ASP.NET CORE-MVC

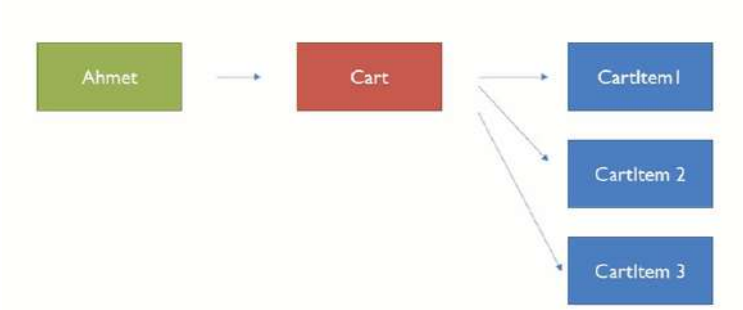
İçindekiler

ALIŞVERİŞ SEPETİ (CARD) İŞLEMLERİ

Alışveriş Sepeti (Card) için Veritabanı Yapısının Düzenlenmesi

Öncelikle bazı kavramları anlamaya çalışalım. Kullanıcı sepetine bir ürün eklediği zaman bu ürünü nerede saklayacağız. Bunun için bilgileri Session'da yada Database'de saklayabiliriz. Yada iki yapıyı aynı anda kullanabiliriz. Session yapısı serverda geçici olarak bilgilerin hafızada tutulduğu bir yapıdır. Eğer kullanıcı giriş yapmadıysa yada kullanıcı işlemleri yoksa, sepete eklediği ürünler geçici olarak Session üzerinde saklanabilir. Şayet kullanıcı girişi yapmadan sessiona eklediği ürünleri almaya karar verirse önce üye girişinin tamamlanması ardından sessionda tuttuğu ürünlerin veritabanına kaydedilip, satın alma sürecine devam edilmesi gerekir. Veritabanına kaydedilen ürünler daha sonradan yada farklı bilgisayarlardan görülebilir fakat sessionda tutulan bilgiler başka bir bilgisayardan görülemez ve belli bir süre sonrada server hafızasından silinir. Burada iki yapıyı beraber kullanmamız en mantıklı olandır. Bu şekilde bir alt yapı hazırlayalım.

Sepet dediğimiz olay bir kullanıcının bağlı olduğu bir Cart nesnesidir. Bu nesneye bağlı olan elemanlarda CartItem lerdir. Yani her bir ürün bir CartItem ı temsil eder.



Tablo yapısı ise aşağıdaki bir sistem olabilir. Burada Ali Su isimli kullanıcının Kartında (sepetinde) 2 ve 4 numaralı ürünlerin olduğunu ve 2 numaralı üründen 1 adet, 4 numaralı üründen ise 2 adet olduğunu görüyoruz. Burada kullanıcı satın alıma geçtiğinde CartItems tablosundaki bilgileri Sipariş (order) tablosuna aktarırız.

Users

id	name
3	Ali Su
7	Oya Ay

Cart

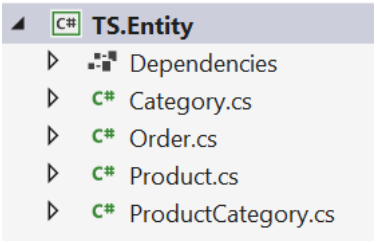
id	userId
3	3
7	7

CartItems

id	CartId	ProductId	Quantity
1	3	2	1
2	3	4	2
3	7	2	1

Burada şöyle bir konu vardır. Kullanıcı ürünü sepetine ekledi. Sepette gösterilen ürüne ait bilgiler ProductId üzerinden products tablosundan gelecektir. Eğer products tablosunda ürünün fiyatı yada herhangi bir bilgisi değişmiş ise kullanıcıda bilgileri o şekilde güncel olarak görecektir. Fakat kullanıcı satın alıma geçerse (Order) o zaman ürünün fiyatının bir daha değişmemesi gerekir. Dolayısı ile ürünün fiyatı order tablosunda satın alacağı fiyat olarak kaydedilmesi gerekir. Tabii burada satın alma sürecinin de ne kadar bir süre aktif kalacağı da ayrı bir konudur.

Şimdi Card ve CartItems entitylerini (tablo nesneleri) projemize ekleyelim. Entity sınıf yapılarımızı TS.Entity katmanı içinde oluşturuyorduk.



Navigation Property

Herhangi bir CardId bilgisini tablo içinde kullanıyoruz ama doğrudan kart bilgilerine ulaşmak içinde tablo içinde Card entitisinin bilgilerine referans yapıyoruz. Yani şu Id li Card bilgileri yerine direk Card entity'sinin adını yazıyoruz. Benzer şekilde ProductId bilgisini tutarken yanında tablo içinde Product entity'sinin bilgilerini de tutuyoruz. Bu işleme "Navigation Property" ekleme diyoruz. (Bu konuda şu makale okunabilir: https://emrecanayar.wordpress.com/2020/10/09/navigation-properties/) . Navigation property'leri iki şekilde düşünebiliriz. Bire-bir ve Bire-çok ilişkisi. Örneğin bir ProductId nin karşılığı olarak bir tane Product entity bulunur. Fakat bir kullanıcının birden çok rolü olabilir. Dolayısı ile bu rollerini de bir liste şeklinde tutabiliriz. Her ikisi içinde tanımlamalar aşağıdaki şekilde yapılabilir.

```
public int ArticleId { get; set; } //Bu yorum hangi makaleye eklenmiş.
0 references
public Article Article { get; set; } //Article(Makale) özelliklerine veya bilgilerine ulaşmak için.
```

```
public int RoleId { get; set; } //Bu kullanıcının rolünü belirtir.
0 references
public Role Role { get; set; } // Kullanıcının rolünün detaylarını almak için. İsim vb.
0 references
public ICollection<Role> Roles { get; set; } //Bir kullanıcının birden fazla rolü olabilir.
0 references
```

Şimdi bu bilgiler ışığında Id özelliklerini verirken yanında entity özelliklerini de tanımlayarak sınıf yapılarımızı oluşturalım.

TS.Entity> **Card.cs**

```
using System.Collections.Generic;

namespace TS.Entity
{
    public class Card
    {
        public int Id { get; set; }
        public int UserId { get; set; }
        public List<CardItem> CardItems { get; set; }
    }
}
```

TS.Entity> **CardItem.cs**

```
namespace TS.Entity
{
    public class CardItem
    {
        public int Id { get; set; }
        public int ProductId { get; set; }
        public Product Product { get; set; }
        public int CardId { get; set; }
        public Card Card { get; set; }
        public int Quantity { get; set; }
    }
}
```

Şu anda sınıf yapılarımız hazır. Oluşturduğumuz bu iki tane entity yi (Card, CardItem) TS.Data>Concrete>TSContext.cs içerisine bunları eklememiz gerekir.

```
public class TSContext : DbContext
{
    public DbSet<Card> Cards { get; set; }
    public DbSet<CardItem> CardItems { get; set; }
```

Veritabanındaki yapılar bu bilgiler üzerinden oluşacak. Veritabanındaki yapıyı oluşturabilmek için önce Migration ları oluşturmamız gerekir. Şimdi onu yapalım. Komut satırından TS.Data klasörü içine geçip orada aşağıdaki komutu çalıştıralım. Oluşturacağımız migration a "AddingCardEntities" adını verelim. Ardından startup proje ve hangi context kullanacağımızı belirtelim.

dotnet ef migrations add AddingCardEntities --startup-project ../TS.webUI --context TSContext

```
C:\Users\pc1\Desktop\TS.com\TS\TS.Data>dotnet ef migrations add AddingCardEntities --startup-project ../TS.webUI --context TSContext
Build started...
Build succeeded.
Done. To undo this action, use 'ef migrations remove'
```

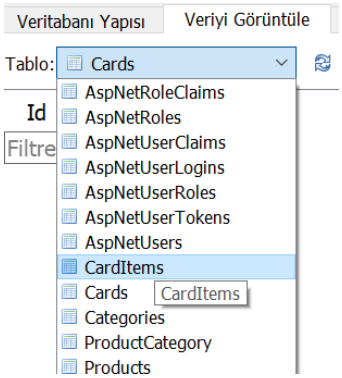
Migration oluştu. Şimdi ona bakalım. Migration açtığımızda içerisinde Card ve CardItem entitilerini görürüz.

Oluşan migrationları çalıştırıp veritabanındaki güncellemeleri yapmış olalım. Bunun için aşağıdaki komutu yazalım. Burada yine startup projeyi ve context i belirtmemiz gerekir.

dotnet ef database update --startup-project ../TS.webUI --context TSContext

```
C:\Users\pc1\Desktop\TS.com\TS\TS.Data>dotnet ef database update --startup-project ../TS.webUI --context TSContext
Build started...
Build succeeded.
Applying migration '20220129142608_AddingCardEntities'.
Done.
```

Veritabanımızdaki tabloları incelediğimizde yeni oluşan Card ve CardItem tablolarını görürüz.



Alışveriş sepeti (Card) Oluşturmak için Kod altyapısının oluşturulması

Bundan önceki konuda Veritabanında sepet işlemleri için gerekli alt yapı oluşturuldu. Alışveriş sepeti bir kullanıcı için ne zaman oluşturulmalıdır? Kullanıcı sonuçta bir düğmeye tıklayarak alışveriş sepetini oluşturmayacaktır. Herhangi bir ürünü almak istediğinde direk sepete ekleyebilmelidir. Dolayısı ile sepet kullanıcı farkında olmadan oluşturulması gereken bir olaydır. Bunun içinde en uygun zaman kullanıcının üyeliği onayladığı anda yani mailine giden linke tıkladığı anda sepet bilgileri oluşturulmalıdır. Eğer kullanıcı mail onayını yapsa bile bir admin onayından vs geçmesi gerekiyorsa o zaman admin onayı verildiğinde oluşturulmalıdır.

Kullanıcının sepetinin olması demek Card tablosu içerisinde kullanıcı Id sine ait bir CardId sinin oluşturulmasıdır. Alışveriş yapmasa bile orada kullanıcıya ait bir CardId oluşmuş olacaktır. Ne zamanki bir alışveriş yaparsa o zaman o ürüne ait bilgilerle CardItem tablosuna bilgiler eklenecektir. Dolayısı ile kullanıcının Card tablosunda bilgilerinin oluşturulması AccountController altında ConfirmEmail() metodu içinde aşağıdaki satırlarda oluşturulmalıdır.

```
var result = await _userManager.ConfirmEmailAsync(user, token);
```

```
        if (result.Succeeded)
        {
                //Card oluşturma bilgileri
```

Şimdi Card işlemleri için Business ve Data katmanında veritabanına bilgileri aktarırken kullanılacak metod yapılarımız için alt yapıları hazırlayalım. Daha önce Entity katmanında Card entitisine ait sınıf yapısını, veritabanı alt yapısını oluştururken hazırlamıştık.

Öncelikle Interface yapılarını hazırlayalım. İki katmanda da (Business, Data) Interface yapısını benzer şekilde oluşturuyorduk. Önce Card başlatma anlamına gelen InitializeCard isminde bir metod oluşturalım. Geri dönüş değeri olmasın. Bizden hangi user için oluşturacağını isteyeceğinden userId girişte verelim.

TS.Business>Abstract>**ICardService.cs**

```
namespace TS.Business.Abstract
{
    public interface ICardService
    {
        public void InitializeCard(int userId);
    }
}
```

Business katmanında oluşturduğumuz IcardService yapımızın aynısının Data katmanındaki adlarını ise IcardRepository ismini veriyoruz. IcardRepository ise IRepository içindeki metod yapılarını kullanacağından ondan türetilmiş olması gerekir. IRepository içindeki metod yapıları ise dışarıdan Generik parametreler almaktaydı. Yani hangi entitiy gönderilirse ona göre işlem yapıyordu. Bu bilgiler ışığında tanımlamasını aşağıdaki şekilde yapmalıyız. IRepository içinde tüm entitiler için kullanabileceğimiz veritabanı işlemlerine yönelik metodlarımız var idi. Eğer eksradan Card için yeni bir metoda ihtiyacımız olursa onu da ICardRepository içine ekleyebiliriz. Şu aşamada ekstra bir metod kullanmayalım. Boş bırakalım.

TS.Data>Abstract>**ICardRepository.cs**

```
```

Interface metodlarımızın dolu versiyonları olan Concrete sınıfını oluşturalım. Business katmanında sınıf adımızı CardManager adını verelim. Bu sınıf IcardService in dolu versiyonu olacağından metod yapısını ordan alacaktır. Daha önce çok kez yaptığımız "Implement Interface" (arabirimi uygula) yöntemiyle metodlarımızı oluşturalım. İçlerinide aşağıdaki şekilde dolduralım.

CardManager business katmanında concrete versiyon idi. Burası veritabanı ile iletişim sağlarken Data katmanındaki karşılığı olan concrete versiyonlar ile iletişim sağlamaz, data katmanındaki Interface versiyonlar ile direk iletişime geçer. Dolayısı ile data katmanındaki Interface yapısının business katmanındaki concrete yapının içerisine injection işlemi yapılması gerekir. Bununla ilgili kodlar CardManager ın baş tarafına yazılmıştır. Burada öncelikle bu dosya içinde kullanılacak olan nesnemizi (`_cardRepository`) tanımlıyoruz. Bu nesnenin tipi (`ICardRepository _cardRepository`) şeklindedir. Inject işlemi yaptığımız metodun yazımına dikkat etmeliyiz. (`public CardManager()`). Başında sadece public vardır metod adı üstteki class adı ile aynıdır. Bu metod içerisine dışarıdan alınan nesne ise (`cardRepository`) şeklinde gösterilir ve bunun tipi de (`ICardRepository cardRepository`) şeklindedir. Dışırıdan alınan nesnenin sınıf içinde kullanılacak nesne içine atılması gerekir. Bu işlem ise (`_cardRepository = cardRepository;`) satırı ile yapılmaktadır. Dikkat edilirse sınıf içinde tüm metodlarda kullanılacak olan (`_cardRepository`) değişkenin tüm metodların üstünde ve sınıfın içinde tanımlanmış olması gerekir. Dolayısı bu nesnenin tanımlanması en yukarıda (`private ICardRepository _cardRepository;`) şeklinde yapılmıştır. Bu kullanımlara dikkat etmek gerekiyor.

TS.Business>Concrete>**CardManager.cs**

```
using TS.Business.Abstract;
using TS.Data.Abstract;
```

```
namespace TS.Business.Concrete
{
    public class CardManager : ICardService
    {
        private ICardRepository _cardRepository;

        public CardManager(ICardRepository cardRepository)
        {
            _cardRepository = cardRepository;
        }

        public void InitializeCard(string userId)
        {
            throw new System.NotImplementedException();
        }
    }
}
```

Benzer şekilde data katmanında da dolu versiyonu (Concrete) EfCoreCardRepository, EfCoreGenericRepository içindeki metodları kullanacaktır. Bu işlemler için ise dışarıdan alacağı hangi entity için işlem yapacağını bilmelidir (Card). Aynı zamanda veritabanına giderken hangi Context kullanacağını bilgisi verilmelidir (TSContext). İçerisinde oluşturulacak metodların yapısı Interface sınıfı olarak ICardRepository sınıfını kullanacağından bu Interface inde verilmesi gerekir.

TS.Data>Abstract>**ICardRepository.cs**

```
using TS.Entity;

namespace TS.Data.Abstract
{
    public interface ICardRepository:IRepository<Card>
    {

    }
}
```

TS.Data>Concrete> **EfCoreCardRepository.cs**

```
using TS.Data.Abstract;
using TS.Entity;

namespace TS.Data.Concrete.EfCore
{
    public class EfCoreCardRepository:EfCoreGenericRepository<Card,TSContext>,ICardRepository
    {

    }
}
```

Not: Business yada Data katmanında yazılan sınıfların diğerinde görülebilmesi için Build etmeyi unutmayın. Yoksa birinde yazılan diğer katmanda görülemez.

Card nesnesi için Veritabanı ile bağlantımızı sağlayacak olan Business ve Data katmanı içindeki Abstract ve Concrete sınıflarımızı oluşturduk. Bu sınıflardan nesne türetip kullanacak olan ise WebUI projemiz içindeki sınıflardır. Card ile ilgili işlemleri AccountController sınıfı içinde kullanacağımızdan bu sınıf ise direk Business katmanındaki Interface versiyon olan ICardService ile haberleşecektir. Dolayısı ile class tipindeki bir sınıfın (yani

AccountController ın) interface tipindeki bir sınıf ile haberleşmesi için (yani ICardService in) AccountController girişinde inject işlemi yapılması gerekir.

```
    public class AccountController : Controller
    {
        private UserManager<User> _userManager;
        private SignInManager<User> _signInManager;
        private IEmailSender _emailSender;
        private ICardService _cardService;
        public AccountController(UserManager<User> userManager, SignInManager<User>
signInManager, IEmailSender emailSender, ICardService cardService)
        {
            _userManager = userManager;
            _signInManager = signInManager;
            _emailSender = emailSender;
            _cardService = cardService;
        }
```

Ardından Card oluşturma işlemini ConfirmEmail() metodu içinde yapacaktık. Oraya gidip Card kayıdımızı oluşturacak kodları yazalım. Burada Card oluşturma işlemi için bu dosya içinde kullandığımız _cardService nesnemizi kullanacağız. Bu nesnenin alt metodu olan InitializeCard() bizden userId yi isteyeceğinden bizde kontroller içinde kullandığımız kullanıcı nesnemizin Id sini kendisine vereceğiz (User.Id).

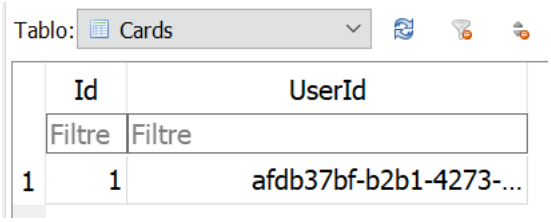
```
    var user = await _userManager.FindByIdAsync(userId);
    if (user != null)
    {
        var result = await _userManager.ConfirmEmailAsync(user, token);
        if (result.Succeeded)
        {
            //Card oluşturma bilgileri buraya konacak
            _cardService.InitializeCard(user.Id);

            TempData.Put("message", new AlertMessage()
            {
                Title = "Hesap sonucu",
                Message = "Hesabınız onaylandı.",
                AlertType = "success"
            });
            return View();
        }
    }
```

DİKKAT: WebUI projesi içerisinde User işlemleri için asp nin kendi hazır sınıfı olan Identity sınıfını kullandığımızdan bu sınıfı UserId bilgilerini STRING tipinde tanımlamıştır. Dolayısı bu sınıfla ilgili ve devamında yapılan tüm Id işlemlerinde hep UserId ler string olarak tanımlanmak durumunda kalınmıştır. Normalde kendi tanımladığımız sınıflarda Id bilgileri tamamen int tipinde tanımlanmıştı.

Startup içerisinde ilgili abstrac lar içindeki Interface sınıflarının hangi concrete sınıfları ile çalışacağını bildirmemiz gerekir. Yani ICardService çağırdığımızda onun dolu versiyonu olan CardManager ın çalıştırılması gerekir. Bununla ilgili kodlarımız Startup içerisinde şu şekilde yazılmıştı.

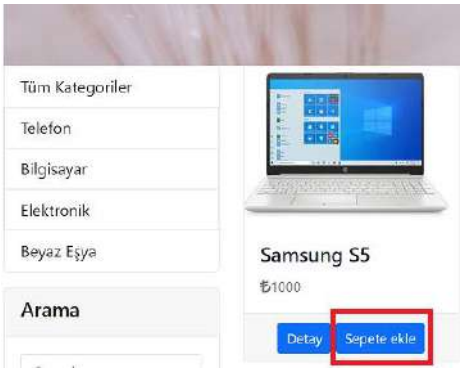
```
    //Data projesi içerisindeki IProductRepository çağrıldığında onun dolu versiyonu olan
EfCoreProductRepository gönderecek.
    services.AddScoped<IProductRepository, EfCoreProductRepository>();
    services.AddScoped<ICategoryRepository, EfCoreCategoryRepository>();
    services.AddScoped<ICardRepository, EfCoreCardRepository>();

    //IProductService çağrıldığında, ProductManager ı gönderecek.
    services.AddScoped<IProductService, ProductManager>();
    services.AddScoped<ICategoryService, CategoryManager>();
    services.AddScoped<ICardService, CardManager>();
```

Card (sepet) oluşturma kodlarımız hazırdır. Deneyelim. 10 dakikalık gecici bir mail adresi alıp dene bir kullanıcı oluşturalım ve bu kullanıcı için Cards tablosunda bilgiler oluşmuş mu görelim. Aşağıdaki şekilde bir Card için Id oluşturulmuş ve hangi kullanıcıya ait olduğu atanmış durumda.



Sepete (Card) Ekle butonun oluşturulması

Sayfalarımızda ürünleri listelerken her kart görünümün altında yada detay sayfasına geçtiğimizde gördüğümüz (Add To Card)(Sepete Ekle) butonlarımız olacaktır. Bu butonlar öncelikle kullanıcı girişi yapmış olan üyelerin karşısına getirilmesi gerekir. Giriş yapmayan kişiler ise sadece Detay sayfasını görebilmelidir. Öncelikle bu düzeltmeyi yapalım. Detay sayfasında Sepete Ekleye tıklarken yanında kaç tane ürün almak istiyorsa o konuda textbox ve butonu da ekleyelim. Bu sayfamız partial sayfa olarak Views>Shared>_Produc.cshtml olarak vardı. Oradan düzenlemeyi yapalım.



