeviriyor küçük harfle girilse bile. İşte bu yüzden NormalizedUserName ve NormalizedEmail alanları var. PasswordHash alanı ise kullanıcın şifresi direkt olarak db de tutulmaz güvenlik sebebiyle. ConcurrencyStamp kısmı ise bu bilgi daha önce güncellenmiş şeklinde hata fırlatmak için var. TwoFactorEnabled ise 3. Parti uygulamalar ile login olunacağı zaman kullanılıyor. Lockout kısımları ise atıyorum kullanıcı 5 kere şifrsini yanlış girerse hesabı kilitlensin olayları için var. Lockoutend kilitlemenin sonlanma tarihidir. LockoutEnabled true ise kilitlenme aktiftir. AccesFailedCount ise hatalı giriş sayısıdır.**
* ****
* **Burada UserManager classı sayesinde bir user oluşturabiliyoruz içerisindeki create metotunu kullanarak. RoleManager aracılığıyla bir role oluşturabiliyoruz. SignInManager ile ilgili kullanıcının giriş çıkış işlemlerini gerçekleştirebiliyoruz. Authentication giriş yetkisidir bu adam giriş yetkisine sahip mi değil mi ? eğer ilgili kişi öğrenci veya öğretmen değilse okula giriş yapamaz. Authorization ise ilgili öğrencinin nerelere girmeye hakkı var atıyorum öğretmenler odasına girmeye hakkı yok ama sınıfa girmeye hakkı var yetkiyi belirlediğimiz kısım burasıdır. Claims ise kişiye özgü bilgileri tutmak için kullanabiliriz mesela en sevdiğiniz renk, gizli sorunuz gibi. Identity mimarisi bu şekildedir.**
* **Biz bu db’ye varsayılan olarak gelen tablolarımızı özelleştirebiliyoruz veya fazladan tablo eklemesi çıkartması vs yapabiliyoruz. Entity dosyamın içerisinde AppUser şeklinde bir entity ekliyorum. Bu entity’i IdentityUser<int> classından kalıtıyorum ve <> içine ıd alanının türünü söylüyorum int diye yani bu Tkey oluyor. Daha sonra entity dosyama rol bilgilerimi tutmak için AppRole entity’si oluşturuyorum ve bunu IdentityRole den kalıtıyorum <int> içerisine yazarak senin primary keyin integer olacak diyorum. Daha sonra en son contextime gidiyorum IdentityDbContext<AppUser, AppRole, int> ekleyerek senin Tuserin artık AppUser, Trole ün artık AppRole ve Tkey in yani dğer tablolaların primary keyi string değil artık int demiş oluyorum. Bunu yaptıktan sonra AppUser yada AppRole i ben dbset olarak belirtmek zorunda değilim. Arka tarafta IdentityDbContext zaten bunlarla beraber ayağa kalkacak. Daha sonra migartion oluşturup değişikliği gönderelim. Db ye baktığımızda User ve Role tablosuna ek olarak eklediğimiz kolon bilgilerinin eklendiğini ve primary key tiplerinin int olduğunu görebiliriz.**
* **Projemize Identity’i eklemek için startup dosyası içinde ConfigureServices metotunda aşağıdaki kodu yazıyoruz.**
* services.AddIdentity<AppUser,AppRole>().AddEntityFrameworkStores<UdemyContext>();
* **Kullanıcı oluştururken Identity paketinden varsayılan olarak gelen bazı validation kuralları var bunları değiştirmek istersek startup dosyası içinde**

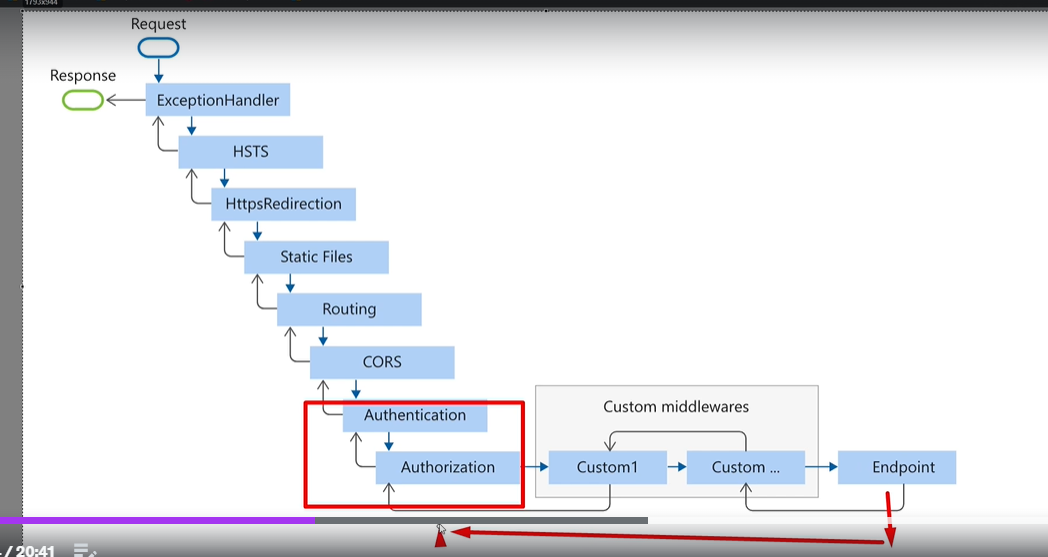
opt.Password.RequireDigit = false;