Burada AddToCard butonumuz bulunmakta fakat bu butonun sadece üye girişi yapanların görmesi gerekir. O yüzden kontrolünü aşağıdaki (@if(User.Identity.IsAuthenticated)) şeklinde yapmalıyız. Fakat bu if bloğunun içinde AddToCard butonuna tıklandığında ilgili Controller altındaki ilgili action metoduna giderken yanımızda bazı bilgileri daha götürmemiz gerekir. Submit butonu ile birlikte bazı bilgilerin birlikte gidebilmesi için <form> etiketleri arasına konulması gerekir. Burada Card işlemleri için CardController.cs sınıfını oluşturalım. Bunun içerisindede AddToCard() metodunu oluşturalım. Bu bilgileri göndereceğimiz form etiketleri içinde aşağıdaki şekilde oluşturmalıyız. Ayrıca submit işlemiyle birlikte göndereceğimiz parametreleri de form içinde hidden şeklinde göndermeliyiz. Hidden şeklinde sakladığımız alanlar yada yanımızda götüreceğimiz alanlar productId, quantity alanları olacaktır.

_product.cshmtl içine eklediğimiz form yapısı aşağıdaki şekilde oldu.

```
<div class="card-footer text-center">
    <a asp-controller="TS" asp-action="Details" asp-route-url="@Model.Url" class="btn btn-
primary btn-sm">Detay</a>

    @if(User.Identity.IsAuthenticated)
    {
        <form asp-controller="Card" asp-action="AddToCard" style="display:inline;" >

            <input type="hidden" name="productId" value="@Model.ProductId" />
            <input type="hidden" name="quantity" value="1" />
            <button type="submit" class="btn btn-primary btn-sm">Sepete ekle</button>
```

```
        </form>
    }
</div>
```

Aynı form yapımızı detay sayfasında da kullanalım. Orada da AddToCard butonu vardır. Fakat orada yanında aynı zamanda ürün sayısını artırabileceğimiz giriş alanı da olsun. Yukarıda ürün listeleme sayfasından eğer kullanıcı giriş yapmışsa gözükecektir ve ürünü sepete eklediğinde varsayılan olarak 1 adet eklenmiş olacaktır. Fakat bir kişi detay sayfasına girip ürünü inceleyip sepete eklemeye çalışırsa artık bu ürünü almak istiyor demektir. Böyle bir kişi şayet kullanıcı girişi yapmamışsa bu durumda önce kullanıcı girişi yapacağı sayfaya yönlendirmek gerekir.

Card la ilgili işlemlerimizi yapmak için Controller olarak CardController.cs oluşturalım. Tüm işlemleri buradan yönetelim. Başlangıçta Card bilgilerini gösterecek olan metodumuz Index() olsun. İkinci metodumuz da AddToCard() olsun. Bu Card direk olarak butondan geleceği için [HttpPost] olarak işaretleyelim.

CardController a sadece Authorize olanlar (giriş yapanlar) ulaşacağından en üste sınıftan önce `[Authorize]` ifadesini ekleyelim. Buna göre CardController yapımız aşağıdaki şekilde oldu. Burada login olmadan bir kişi AddToCard butonuna tıklarsa ne olur? Bu buton tabiki buraya yani controllera yönlendirecektir. Controller başında ise authorize işlemi vardır. Kişinin Authorize ı yoksa (yani bilgisayarında cookileri yoksa) bu durumda direk Login sayfasına yönlendirme olacaktır. Bununla ilgili alt yapıyı daha önceden kurmuştuk. {Bu alt yapı nasıl çalışıyordu tekrar bak burada açıkla}

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace TS.WebUI.Controllers
{
    [Authorize]
    public class CardController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }

        [HttpPost]
        public IActionResult AddToCard()
        {
            return View();
        }
    }
}
```

Şu aşamada biz Karta bilgi eklerken kullanıcı girişimiz zorunlu oldu. Çünkü bir karta bilgi ekleyebilmek için (CardItems tablosuna) Card tablosunda kişinin bilgisinin olması gerekir. Yani bir cardId olması gerekir. Böyle bir işlem içind kullanıcı oluşturulmuş olmalıdır (mail kontrolü de yapılmış olacak).

Bir sonraki derste veritabanına elle bazı ürün bilgilerini girelim ve kullanıcının karşına kart bilgilerini yani CardItems deki bilgileri gösterelim. Ondan sonraki derstede AddToCard işlemi ile CardItems tablosuna ürün bilgilerini ekleme kodlarını tamamlayalım {Anlatımda buradaki konular sıralamada yer değiştirse iyi olur.}.

Sepete Eklenen Ürünlerin, Sayfaya Getirilmesi

Henüz daha Sepete ekleme işlemini yapmadık. Onu bir sonraki derste tamamlayalım. Burada veritabanına birkaç ürünü elle ekleyelim ve bu ürünleri sepet sayfasında gösterelim.

Şu anda iki tane user ımız var.

Tablo: AspNetUsers

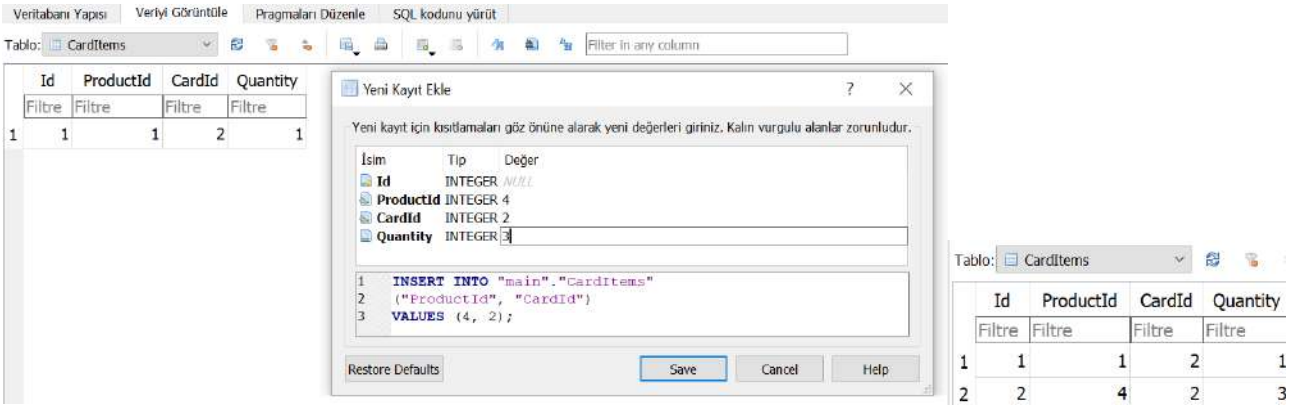
	Id	FirstName	LastName	UserName	Normal
	Filtre	Filtre	Filtre	Filtre	Filtre
1	0fb47c96-ac4f-49df-...	Ali	Su	admin	ADMIN
2	cdf918de-891a-4932-...	dene4	dene4	dene4	DENE4

2 numaralı user oluştururken mail onayı verdiğinde Cards tablosunda bilgisi oluşmuştur.

Tablo: Cards

	Id	UserId
	Filtre	Filtre
1	2	cdf918de-891a-4932-...

Şimdi 2 numaralı user birkaç tane ürün satın aldığını varsayarak CardItems tablosuna kayıt yapalım.



Buradaki amacımız bu bilgileri veritabanından okutup sayfamızda görüntülemektir. Bu bilgileri form üzerinden ekleme işlemini daha sonraki derste yapacağız.

Şimdi kullanıcının CardItems tablosundaki bilgileri CardController altındaki Index() metodu içinden getirelim. Yani kullanıcı Card (sepet) butonuna tıkladığında bu metod aracılığı ile sayfada bilgilerini görelim.

TS.Entity>Card.cs sınıfını oluşturuken sınıf yapısı içinde CardItems lara kolayca erişmek için liste şeklinde bu bilgilerin konulacağı yapıyı oluşturmuştuk. Dolayısı ile ürünlerini getireceğimiz Card ın Id bilgilerini biliyorsak ona bağlı CardItems lerine (ürünleri de) kolayca getirebiliriz. CardItem bilgisine ulaşınca ise oradan da Product (ürün detay bilgilerine) ulaşmış oluruz. CardItems tablosundan quantity leride alarak sepet toplamı için bir hesaplama yaparız.

```
using System.Collections.Generic;

namespace TS.Entity
{
    1 reference
    public class Card
    {
        0 references
        public int Id { get; set; }
        0 references
        public string UserId { get; set; } //hazır olarak kull
        0 references
        public List<CardItem> CardItems { get; set; }
    }
}
```

```
namespace TS.Entity
{
    1 reference
    public class CardItem
    {
        0 references
        public int Id { get; set; }
        0 references
        public int ProductId { get; set; }
        0 references
        public Product Product { get; set; }
        0 references
        public int CardId { get; set; }
        0 references
        public Card Card { get; set; }
        0 references
        public int Quantity { get; set; }
    }
}
```

İlk olarak userId bilgisini vererek kullanıcının Card bilgisini getirecek metodlarımızı yazalım. Metodlarımız Business ve Data katmanları içinde oluşturalım. Daha önceden çok kez yaptığımız Interface ve Class yapıları üzerinden bu bilginin gelmesini sağlayalım.

TS.Business.Abstract> **ICardService.cs**

```
using TS.Entity;

namespace TS.Business.Abstract
{
    public interface ICardService
    {
        public void InitializeCard(string userId);
        Card GetCardByUserId(string userId);
    }
}
```

TS.Business.Concrete> **CardManager.cs**

```
using TS.Business.Abstract;
using TS.Data.Abstract;
using TS.Entity;

namespace TS.Business.Concrete
{
    public class CardManager : ICardService
    {
        private ICardRepository _cardRepository;

        public CardManager(ICardRepository cardRepository)
        {
            _cardRepository = cardRepository;
        }

        public Card GetCardByUserId(string userId)
        {
            return _cardRepository.GetByUserId(userId);
        }

        public void InitializeCard(string userId)
        {
            _cardRepository.Create(new Card() { UserId = userId });
        }

    }
}
```

TS.Data.Abstract> **ICardRepository.cs**

```
using TS.Entity;

namespace TS.Data.Abstract
{
    public interface ICardRepository:IRepository<Card>
    {
        Card GetCardByUserId(string userId);

    }
}
```

TS.Data>Concrete>EfCoreCardRepository() metodu içine yazılan kodları biraz daha detaylı inceleyelim. Veritabanına bağlantıyı sağlayan context nesnemizi oluşturalım. Oluşturduğumuz bu nesnenin kullanımı bittikten sonra hafızadan atılması için using() fonksiyonu içinde yazmalıyız. Bu fonksiyon içinde kullanılıp atılan sınıfların IDisposable interface ne sahip olması gerekir.

Kodun detayında (`context.Cards`) ifadesiyle context üzerinden Cards tablosuna ulaşıyoruz. Ardından bu tablo üzerinden (`Include(i=>i.CardItems)`) ifadesiyle CardItems tablosuna geçiş yapıyoruz. Daha sonra gelen (`.ThenInclude(i=>i.Product)`) ifadesiyle CardItems tablosundan da Product tablosuna geçiş yapıyoruz. Daha önce açıkladığımız gibi bu tablolar arasında geçişi sağlamak için sınıf yapılarının içerisine nesne olarak tanımlamalar yapmıştık (Buna Navigation Property denir). Bu sayede tablodan diğer bir tabloya geçiş yapıyoruz. Bu işlemin sonucunda bütün Card bilgileri gelir. Onun bağlı olduğu CardItems ve Productlarıyle birlikte. Böyle bir şeyi istemeyiz. Sadece bizim ilgilendiğimiz Card a ait bilgilerin gelmesini isteriz. Bu nedenle sorgunun en sonuna (`.FirstOrDefault()`) ifadesini en son da kullanmalıyız. Bu sınıf üstte Linq namespaceini ister.

FirstOrDefault ifadesi bulduğu ilk kaydı alacaktır. Fakat biz tablolarda bulunan ilk kaydı istemeyiz. Bizim UserId si ile eşleşen kaydı almamız gerekir. Bu nedenle ifadenin içerisine ek bir sorgu daha yazmalıyız. (`.FirstOrDefault(i => i.UserId == userId);`) En sona bu ifadeyi eklemek yerine Cards ifadesinden sonra bir Where koşul ifadesi de ekleyebilirdik. Yine aynı işlemi yapmış olurduk. Bu ifadenin kullanımını daha iyi anlamak için şu iki yazım şeklinin aynı olduğunu bilelim.

```
filteredPeopleList = people.Where(person => person.Name == "John");
```

```
foreach (var person in people)
{
    if (person.Name == "John")
    {
        filteredPeopleList.Add(person);
    }
}
```

Yazdığımız kodların son hali şu şekilde olur.

TS.Data>Concrete>**EfCoreCardRepository.cs**

```
using Microsoft.EntityFrameworkCore;
using System.Linq;
using TS.Data.Abstract;
using TS.Entity;

namespace TS.Data.Concrete.EfCore
{
    public class EfCoreCardRepository : EfCoreGenericRepository<Card, TSContext>,
ICardRepository
    {
        public Card GetCardByUserId(string userId)
        {
            using (var context = new TSContext())
            {
                return context.Cards
                              .Include(i => i.CardItems)
                              .ThenInclude(i => i.Product)
                              .FirstOrDefault(i => i.UserId == userId);
            }
        }
    }
}
```

Şimdi veritabanı işlemlerini yapacak olan bu sınıfları hazırladıktan sonra yapıyı kullanacak olan CardController içindeki metodlarımızı yazalım.

CardController bizden veritabanı işlemlerini yapabilmesi için bir service (business katmanında oluşturduğumuz nesneler) bekleyecektir. Bu service yapsını Constructor yöntemiyle ekleyelim. Kodlarımız şu şekilde olacaktır. Buradaki yapıyı inceleyecek olursak (`private ICardService _cardService;`) ifadesiyle sınıf içinde ICardService sınıfından türetilmiş _cardService nesnesini kullanacağız demektir. _cardService nesnesi sadece bu sınıfı içinde kullanılabilecek. Fakat sınıfa dışarıdan bir bilgi gelmesi gerekir. Gelen bu bilgi ise (`ICardService`) yapısında olmalıdır. Dolayısı ile sınıfın Constructor girişine (`ICardService cardService`) yazarak dışarıdan alınacak olan cardService nesnesi ki bu nesne ICardService tipinde olmuş oluyor. Daha sonra dışarıdan alınan bilginin sınıf içinde kullanılacak nesnenin içine atılması gerekir. Bu işlemi de (`_cardService = cardService;`) satırı ile yapıyoruz.

```
public class CardController : Controller
{
    private  ICardService _cardService;

    public CardController(ICardService cardService)
    {
        _cardService = cardService;
    }
```

Controller içindeki Index() metodu içinde kullanıcının card bilgilerini almaya çalışalım. Card bilgilerini alırken aşağıdaki satırı kullanıyoruz fakat UserId yi getirebilmek için _userManger nesnesine ihtiyacımız oldu. Dolayısı ile bu nesneyi controller için constructor yapısı ile ekledik.

```
private  ICardService _cardService;
private UserManager<User> _userManager;

public CardController(ICardService cardService, UserManager<User> userManager)
{
    _cardService = cardService;
    _userManager = userManager;
}

   var card = _cardService.GetCardByUserId(_userManager.GetUserId(User));
```

Burada kullanılan User bilgisi veritabanından gelen bir bilgi değil. Identity ile ilişkili giriş yapan kullanıcının bilgileridir. Dolayısı ile session üzerinden gelen bir bilgidir {Bu ifadeyi doğrulamak gerekecek}

Artık CardController>Index() metodumuz içinde kullanıcının kart (sepet) bilgilerini alabiliyoruz. Bu bilgileri uygun bir model içerisinde View sayfasına taşıyabilmemiz gerekir. O zaman Models klasörü içerisinde yeni bir model sınıfı daha oluşturalım.

Not: Models klasörü içindeki model sayısı arttıkça bulmak zorlaşacağından bu klasör içerisine alt klasörler oluşturarak bunları daha hiyerarşik yapabiliriz. Yani Admin ile alakalı, Identity ile alakalı modelleri farklı başlıklar altında toplayabiliriz.

Models klasörü içerisine CardModel isminde bir class ekleyelim. İçerisinde Card bilgisi ile CartItems bilgilerini taşıyacak şekilde yapıyı oluşturalım.

TS.WebUI.Models> **CardModel.cs**

```
using System.Collections.Generic;

namespace TS.WebUI.Models
{
    public class CardModel
    {
        public int CardId { get; set; }
        public List<CardItemsModel> CardItems { get; set; }
    }

    public class CardItemsModel
```

```
    {
        public int CardItemId { get; set; }
        public int ProductId { get; set; }
        public string Name { get; set; }
        public string Price { get; set; }
        public string ImageUrl { get; set; }
        public int Quantity { get; set; }
    }
}
```

Bu model içinde bilgileri View sayfasına taşımak için action metodu içinde içlerini doldurmak gerekir. Doldurma işlemini direk return içinde yapalım. Sayfaya gidecek model içinde CardId bilgisi zaten metod içindeki card nesninin Id bilgisi olacaktır (`CardId = card.Id,`). Model içinde gidecek olan CardItems bilgilerini elde etmek biraz daha karmaşık. Biz zaten sayfamızdaki card nesnesinin içinde onun bağlı olduğu cardItems bilgilerini elde ediyorduk. Hatırlarsak buna Navigation Property demiştik. O zaman bu bilgileri buradan okuyup okuduğumuz her bir cardItem bilgisini sayfaya götüreceğimiz CardItemsModel içinde ayrı ayrı oluşturup taşıyabiliriz. Bu işlem için aşağıdaki gibi bir select sorgusu yazacağız. En sonunda oluşan herbir CardItems elemanını listeye çevirip zaten liste şeklinde olan model içindeki (`List<CardItemsModel> CardItems`). İçerde cardItem ları döndürürken her bir eleman i nin içinde olacaktır. Yani döngüde her bir cardItem i şeklinde gösterilmiş olacak ve bizde cardItem a ait bilgileri çekerken i değişkenini kullanacağız.

Burada fiyat kısmında şuna dikkat etmeliyiz. Product.Price nullable şeklinde double olarak tanımlanmıştı. Double = ?double içine direk atılamıyor. Bir tanesi nullable olunca direk atılamıyor o yüzden dönüşüm yaparak atmamız gerekir. Yani (`Price =(double)i.Product.Price,`) şeklinde yazmalıyız.

CardController>Index() metodumuzun son hali aşağıdaki şekilde oldu. Buradaki CardModel içindeki bilgiler artık View sayfasına gidecektir.

```
public IActionResult Index()
{
    var card = _cardService.GetCardByUserId(_userManager.GetUserId(User));
    return View(new CardModel()
    {
        CardId = card.Id,
        CardItems = card.CardItems.Select(i => new CardItemsModel()
        {
            CardItemId = i.Id,
            ProductId = i.ProductId,
            Name = i.Product.Name,
            Price =(double)i.Product.Price,
            ImageUrl = i.Product.ImageUrl,
            Quantity = i.Quantity

        }).ToList()
    });
}
```

Şimdi View sayfasını hazırlayalım. CardController için Views klasörü altında Card isminde bir klasör öncelikle bulunmalıdır. Bu klasör içinde ise Index() metodunun karşılığı olarak da Index.cshtml dosyasını oluşturalım.

Views>Card>**Index.cshtml**

```
@model CardModel

<h1 class="h3">Alışveriş Sepeti</h1>
<hr>
<div class="row">
    <div class="col-md-8">
        <table class="table hover">
            <thead>
```

```
                <tr>
                    <th></th>
                    <th>Ürün Adı</th>
                    <th>Fiyatı</th>
                    <th>Adedi</th>
                    <th>Toplam</th>
                    <th></th>
                </tr>
            </thead>
            <tbody>
                @foreach (var item in Model.CardItems)
                {
                    <tr>
                        <td>
                            <img src="~/img/@item.ImageUrl" width="80" alt="">
                        </td>
                        <td>@item.Name</td>
                        <td>@item.Price</td>
                        <td>@item.Quantity</td>
                        <td>@(item.Quantity * item.Price)</td>
                        <td>
                            <button class="btn btn-danger btn-sm">
                                <i class="fa fa-times fa-fw"></i>
                            </button>
                        </td>
                    </tr>
                }
            </tbody>
        </table>
    </div>
    <div class="col-md-4"></div>
</div>
```

Startup içerisindeki Route bilgimizi yazalım (Link desenini). Bu ifadeye göre "Localhost/card" dediğimizde CardController altında Index() metoduna gidecektir.

```
    endpoints.MapControllerRoute(
        name: "card",
        pattern: "card",
        defaults: new { controller = "Card", action = "Index" }
    );
```

Sepet Toplamının Hesaplanması

Bir önceki dersimizde kullanıcının sepetindeki ürünleri gösterdik. Bu derste ise ürünlerin toplam fiyatını gösterelim.

View sayfasına götürdüğümüz Model içinde (CardModel) cardItems (ürünler) ile ilgili tüm tanımlamalar vardır. Aynı şekilde tıpkı bir property (özellik) tanımlıyor gibi "decimal TotalPrice()" şeklinde bir metod tanımlayabiliriz. Sonuçta Model dediğimiz dosyada bir Class dır ve içinde bir metod tanımlanabilir. Tabii metodumuz geri sayı döndüreceğinden, virgüllü sayıları göndersin ve diye double tipinde tanımlıyoruz.

CardItems dediğimiz nesne içinde CardItem leri tutan bir liste idi. Dolayısı ile CardItems.sum() fonksiyonunu kullanarak içinde hangilerinin toplamını alacağımız bildirebiliriz. Listenin içindeki her bir eleman i ile temsil edilir. i.price dediğimizde aslında CardItem.Price demiş oluyoruz. Dolayısıyle (i=>i.price * i.quantity) ifadesi bir tane CardItem in hesaplar. Fakat parantez dışında CardItems.Sum() fonksiyonu dizinin toplamını alır.

```
   public class CardModel
    {
        public int CardId { get; set; }
        public List<CardItemsModel> CardItems { get; set; }

        public double TotalPrice()
        {
```

```
            return CardItems.Sum(i=>i.Price * i.Quantity);
        }
    }
```

Model sınıfı içerisine bu metodu ekledikten sonra artık bunu sayfamızda doğrudan kullanabiliriz. Şimdi View sayfamızı (Card>Index.cshtml) sayfamızı biraz daha düzenleyelim. Sayfayı 8 lik ve 4 sütunlar olarak ikiye ayırmıştık. 8 lik kısımda CardItems ları (ürünleri) 4 lük kısımda ise Card Summary (Sepet özetini) gösterelim. Son durumda sayfa kodlarımız şu şekilde oldu.

Views>Card>**Index.cshtml**

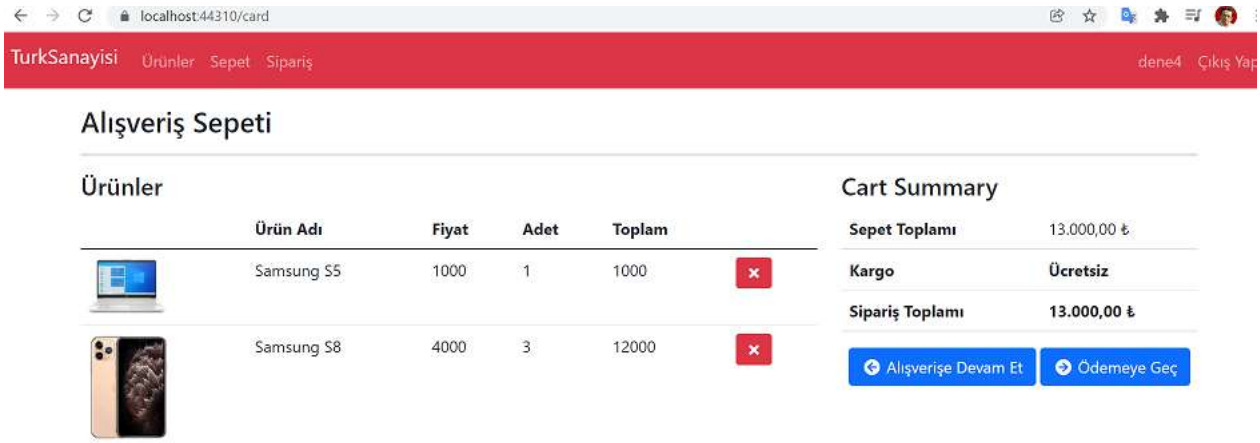
```
@model CardModel

<h1 class="h3">Alışveriş Sepeti</h1>
<hr>
<div class="row">
    <div class="col-md-8">
        <div class="text-left">
            <h4>Ürünler</h4>
        </div>
        <table class="table table-hover">
            <thead>
                <tr>
                    <th></th>
                    <th>Ürün Adı</th>
                    <th>Fiyat</th>
                    <th>Adet</th>
                    <th>Toplam</th>
                    <th></th>
                </tr>
            </thead>
            <tbody>
                @foreach (var item in Model.CardItems)
                {
                    <tr>
                        <td>
                            <img src="~/img/@item.ImageUrl" width="80" alt="">
                        </td>
                        <td>@item.Name</td>
                        <td>@item.Price</td>
                        <td>@item.Quantity</td>
                        <td>@(item.Quantity * item.Price)</td>
                        <td>
                            <button class="btn btn-danger btn-sm">
                                <i class="fa fa-times fa-fw"></i>
                            </button>
                        </td>
                    </tr>
                }
            </tbody>
        </table>
    </div>
    <div class="col-md-4">
        <div class="text-left">
            <h4>Cart Summary</h4>
        </div>
        <table class="table">
            <tbody>
                <tr>
                    <th>Sepet Toplamı</th>
                    <td>@Model.TotalPrice().ToString("c")</td>
                </tr>
                <tr>
                    <th>Kargo</th>
                    <th>Ücretsiz</th>
                </tr>
                <tr>
                    <th>Sipariş Toplamı</th>
                    <th>@Model.TotalPrice().ToString("c")</th>
                </tr>
            </tbody>
        </table>
        <div class="text-center">
            <a href="/" class="btn btn-primary">
                <i class="fa fa-arrow-circle-left fa-fw"></i> Alışverişe Devam Et
            </a>
```

```
            <a href="/checkout" class="btn btn-primary">
                <i class="fa fa-arrow-circle-right fa-fw"></i> Ödemeye Geç
            </a>
        </div>
    </div>
</div>
```

Bu arada NavBar içinde "Sepet" linkine tıkladığımızda sayfamıza link gitsin.

```
<li class="nav-item">
       <a href="/card" class="nav-link">Sepet</a>
</li>
```

Kodları deneyelim. Genel olarak sayfa yapısı oluştu.



Sepete Ürün Ekleme

Ürünleri listelediğimiz yerde yada Ürün detay sayfasında geçen bir "Sepete Ekle" butonu (AddToCard) na tıklayınca butonla birlikte bazı bilgiler form içerisinden alınıp post tipindeki action AddtoCard() metoduna gelmesi gerekir. Daha önceki uygulamalarımızda ürün detay sayfasındaki "sepete ekle" butonunu bir form içine almıştık ve form içinden ProductId bilgisini hidden olarak, Quantity bilgisini ise gösterek koymuştuk. Submit butonuna tıkladığımızda ise bu bilgiler action metodumuza gelecektir. Burada bir de kullanıcının Card bilgisine (CardId) ihtiyamız vardır. Fakat zaten login olmuş bir kullanıcının her an bilgilerine User nesnesi aracılığı ile ulaşabiliyoruz. Yani sayfadan sayfaya taşıma ihtiyacımız yoktur. Bu nedenle kullanıcının kart bilgilerini UserId aracılığı ile alabiliriz.

Veritabanına kayıt işlemi yapacağımızdan Business, Data katmanları içinde oluşturduğumuz Abstract ve Concret içindeki sınıflarımızı oluşturalım. Burada oluşturacağımız sınıf dışarıdan UserId, ProductId ve Quantity alsın ve bu bilgileri veritabanında CardItems tablosunun içerisine eklesin. Hazırlaayacağımız sınıf yapısı şimdilik geri değer döndermesin. Yani kayıttan sonra bir sonuçda gönderebilir. Bununla şimdilik uğraşmayalım. O zaman sınıf yapılarımızı aşağıdaki şekilde oluşturalım.

TS.Business.Abstract> ICardService

```
using TS.Entity;

namespace TS.Business.Abstract
{
    public interface ICardService
    {
        public void InitializeCard(string userId);
        Card GetCardByUserId(string userId);

        void AddToCard(string userId, int productId, int quantity);
    }
```

```
}
```

Interface sınıfının dolu versiyonunu CardManager içinde oluşturalım. Bunun için arabirimi uygula diyerek çerçeve ortaya çıksın.

CardManager içerisinde daha önceden oluşturduğumuz (`public Card GetCardByUserId(string userId)`) metodumuz vardı. Bu metodu AddToCard içinde kullanarak kullanıcının card bilgilerini alalım. İşlem yaparken card!=null kontrolünü oluşturalım.

Eğer kartı varsa bu durumda ne yapacağız. İki seçenek ortaya çıkacaktır. Eklenecek ürün daha önceden eklediği ürünlerin içinde var ise burada sadece o ürünün quantity sini artırak. Bu durumda CardItems tablosunda güncelleme işlemi yapacak. Peki eklenecek ürün yok o zaman tabloya yeni bir kayıt eklemiş olacak. Bu yapıyı oluşturalım.

Öncelikle eklenecek olan ürün CardItems tablosu içinde varmı kontrol edelim. Bunun için CardItem ın tüm bilgilerini getirmemize gerek yok. Sadece index numarasını (sıra numarasını) sorgulasak yetecektir. Dolayısı ile elimde Card nesnesi var ve nesnenin üzerinde CardItems tablosuna geçip orada FindIndex() metodunu kullanarak bir index numarası döndürmeye çalışırız. Şayet pozitif bir sayı geliyorsa bu ürün var demektir. Negatif bir sayı dönerse bu ürün yok demektir. FindIndex metodunu kullanırken tüm itemleri taramasın için i=>i.ProductId şeklinde yazarak tablodaki tüm ProductId bilgilerini tararız. Bunu yaparken dışarıdan alından productId bilgisi ile karşılaştırırsak sonucu pozitif yada negatif bir sayı olarak alabiliriz. İlgili satırımız şu şekilde olacaktır.

```
var index = card.CardItems.FindIndex(i => i.ProductId == productId);
```

Eğer index numarası negatifse yeni bir kayıt ekleyelim. Kodlar aşağıdaki şekilde olacaktır.

```
if(index<0) //eğer negatifse
{
    card.CardItems.Add(new CardItem() {
        ProductId = productId,
        Quantity = quantity,
        CardId = card.Id
    });
}
```

Şayet geri dönen index numarası pozitif ise tabloya bu üründen daha önce eklemiş demektir. Buna göre Quantity bilgisini dışarıdan aldığı quantity sayınca artıracak.

```
else
{
      card.CardItems[index].Quantity += quantity;
}
```

Card bilgilerini update etmek için aşağıdaki satırıda eklemeliyiz. Diğerlerinde bilgilerin içini doldurduk. Birinde veritabanıdan gelen card bilgisi yoktu kendimiz yeni bir boş kart nesnesi oluşturup içerisine bilgileri doldurduk. Update işleminde ise veritabanındaki cardla ilgili bilgileri aldık ve bunun quantitsini değiştirdik. fakat veritabanında henüz güncelleme olmadı. Update etmek içinde _cardRepository içine yazdığımız ve onların içinde entityframework un varsayılan metodlarından olan Update() metodunu kullanacağız. Bu metod ya yeni bir kart ekleyecek yada varolan kolonu güncelleyecek. Dikkat edersek güncelleme işlemi için Card tablosunun bilgisi verildi. CardItems tablosu da ona bağlı olduğundan onlarda güncellenmiş olmaktadır. Kodlarımızın son hali şu şekilde olacaktır.

```
public class CardManager : ICardService
{

    public void AddToCard(string userId, int productId, int quantity)
```

```
        {
            var card = GetCardByUserId(userId);
            if(card !=null)
            {
                var index = card.CardItems.FindIndex(i => i.ProductId == productId);
                if(index<0) //eğer negatifse
                {
                    card.CardItems.Add(new CardItem() {
                        ProductId = productId,
                        Quantity = quantity,
                        CardId = card.Id
                    });
                }
                else
                {
                    card.CardItems[index].Quantity += quantity;
                }
                _cardRepository.Update(card);
            }
        }
```

Data katmanındaki işlemlere devam edelim. Update işlemi için Data>Concrete>EfCoreGenericRepository içindeki hazır olarak Entityframework un metodu olan Update metodunu kullanıyoruz. Ancak biz burda gelen entity nin durumunu modified olarak değiştiriyoruz. Ancak bu şekilde bir yöntem kullandığımızda mevcut entity ile bağlantılı kayıtlar üzerinde bir işlem yapmıyor. Yani buradaki uygulamada Card tablosu üzerinde bir update işlemi yapıyoruz fakat aynı zamanda bu tablo ile ilişkili olan CardItems tablosundaki güncellemeleride yapmak istiyoruz. Bu durumda bu hazır metod bizim işimizi görmüyor. Dolayısı ile içeriğini kendimizin yazdığı yeni bir metod oluşturmalıyız. Tabii bu metodun adını Update ile aynı ismi veremeyiz. O zaman mevcut metodla çakışır. Peki bu durumda ne yapacağız, şimdi ona bakalım.

```
    public void Update(TEntity entity)
    {
        using (var context = new TContext())
        {
            context.Entry(entity).State = EntityState.Modified;
            context.SaveChanges();
        }
    }
```

Şimdi bu metodun farklı bir versiyonunu yazalım. Kendimize ait metodlarımızı yazdığımız yer EfCoreCardRepository nin içerisidir. Burada daha öncede kendimize ait metodlar yazmıştık. Fakat hatırlarsa burada oluşturacağımız bir metodun Interface versiyonlarını da Abstract klasörü içinde oluşturmamız gerekir. Eğer ismini değiştirmezsek buna ihtiyaç kalmaz. Fakat bu seferde aynı isimde olan bizim oluşturduğumuz metod ile EfCore kendi içinde olan metodlar çakışacaktır.

Böyle bir durumda aynı metodun iki farklı versiyonunu oluşturabileceğimiz, Virtual ve Override tiplerini kullanarak yapabiliriz. Bu konuyu alternatif bir bilgi olması için burada anlatalım. O zaman yapmamız gereken ilk orijinal metodu Virtual olarak tanımlayacağız. Bu metod generik olarak oluşturulmuştu. (`public virtual void Update(TEntity entity)`) şeklinde yazdığmız daha artık Update metodunu override edebiliyoruz (ezebiliyoruz).

Bizim override tipinde oluşturduğumuz metod içinde biz orijinal (virtual) metodundaki işlemleri yapmak istemiyoruz. Sadece onun isim hakkını kullanıyoruz. İçeriğini tamamen kendi istediğimiz şekilde dolduruyoruz. Zaten aynı içeriği kullanacak olsak bu metoda ihtiyaç kalmazdı.

Biz kendi yazdığımız metod içinde yen eklemek istediğimiz işlemleri yaptıktan sonra devamında mevcut orijinal metoddaki işlemleri yapmak istersek bu sefer override olarak hazırladığımız metodun içinde virtual olarak bulunan orijinal metodu kullanabiliriz. Yani bir override Update() metodu içinde yine aynı isimde virtual Update() metodunu kullanacağız. Böyle bir kullanımda base.Update(entity) yazarek orijinal metodu çağırmamız gerekir.

Her iki metodumuzu da yanyana yazalım. Birinci orijinal (virtual) metodumuz sadece entitynin kendisinde güncelleme yapar (`.State = EntityState.Modified;`) ifadesi buna sebep oluyor. Yani bu kullanımda sadece Card tablosunda güncelleme yapar. İkinci override metodu kullandığımızda ise bu ifadeyi kaldırıp yazdık. Böylece bir entity güncellenirken onun içerisinde bulunan bağlantılı başka entitileride güncellemiş olacak. Burada ana entity miz Card dır. Onun bağlantılı olduğu entity ise CardItems dir. Çünkü sınıf yapısı içinde bu bağlantıyı kullandık (buna Navigation Property dedik). Tabii burada TContext değilde yine kendi yazdığımız TSContext imizi kullandık.

Virtual Update metodu	Override Update metodu

`public virtual void Update(TEntity entity)`
`{`
`  using (var context = new TContext())`
`  {`
`  context.Entry(entity).State = EntityState.Modified;`
`  context.SaveChanges();`
`  }`
`}`
	`public override void Update(Card card)`
	`{`
	`  using (var context = new TSContext())`
	`  {`
	`  context.Cards.Update(card);`
	`  context.SaveChanges();`
	`  }`
	`}`

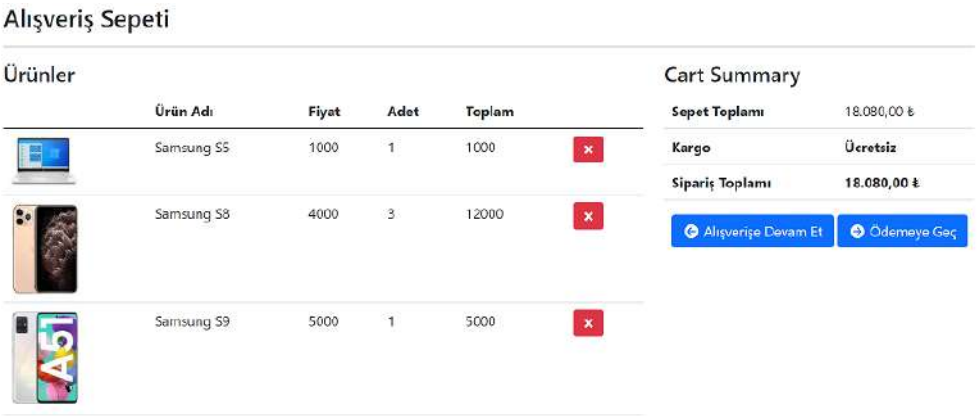
Artık controller içindeki işlemlerimizi yapalım. Nerde kalmıştık. En son kullanıcı bir ürünü kartına eklemek istediğinde yani AddToCard butonuna tıkladığında form içinden alınan productId ve quantity bilgilerini [HttpPost] tipindeki AddToCard() metoduna görüyorduk. Burada Update sınıfı için oluşturduğumuz yukarıdak yapıyı kullanarak kayıt işlemini gerçekleştireceğiz.

Formdan submit butonuna tıkladığımızda iki tane değişken geliyordu. productId ve quantity (name içinde yazan bilgiler). Bunları action metod içine girişte alalım. Ortamda giriş yapmış olan kullanıcının bilgileri User içinde tutulmaktadır. Bu bilgiyi kullanarak _userManager içindeki GetUserId() metodunu kullanarak userId bilgisini alabiliriz. AddToCard metodlarını hazırlarken artık vereceğimiz üç tane bilgiyi gönderebiliriz. İki tanesi formdan gelmiş oldu, bir taneside ortamda giriş yapmış olan userId bilgisi oldu.

```
[HttpPost]
public IActionResult AddToCard(int productId, int quantity)
{
    var userId = _userManager.GetUserId(User);
    _cardService.AddToCard(userId, productId, quantity);
    return RedirectToAction("Index");
}
```

Kodlarımızı deneyelim.

Dikkat: Admin userı ilk veritabanı oluşturulurken seed bilgileri ile oluşturmuştuk. Dolayısı ile Admin kullanıcının Card tablosunda herhangi bir kartı olmayacaktır. Sonradan kaydolan kullanıcıların ise mail onaylarını yaptıklarında Card tablosunda bilgileri oluşmaktadır. O nedenle admin sepete ekleme uygulamalarını yapamaz. Sitenin gerçek uygulamasında admin kullanıcının Card bilgilerini de başlangıçta oluşturmak en doğru olandır.



Sepetten Ürün Silme

Sepet içerisinde ürünleri listelerken (Card>Index.cshtml) silme işlemi için aşağıda kodları verilen butonu hazırlamıştık. Kırmızı çarpı şeklinde bir buton koyduk.

```
<button class="btn btn-danger btn-sm">
       <i class="fa fa-times fa-fw"></i>
</button>
```

Bu butona tıkladığımızda gideceğimiz yer CardController altında DeleteFromCard isminde bir action metodu olsun. Fakat butondan bu metoda giderken yanımızda götürmemiz gereken bilgileri götürebilmek için submit butonun ve gidecek olan gerekli bilgilerin bir form içinde olması gerekir.

Yanımızda giderken götürmemiz gereken bilgiler hidden alanı olarak tanımlanır. Ürünü Card dan silmek için bize productId bilgisi olsa yeterlidir. Kodlarımız aşağıdaki şekilde oluştu.

```
<form asp-controller="CardController" asp-action="DeleteFromCard" method="post">
    <input type="hidden" name="productId" value="@item.ProductId" />
    <button type="submit" class="btn btn-danger btn-sm">
        <i class="fa fa-times fa-fw"></i>
    </button>
</form>
```

Action metodunu oluşturmaya başlayalım. Action metodunun giriş kısmında silinecek olan productId yi almak yeterli olur. Zaten aktif olan kullanıcı ortamdan alınmakta (User). Kullanıcı bilgisindenden userId ye ulaşbiliyoruz. Dolayısı ile silme işlemi için yazacağımız sınıf yapısına bu iki bilgiyi götürerek ilgili kaydı silebiliriz.