opt.Password.RequiredLength = 1;

opt.Password.RequireLowercase = false;

opt.Password.RequireUppercase = false;

opt.Password.RequireNonAlphanumeric = false;

* Default olarak gelen tokeni kullanabilmemiz için ilgili controller’ın üstüne [AutoValidateAntiforgeryToken] bunu dedikten sonra artık bu sunucunun üretmediği x bir noktadan belirli istekler yapılamaz hale gelecektir daha güvenli olacaktır proje.
* 
* Yetkilendirme için bunları startup dosyasına ekliyor olmam gerekiyor.
* User özelliğni kullanabilmek için [Authorize] gereklidir. Layoutta bunu [Authorize] olmadan kullanabiliyoruz çünkü ilgili metotta zaten [Authorize] var ordan alıyor o bilgiyi. Bunun yerine user bilgisini çekerek \_userManager üzerinden .GetRolesAsync metotunu kullanabiliriz.

//kullanıcı kaydı için userManagerdan faydalanmak

private readonly UserManager<AppUser> \_userManager;

//kullanıcı giriş için SignInManagerdan faydalanmak

private readonly SignInManager<AppUser> \_signInManager;

//rol ekleyebilmek için

private readonly RoleManager<AppRole> \_roleManager;

//cookie ayarlarını değiştirmek

services.ConfigureApplicationCookie(opt =>

{

//true yazdığımızda diğer şahıslar tarafından javascript aracılığıyla çekilemiyor cookie.

opt.Cookie.HttpOnly = true;

//Sadece ilgili domainde kullanabilir

opt.Cookie.SameSite = SameSiteMode.Strict;

//always dersek sadece https'de çalışır SameAsRequest dersek http ile gelirse httpde, https ile gelirse https'de çalışır

opt.Cookie.SecurePolicy = CookieSecurePolicy.SameAsRequest;

//Cookie'nin name'ini değiştirmek

opt.Cookie.Name = "UdemyCookie";

//cookkie'nin ayakta kalma süresini ayarlamak

opt.ExpireTimeSpan = TimeSpan.FromDays(25);

//bir kullanıcının yetkisi olmadan bir alana giriş yapmaya çalıştığında onu nereye göndereceğimizi ayarlayalım

opt.LoginPath = new PathString("/Home/SignIn");

* });

**CUSTOM COOKİE**

* Identity yapısını tablolarını vs kendimiz oluşturmak istersek bu mimariyi,
* Authentication için https://learn.microsoft.com/en-us/aspnet/core/security/authentication/cookie?view=aspnetcore-7.0