```
[HttpPost]
 public IActionResult DeleteFromCard(int productId)
 {
     var userId = _userManager.GetUserId(User);
     _cardService.DeleteFromCard(userId, productId);
     return RedirectToAction("Index");
 }
```

Servis katmanındaki metodlarımızı yazalım. Önce Abstract içindeki Interface

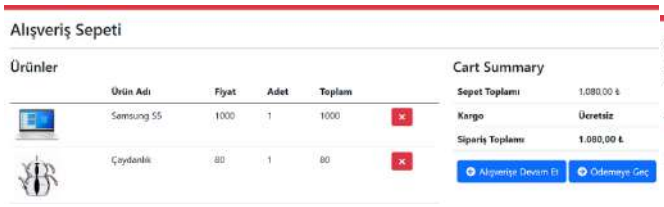
```
public interface ICardService
    {
        void DeleteFromCard(string userId, int productId);
    }
```

```
public class CardManager : ICardService
{
    public void DeleteFromCard(string userId, int productId)
    {
        var card = GetCardByUserId(userId);

        _cardRepository.DeleteFromCard(card.Id, productId);
    }
```

```
    public interface ICardRepository:IRepository<Card>
    {
        void DeleteFromCard(int cardId, int productId);
    }
```

```
    public class EfCoreCardRepository : EfCoreGenericRepository<Card, TSContext>,
ICardRepository
    {

        public void DeleteFromCard(int cardId, int productId)
        {
            using (var context = new TSContext())
            {
                var cmd = "DELETE FROM CardItems WHERE CardId=@p0 AND ProductId=@p1";

                context.Database.ExecuteSqlRaw(cmd,cardId,productId);
            }
        }
```

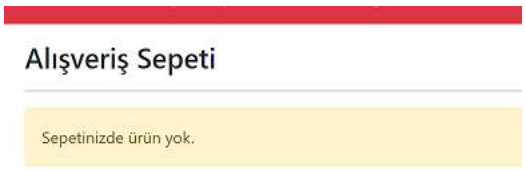
Kodları deneyelim. Tıkladığımızda direk ürün sepetten çıkarılmaktadır.



Yalnız burada kullanıcı kartındaki tüm ürünleri silerse yine sepet görüntüsü olan bu sayfanın gelmemesi gerekir. Farklı bir sayfayı kullanıcının karşısına çıkaralım.

Bunun için index sayfasında ürünlerin görüntülendiği <row> sepette bir ürün varsa çıksın. Eğer yoksa farklı bir <row> etiketi arasını görüntüleyelim. Tüm sayfanın

Kalan son ürünü sildiğimizde sayfa bu şekilde açılacaktır.



TS.WebUI>Views>Card>**Index.cshtml**

```
@model CardModel

<h1 class="h3">Alışveriş Sepeti</h1>
<hr>

@if (Model.CardItems.Count==0)
{
    <div class="row">
        <div class="col-12">
            <div class="alert alert-warning">
                Sepetinizde ürün yok.
            </div>
        </div>
    </div>
}
else
{
    <div class="row">
        <div class="col-md-8">
            <div class="text-left">
                <h4>Ürünler</h4>
            </div>
            <table class="table table-hover">
                <thead>
                    <tr>
                        <th></th>
                        <th>Ürün Adı</th>
                        <th>Fiyat</th>
```

```
                        <th>Adet</th>
                        <th>Toplam</th>
                        <th></th>
                    </tr>
                </thead>
                <tbody>
                    @foreach (var item in Model.CardItems)
                    {
                        <tr>
                            <td>
                                <img src="~/img/@item.ImageUrl" width="80" alt="">
                            </td>
                            <td>@item.Name</td>
                            <td>@item.Price</td>
                            <td>@item.Quantity</td>
                            <td>@(item.Quantity * item.Price)</td>
                            <td>
                                <form asp-controller="Card" asp-action="DeleteFromCard" method="post">
                                    <input type="hidden" name="productId" value="@item.ProductId" />
                                    <button type="submit" class="btn btn-danger btn-sm">
                                        <i class="fa fa-times fa-fw"></i>
                                    </button>
                                </form>
                            </td>
                        </tr>
                    }
                </tbody>
            </table>
        </div>
        <div class="col-md-4">
            <div class="text-left">
                <h4>Cart Summary</h4>
            </div>
            <table class="table">
                <tbody>
                    <tr>
                        <th>Sepet Toplamı</th>
                        <td>@Model.TotalPrice().ToString("c")</td>
                    </tr>
                    <tr>
                        <th>Kargo</th>
                        <th>Ücretsiz</th>
                    </tr>
                    <tr>
                        <th>Sipariş Toplamı</th>
                        <th>@Model.TotalPrice().ToString("c")</th>
                    </tr>
                </tbody>
            </table>
            <div class="text-center">
                <a href="/" class="btn btn-primary">
                    <i class="fa fa-arrow-circle-left fa-fw"></i> Alışverişe Devam Et
                </a>
                <a href="/checkout" class="btn btn-primary">
                    <i class="fa fa-arrow-circle-right fa-fw"></i> Ödemeye Geç
                </a>
            </div>
        </div>
    </div>

}
```

SİPARİŞ SÜRECİ ve KREDİ KARTI UYGULAMASI

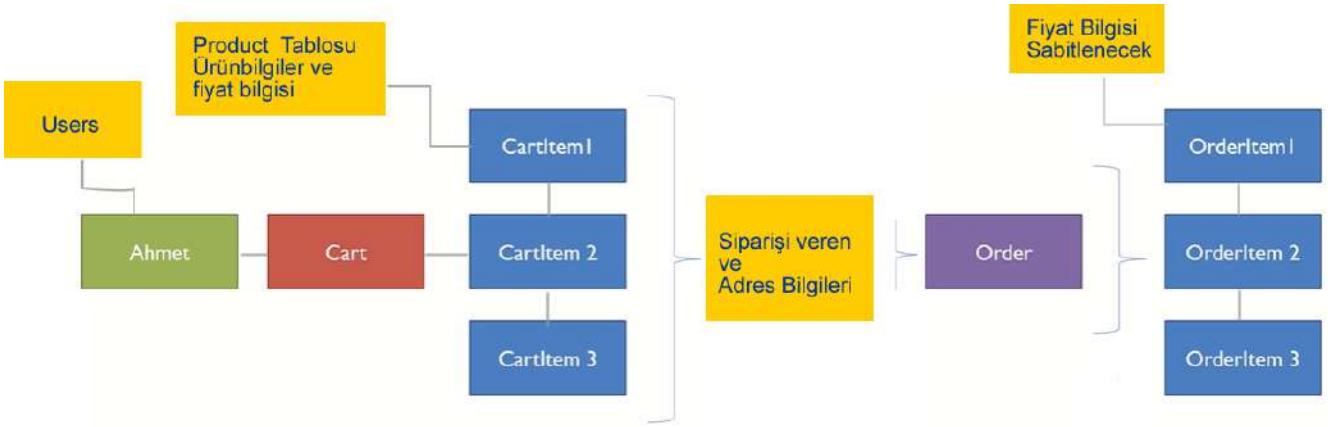
Sipariş İçin Veritabanı Altyapısının Hazırlanması

Veritabanındaki tablolarımızı oluşturmadan önce alışveriş işleyişinin mantığını bir oturtmaya çalışalım. Kullanıcılar tablosundan gelen bir Ahmet kullanıcımız olsun. Bu kullanıcının üye olurken mail onayını aldığımız anda Card tablosunda (Sepet) ona ait bir satırı otomatik olarak oluşturuyoruz. Ne zamanki bu kullanıcı bazı ürünleri almak

üzere sepetine eklerse her bir eklediği ürün Product tablosundan gelmekte ve eklenen bu ürünler CartItem Tablosunda tutulmaktadır.

Kişi sepetindeki ürünleri almak üzere sipariş vere (order) tıklarsa burada ürünün alan kişi ve adres bilgilerini istemesi gerekir. Bazen üye kendi hesabaından bir başkasının adına alım satım yapıyor olabilir. Ayrıca tek bir adresi olmayabilir. Kargonun gideceği adres ve isimler bu kısımda alınmalıdır.

Aynı Cart ve CarItem tablolarında olduğu gibi üye için Order ve OrderItem tabloları oluşturulmalıdır. Order tablosu üyeye ait ana sipariş tablosu, OrderItem larda sipariş verilen ürünlerin tablosu olur. OrderItem tablosundaki ürün bilgileri Product tablosundan gelir fakat fiyat bilgisi son değişmeyen fiyat olması için burada ayrıca tutulmalıdır. Çünkü kişi sipari vermeye başladıktan sonra ürünün fiyatı değişirse olmaz artık alıma geçmiştir. O yüzden son fiyat burada kalmalıdır. CartItem tablosundaki ürün fiyatları ise Product tablosundan gelir ve fiyat değişirse orada artık fiyat yeni haliyle yansıtılır.



Entity lerimizi oluşturmaya başlayalım. TS.Entity katmanı içerisinde iki tane entity ye (tabloda diyebiliriz) ihtiyacımız var.

Order (sipariş) tablosu içinde şipariş numarası olarak Order>Id bilgisi de kullanılabilir yada belli sayıdan oluşan bir numarada verilebilir. Siparişin verildiği tarih tutulabilir. Sipariş veren kullanıcının Id si tutulabilir ama bu kargolanacak kişiyi temsil etmeyecek. Kargolanacak kişinin bilgileri ise altında ayrıca adı, soyadı, adresi, telefonu ve maili olacak şekilde kaydedilecektir. Bu bilgiler kullanıcının bilgileri ile aynı ise oradan otomatik olarakda çekilebilir. Ayrıca siparişin durumunu belirlemek için bir liste şeklinde tanımlamada oluşturulabilir. Buradaki maddeler dahada artırılabilir. Waiting ifadesi alışveriş ödeme butonuna tıklamış ama henüz daha ödeme yapmamış durumunu belirtebilir. Ödenmiş durumu Unpaid ile sipariş adrese ulaşmış ve onay alındı ise completed ifadesi ile belirtilebilir. Bunların karşılarına gelecek sayılarda farklı anlamlarla eşleştirilebilir.

TS.Entity> **Order.cs**

```
using System;
using System.Collections.Generic;

namespace TS.Entity
{
    public class Order
    {
        public int Id { get; set; }
        public string OrderNumber { get; set; }
        public DateTime OrderDate { get; set; }
        public string UserId { get; set; }

        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Address { get; set; }
        public string City { get; set; }
        public string Phone { get; set; }
        public string Email { get; set; }
```

```
        public string Note { get; set; }

        public EnumOrderState OrderState { get; set; }

        public List<OrderItem> OrderItems { get; set; }
    }

    public enum EnumOrderState
    {
        waiting = 0,
        unpaid = 1,
        completed = 2
    }
}
```

OrderItem tablosu ise sipariş verilen herbir ürünü temsil edecektir. Ürünün hangi Order tablosuna bağlı olduğunu belirtmek için OrderId özelliği eklendi. Order tablosundaki diğer bilgiler ulaşmak istediğimizde ise direk nesne olarak bağlantı sağlamak için Order Order özelliği eklendi (bu işleme Navigation Property diyorduk). Ürünle ilgili bilgilere ulaşmak için ProductId özelliği eklendi. Product Product bilgisi ile de ürünün tüm özelliklerini nesne olarak çekmek için eklendi. Ayrıca ürünün fiyatı product tablosundan gelmeyecek burada artık en son sipariş verilen meblağ tutulacak. Ayrıca ürünün kaç tane olduğu bilgisi yine burada tutulacak. Hatırlanırsa CardItem içinde de bu bilgi tutuldu. Quantity bilgisi aslında değişmeyeceği için (fiyat gibi) Order tablosu üzerinden OrderItem tablosuna geçiş yaparakda alınabilir. Buraya kaydetmek bir son işlemi göstereceğinden tutulabilir.

TS.Entity> **OrderItem.cs**

```
namespace TS.Entity
{
    public class OrderItem
    {
        public int Id { get; set; }

        public int OrderId { get; set; }
        public Order Order { get; set; }

        public int ProductId { get; set; }
        public Product Product { get; set; }

        public double Price { get; set; }
        public int Quantity { get; set; }
    }
}
```

Ardından TSContext içindeki oluşacak tablolarımızın adlarını verelim.

```
    public class TSContext : DbContext
    {
        //Aşağıdaki DbSet ler veritabanında oluşturulacak olan tabloları ifade eder.
        public DbSet<Product> Products { get; set; }
        public DbSet<Category> Categories { get; set; }
        public DbSet<Card> Cards { get; set; }
        public DbSet<CardItem> CardItems { get; set; }
        public DbSet<Order> Orders { get; set; }
        public DbSet<OrderItem> OrderItems { get; set; }
```

Entity katmanı içerisindeki entity (veritabanındaki tablolara karşılık gelen) sınıf yapılarımızı oluşturduktan sonra artık Migration (veritabanı tablolarını oluşturacak olan kod alt yapısı) larımızı oluşturup veritabanındaki tabloları oluşturmaya geçebiliriz.

Her şeyi sıfırdan başlatalım.

Veritabanını tamamen kaldır.

TS.WebUI> dotnet ef database drop --force

Veritabanını yeniden oluştur

TS.WebUI> dotnet ef database update

Data katmanına geçelim. Daha önceden Migration var ise onları kaldıralım. Sıfırdan oluşturacağız. (Not öncesinde Entity ve data katmanlarında Build işlemi çalıştırılmalıdır. Exe üzerinden oluşturmalar yapılacaktır).

TS.Data> dotnet ef migrations remove

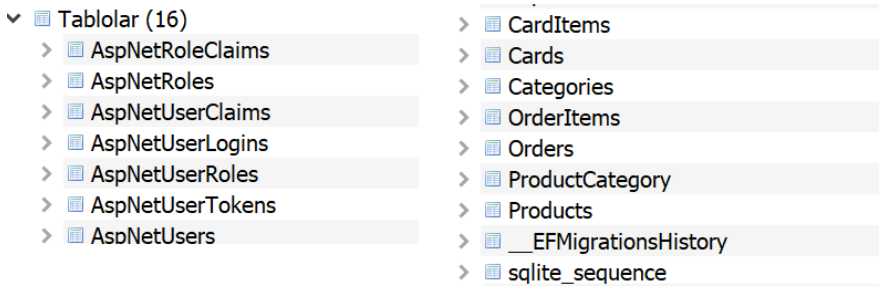
Data katmanına geçip Migrationslarımızı oluşturalım.

TS.Data> dotnet ef migrations add AddingOrderEntities --startup-project ../TS.WebUI --context TSContext

Migrationları çalıştıralım.

TS.Data> dotnet ef database update (Bu şekilde çalışmıyor. Startup proje ve contex sınıfınıda vermeliyiz.)

Veritabanımızı açtığımızda tablolarımızın oluştuğunu göreceğiz.



Sipariş Sayfasının Hazırlanması

Önceki dersimizde sipariş işlemleri için veritabanı tablolarımızı hazırladık. Bu dersimizde ise form sayfamızı oluşturalım. Sepetteki ürünlerimizi Views>Card>Index.cshmtl sayfamızda gösteriyorduk. O sayfan kullanıcı ürünlerini gördükten sonra alttaki “chek out=ödeme sayfasına” geç şeklinde gösterilen butona tıklayacak. Bu butona tıkladığında CardController>Checkout action metoduna götürecek. Bu metod ise bize bir sayfa getirecek.

Linkin controller içindeki action metoda yönlenmesi için startup içinde Rout desenimizi oluşturalım.

```
endpoints.MapControllerRoute(
    name: "checkout",
    pattern: "checkout",
    defaults: new { controller = "Card", action = "checkout" }
);
```

Controller altındaki Get tipindeki Checkout() metodu aşağıdaki şekilde olsun. Bu sayfaya sadece link üzerinden bir yönlendirme vardır. Herhangi bir parametre alınmamaktadır.

```
public class CardController : Controller
{
    public IActionResult CheckOut()
    {
        return View();
    }
```

Sayfamızın model sınıf yapısını oluşturalım. Models>OrderModel.cs dosyamızı oluşturalım. Bu model içinde daha önceden oluşturduğumuz CardModel.cs içindeki yapıyı sipariş özetini gösterirken aynen kullanabiliriz. Dolayısı ile diğer sınıfı buraya aynen ekleyelim. Bunun haricinde modelimiz içinde olması gereken bilgiler kargonun gideceği kişi ve ileşim bilgileri olacaktır.

TS.WebUI.Models> **OrderModel.cs**

```
namespace TS.WebUI.Models
{
    public class OrderModel
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Address { get; set; }
        public string City { get; set; }
        public string Phone { get; set; }
        public string Email { get; set; }
        public CardModel CardModel { get; set; }
    }
}
```

Şimdi Checkout() action metodumuzun içini dolduralım. Bu metotdan sayfamıza CardItems tablosundaki bilgileri götüreceğiz. Dolayısı ile bu bilgileri daha önce Index() metodu içinde almıştık. Ordaki yapıyı ufak bir değişiklikle kullanalım. Bu yapı iki metod içindede kullanıldığından GetItems şeklinde alt bir metod içine de alabiliriz. Burada oluşturduğumuz OrderModel in içinde CardModel yapısını kullanıyoruz. Dolayısı ile önce OrderModelden bir nesne türetelim. Bu nesenin alt özelliği CardModel olduğu için onuda bilgileri getiren CartModele eşitleyelim. Artık sayfaya içinde CardModelden gelen CardItems bilgileri olduğu şekliyle gönderebiliriz. View sayfamızda da bu modeli girişte bekliyor olmamız gerekiyor.

```
public IActionResult CheckOut()
{
    var card = _cardService.GetCardByUserId(_userManager.GetUserId(User));

    var orderModel = new OrderModel();

    orderModel.CardModel = new CardModel()
    {
        CardId = card.Id,
        CardItems = card.CardItems.Select(i => new CardItemsModel()
        {
            CardItemId = i.Id,
            ProductId = i.ProductId,
            Name = i.Product.Name,
            Price = (double)i.Product.Price,
            ImageUrl = i.Product.ImageUrl,
            Quantity = i.Quantity

        }).ToList()
    };

    return View(orderModel);
}
```

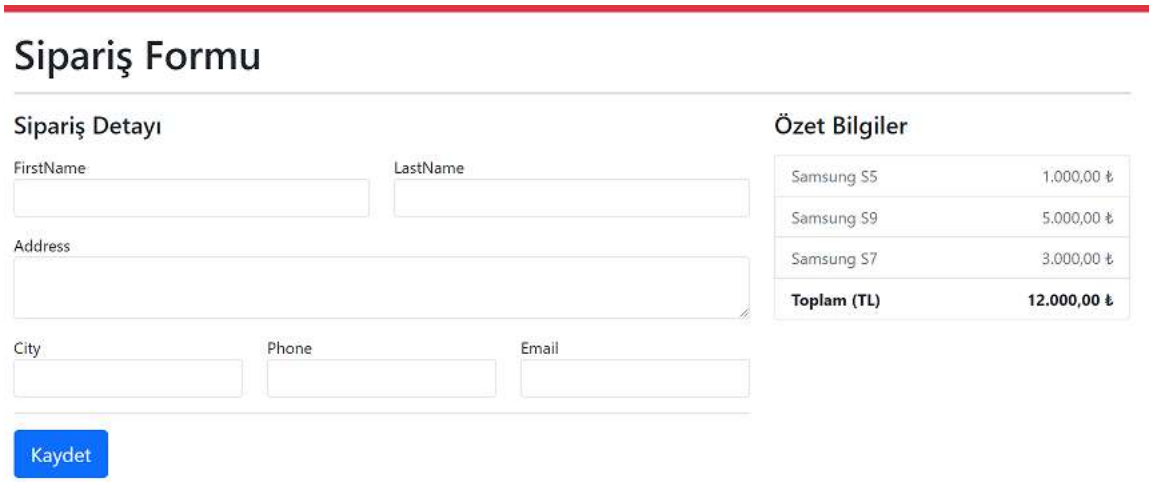
Action metodun gideceği View sayfamızı hazırlayalım.

Views>Card>**Checkout.cshtml**

```
@model OrderModel

<h1>Şiparis Formu</h1>
<hr>
<div class="row">
```

```
    <div class="col-md-8">
        <h4 class="mb-3">Sipariş Detayı</h4>
        <form method="POST">
            <div class="row mb-3">
                <div class="col-md-6">
                    <label asp-for="@Model.FirstName"></label>
                    <input asp-for="@Model.FirstName" class="form-control">
                </div>
                <div class="col-md-6">
                    <label asp-for="@Model.LastName"></label>
                    <input asp-for="@Model.LastName" class="form-control">
                </div>
            </div>
            <div class="mb-3">
                <label asp-for="@Model.Address"></label>
                <textarea asp-for="@Model.Address" class="form-control"></textarea>
            </div>
            <div class="row mb-3">
                <div class="col-md-4">
                    <label asp-for="@Model.City"></label>
                    <input asp-for="@Model.City" class="form-control">
                </div>
                <div class="col-md-4">
                    <label asp-for="@Model.Phone"></label>
                    <input asp-for="@Model.Phone" class="form-control">
                </div>
                <div class="col-md-4">
                    <label asp-for="@Model.Email"></label>
                    <input asp-for="@Model.Email" class="form-control">
                </div>
            </div>
            <hr class="mb-3">
            <button type="submit" class="btn btn-primary btn-lg btn-block">Submit</button>
        </form>
    </div>
    <div class="col-md-4">
        <h4 class="mb-3">
            <span>Özet Bilgiler</span>
        </h4>
        <ul class="list-group mb-3">

            @foreach (var item in Model.CardModel.CardItems)
            {
            <li class="list-group-item d-flex justify-content-between">
                <span class="text-muted">@item.Name</span>
                <span class="text-muted">@item.Price.ToString("c")</span>
            </li>
            }

            <li class="list-group-item d-flex justify-content-between">
                <strong>Toplam (TL)</strong>
                <strong>@Model.CardModel.TotalPrice().ToString("c")</strong>
            </li>
        </ul>

    </div>
</div>
```

Kodlarımızı deneyelim.



Burada ürün bilgileri listelenirken yanında adet sayıları da getirilebilir. Ayrıca Adres kısmı daha önce kaydedilen adresler şeklinde getirilip kişinin sadece listeden seçmesi de sağlanabilir. Ayrıca şehirler otomatik olarak getirilebilir.

Kredi Kartı Entegrasyonu

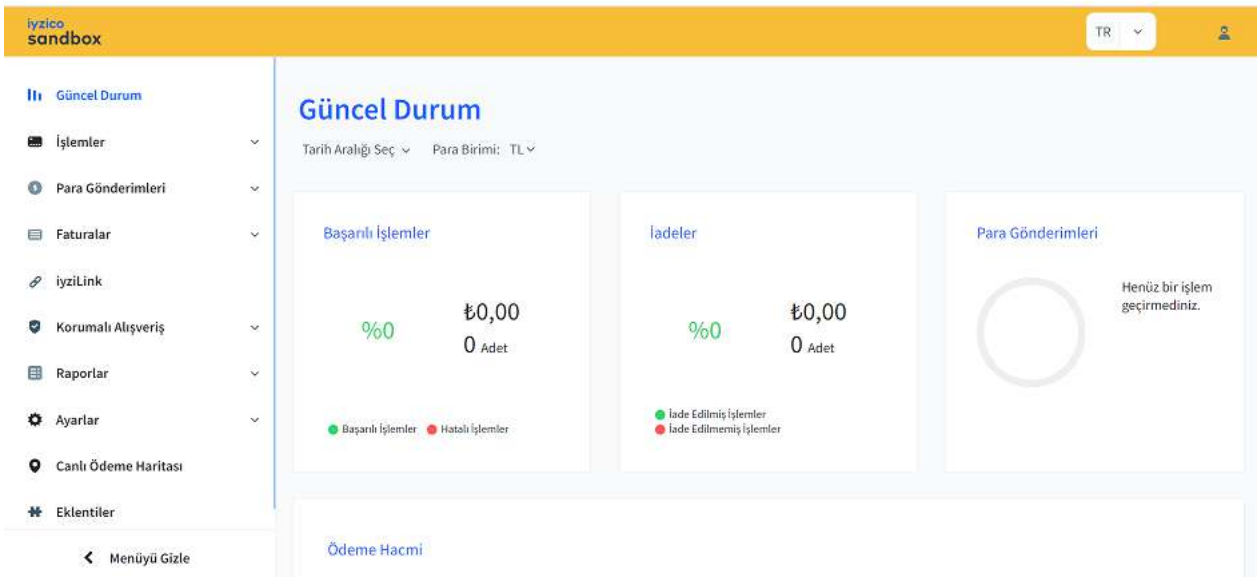
Bir önceki dersimizde Sipariş formunu hazırladık. Bu dersimizde ise Kredi kartı ödeme formunu ekleyelim ve alt yapısını hazırlayalım.

Kredi kartı işlemlerini yapabilmek için kullanabileceğimiz Api imkanı ve test ortamı sunan bazı firmalar var. Bu firmalar üzerinden yaptığımız işlemler yine bir banka üzerinden yapacağımız gerçek bir kredi kartı işlemlerinin bir benzeridir. Bunlardan bir tanesi https://www.Stripe.Com sitesi. Ayrıca Türkçe bir site olan https://www.iyzico.com/ adresi de kullanılabiliriz. Biz bu sitenin deneme api sini kullanacağız.

Bunun için sitenin içindeki geliştirici kısmına geçip oradan Sandbox(test) kısmındaki verilen bilgiler üzerinden bir test hesabı oluşturacağız. (sandbox kullanımı için ek video: https://www.youtube.com/watch?v=A1netAd_Ld0)

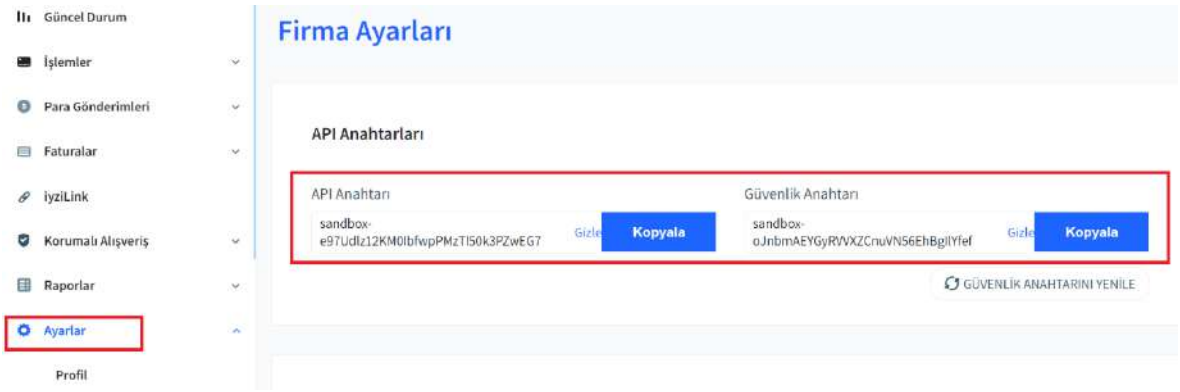
Deneme bölümü için (sandbox) Önce bir üyelik oluşturalım (https://sandbox-merchant.iyzipay.com/auth/login)

Giriş yaptığımızda aşağıdaki ekranı görürüz.



Sandbox (deneme ile ilgili sayfalara ulaşmak için Iyzico.com un en altındaki fooder kısmındaki linkleri kullanabiliriz.

Site içerisinde kullandığımız kodları ile Iyzico nun güvenli bir şekilde iletişimi işin Api anahtlarını bu linkten alabiliriz (https://sandbox-merchant.iyzipay.com/merchant-settings).



Kredi kartı ile ilgili deneme kodlarımızı ise (https://dev.iyzipay.com/tr) adresinden alıp kendi sayfamız içerisine entegre edebiliriz.

Buradan aldığımız kredi kartı bilgilerini kullanmak için Şimdi form alt yapımızı ve controller alt yapımızı hazırlayalım. Checkout.cshtml sayfamızdaki form etiketlerinin gideceği adres olan controller ve action isimlerini verelim.

```
<form method="POST" asp-controller="Card" asp-action="Checkout">
```

Formdan gelen bilgilerin gideceği [HttpPost] tipindeki Checkout() metodunu oluşturalım. (Not: Aynı isimde iki tane metod olabilmesi için (Get tipinde ve diğeride Post tipinde atarken) en azından parametrelerinin farklı olması gerekir. Buna aşırı yüklenmiş ismi (override) verilir.)

```
[HttpPost]
public IActionResult Checkout(int cardId)
```

```
        {
            return View("Success");
        }
```

Metodumuzun içerisine Iyzico dan alınan ödeme için alınan örnek kodları kopyalayalım.

Kopyaladığımız kodların kütüphanesinin olmadığını görüyoruz. Bazı kodlar çalışmamaktadır.

```
    BasketItem thirdBasketItem = new BasketItem();
    thirdBasketItem.Id = "BI103";
    thirdBasketItem.Name = "Usb";
    thirdBasketItem.Category1 = "Electronics";
    thirdBasketItem.Category2 = "Usb / Cable";
    thirdBasketItem.ItemType = BasketItemType.PHYSICAL.ToString();
    thirdBasketItem.Price = "0.2";
    basketItems.Add(thirdBasketItem);
    request.BasketItems = basketItems;

    ThreedsInitialize threedsInitialize = ThreedsInitialize.Create(

    return View("Success");
}
```

Yine aynı siteden yüklenecek paketin adını öğrenelim. Package Manager Consule a şu komutu yazıp çalıştırabiliriz.

Install-Package Iyzipay

Yada şu adrese gidip oradaki (https://www.nuget.org/packages/Iyzipay) oradaki linklerden hangisini kullanacaksak onu kopyalım.

Install-Package Iyzipay -Version 2.1.39 (Package Manager için)

dotnet add package Iyzipay --version 2.1.39 (Dotnet için)

Not: Paketlerimizi TS.WebUI projesi içerisine yükleyeceğiz. O zaman komut satırından bu proje içerisine gidip dotnet kullanarak en son versiyon paketimizi yükleyelim.

```
C:\Users\pc1\Desktop\TS.com\TS\TS.WebUI>dotnet add package Iyzipay --version 2.1.39
  Geri yüklenecek projeler belirleniyor...
  Writing C:\Users\pc1\AppData\Local\Temp\tmpDFF8.tmp
info : 'C:\Users\pc1\Desktop\TS.com\TS\TS.WebUI\TS.WebUI.csproj' projesine 'Iyzipay' paketi için
yor.
info : C:\Users\pc1\Desktop\TS.com\TS\TS.WebUI\TS.WebUI.csproj için paketler geri yükleniyor...
```

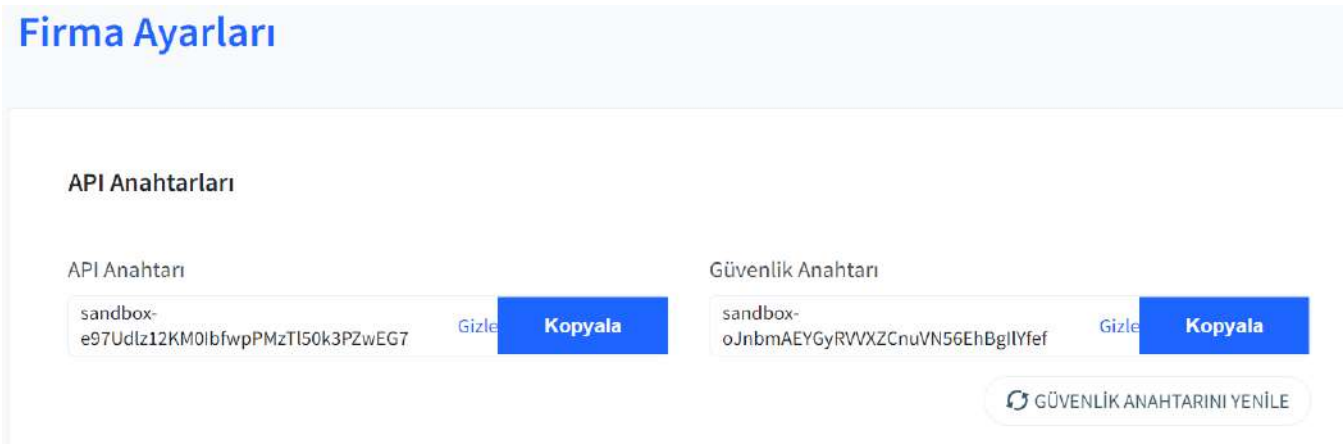
Sırasıyla çalışmayan kodlarımızın Namespacelerini yukarıya ekleyelim.

```
[HttpPost]
0 references
public IActionResult Checkout(int cardId)
{
    CreatePaymentRequest request = new CreatePaymentRequest();
    requ.Locale = Locale.TR.ToString();
    requ
    requ
```

using Iyzipay.Request;

CS0246 The type or namespa

Daha sonra sitenin (Iyzico) bize verdiği Api anahtarlarını değiştirmemiz gerekiyor.



Burdan aldığımız Api anahtarlarını aşağıdaki kodlarımızın olduğu yere yapıştırıyoruz.

```
Options options = new Options();
options.ApiKey = "sandbox - e97Udlz12KM0IbfwpPMzTl50k3PZwEG7";
options.SecretKey = "sandbox-oJnbmAEYGyRVVXZCnuVN56EhBgIlYfef";
options.BaseUrl = "https://sandbox-api.iyzipay.com";
```

Not: Buradaki (`options.BaseUrl = "https://sandbox-api.iyzipay.com";`) adresimiz gerçek bir kullanıma geçtiğimizde (`options.BaseUrl = "https:// api.iyzipay.com";`) şeklinde değiştirilmesi gerekir.

(`request.ConversationId = "123456789";`) bilgisi firma sahibinin yapılan kredi kartı ile ilgili sorgulamalar için iyzipay den yapma işinde kullandığı bir numaradır. Firmanın kendi veritabanınada bu numarayı kaydetmesi ve servis aldığ Iyzipay daki kayıtlarla aynı olması güvenlik açısından önemlidir.

Metod içindeki örnek kodlarımız aşağıdaki şekilde oldu.

```
[HttpPost]
 public IActionResult Checkout(int cardId)
 {
     Options options = new Options();
     options.ApiKey = "sandbox-RStpAR6sBz86RpFTuWPyF7hjh8c8g9E4";
     options.SecretKey = "sandbox-0zu0oxpXipwFsSGu1FrSofDVhTrWig2p";
     options.BaseUrl = "https://sandbox-api.iyzipay.com";

     CreatePaymentRequest request = new CreatePaymentRequest();
     request.Locale = Locale.TR.ToString();
     request.ConversationId = "123456789";
     request.Price = "1";
     request.PaidPrice = "1.2";
     request.Currency = Currency.TRY.ToString();
     request.Installment = 1;
     request.BasketId = "B67832";
     request.PaymentChannel = PaymentChannel.WEB.ToString();
     request.PaymentGroup = PaymentGroup.PRODUCT.ToString();
     request.CallbackUrl = "https://www.merchant.com/callback";

     PaymentCard paymentCard = new PaymentCard();
     paymentCard.CardHolderName = "Ali SU";
     paymentCard.CardNumber = "5528790000000008";
     paymentCard.ExpireMonth = "12";
     paymentCard.ExpireYear = "2030";
     paymentCard.Cvc = "123";
     paymentCard.RegisterCard = 0;
     request.PaymentCard = paymentCard;

     Buyer buyer = new Buyer();
     buyer.Id = "BY789";
     buyer.Name = "John";
```

```
buyer.Surname = "Doe";
buyer.GsmNumber = "+905350000000";
buyer.Email = "email@email.com";
buyer.IdentityNumber = "74300864791";
buyer.LastLoginDate = "2015-10-05 12:43:35";
buyer.RegistrationDate = "2013-04-21 15:12:09";
buyer.RegistrationAddress = "Nidakule Göztepe, Merdivenköy Mah. Bora Sok. No:1";
buyer.Ip = "85.34.78.112";
buyer.City = "Istanbul";
buyer.Country = "Turkey";
buyer.ZipCode = "34732";
request.Buyer = buyer;

Address shippingAddress = new Address();
shippingAddress.ContactName = "Jane Doe";
shippingAddress.City = "Istanbul";
shippingAddress.Country = "Turkey";
shippingAddress.Description = "100 Yıl. Sanayi Sok. No:1";
shippingAddress.ZipCode = "34742";
request.ShippingAddress = shippingAddress;

Address billingAddress = new Address();
billingAddress.ContactName = "Jane Doe";
billingAddress.City = "Istanbul";
billingAddress.Country = "Turkey";
billingAddress.Description = "Nidakule Göztepe, Merdivenköy Mah. Bora Sok. No:1";
billingAddress.ZipCode = "34742";
request.BillingAddress = billingAddress;

List<BasketItem> basketItems = new List<BasketItem>();
BasketItem firstBasketItem = new BasketItem();
firstBasketItem.Id = "BI101";
firstBasketItem.Name = "Binocular";
firstBasketItem.Category1 = "Collectibles";
firstBasketItem.Category2 = "Accessories";
firstBasketItem.ItemType = BasketItemType.PHYSICAL.ToString();
firstBasketItem.Price = "0.3";
basketItems.Add(firstBasketItem);

BasketItem secondBasketItem = new BasketItem();
secondBasketItem.Id = "BI102";
secondBasketItem.Name = "Game code";
secondBasketItem.Category1 = "Game";
secondBasketItem.Category2 = "Online Game Items";
secondBasketItem.ItemType = BasketItemType.VIRTUAL.ToString();
secondBasketItem.Price = "0.5";
basketItems.Add(secondBasketItem);

BasketItem thirdBasketItem = new BasketItem();
thirdBasketItem.Id = "BI103";
thirdBasketItem.Name = "Usb";
thirdBasketItem.Category1 = "Electronics";
thirdBasketItem.Category2 = "Usb / Cable";
thirdBasketItem.ItemType = BasketItemType.PHYSICAL.ToString();
thirdBasketItem.Price = "0.2";
basketItems.Add(thirdBasketItem);
request.BasketItems = basketItems;

Payment payment = Payment.Create(request, options);

if (payment.Status == "success")
{
    return View("Success");
}
```

```
        return View();
    }
```

İşlem bittikten sonra gidilecek olan Success sayfasını oluşturalım. Boş mesaj içeren bir sayfa olsun şimdilik.

Views>Card>**Success.cshtml**

```
<h3>İşlem Başarılı</h3>
```

Şu anda buraya kadar yazılan kodlar iyzipay üzerinden ödemeyi gönderecektir. Gerçek bir uygulama yapsak bunu görebilirdik. Bundan sonrası için form üzerinden (checkout) gönderilen

Sipariş Bilgilerinin Kaydedilmesi

Bir önceki derste Kredi Kartı bilgilerini kod içerisinde vermiştik. Oysa bu bilgileri kullanıcı formdan dolduracaktır. Dolayısı ile Form üzerinden kredi kartı bilgilerini alıp bunları Iyzipay sitesine götürecek kodlamayı yapalım. Kullanıcı kredi kartı bilgilerini girip ödeme alındıysa ve sonuç success olarak alınmışsa bu sipariş verilen ürünlerin kaydedilip sepetin (CardItems tablosunun) sıfırlanması gerekir. Ardından sipariş bilgilerininde Order ve OrderItems tablolarına kaydedilmiş olması gerekir.

Öncelikle OrderModel.cs sınıfı içindeki alanlara Kredi kartı ile ilgili alanlarımızı ekleyelim.

TS.WebUI.Models> **OrderModel.cs**

```
namespace TS.WebUI.Models
{
    public class OrderModel
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Address { get; set; }
        public string City { get; set; }
        public string Phone { get; set; }
        public string Note { get; set; }
        public string Email { get; set; }
        public string CardName { get; set; }
        public string CardNumber { get; set; }
        public string ExpirationMonth { get; set; }
        public string ExpirationYear { get; set; }
        public string Cvc { get; set; }

        public CardModel CartModel { get; set; }
    }
}
```

View sayfamızın yapısı içerisine Kredi kartı ile ilgili form elemanlarını koyalım.

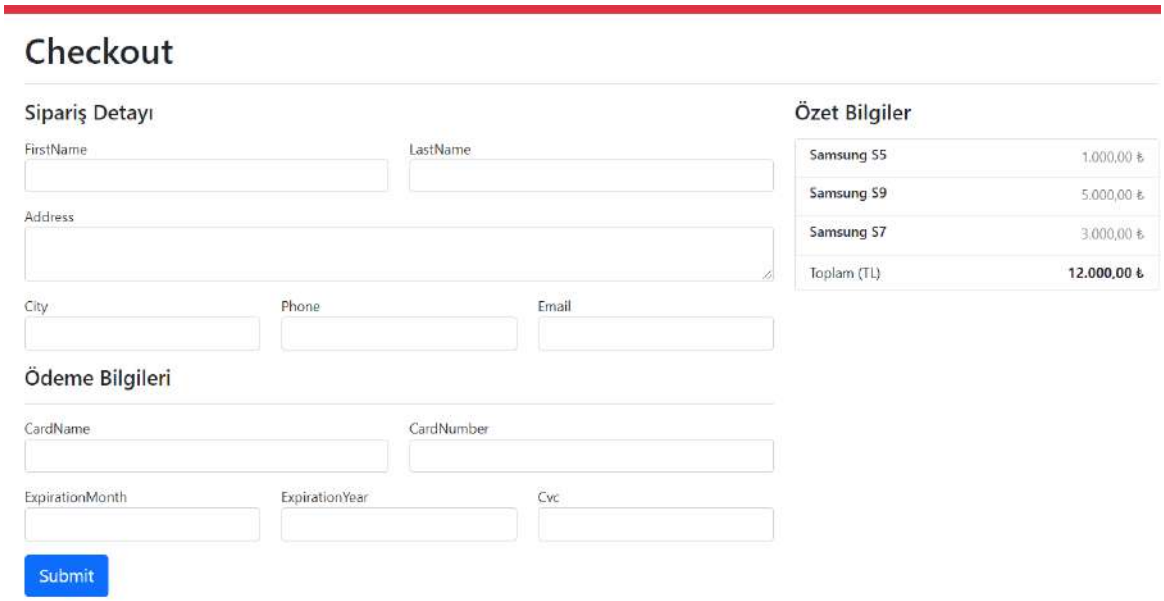
Views>Card>**Checkout.cshtml**

```
@model OrderModel

<h1>Sipariş ve Ödeme</h1>
<hr>
<div class="row">
    <div class="col-md-8">
        <h4 class="mb-3">Sipariş Detayı</h4>
        <form method="POST" asp-controller="Card" asp-action="Checkout">
            <input type="hidden" name="cardId" value="0">
            <div class="row mb-3">
```

```
                <div class="col-md-6">
                    <label asp-for="@Model.FirstName"></label>
                    <input asp-for="@Model.FirstName" class="form-control">
                </div>
                <div class="col-md-6">
                    <label asp-for="@Model.LastName"></label>
                    <input asp-for="@Model.LastName" class="form-control">
                </div>
            </div>
            <div class="mb-3">
                <label asp-for="@Model.Address"></label>
                <textarea asp-for="@Model.Address" class="form-control"></textarea>
            </div>
            <div class="row mb-3">
                <div class="col-md-4">
                    <label asp-for="@Model.City"></label>
                    <input asp-for="@Model.City" class="form-control">
                </div>
                <div class="col-md-4">
                    <label asp-for="@Model.Phone"></label>
                    <input asp-for="@Model.Phone" class="form-control">
                </div>
                <div class="col-md-4">
                    <label asp-for="@Model.Email"></label>
                    <input asp-for="@Model.Email" class="form-control">
                </div>
            </div>
            <h4 class="mb-3">Ödeme Bilgileri</h4>
            <hr class="mb-3">
            <div class="row mb-3">
                <div class="col-md-6">
                    <label asp-for="@Model.CardName"></label>
                    <input asp-for="@Model.CardName" class="form-control">
                </div>
                <div class="col-md-6">
                    <label asp-for="@Model.CardNumber"></label>
                    <input asp-for="@Model.CardNumber" class="form-control">
                </div>
            </div>
            <div class="row mb-3">
                <div class="col-md-4">
                    <label asp-for="@Model.ExpirationMonth"></label>
                    <input asp-for="@Model.ExpirationMonth" class="form-control">
                </div>
                <div class="col-md-4">
                    <label asp-for="@Model.ExpirationYear"></label>
                    <input asp-for="@Model.ExpirationYear" class="form-control">
                </div>
                <div class="col-md-4">
                    <label asp-for="@Model.Cvc"></label>
                    <input asp-for="@Model.Cvc" class="form-control">
                </div>
            </div>
            <button type="submit" class="btn btn-primary btn-lg btn-block">Submit</button>
        </form>
    </div>
    <div class="col-md-4">
        <h4 class="mb-3">
            <span>Özet Bilgiler</span>
        </h4>
        <ul class="list-group mb-3">

            @foreach (var item in Model.CardModel.CardItems)
            {
                <li class="list-group-item d-flex justify-content-between">
                    <div>
                        <h6>@item.Name</h6>
                    </div>
                    <span class="text-muted">@item.Price.ToString("c")</span>
                </li>
            }

            <li class="list-group-item d-flex justify-content-between">
                <span>Toplam (TL)</span>
                <strong>@Model.CardModel.TotalPrice().ToString("c")</strong>
            </li>
        </ul>
```

```
    </div>
</div>
```

Sayfa görünümüne bakalım.



Şimdi bu formdan gelen bilgileri post tipindeki action metod içerisinde işlenmesini yapalım (`public IActionResult Checkout(OrderModel model)`). Bu metod içerisinde kredi kartı bilgilerini Iyzipay götüren ve kredi kartı bilgilerini içeren kodları alt bir fonksiyon içerine alalım. Böylece Checkout) metodumuz daha sade gözüksün.

Ödeme başarılı ise paymetn bilgilerini ve conversation bilgilerini kaydedeceğiz. Ödeme alınıp kayıtlar yapıldıktan sorna kullanıcının kartını (sepetini) yeni alışverişler için temizliyor olmamız gerekir.

```
if (payment.Status == "success")
{
       SaveOrder(model, payment, userId);

       ClearCard(model.CardModel.CardId);
```

Metod içerisinde kullanıcının hangi ürünleri aldığını, view sayfasından getirebilirdik. Tabii orada özet bilgi şeklinde sağ tarafta tutuyoruz ve bu bilgiler form etiketleri dışındaydı. Formla birlikte gelmesi için hiddenlar içinde tutup getirebiliriz. Ama onun yerinde direk olarak kodlar içerisinden cardItems içerisinden alabiliriz. Onunla ilgili kodları şu şekilde oluşturabiliriz.