**Uygulama geliştirme süreci**

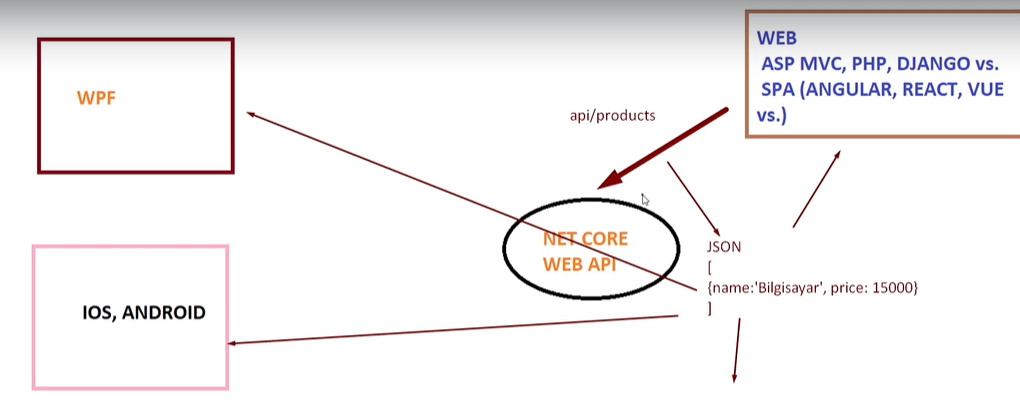
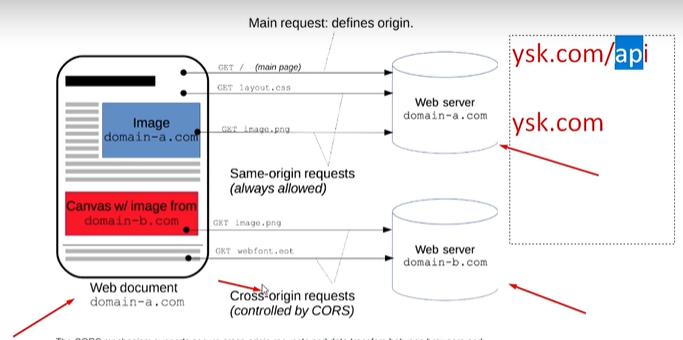
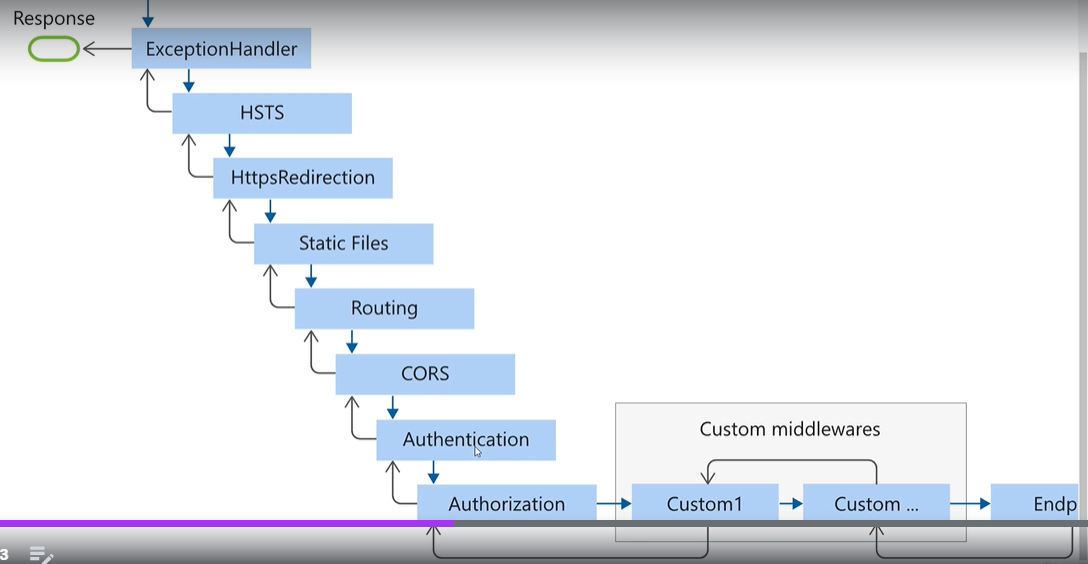
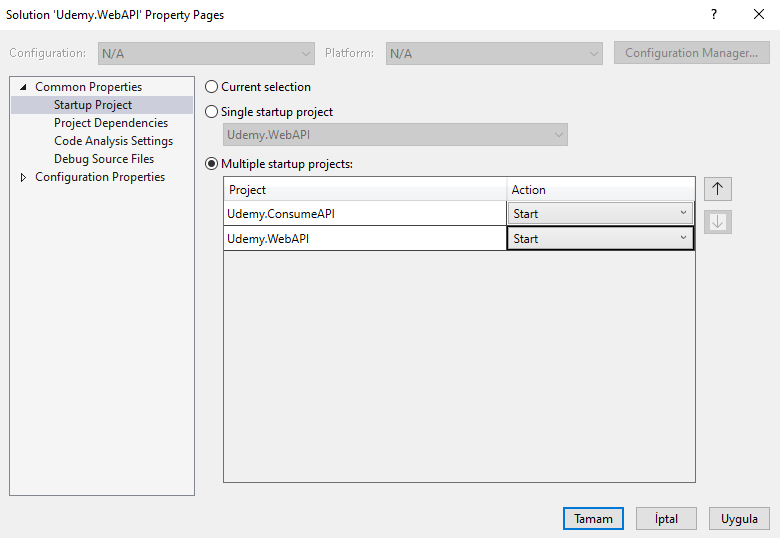
* **TEMEL AYARLAMALARDAN SONRA REPOSİTORY ENTİTY STARTUP VS VS.**
* Önce Dto’yu oluştur, Dto katmanında
* Dto’nun Validatorlarını oluştur, ValidationRules’ta
* Validatorları DependencyExtension’a kaydet.
* Daha sonra bu arkadaşın MapProfile’ını oluştur, Mappings’te
* Bu profile’ı git DependencyExtension’a kaydet.
* Sonra gel ilgili Service’ini ve İnterface’ini yaz.
* Sonra DependencyExtension’a servicesnin scopunu kaydet.
* Sonra gel UI’da kullan.

**\*\*\*\*\***Lookup tablolar kolay kolay ilgili verisinin değişmediği tablolardır.(Gender),(Statu) tabloları örnek olarak verilebilir. Bunların bilgilerini Enum’a kolaylıkla çevirebiliriz.

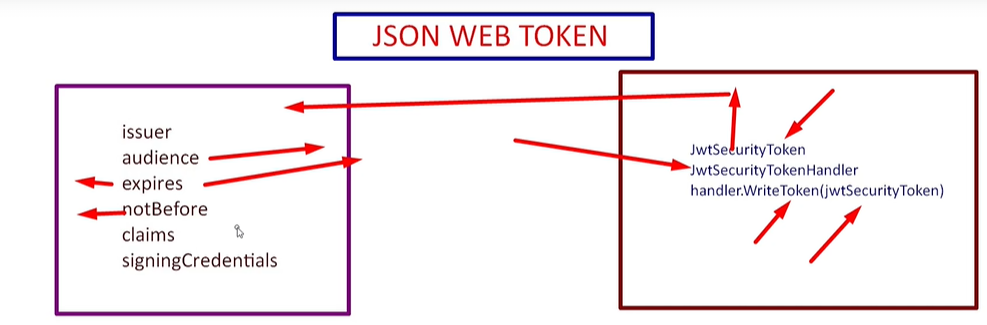
\*\*\*\*\*\*Enumları 2 noktada kullanabiliriz.

* 1-Db’ye sürekli olarak gidip gelmemek için. Özellikle lookup table’larda bu durum var.
* 2- Clean Code = if(GenderId == 1) yanlış bir kullanımdır. 1 neye karşılık geliyor bilemeyiz. Bu durumda if(GenderId == (int)GenderIdType.Erkek) şeklinde bir enum kullanımı yapmak çok daha sağlıklıdır.