```
    model.CardModel = new CardModel()
    {
        CardId = card.Id,
        CardItems = card.CardItems.Select(i => new CardItemModel()
        {
            CardItemId = i.Id,
            ProductId = i.ProductId,
            Name = i.Product.Name,
            Price = (double)i.Product.Price,
            ImageUrl = i.Product.ImageUrl,
            Quantity = i.Quantity
        }).ToList()
    };
```

SaveOrder() alt metodu içerisinde sipariş bilgilerini kaydedelim. Order entity içerisine içerisine Payment (ödemeden) gelen bazı bilgileri de ekleyelim. Böylece siparişler incelenirken ödemenin nasıl yapıldığı hakkında da bilgileri tutmuş oluruz. Entity sınıfları içinde yapılan bir değişiklik direk veritabanındaki tabloları temsil ettiği için Migration üzerinden bu güncellemelerin yapılması gerekir. Daha sonra onu da yapacağız.

TS.Entity> **Order.cs**

```
using System;
using System.Collections.Generic;

namespace TS.Entity
{
    public class Order
    {
        public int Id { get; set; }
        public string OrderNumber { get; set; }
        public DateTime OrderDate { get; set; }
        public string UserId { get; set; }

        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Address { get; set; }
        public string City { get; set; }
        public string Phone { get; set; }
        public string Email { get; set; }
        public string Note { get; set; }
        public string PaymentId { get; set; }
        public string ConversationId { get; set; }
        public EnumPaymentType PaymentType { get; set; }
        public EnumOrderState OrderState { get; set; }
        public List<OrderItem> OrderItems { get; set; }
    }

    public enum EnumPaymentType
    {
        CreditCard = 0,
        Eft = 1
    }

    public enum EnumOrderState
    {
        waiting = 0,
        unpaid = 1,
        completed = 2
    }
}
```

Şimdi cardController içindeki alt fonksiyon olarak yazacağımız ve siparişleri veritabanına kaydedeceğimiz SaveOrder() metodunu oluşturalım. View sayfasından gelen bilgiler model içerisinde tutuluyordu. Bu metoda o bilgiler yine girişte alınacaktır. Sayfa içinde Order entitysinden yeni bir nesne türetelim bu nesnenin içerisine konulacak alanları hem modelden gelen bilgiler hemde kullanıcı ve payment den gelen bilgilerle dolduralım. OrderNumber olarak 6 rakamlı rastgele bir sayı üretebiliriz.

OrderState ve PaymentType özellikleri içinde Enum kullanımlarının nasıl yapıldığına dikkat edin. Bu iki özellik yukarıda Order entitisi içinde alt metod olarak tanımlanmıştı. Bu metodlar içinde atanan özelliklerin içerisine integer sayılar yüklendi. Böylece kod içerisinde özellikl adı ile anılırken veritabanına ise onun karşılığı olan sayı atanmış olmaktadır. Örneğin (`order.OrderState = EnumOrderState.completed;`) dersek OrderState= 2 değeri veritabanına atanmış olacaktır. Çünkü (`EnumOrderState.completed;`) den gelen değer 2 dir.

Metod girişinde kredi kartı ve ödeme ile ilgili olarak cart servis hizmeti veren firmanın nesnesi içinde tutulan bazı önemli bilgileri almış oluyoruz. Yani (`Payment payment`) ifadesinden bize Iyzco yu kullanıyorsak o servisin kullandığı bazı bilgileri almış oluyoruz. Bizde kendi veritabanımızda Order tablosu içerisinde ödeme ile ilgili olarak PaymentId ve ConversationId şeklinde iki bilgiyi tutmak istiyoruz. ConversationId şu anlama geliyor. Kişi hizmet aldığı servis üzerinden (Iyzipay) ödemelerini kontrol ederken hangi numarayı kullanacak, bunun için kullanılıyor. Yani kişiye PaymentId gösterilmiyor, ConversationId bilgisi ile kontrol etmesi isteniyor.

```
        order.PaymentId = payment.PaymentId;
        order.ConversationId = payment.ConversationId;
```

OrderDate (sipariş tarihi) olarak o anki tarih-saati atayabiliriz.

OrderItems lar yani ürünleri bir döngü ile model içinden alalım ve atayalım. Aslında bu metod içinde model dediğimiz nesne Order entitysidir. Bu entity nin içinde ise Card entitisi kullanılır. Onun da içinde her bir ürünü tutan CardItems tutulur. Dolayısı ile model içinde bilgilerin alınması (`model.CardModel.CardItems`) şeklinde olur. Tüm anlatılan kodlar aşağıdaki şekilde olacaktır.

model	model.CardModel	model.CardModel.CardItems
`public class OrderModel { public CardModel CardModel { get; set; }`	`public class CardModel { public List<CardItemModel> CardItems { get; set; }`	`public class CardItemModel {`

(`new TS.Entity.OrderItem()`) şeklinde geçen yerde OrderItem iki farkl yerde tanımlanmış ve çakışma vermiştir. O nedenle tam yolu yazarak tanımlanmıştır.

```
    private void SaveOrder(OrderModel model, Payment payment, string userId)
    {
        var order = new Order();

        order.OrderNumber = new Random().Next(111111, 999999).ToString();
        order.OrderState = EnumOrderState.completed;
        order.PaymentType = EnumPaymentType.CreditCard;
        order.PaymentId = payment.PaymentId;
        order.ConversationId = payment.ConversationId;
        order.OrderDate = new DateTime();
        order.FirstName = model.FirstName;
        order.LastName = model.LastName;
        order.UserId = userId;
        order.Address = model.Address;
        order.Phone = model.Phone;
        order.Email = model.Email;
        order.City = model.City;
        order.Note = model.Note;

        order.OrderItems = new List<Entity.OrderItem>();

        foreach (var item in model.CardModel.CardItems)
        {
            var orderItem = new TS.Entity.OrderItem()
            {
                Price = item.Price,
                Quantity = item.Quantity,
                ProductId = item.ProductId
            };
            order.OrderItems.Add(orderItem);
        }
        _orderService.Create(order);
    }
```

orderService için hem business hemde Data katmanındaki metodlarımızı oluşturalım.

TS.Business.Abstract> **IorderService.cs**

```
using System.Collections.Generic;
using TS.Entity;
```

```
namespace TS.Business.Abstract
{
    public interface IOrderService
    {
        void Create(Order entity);
    }
}
```

OrderManager içindeki concrete versiyonumuz. Burası Interface olarak IOrderService kullanacak. Bilgileri ise IOrderRepository üzerinden çekecek (inject edecek) . Bunun için constructor kullanımını en üste yazıyoruz.

TS.Business.Concrete> **OrderManager.cs**

```
using System.Collections.Generic;
using TS.Business.Abstract;
using TS.Data.Abstract;
using TS.Entity;

namespace TS.Business.Concrete
{
    public class OrderManager : IOrderService
    {
        private IOrderRepository _orderRepository;

        public OrderManager(IOrderRepository orderRepository)
        {
            _orderRepository = orderRepository;
        }
        public void Create(Order entity)
        {
            _orderRepository.Create(entity);
        }
    }
}
```

Data katmanındaki Interface metodumuz; Create metodumuz generik yapıyı temsil eden IRepository içinde bulunduğundan buraya tekrardan eklenmedi. Buraya eklenen metodlar orada genel olarak kullanıma uygun olmayan metodlar olacaktır.

TS.Data.Abstract> **IOrderRepository.cs**

```
using System.Collections.Generic;
using TS.Entity;

namespace TS.Data.Abstract
{
    public interface IOrderRepository : IRepository<Order>
    {

    }
}
```

Data katmanında bu Interface in dolu versiyonu ise; Burada yine Create metodu oluşturulmamıştır. Çünkü metodun tüm entiyler için kullanılabileceği yapı EfCoreGenericRepository içinde oluşturulmuştur. Bu generik repository ise bizden bir entity ve bir context isteyecektir. Bunlarda (`<Order, TSContext>`) şekinde verilmiştir. Buradaki EfCoreOrderRepository sınıfımızın metod yapısı IOrderRepository deki metod yapısında olacağından en sona (`IOrderRepository`) şeklinde onu da ekledik.

TS.Data.Concrete.EfCore> **EfCoreOrderRepository.cs**

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using TS.Data.Abstract;
```

```
using TS.Entity;

namespace TS.Data.Concrete.EfCore
{
    public class EfCoreOrderRepository : EfCoreGenericRepository<Order, TSContext>,
IOrderRepository
    {
    }
}
```

Startup içerisinde hangi Interfacein (abstract klasöründe) hangi dolu veriyonu çağıracağını vermemiz gerekiyor. Bunun için aşağıdaki iki satırıda eklemeliyiz.

```
public class Startup
{
       services.AddScoped<IOrderRepository, EfCoreOrderRepository>();
       services.AddScoped<IOrderService, OrderManager>();
```

Bu yapıyı CardController içinde kullanacağımızdan orada da bir injection işlemi yapmamız gerekir. Bunun içinde şu satırlar yazıldı.

```
    public class CardController : Controller
    {
        private ICardService _cardService;
        private UserManager<User> _userManager;
        private IOrderService _orderService;
        public CardController(ICardService cardService, UserManager<User> userManager,
IOrderService orderService )
        {
            _cardService = cardService;
            _userManager = userManager;
            _orderService = orderService;
        }
```

PaymentProcess() metodu içinde kredi kartı bilgilerini service gönderdiğimiz ve parametrelerini atadığımız bilgiler vardı. Bu parametrelerden bazılarını şu şekilde detaylandıralım. Apikey, SecretKey ve `BaseUrl` parametreleri bize servis sağlayacının verdiği anahtarlardır (Iyzipay). Bu bilgileri buraya yazmak yerine appSettings.json dosyasının içerisine yazmak ve oradan çekmek gerekir.

```
       options.ApiKey = "sandbox-RStpAR6sBz86RpFTuWPyF7hjh8c8g9E4";
       options.SecretKey = "sandbox-0zu0oxpXipwFsSGu1FrSofDVhTrWig2p";
       options.BaseUrl = "https://sandbox-api.iyzipay.com";
```

Buradaki ConversationId bilgisi bizim tarafımızdan üretilen rastgele bir sayı olsun. Bu sayı kredi kartından para çekildiğinde servis sağlayacının veritabanına kaydedilecektir. Bizde buraya kaydedilen sayının bizim tarafımızdan üretilip üretilmediğini kontrol edebilmemiz için (güvenlik amaçlı) bu sayıyı aynı zamanda kendi veritabanımıza da kaydetmeliyiz. Servis sağlayıcının veritabanındaki bu bilgi ile bizim veritabanımızdaki bu bilgi aynı değilse bir güvenlik sorunu var demektir. Bu işlem için 9 haneli rastgele bir sayı üretelim.

```
       request.ConversationId = new Random().Next(111111111, 999999999).ToString();
```

Fiyat bilgisini model içindeki CartModel altından alabiliriz. Eğer ürün fiyatının üzerine kargo bilgisi gibi ek ücretler var ise ödenecek ücret farklı olur. Bu kargo ücretlerinide başka bir tablodan alıp buraya üzerine eklenebilir.

```
       request.Price = model.CardModel.TotalPrice().ToString();
       request.PaidPrice = model.CardModel.TotalPrice().ToString();
```

Ödemede para birimi ve taksit sayısı gibi bilgilerede ihtiyaç olacaktır. Bunları da aşağıdaki şekilde alabiliriz.

```
       request.Currency = Currency.TRY.ToString();
```

```
request.Installment = 1;
```

SepetId si için random bir sayı üretilebilir.

```
request.BasketId = "B67832";
```

Kart bilgilerinide aşağıdaki şekilde formdan gelen bilgiler olarak alırız.

```
PaymentCard paymentCard = new PaymentCard();
paymentCard.CardHolderName = model.CardName;
paymentCard.CardNumber = model.CardNumber;
paymentCard.ExpireMonth = model.ExpirationMonth;
paymentCard.ExpireYear = model.ExpirationYear;
paymentCard.Cvc = model.Cvc;
paymentCard.RegisterCard = 0;
request.PaymentCard = paymentCard;
```

Diğer satın alanın kendi adres bilgileri (buyer), kargo adres bilgileri (shipping adress) (satın alan başkasına gönderebilir), faturanın gideceği yerin adres bilgileri (billing adress) bilgilieri de model içinden alınır.

Ürünleri döngü şeklinde alalım. Her birini BasketItem nesnesi üzerinden alarak BasketItems nesnesine ekleyelim.

```
List<BasketItem> basketItems = new List<BasketItem>();
BasketItem basketItem;

foreach (var item in model.CardModel.CardItems)
{
    basketItem = new BasketItem();
    basketItem.Id = item.ProductId.ToString();
    basketItem.Name = item.Name;
    basketItem.Category1 = "Telefon";
    basketItem.Price = item.Price.ToString();
    basketItem.ItemType = BasketItemType.PHYSICAL.ToString();
    basketItems.Add(basketItem);
}
request.BasketItems = basketItems;
```

Şimdi kodlarımızı denemeden önce Veritabanında Order tablosunda yapılan değişiklikleri güncellemek için Migrationımızı oluşturalım ve updata edelim.

TS.Data> dotnet ef migrations add UpdateOrderEntities --startup-project ../TS.WebUI --context TSContext

TS.Data> dotnet ef database update --startup-project ../TS.WebUI --context TSContext

Yapılan değişiklik sadece Order entitysi (tablosu) üzerinde olduğundan sadece onu güncelleyecektir. Diğer tablolara dokunmayacaktır .

Chekout ile ilgili tüm kodlarımızı verelim

```
[HttpPost]
public IActionResult Checkout(OrderModel model)
{
    if (ModelState.IsValid) //sayfadan gelen bilgiler doğrumu kontrol edelim.
    {
        var userId = _userManager.GetUserId(User);
        var card = _cardService.GetCardByUserId(userId);

        //Kullanıcının hangi ürünleri aldığını burada alıp CardItems lar içerisinde
tutuyoruz.
        model.CardModel = new CardModel()
        {
```

```
                    CardId = card.Id,
                    CardItems = card.CardItems.Select(i => new CardItemModel()
                    {
                        CardItemId = i.Id,
                        ProductId = i.ProductId,
                        Name = i.Product.Name,
                        Price = (double)i.Product.Price,
                        ImageUrl = i.Product.ImageUrl,
                        Quantity = i.Quantity
                    }).ToList()
                };

                //Ödeme işlemine gönderiyor..
                var payment = PaymentProcess(model);

                //ödemeden gelen bilgi başarılı ise siparişi veritabanına kaydedecek, ardından
Card (sepetteki bilgileri) sıfırlayacak.
                if (payment.Status == "success")
                {
                    SaveOrder(model, payment, userId);
                    ClearCard(model.CardModel.CardId);
                    return View("Success");
                }
                else //ödemeden gelen bilgi başarısızsa hata mesajını verecek.
                {
                    var msg = new AlertMessage()
                    {
                        Message = $"{payment.ErrorMessage}",
                        AlertType = "danger"
                    };

                    TempData["message"] = JsonConvert.SerializeObject(msg);
                }
            }
            return View(model); //gelen bilgiler valid değilse sayfaya geri gönderiyoruz.
        }
```

```
        private void ClearCard(int cardId)
        {
            _cardService.ClearCard(cardId);
        }
```

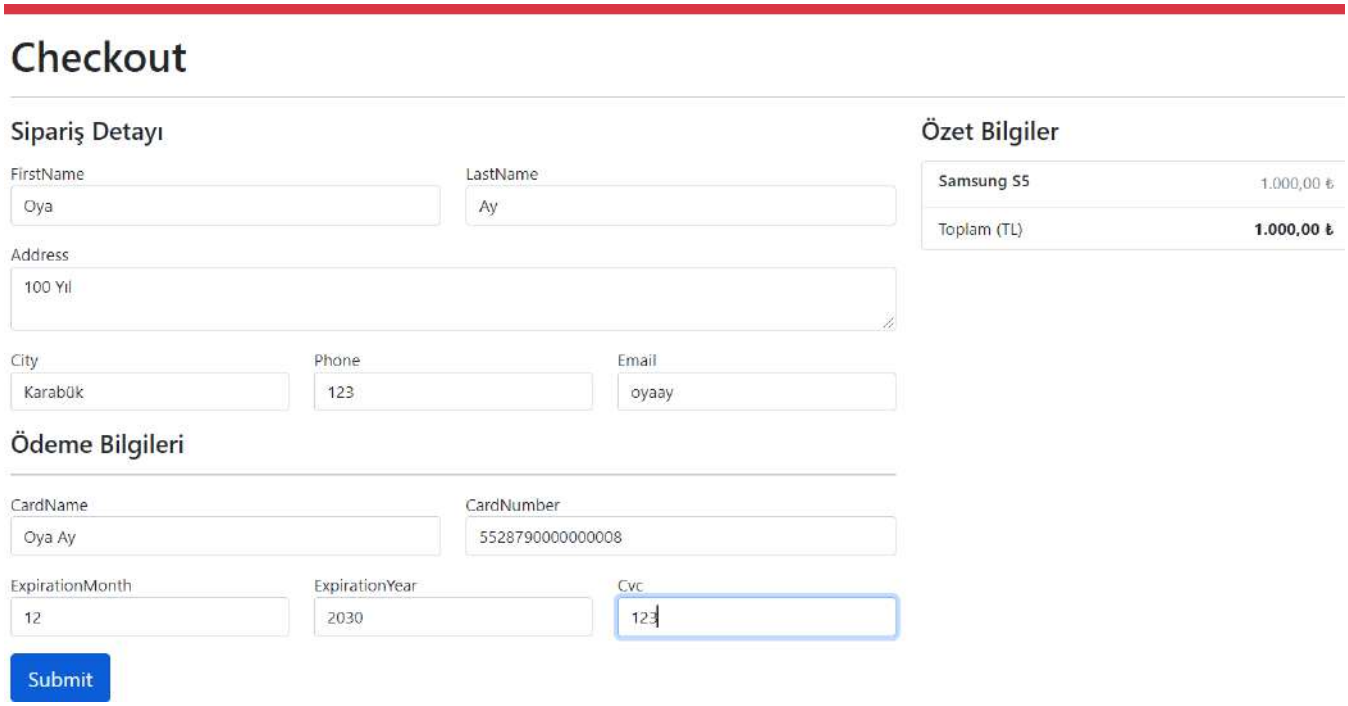
```
        private void SaveOrder(OrderModel model, Payment payment, string userId)
        {
            var order = new Order();

            order.OrderNumber = new Random().Next(111111, 999999).ToString();
            order.OrderState = EnumOrderState.completed;
            order.PaymentType = EnumPaymentType.CreditCard;
            order.PaymentId = payment.PaymentId;
            order.ConversationId = payment.ConversationId;
            order.OrderDate = new DateTime();
            order.FirstName = model.FirstName;
            order.LastName = model.LastName;
            order.UserId = userId;
            order.Address = model.Address;
            order.Phone = model.Phone;
            order.Email = model.Email;
            order.City = model.City;
            order.Note = model.Note;

            order.OrderItems = new List<Entity.OrderItem>();

            foreach (var item in model.CardModel.CardItems)
            {
```

```
                var orderItem = new TS.Entity.OrderItem()
                {
                    Price = item.Price,
                    Quantity = item.Quantity,
                    ProductId = item.ProductId
                };
                order.OrderItems.Add(orderItem);
            }
            _orderService.Create(order);
        }
```

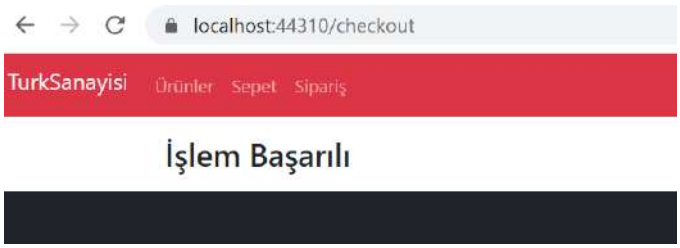
```
        private Payment PaymentProcess(OrderModel model)
        {
            Options options = new Options();
            options.ApiKey = "sandbox-e97Udlz12KM0IbfwpPMzTl50k3PZwEG7";
            options.SecretKey = "sandbox-oJnbmAEYGyRVVXZCnuVN56EhBgIlYfef";
            options.BaseUrl = "https://sandbox-api.iyzipay.com";

            CreatePaymentRequest request = new CreatePaymentRequest();
            request.Locale = Locale.TR.ToString();
            request.ConversationId = new Random().Next(111111111, 999999999).ToString();
            request.Price = model.CardModel.TotalPrice().ToString();
            request.PaidPrice = model.CardModel.TotalPrice().ToString(); //Kargo bilgisi varsa
buraya eklenmeli.
            request.Currency = Currency.TRY.ToString();
            request.Installment = 1;
            request.BasketId = "B67832";
            request.PaymentChannel = PaymentChannel.WEB.ToString();
            request.PaymentGroup = PaymentGroup.PRODUCT.ToString();

            PaymentCard paymentCard = new PaymentCard();
            paymentCard.CardHolderName = model.CardName;
            paymentCard.CardNumber = model.CardNumber;
            paymentCard.ExpireMonth = model.ExpirationMonth;
            paymentCard.ExpireYear = model.ExpirationYear;
            paymentCard.Cvc = model.Cvc;
            paymentCard.RegisterCard = 0;
            request.PaymentCard = paymentCard;

            // *** Formda deneme için kullanılacak kart numaraları.Iyzipay bu kartları tanıyor
***
            //paymentCard.CardNumber = "5528790000000008";
            //paymentCard.ExpireMonth = "12";
            //paymentCard.ExpireYear = "2030";
            //paymentCard.Cvc = "123";

            Buyer buyer = new Buyer();
            buyer.Id = "BY789"; //UserId buraya atanmalı. Bilgi modele eklenmeli.
            buyer.Name = model.FirstName;
            buyer.Surname = model.LastName;
            buyer.GsmNumber = "+905350000000";
            buyer.Email = "email@email.com";
            buyer.IdentityNumber = "74300864791";
            buyer.LastLoginDate = "2015-10-05 12:43:35";
            buyer.RegistrationDate = "2013-04-21 15:12:09";
            buyer.RegistrationAddress = "100 Yıl, No:1";
            buyer.Ip = "85.34.78.112";
            buyer.City = "Karabük";
            buyer.Country = "Turkey";
            buyer.ZipCode = "78732";
            request.Buyer = buyer;

            Address shippingAddress = new Address();
            shippingAddress.ContactName = "Ali Su";
```

```
            shippingAddress.City = "Karabük";
            shippingAddress.Country = "Turkey";
            shippingAddress.Description = "100 Yıl, No:1";
            shippingAddress.ZipCode = "78732";
            request.ShippingAddress = shippingAddress;

            Address billingAddress = new Address();
            billingAddress.ContactName = "Ali Su";
            billingAddress.City = "Karabük";
            billingAddress.Country = "Turkey";
            billingAddress.Description = "100 Yıl, No:1";
            billingAddress.ZipCode = "78742";
            request.BillingAddress = billingAddress;

            List<BasketItem> basketItems = new List<BasketItem>();
            BasketItem basketItem;

            foreach (var item in model.CardModel.CardItems)
            {
                basketItem = new BasketItem();
                basketItem.Id = item.ProductId.ToString();
                basketItem.Name = item.Name;
                basketItem.Category1 = "Telefon";
                basketItem.Price = item.Price.ToString();
                basketItem.ItemType = BasketItemType.PHYSICAL.ToString();
                basketItems.Add(basketItem);
            }
            request.BasketItems = basketItems;

            return Payment.Create(request, options);
        }
```

Kodlarımızı deneyelim.