**WEB API**

* API: Application Programming Interface – Uygulama Geliştirme Arayüzü.
* Web API’ ler istemci – sunucu mimarisinde web tabanlı sistemler arasında veri ve işlevsellik sağlamaktadır. Bu API’ ler genellikle http ya da HTTPS protokolünü kullanarak web uygulamalarından gelen istekleri ve cevapları iletir.
* Benim bir uygulamam var diyelim. Uygulamamın admin kısmını .netcore ile kullanıcı kısmını ise angular ile geliştireceğim diyelim. Aynı zamanda bu uygulamanın mobil ayağı’da olacak, masaüstü ayağı’da olacak. İşte bu noktada her bir parçaya tek tek backend yazmak istemiyorum. Bütün olay burda başlıyor. Bunların hepsinin anlaşabileceği hepsinin kullanabileceği bir altyapı, hizmetçi, bunlara hizmet edecek bir yapı hazırlamak istiyorum. Tam olarak WEB API dediğimiz şey bu. WEB API herkesin anlayabileceği datayı geriye dönüyor ve biz bu datayı tüketiyoruz. Herkesin ortak anlayabileceği data dili JSON ve XML oluyor. Öetle WEB API diğer uygulamalara bir şekilde hizmet veren bir başka uygulamadır.
* ****
* WEB API ile birlikte gelen REST kavramı vardır;
* REST kavramının amacı veriye anlamlı bir şekilde yaklaşmaktır.
* GET : Veriyi çekmek için kullanıyoruz.
* POST : Bir veriyi eklemek için kullanıyoruz.
* PUT: Bir veriyi güncellemek için kullanıyoruz.
* DELETE: Bir veriyi silmek için kullanıyoruz.
* Ben veriyi başarılı bir şekilde oluşturduysam geriye CREATED dönerim yani 201.
* Ben veriyi başırılı bir şekilde güncellediysem veya sildiysem NO CONTENT dönerim yani 204.
* Ben veriyi başarılı bir şekilde listelediysem birini veya birden fazlasını listelediysem geriye 200 dönerim.
* Benim bir verim validation hatasına takılırsa geriye 400 dönücem yani BAD REQUEST
* Eğer ben bir veriyi eklerken yüklerken veya getirirken herhangi bir sıkıntıyla karşı karşıya kalırsam PROBLEM dönücem yani 500 dönücem.
* Eğer ben bir veriyi güncellerken veya bir tanesini getirirken o veriyi bulamazsam NOT FOUND dönücem.
* API Yapısında controller’ın API olduğunu belirten şey üstündeki [ApiController] attrubute’dır. [Route("api/[controller]")] attribute’u ise routun bu şekilde çalışacaını belirtiyor. Ve bu Contoller sınıfı ControllerBase’den kalıtılmıştır. Bizim standar kullandığımız controller’larda controller’ımızı Controller sınıfından kalıtıyorduk bu Controller sınıfı aslında ControllerBase sınıfından kalıtırlmıştır. O yüzden aslında yapacağımız şeyler normal bir .netcore mvc projesiyle aynıdır arada çok ufak değişiklikler vardır. Sadece JsonWebToken’Da bir efor harcayacağız.
* Startup tarafında ConfigureServices metotunda biz normalde services.ControllerWithViews eklerdik ama Apı uygulamasında services.AddControllers eklenmiş sadece. Çünkü bizim bir View ile bir bağlantımız yok. Ve uygulama geliştirme aşamasındayken Swagger apisini kullan diye bir ayar geçilmiş otomatik olarak. Bizim controllerların üstünde [Route("api/[controller]")] diyerek route ayarını orda geçme sebebimiz startup’ta endpoints. Controller’larla maple demiş route’a. Ancak bu controllerda bulunan Route şemasında herhangi bir action name bulunmuyor. Sadece Controller’ın name’ini route şemasına alıyor dikkat edelim. Bu sebeple aslında bizim Action Metotlara verdiğimiz isimlerin bir önemi bulunmuyor. Buradaki action metotlarımız aslında bizim birer endpointlerimizdir WEB API bu endpointleri bize sunar ve biz bu endpointleri tüketiriz.
* 
* Diyelim ki 2 tane web uygulamam var a ve b şeklinde bunlardan a olanı domain a api servisine bağlanabilir eğer domain b api servisine bağlanmak isterse Croos Origin Request yani CORS’a düşer bununla ilgili bir hata söz konusu olacak. Domain b Api sadece sadece b uygulamasına açıktır. Biz istriyoruz ki başka bir web servisi ya da başka bir fornetend uygulama veya mobil uygulamada bizim api’ımızı tüketebilsin işte tam olarak bu noktoda bizim CORS configurasyonlarımızı yapmamız gerekiyor. Bunun için sturtap dosyasında aşağıdaki şema gib bir yol izlememiz gerekiyor.
* 
* Startup tarafında ilgili ayarlamaları yaptıktan sonra ilgili controller’ın üstünde [EnableCors] kullanarak ilgili Cors configurationumuzu yapmış ve kullanmış oluyoruz.
* Api kullanımlarında bazen birden fazla kez veri çekerken vs 500 hatasına düşeriz yani bizim json objemiz içerisindeki başka bir json ile sürekli bir cycle oluşturuyor geriye dönemiyor şeklinde hata alırız. Bu noktoda Microsoft.AspNetCore.Mvc.NewtonSoft paketinden faydalanırız daha sonra startup içinde ilgili ignore configure’sini yaparız ve bu durumdan kurtuluruz.
* İlgili Api uygulamamızı oluşturduktan sonra bu uygulamayı tüketebilecek ayrı bir uygulamamız olması gerekiyor.
* Api uygulamamızı web uygulamamıza bağlamak için solution’a sağ tıklayıp özellikler diyip startup kısmında aşağıdaki gerekli ayarlamayı yapmalıyız. Bu sayede biz uygulamayı çalıştırdığımız zaman aynı anda hem web projesi hemde api projesi ayağa kalkacaktır.
* 
* Daha sonra API’a 2 şekilde yaklaşabiliyoruz ilki TypedClient ve ikincisi HttpClientFactory doğrudan kullanma olayı söz konusu. Biz HttpClientFactory doğrudan kullanmayı yapalım. Bunu web uygulamamın startup dosyası içerisinde ConfigureServices metotunda services.AddHttpClient() demek. Daha sonra ilgili controller içerisinde httpClientFactory’i dependency injectin olarak alabiliyorum.