Formu gönderince kredi kartı uygulamasının başarılı olduğunu görebiliyoruz.



Iyzico nun sitesinden hesabımıza girip baktığımızda deneme modülünde (sandbox) bir ürünün parasının yattığını görebiliyoruz (https://sandbox-merchant.iyzipay.com/transactions?currencyCode=TRY).

Ödeme Sonrası Sepetteki Ürün Bilgilerinin Silinmesi

Kişi siparişlerinin (orders) parasını ödedikten sonra (chekout) sepetindeki ürünlerin sıfırlanması gerekir. Bu işlemi yapalım. Eğer ödeme onaylanmışsa (`if (payment.Status == "success")`) iki temel işlemi yapmalıyız. Birisi siparişleri yani parası alınmış ürünleri sipariş tablosuna kaydedip o ürünleri kargoya vermek üzere bilgilerini saklamalıyız. Bir diğeri ise yeni ürünler alması için Card tını (sepetini) boşaltmalıyız.

```
if (payment.Status == "success")
{
    SaveOrder(model, payment, userId);
    ClearCard(model.CardModel.CardId);
```

ClerdCard işlemine ait Business ve Data katmanındaki sınıf ve metodlarımızı oluşturalım. Biz burada Card entity sini silmeyeceğiz. Buna bağlı CardItems leri sileceğiz. Bunun için bize CardId bilgisi olsa yeterlidir. CardItems tablosu içinden CardId ye bağlı satırları sileriz.

```
public interface ICardService
{

    void ClearCard(int cartId);
```

```
public class CardManager : ICardService
{
    public void ClearCard(int cardId)
    {
        _cardRepository.ClearCard(cardId);
```

```
        }
```

```
    public interface ICardRepository : IRepository<Card>
    {
        void ClearCard(int cartId);
```

```
    public class EfCoreCardRepository : EfCoreGenericRepository<Card, TSContext>,
ICardRepository
    {
        public void ClearCard(int cardId)
        {
            using (var context = new TSContext())
            {
                var cmd = @"delete from CardItems where CardId=@p0";
                context.Database.ExecuteSqlRaw(cmd, cardId);
            }
        }
```

Kodları denediğimizde ödeme yaparsak sonrasında kişinin sepetindeki bilgilerin temizlendiğini göreceğiz.

Müşteri ve Yönetim Sayfasında Siparişlerin Listelenmesi

Siteden alışveriş yapan tüm kullanıcıların vermiş oldukları siparişlerin yönetim sayfasından (Admin sayfasından) listelenebilmesi gerekir. Kimler sipariş verdi, siparişlerin kargoya verilip verilmediği gibi süreci gösteren işlemlerinde yapılabilmesi gerekir. Eğer kullanıcının Role özelliği Admin ise o zaman tüm siparişleri listeleyebiliriz. Listelediğimiz siparişler üzerinde bazı değişiklikleri yapabiliyor olmamız gerekir. Tedarik sürecinde, Kargoya verildi, tamamlandı gibi. Kullanıcı sayfasında ise bu işlemler basit olarak listelenmeli. Süreçler ise sadece bilgilendirme amacıyla verilmeli değişiklik yapılamamalı.

Burada uygulamayı basit tutmak için önce Kullanıcının sayfasında siparişlerini listeleyeceğiz. Benzer şekilde yönetim sayfasında da bu işlemleri yapacağız.

İlk olarak Startup içinden linkimizi (Route desenimizi) düzenleyelim. Order ile ilgili işlemleri OrderController içinden yönetelim.

```
    endpoints.MapControllerRoute(
        name: "orders",
        pattern: "orders",
        defaults: new { controller = "Order", action = "GetOrders" }
    );
```

NavBar üzerindeki linkimizi de buna uygun düzenleyelim. Kullanıcı giriş yapmışsa (login olmuşsa) kendisine ait orderleri (siparişleri) listeleyebilsin.

```
    @if (User.Identity.IsAuthenticated)
    {
        <li class="nav-item">
            <a href="/orders" class="nav-link">Siparişler</a>
        </li>
```

Aynı işlemleri yönetim sayfasında yapabilmek için NavBar içine ikinci bir link daha eklememize gerek yok. Çünkü kullanıcı Idsi metodlarımıza geldiğinde Role özelliğine bakarsak Admin dışında bir role sahipse sadece o kullanıcıya ait order ları listeleyebiliriz.

Öncelikle OrderController sınıfımızı oluşturalım Bu sınıf içinde GerOrders() şeklinde metodumuzu oluşturalım. Ve bu metodun bilgileri getirirken kullanacağı Business ve Data katmanlarındaki sınıf yapılarını oluşturalım. En baştan itibaren alırsak Entity katmanındaki sınıf yapısınıda verelim. Bu yapı veritabanındaki tablolara karşılık gelmektedir.

TS.Entity> **Order.cs**

```
using System;
using System.Collections.Generic;

namespace TS.Entity
{
    public class Order
    {
        public int Id { get; set; }
        public string OrderNumber { get; set; }
        public DateTime OrderDate { get; set; }
        public string UserId { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Address { get; set; }
        public string City { get; set; }
        public string Phone { get; set; }
        public string Email { get; set; }
        public string Note { get; set; }
        public string PaymentId { get; set; }
        public string ConversationId { get; set; }
        public EnumPaymentType PaymentType { get; set; }
        public EnumOrderState OrderState { get; set; }
        public List<OrderItem> OrderItems { get; set; }
    }

    public enum EnumPaymentType
    {
        CreditCard = 0,
        Eft = 1
    }

    public enum EnumOrderState
    {
        waiting = 0,
        unpaid = 1,
        completed = 2
    }
}
```

TS.Business.Abstract> **IOrderService.cs**

```
using System.Collections.Generic;
using TS.Entity;

namespace TS.Business.Abstract
{
    public interface IOrderService
    {
        void Create(Order entity);
```

```
        List<Order> GetOrders(string userId);
    }
}
```

TS.Business.Concrete> **OrderManager.cs**

```
using System.Collections.Generic;
using TS.Business.Abstract;
using TS.Data.Abstract;
using TS.Entity;

namespace TS.Business.Concrete
{
    public class OrderManager : IOrderService
    {
        private IOrderRepository _orderRepository;

        public OrderManager(IOrderRepository orderRepository)
        {
            _orderRepository = orderRepository;
        }
        public void Create(Order entity)
        {
            _orderRepository.Create(entity);
        }

        public List<Order> GetOrders(string userId)
        {
            return _orderRepository.GetOrders(userId);
        }
    }
}
```

TS.Data.Abstract> **IOrderRepository.cs**

```
using System.Collections.Generic;
using TS.Entity;

namespace TS.Data.Abstract
{
    public interface IOrderRepository : IRepository<Order>
    {
        List<Order> GetOrders(string userId);
    }
}
```

Aşağıda orders ları getirirken önce bu tablodan OrdersItems tablosuna geçiş yapıyor (`.Include(i => i.OrderItems)`) sonra ondan da Product tablosuna geçiş yapıyor ve tüm bilgileri alıyor. Alınan bilgileri daha sonra sorgulamada kullanmak için AsQueryable olarak işaretliyor. Böylece bu üstteki satır ile tüm orderları getirmiş oluyor.

Bir alt satırlarda ise eğer kullanıcı Id si metoda gelmişse bu isteği bir customer istiyor demektir. Oda kendi orderlarını görebilir. O zaman bir önceki satırlarda Queryable yapılan orderlar içinden bir daha sorgulama ile sadece bu müşteriye ait olan orderler yükleniyor. Ve geriye bu liste gönderiliyor.

TS.Data.Concrete.EfCore> **EfCoreOrderRepository.cs**

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using TS.Data.Abstract;
using TS.Entity;

namespace TS.Data.Concrete.EfCore
{
    public class EfCoreOrderRepository : EfCoreGenericRepository<Order, TSContext>,
IOrderRepository
    {
        public List<Order> GetOrders(string userId)
        {
            using (var context = new TSContext())
            {

                var orders = context.Orders
                                    .Include(i => i.OrderItems)
                                    .ThenInclude(i => i.Product)
                                    .AsQueryable();

                if (!string.IsNullOrEmpty(userId))
                {
                    orders = orders.Where(i => i.UserId == userId);
                }

                return orders.ToList();
            }
        }
    }
}
```

TS.WebUI.Controllers> **OrderController.cs**

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using System.Linq;
using TS.Business.Abstract;
using TS.WebUI.Identity;
using TS.WebUI.Models;

namespace TS.WebUI.Controllers
{
    public class OrderController : Controller
    {
        //private ICardService _cardService;
        private UserManager<User> _userManager;
        private IOrderService _orderService;
        public OrderController(UserManager<User> userManager, IOrderService orderService)
        {
            _userManager = userManager;
            _orderService = orderService;
        }

        public IActionResult GetOrders()
        {
           var userId = _userManager.GetUserId(User);
            var roleUserId = userId;
            if (User.IsInRole("Admin"))//Kullanıcı Admin ise UserId=null olacak götürecek ve
tüm Orderları listeleyecek.
            {
```

```
                roleUserId = null;
            }
            var orders = _orderService.GetOrders(roleUserId);
            var orderListModel = new List<OrderListModel>();
            OrderListModel orderModel;            foreach (var order in orders)
            {
                orderModel = new OrderListModel();

                orderModel.OrderId = order.Id;
                orderModel.OrderNumber = order.OrderNumber;
                orderModel.OrderDate = order.OrderDate;
                orderModel.Phone = order.Phone;
                orderModel.FirstName = order.FirstName;
                orderModel.LastName = order.LastName;
                orderModel.Email = order.Email;
                orderModel.Address = order.Address;
                orderModel.City = order.City;
                orderModel.OrderState = order.OrderState;
                orderModel.PaymentType = order.PaymentType;

                orderModel.OrderItems = order.OrderItems.Select(i => new OrderItemModel()
                {
                    OrderItemId = i.Id,
                    Name = i.Product.Name,
                    Price = (double)i.Price,
                    Quantity = i.Quantity,
                    ImageUrl = i.Product.ImageUrl
                }).ToList();

                orderListModel.Add(orderModel);
            }
            return View("Order", orderListModel);
        }
    }
}
```

GetOrders() metodumuz içinden alınan bilgiler orderListModel içinde sayfaya taşınacaktır. Sayfamızın adını "Orders" verelim. İlla "GetOrders" şeklinde olması gerekmez. View sayfamızın adı metod isminden farklı ise parantezler içinde View sayfamızın adını direk yazdığımıza da dikkat edelim. View sayfamızı Views>Order>order.cshmtl şeklinde yol içine aşağıdaki şekilde oluşturalım.

Views>Order>**Order.cshtml**

```
@model List<OrderListModel>

<h1>Siparişler</h1>
<hr>

@foreach (var order in Model)
{
    <table class="table table-bordered table-sm mb-3">
        <thead class="bg-primary">
            <tr>
                <td colspan="2">Order Id: #@order.OrderNumber</td>
                <th>Price</th>
                <th>Quantity</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var orderItem in order.OrderItems)
            {
```

```
                <tr>
                    <td><img src="~/img/@orderItem.ImageUrl" alt="" width="50"></td>
                    <td>
                        @orderItem.Name
                    </td>
                    <td>@orderItem.Price</td>
                    <td>@orderItem.Quantity</td>
                </tr>
            }
        </tbody>
        <tfoot>
            <tr>
                <td colspan="2">Customer Name</td>
                <td>@order.FirstName @order.LastName</td>
                <td rowspan="6">Total: @order.TotalPrice().ToString("c")</td>
            </tr>
            <tr>
                <td colspan="2">Address: </td>
                <td>@order.Address</td>
            </tr>
            <tr>
                <td colspan="2">Email: </td>
                <td>@order.Email</td>
            </tr>
            <tr>
                <td colspan="2">Phone: </td>
                <td>@order.Phone</td>
            </tr>
            <tr>
                <td colspan="2">Order Status: </td>
                <td>@order.OrderState</td>
            </tr>
            <tr>
                <td colspan="2">Payment Type </td>
                <td>@order.PaymentType</td>
            </tr>
        </tfoot>
    </table>
}
```

Sayfaya bilgileri taşımada kullandığımız OrderListModel.cs yapısı ise şu şekilde olacaktır.

TS.WebUI.Models> **OrderListModel.cs**

```
using System;
using System.Collections.Generic;
using System.Linq;
using TS.Entity;

namespace TS.WebUI.Models
{
    public class OrderListModel
    {
        public int OrderId { get; set; }
        public string OrderNumber { get; set; }
        public DateTime OrderDate { get; set; }
        public string UserId { get; set; }

        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Address { get; set; }
        public string City { get; set; }
        public string Phone { get; set; }
        public string Email { get; set; }
```

```
        public string Note { get; set; }
        public EnumPaymentType PaymentType { get; set; }
        public EnumOrderState OrderState { get; set; }
        public List<OrderItemModel> OrderItems { get; set; }

        public double TotalPrice()
        {
            return OrderItems.Sum(i => i.Price * i.Quantity);
        }
    }

    public class OrderItemModel
    {
        public int OrderItemId { get; set; }
        public double Price { get; set; }
        public string Name { get; set; }
        public string ImageUrl { get; set; }
        public int Quantity { get; set; }
    }
}
```

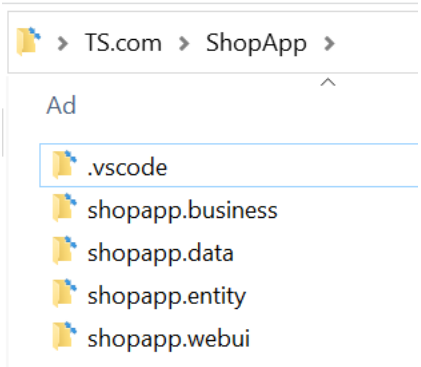
Denediğimiz zaman kodların çalıştığını görürüz. Admin olarak denediğimizde tüm Orderlar gelir. Müşteri olarak

PROJENİN DÜZENLENMESİ VE SİTENİN YAYINLANMASI

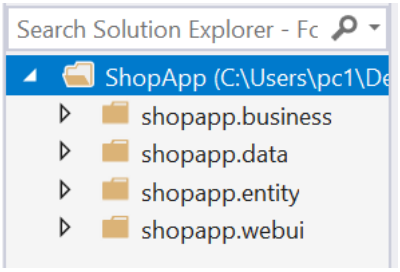
Proje Dosyalarının Çalışır Hale Getirilmesi

Proje dosyalarının indirilmesi .net core kurulması ve npm modülün yüklenmesi

Proje dosyaları biribirinden bağımsız bir klasör yapısı içinde iken çalışır durumda değildir. Bunların birbirini görecek şekilde proje yapısına dönüştürülmesi gerekir. Proje dosyalarımızı aşağıdaki klasör yapısı şeklinde bir kaynaktan indirdiğimiz varsayalım.



Buraya kaydedilen dosyaları Visual Studio içinden Open>Open folder kısmında açarak aşağıdaki görüntüyü elde edelim.

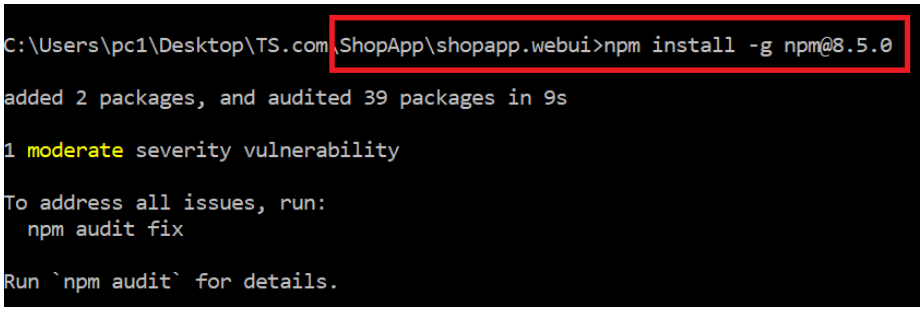


WebUI içindeki Package.json dosyasını açtığımızda bazı yüklü olan programları versiyonları ile görmekteyiz. Fakat bize gerekli olan nodemodulu burada görmüyoruz. Öncelikle bunu proje içerisine yüklemeliyiz.

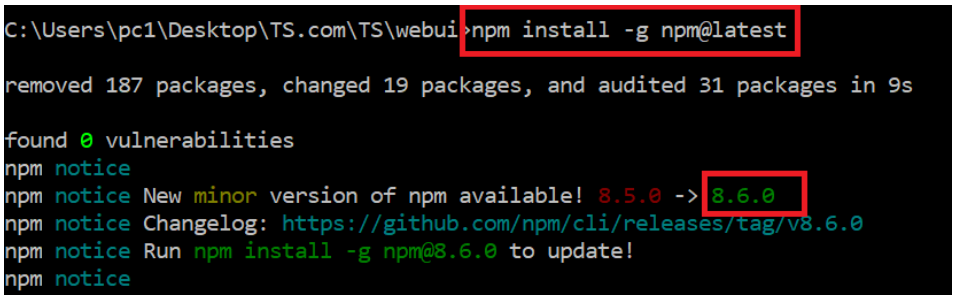
Console üzerinden webui projesine geçip

npm install -g npm@8.5.0 (-g işareti global yükleneceğini gösterir. En son versiyonu yüklemek için npm install -g npm@latest şeklinde yazılabilir).

şeklinde versiyon belirterek paketleri yükleyelim.



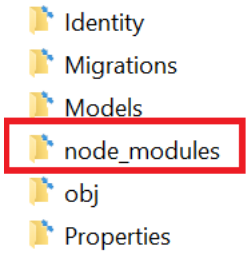
En son versiyonu başka bir zaman yüklemeyi denediğimizde aşağıdaki gibi bir görüntü aldık. Ama yüklemedi. Uyarıda npm install -g npm@8.6.0 şeklinde yazın diyor.



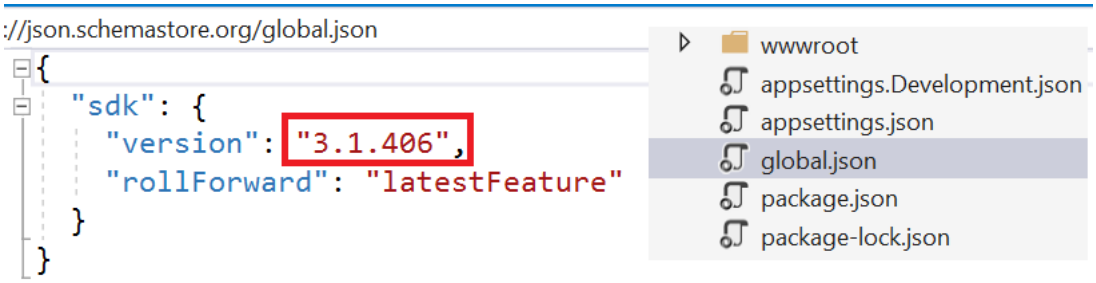
npm install -g npm@8.6.0 Denenince bu durumda yükledi.

```
C:\Users\pc1\Desktop\TS.com\TS\webui>npm install -g npm@8.6.0

changed 14 packages, and audited 201 packages in 7s

10 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Klasör yapısı içinde node modülün yüklendiğini görüyoruz. (Not sonraki denemelerde henüz node_modulus gözükmedi. Projeyi VS içinde çalışır hale getirip ilk run elde edilene kadar oluşmuyor. Aşağıda gelecek.)



Global.json dosyasını açıp içinde asp core un hangi versiyonunu kullandığımızı öğrenmemiz gerekir. Şu anda 3.1 versiyonunu kullanıyoruz. 3.1 den sonraki sayılar önemli değildir.



Biz şuanda 3.1 versiyonu kullanıyoruz. Dolayısı ile ".net core" versiyonu olarak 3.1 Sdk yı indiriyor olmamız gerekir.

https://dotnet.microsoft.com/en-us/download/dotnet

(Sonradan başka bir uygulamada **6.0.201** şu adresten yükledim. https://download.visualstudio.microsoft.com/download/pr/1eb43f77-61af-40b0-8a5a-6165724dca60/f12aac6d4a907b4d54f5d41317aae0f7/dotnet-sdk-6.0.201-win-x64.exe)

Supported versions

Version	Status	Latest release	Latest release date	End of support
.NET 6.0 (recommended)	LTS	6.0.2	February 8, 2022	November 08, 2024
.NET 5.0	Current	5.0.14	February 8, 2022	May 08, 2022
.NET Core 3.1	LTS	3.1.22	December 14, 2021	December 03, 2022

Bilgisayarımızda kurulu olan dotnet versiyonu kontrol edelim. Herhangi bir konumda iken aşağıdaki komutu yazalım.

dotnet –-version (yani dotnet asp.net core olmuş oluyor)

```
C:\Users\pc1\Desktop\TS.com\ShopApp\shopapp.webui>dotnet --version
3.1.407
```

Bilgisayarımızda kurulu tüm versiyonları görmek içinde aşağıdaki komutu kullanabiliriz.

dotnet --list-sdks

```
C:\Users\pc1>dotnet  --list-sdks
3.1.101 [C:\Program Files\dotnet\sdk]
3.1.407 [C:\Program Files\dotnet\sdk]
```

Eğer bilgisayarımıza 5.0 yada 6.0 versiyonlarıda kurarsak o seviyedeki projeleride çalıştırabiliriz.

webUI>global.json içindeki dosyada geçen "latestFeature" ifadesi sonraki versiyonlarında bu proje için kullanılabileceğini gösterir.