**JSON WEB TOKEN**

* 2 Tip Authontacation yöntemi vardır. Geneleneksel olarak gördüğümüz tip Cookie Based Authontacation’dur. Birde modern olarak gördüğümüz tip ise Jason Web Token yani Token bazlı bir giriş sisteminden bahsediliyor.
* Normal şartlarda Cookie’nin mantığı ilgili kullanıcımız kullanıcıname ve password’u ile giriş yapıyordu. Bizde kontrol ediyorduk ilgili kullanıcı sistemde var mı diye eğer sistemde varsa bir Cookie oluşturup Cookide ilgili kullanıcın bilgisini tutuyorduk. JsonWebToken’de ise diyelim ki user’ın bilgisini check eden bir endpointimiz var ve bu endpointine yine biz bir username ve password ile gidiyoruz. Gittiğimiz zaman bu sefer bize ilgili endpoint bize bir token bir bilet veriyor ve biz bu biletle aslında senin bilgilerin doğruysa al sen bu bileti ilgili sinema saloluna filme girebilirsin veya filmlere girebilirsin şeklinde bir bilet veriyoruz. Buradaki doğrulama şemamız tamamiyle Token bazlı oluyor.Artık O kişide o token varsa o tokenın bitiş süresine kadar yetkili oluğu kısımlarda tüketebiliyor. Burdaki mantık ben username ve password ile gidiyorum enpointe ve ilgili endpoint bana bir token üretiyor ve token çıktısını bana veriyor. Token çıkıtsı 3 parçadan oluşuyor birincisi Header ikincisi PAYLOAD üçüncüsü SIGNATURE. Bu tokenın bağlı olduğu standartlar var bunada işte RFC 7519 deniyor.
* Header kısmında bulunan alg tokenin şifrelendiği algoritma türüdür H256 şifreleme algoritmasıdır RFC tarafında tavsiye edilen bir şifrelemedir. Typ kısmı ise tokenın tipi JasonWebToken.
* Payload kısmı ise user’ın datası veya rol bilgilerini tuttuğum kısımdır.
* Signature kısmı ise imzadır. HMACHA256 ile şifreleniyor ve şifrelenirken encode ediliyor. Encode edildikten sonra şifreliyor ve diyor ki senin secret codun bu keye göre şifreliyor.
* .NET CORE tarafında JwtSecurityHandler nesnesi aracılığıyla kolaylıkla bir token üretebiliyoruz. Bu nesnenin içerisindeki WriteToken metotuna bir bir başka nesne aracılığıyla jwtSecurityToken nesnesi ile belirtmiş olduğumuz tokeni içerisine parametre olarak verip üretebiliyoruz. Ve bu nesnenin içerisinde bir takım parametreler var.
* İssuer: Bu tokenı oluşturan
* Audience : Bu tokenı kullanacak olanlar
* Expires: Token’ın geçerlilik süresi
* NotBefore: Ne zamandan itibaren
* Claims: İlgili user’ın veya başka dataları tutabiliyoruz yani istediğimiz dataları Claims’te tutabiliyoruz Claimsler ile bu bilgileri taşıyabiliyoruz.
* SigningCredentials: İlgili token’ın imzasını tuttuğumuz kısımdır.
* 
* JasonWebTokeni kullanabilmek için api projesinde startup tarafında configure etmemiz gerekiyor. Bunu yapmak için manage nuget packettan Microsoft.AspNetCore.Authentication.JwtBearer paketini api projeme indirmem gerekiyor.
* JsonWebToken https ile beraber çalışır uygulamamın içinde kullanırsam.
* 3 Tip şifreleme türü vardır. İlki Simetrik Diğeri Asimetrik diğeri ise Hash tipi şifrelemeler vardır.
* Hash tipinde siz bir datayı bir kere şifrelediğiniz zaman o datayı kaybedersiniz tekrar çözmeniz mümkün değildir.
* Simetrik şifrelemede bir key vasıtasıyla, şifreleyende şifreyi çözedende aynı keye sahip olması gerekiyor bu bir key aracığıyla siz simetrik şifrelemeyi çözebiliyorsunuz.
* Asimetrikte ise işler biraz daha kompleks şifeyi çözende şifreleyende farklı keyler kullanıyor.
* Biz JsonWebToken’ simetrik olarak şifreleyip veriyoruz.
* Postman aracılığıyla endpointlere istekler atabiliyoruz. Ordaki dönüşleri görebiliyoruz. Swagger’Daki arayüzü kullanmak yerine daha çok Postman kullanılıyor. Swagger’ı daha çok bir dökümantasyon gibi düşünebiliriz. ,
* Swagger’da 401 hatası sen giriş yapmadın anlamına gelmektedir. Request Url adresi kopyalanıp Postman’e yapıştırılır. Post metotu olan Login metotuna postman ile istek yapıldığında bize bir token üretir. Daha sonra 401 hatası aldığım yerde bu tokenı headers kısmında key kısmına Authorization yazdıtkan sonra Value kısmına Bearer yazıp bir boşluk bırakıp kopyaladığım token bilgisini yapıştırıyorum. Ve artık Get listeleme isteğim giriş ypamadın yani 401 hatası vermiyor ilgili listeleme isteğini gerçekleştiriyor.
* 403 hatası ise yetkin yok hatasıdır.

**COMMAND QUERY RESPONSİBLİTY SEGREGATİON (CQRS)**

* Bir db miz var diyelim. Ve bir service’miz var diyelim ve tabi bu service’e UI’dan bir istek geldiğini düşünün. Bu istek eğer bir veriyi getirme isteğiyse bunn adı QUERY eğer bu istek bir güncelleme , ekleme veya silme isteğiyse bunun adı COMMAND’dır. Ben hem select yapıp hemde ekleme yapıyorsam bunada COMMAND deriz bu bir Transactional işlemdir yani birliktedir.
* CQRS’in yapısında bir COMMAND BİR COMMANDRESULT yani diyelim ki bir update işlemi ve bu update işlemi geriye bir sonuç dönüyorsa geriye döneceğimiz result’tır ve bu commandı işleyen COMMANDHANDLER vardır. Bu yapı aynı şekilde QUERY içinde geçerlidir. COMMAND VE QUERY bizim için aslında request anlamına geliyor. Mesela bir query oluşturdum query ile id’si 2 olan ürünü çekmek istiyorum işte o query içerisinde id’yi parametre olarak alıcam veya bütün ürünleri çekmek istiyorum o zaman herhangi bir parametre geçmeden o query ile onu çekicem benim bu business işini yaptığım yer HANDLER’dır HANDLER’DA gerekli işi yapıcam ve geriye QUERYRESULT dönücem
* N katmanlı mimaride diyelim ki 2 tür ekleme işlemi yapmak istiyorum birinde 2 alanlı ekleme işlemi yapmak istiyorum diğerinde 4 alanlı bir ekleme işlemi yapmak istiyorum bu durumda naparım ilgili service içinde 2 ayrı metot yazarım. Bu günün sonunda Domain spesifik iş yapmamızı zorlaştırıyor çok fazla dto ihtiyacı çok fazla viewModel ihtiyacı doğruyor. İşte bu noktada CQRS’in hizmet ettiği şey Object Mapping.
* Bu yapıda controller’da bir sürü dependency inejction değişkenlerimiz oluyor bir karmaşıklık oluyor işte bundan kurtulmak için Mediator Pattern’a ihtiyaç duyuyoruz. Bu yapıya bir request geliyor ve yapı requestin isteğine göre diyelim ki request create yapmak istesin otomatik olarak createHandler’a yöneldirme yapıyor. Yani bana bir updatestudentcommend gelirse ben giderim updatestudemtcommendHandler’ı çalıştırırım diyor. Buna arabulucu anlamındaki MEDİATOR PATTERN diyoruz. Bunu projeye dahil etmek için nugetten MediatR.Extensions.Microsoft.DependencyInjection paketini indiriyorum. Daha sonra bizim bunu startuptaki services’lara eklememiz gerekiyor. Daha sonra ilgili Query ve Commandlarımı Irequest sınıfından kalıtıyorum ve <> içinde result’ını belirtiyorum ve daha sorna ilgili handler sınıfınıda IrequestHandler sınıfından kalıtıp <> içerisinde query veya commandımı ve bir de resultını belirtiyorum. Daha sonra controllerımda dependency injection olarak sadece Imediator’u ekliyorun. Burda örnek alacağını startup tarafında services olarak yaptığımız configuration’dan biliyor.

**.NETCORE 6.0**

Artık .NetCore 6.0 ile birlikte startup classımız yok. Bu startup classı program.cs classının içerisine taşınmıştır. Normalde direkt Services üzerinden gidiyorduk dependency injection ile ama artık builder.Services şeklinde gidiyoruz. Bunuda CreateBuilder metotu sayesinde ele alıyoruz. app.Run() dediğimiz zamanda uygulama ayağa kalkıyor. Aslında değişen bir şey yok sadece builder.Services kullanıcaz arıtk. Artık proje içinde string kullanırken nullable olarak işaretlemeliyiz ? koyarak. Artık entitylerimde ilişki belirttikten sonra ctor kullanrak sen bunu null olarak işaretleme en kötü ihtimalle boş olarak örnekle diyoruz. AppRole'e diyeceğim ki sen null olamazsın yani en kötü boş bir nesne örneği taşırsın bu class örneklendiği zaman.

//dbsetleri oluştururken .netcore6.0 ile null değerlere karşı önlemler arttı biraz daha korunmamız gerekiyor.

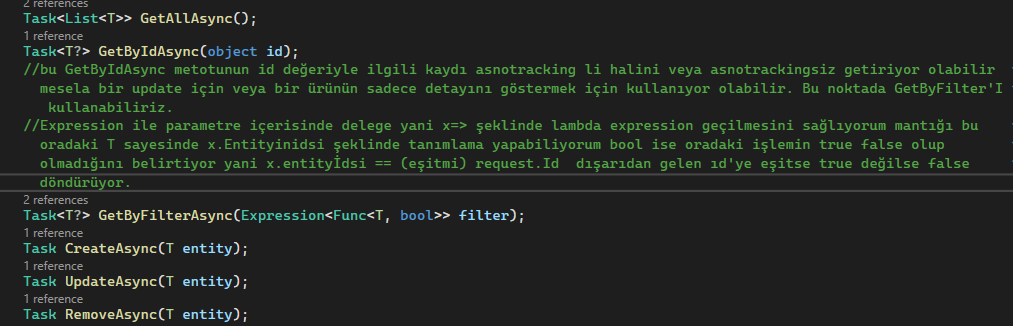
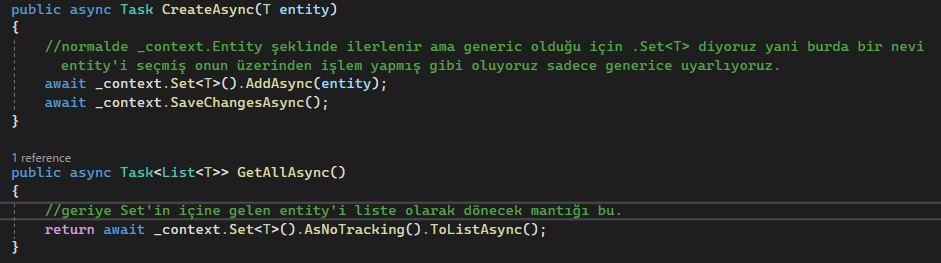
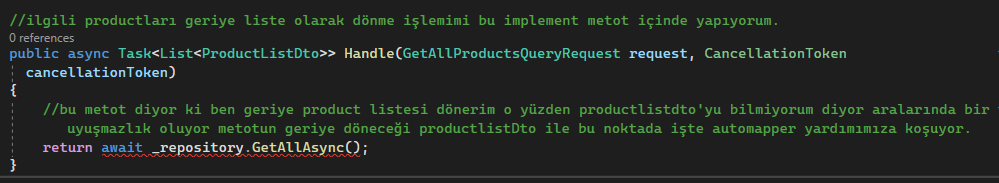
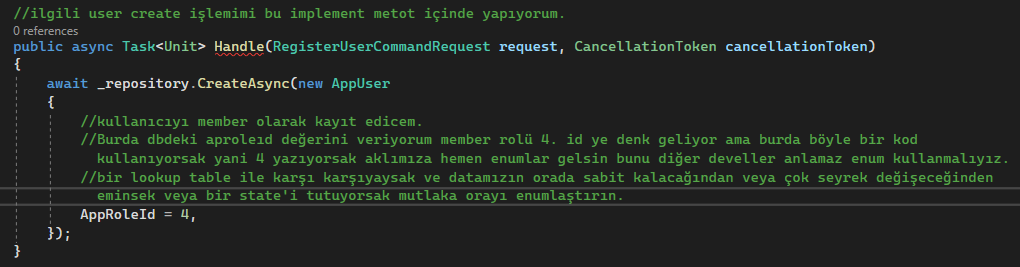
//bunların setini açmak çok doğru bir yaklaşım değil sonuç olarak ben Product'ı gelip setlemicem ben bu veritabanındaki verileri getiricem bir şekilde yani setter'ının açık olması çok mantıklı değil DbSet<Product>? yazarak bu önlemden kurtulabiliriz veya aşağıdaki gibi bir işlem yapıp setter'ı kapatıp null önlemlerden kurtulabiliriz. Kullanım aşağıdadır.

public DbSet<Product> Products => this.Set<Product>();

public DbSet<Category> Categories => this.Set<Category>();

public DbSet<AppUser> AppUsers => this.Set<AppUser>();

public DbSet<AppRole> AppRoles => this.Set<AppRole>();

* GetByFilter NEDEN VAR SORUSU \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
* Ayrıca burada T yi ? olark nullable olarak işaretleme sebebimizde .netcore 6.0 ile alakalı metot içinde singleordefault kullandığımız için default yani null gelebilri diyor.
* Ayrıca burda Expression neden kullanılıyor sorusunun cevabıda var.
* 
* Set<> konusunu anlamadıysan aşağıda:
* 
* AutoMapper’in lazım olduğu bir noktadan örnek sana:
* 
* Enum ve LookUp Table kullanımı ile ilgili bir örnek sana;
* 
* UTC İLU ÇALIŞMAK: Diyelim ki bir giriş yapma özellikli uygulmama var ve ben buna bir loglama yaptım dedim ki eğer kullanıcı 5 kere yanlış şifre girerse hesabı DateTime.Now yani localin şuanki saatine göre veya türkiyenin saatine göre şu kadar saat bloklansın. Bu kötü niyetli kullanıcı sürekli şifre denemesi yapıp hesabı bu zaman dilimine göre bloklandıktan sonra bölgesini değiştirerek bu bloklama süresinde değişim yapabilir. Bu durumda Daha global bir time’ı ele almalıyız. İşte tam olarak bu noktoda UTC denilen bir kavram var bu tüm dünyanın kabul ettiği bir saat dilimidir ve herkes için ortaktır. Dolayısıyla DateTime.Now işlemlerimi DateTime.UtcNow ile yapmam en sağlıklısıdır.
* Backoffice tarafında yani api tarafında yaptığımız token işlemlerini forentend uygulamasına taşımak için frontend tatafında Microfost.AspNetCore.Authentication.JwtBearer paketine ihtiyacımız var ve handler.ReadJwtToken kullanıcaz api tarafıdna ise Write kullanmıştık.