```
{
  "sdk": {
    "version": "3.1.406",
    "rollForward": "latestFeature"
  }
}
```

Öncelikle Shopapp.webUI projemizi derleyelim (build edelim) bir problem var mı kontrol edelim.

Shopapp.WebUI> dotnet build

```
C:\Users\pc1\Desktop\TS.com\ShopApp\shopapp.webui>dotnet build
.NET için Microsoft (R) Build Engine sürüm 16.7.2+b60ddb6f4
Telif Hakkı (C) Microsoft Corporation. Tüm hakları saklıdır.

  Geri yüklenecek projeler belirleniyor...
  C:\Users\pc1\Desktop\TS.com\ShopApp\shopapp.webui\shopapp.webui.csproj geri yüklendi (3,91 sec içinde).
  C:\Users\pc1\Desktop\TS.com\ShopApp\shopapp.entity\shopapp.entity.csproj geri yüklendi (6,86 sec içinde).
  C:\Users\pc1\Desktop\TS.com\ShopApp\shopapp.data\shopapp.data.csproj geri yüklendi (6,89 sec içinde).
  C:\Users\pc1\Desktop\TS.com\ShopApp\shopapp.business\shopapp.business.csproj geri yüklendi (6,9 sec içinde).
  shopapp.entity -> C:\Users\pc1\Desktop\TS.com\ShopApp\shopapp.entity\bin\BNB\Debug\netstandard2.0\shopapp.entity.dll
  shopapp.data -> C:\Users\pc1\Desktop\TS.com\ShopApp\shopapp.data\bin\BNB\Debug\netstandard2.0\shopapp.data.dll
  shopapp.business -> C:\Users\pc1\Desktop\TS.com\ShopApp\shopapp.business\bin\BNB\Debug\netstandard2.0\shopapp.business
.dll
  shopapp.webui -> C:\Users\pc1\Desktop\TS.com\ShopApp\shopapp.webui\bin\BNB\Debug\netcoreapp3.1\shopapp.webui.dll
  shopapp.webui -> C:\Users\pc1\Desktop\TS.com\ShopApp\shopapp.webui\bin\BNB\Debug\netcoreapp3.1\shopapp.webui.Views.dll


Oluşturma başarılı oldu.
    0 Uyarı
    0 Hata

Geçen Süre 00:00:26.01
```

Bir hata bulunmadı. Webui projesi içindeyken dotnet run diyerek projeyi çalıştıralım.

dotnet run

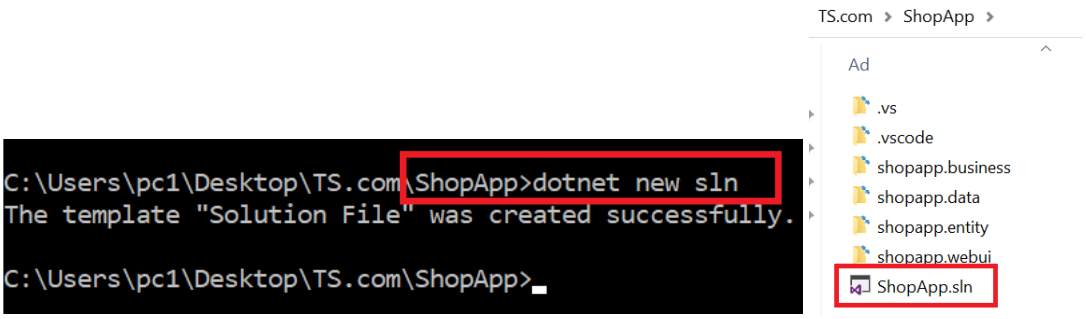
```
C:\Users\pc1\Desktop\TS.com\ShopApp\shopapp.webui>dotnet run
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\pc1\Desktop\TS.com\ShopApp\shopapp.webui
```

Projenin VS içinde çalıştırılması için Sln dosyasının oluşturulması

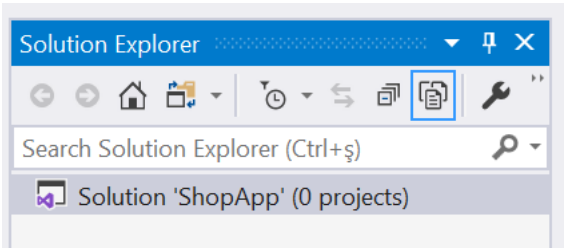
Yukarıdaki ekrana göre projemiz çalışabilir durumdadır ama Visual Studio kullanıyorsak sln dosyasını hazırlayıp öyle çalıştırabiliriz. Visual Studio denenirken bir sln (solution dosyasının) olması gerekir. Şimdi bu dosyayı oluşturalım.

Uygulamanın ana dizinindeyken yani solution dizinindeyken (shopapp altında)sln dosyasını oluşturalım. Bu işlemi yine komut satırından yapalım. Şu komutu yazdığımızda sln dosyası oluşacaktır.

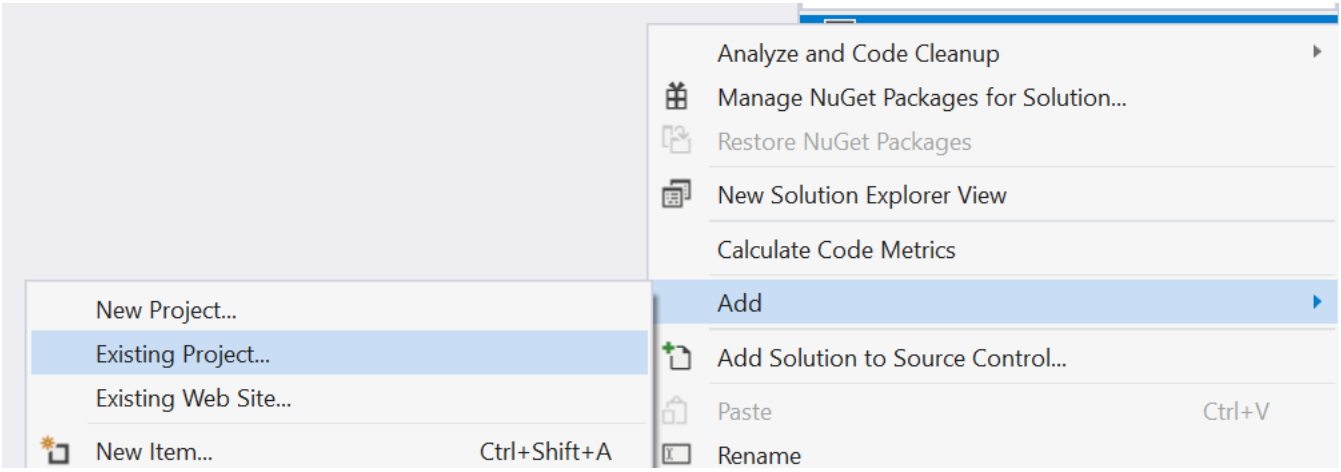
Shopapp> dotnet new sln



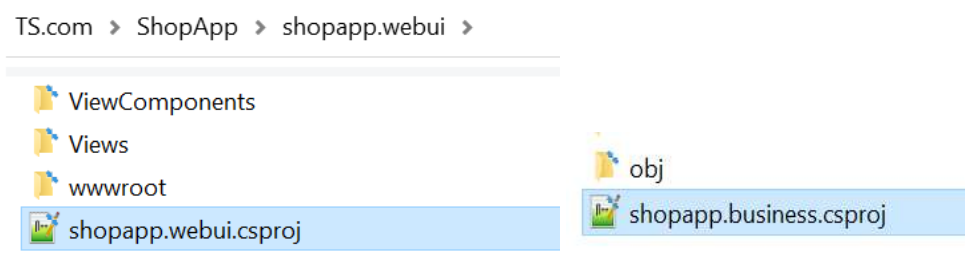
Visual Studio dan çıkıp, sln dosyasına tıklayarak solutionı açtığımızda içerisinde projelerin olmadığını görüyoruz. Explorer kısmı boş geldi.



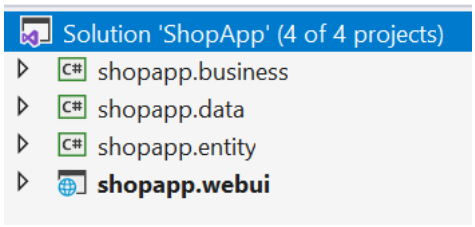
Solution içine proje eklemek için üzerien sağ tuşana tıklayıp Existing Project seçeneğinden projelerimizi ekleyelim.



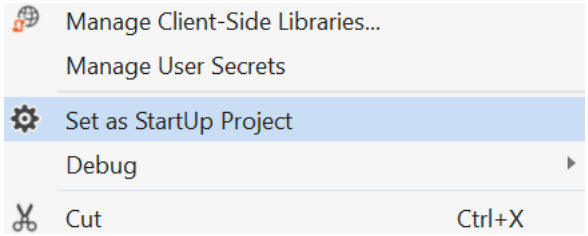
Her bir proje klasörünü açtığımızda içindeki csproj dosyalarını seçerek projeyi solution içine eklemiş oluyoruz.



Bu şekilde solution içindeki 4 adet projeyi de ekledik.



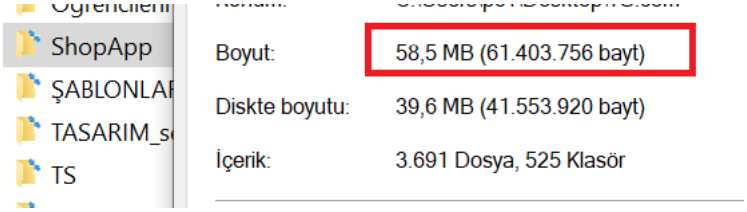
Tüm projeleri ekledikten sonra çalıştırılabilir proje olarak da WebUI projesini seçelim. Bunun için WebUI üzerine sağ tuşa tıklayıp Startup Project seçeneğini işaretleyelim. Böylece solution çalıştırıldığında webui projesi çalışacaktır.



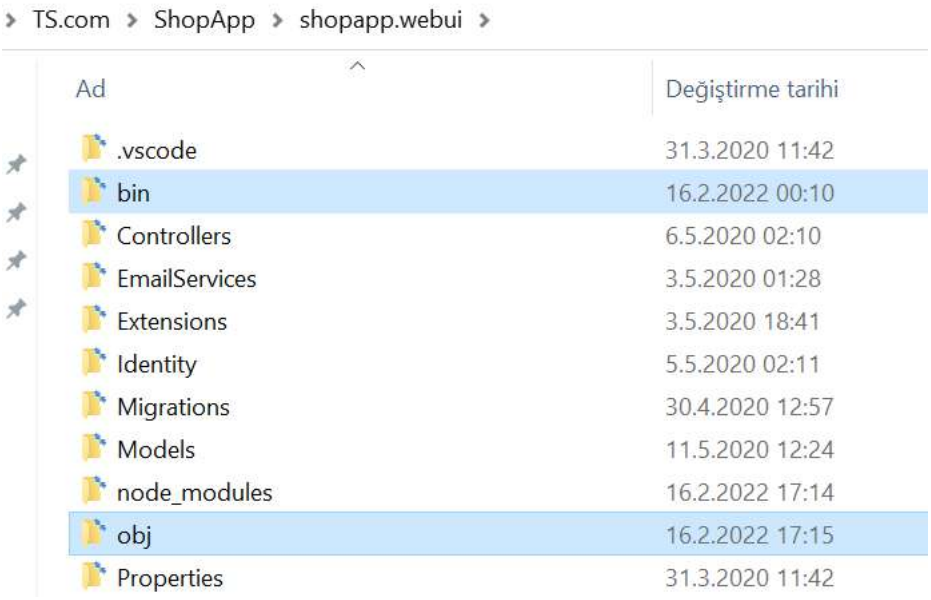
Artık F5 tıklayıp solution çalıştırmaya başladığımızda projemizin çalıştığını görmekteyiz.

Proje Dosyalarını Taşırken Boyutunu Düşürme

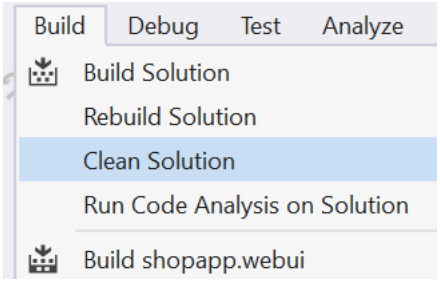
Projemizin ne kadar yer kapladığına baktığımızda 58 MB yer kabladığını görüyoruz.



Projenin boyutunu düşürmek için örneğin bir yere taşımak istediğimizde bin ve obj klasörlerini kaldırıp boyutunu düşürebiliriz. Proje taşındığı yerde tekrar çalıştırıldığında bu klasörler oluşacaktır. Ayrıca node_modules klasörünü de kaldırabiliriz. Fakat taşındığı yerde yeniden kurmamız gerekir (yukarılarda kurulumu anlatılmıştı).



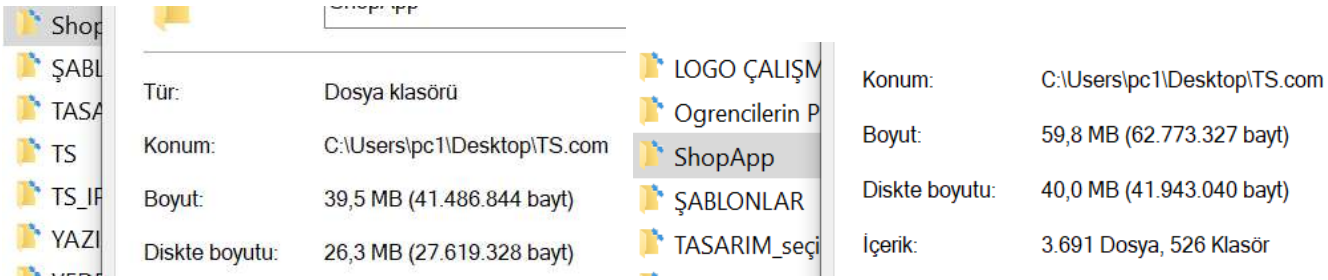
Başka bir işlem olarak Build altında "Clean Solution" seçeneğini kullanabiliriz.



Yada komut satırında ana dizindeyken (shopapp) Aşağıdaki komutu çalıştırabiliriz.

Shopapp> dotnet clean

Bunlardan VS içinde Build> Clean solution komutunu denediğimizde 39.5 MB ta düştüğünü görüyoruz. Solution tekrar çalıştırda ise 59.8 MB çıktığını görüyoruz.



Bağlantı Cümlesinin AppSettings dosyasına alınması.

Veritabanı bağlantı cümlesi, yani veritabanın adı ve hangi tür olduğunu gösteren cümlelerimiz Data projesi içinde TSContext içinde verilmişti.

```
    public class TSContext : DbContext
    {
        //Veritabanın adı ve tipi burada veriliyor.
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlite("Data Source=TSDb");
        }
```

Veritabanı adresini bu şekilde data projesi içinde vermemiz projenin ilerleyen aşamalarında kullanışlı olmayabilir. Solutiona çeşitli projeleri ekleyebiliriz. Nasılki burada Web projesi eklediysek ileriki aşamalarda WebApi projeside ekleyebiliriz. Dolayısı ile hangi projeyi eklersek ekleyelim, kullanacağımız veritabanının adresini o proje içinde tanımlarsak ve Data projesi içindeki contexe bunu gönderirsek, veritabanı bağlantısı ile ilgili değişiklikleri uygulamadan yapmış oluruz.

Veritabanı adres cümlesini WebUI projesi içinde Appsettings.json dosyasından göndereceğiz. Daha önceden bu dosya içinde mail gönderme için şifrelermizi, adminin seed bilgileri için temel özellikleri vs eklemiştik.

Önce Data projesi içindeki TSContext dosyamızın içindeki veritabanı adresinin bulunduğu ilgili satırları kaldıralım.

```
    //Veritabanın adı ve tipi burada veriliyor.
    //protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    //{
    //    optionsBuilder.UseSqlite("Data Source=TSDb");
    //}
```

Bu sınıf içerisine dışarıdan bilgi almak için bir constructor ekleyelim. Constructor dışarıdan aldığı bilgilerin tipi DbContextOptions şeklinde olsun ve giriş parametresine de options ismini verelim. Buradaki yapı base sınıfındaki yapı ile aynı olacak. Oradaki tanımlandığı şekilde bir kullanıma sahip olacak.

```
    public class TSContext : DbContext
    {
        public TSContext (DbContextOptions options): base(options)
        {

        }
```

Veritabanımızın adresini Startup içerisinden verelim. Projemizde iki tane Context kullanıyorduk. Bir tanesi Data katmanı içindeki TSContext, diğeri ise WebUI içinde ApplicationContext dir. Bu iki Contexin veritabanına ulaşabilmesi için adres satırlarımızı Startup içerisinde aşağıdaki şekilde oluşturabiliriz.

```
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDbContext<ApplicationContext>(options => options.UseSqlite("data
source=TSDb"));
        services.AddDbContext<TSContext>(options => options.UseSqlite("data source=TSDb"));
```

Bu şekilde veritabanının adresini iki Context e de buradan (startup içerisinden) gönderebiliriz. Fakat biz burada verdiğimiz adresleri tek bir yere yazmak için tamamen **AppSettings** içinden almak istiyoruz. Bu dosyanın içine aşağıdaki satırları ekleyelim. Burada şu aşamada Sqlite adresini verdik. İlerleyen aşamalarda MySql yada MsSql veritabanlarına geçilirse diğer adres etikeleride kullanılabilir.

```
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "SqliteConnection": "data source=TSDb",
    "MySqlConnection": "",
    "MsSqlConnection": ""
  }
```

Peki StartUp.cs dosyası içinden buradaki adresi nasıl alabiliriz. Onunla ilgili satırlarda şu şekilde olacaktır.

```
public void ConfigureServices(IServiceCollection services)
{
       services.AddDbContext<ApplicationContext>(options => options.UseSqlite
(_configuration.GetConnectionString("SqliteConnection")));
       services.AddDbContext<TSContext>(options => options.UseSqlite
(_configuration.GetConnectionString("SqliteConnection")));
```

Bu satırlar ile Context içine bilgiler gönderilirken "options" içerisinde göndermiş oluyoruz. Context ler içerisine ise gönderilen bu options nesnesi constructor içerisinden alınıyor. Yani sınıf yapılarının içinden kullanılacak dışarıdan alınan parametreler constructor ile alınmış oluyor. Dolayısı ile biz artık TSContext ten oluşturduğumuz nesne oluşturma yerlerini kaldırmalıyız. Nesne oluştururken kullandığımızın Using() işlemine ait yapıyıda kaldıracağız. Bu tip bir nesne oluşturma çok EfCore içindeki klasörde çok sayıda sınıf içinde kullandığımızdan hepsini düzeltmeliyiz.

EfCoreGenericRepository nin içerisi şu şekilde oldu. Üst kısma constructor eklendi ve using şeklindeki kullanımların hepsi kaldırıldı.

EfCoreGenericRepository.cs

```
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using TS.Data.Abstract;

namespace TS.Data.Concrete.EfCore
{
    public class EfCoreGenericRepository<TEntity > : IRepository<TEntity>
        where TEntity:class
    {
        protected readonly DbContext context;

        public EfCoreGenericRepository (DbContext ctx)
        {
            context = ctx;
        }

        public void Create(TEntity entity)
        {
            context.Set<TEntity>().Add(entity);
            context.SaveChanges();
        }

        public void Delete(TEntity entity)
        {
            context.Set<TEntity>().Remove(entity);
            context.SaveChanges();
        }

        public List<TEntity> GetAll()
        {
            return context.Set<TEntity>().ToList();
        }

        public TEntity GetById(int Id)
        {
            return context.Set<TEntity>().Find(Id);
        }

        public virtual void Update(TEntity entity)
        {
            context.Entry(entity).State = EntityState.Modified;
            context.SaveChanges();
        }
    }
```

```
}
```

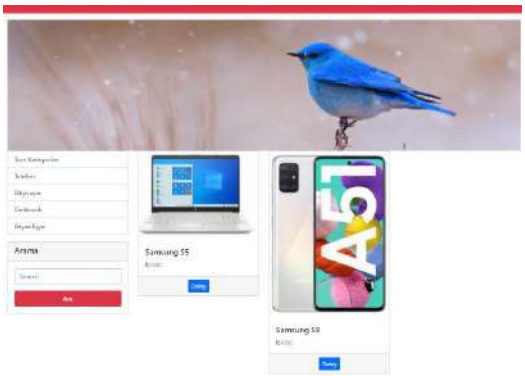
Burası generic versiyon olan ana sınıfımızdı. Ve bundan türettiğimiz (implement ettiğimiz) diğer sınıflar içinde de gerekli düzeltmeleri yapalım.

EfCoreProductRepository içindeki bilgileri düzenleyelim. Bu sınıfa TSContext artık girişten alınmayacak. Bunu constructor içinden alacağız. Constructor içinden TSContext bilgisini alıp context içine atacağız ve bu bilgiyi base sınıfına göndereceğiz. context ifadesini hemen constructorın altında private olarak tanımlıyoruz. Yani sadece bu dosya içinde kullanılacak. Public diyerek dışarıya da açabiliriz. Artık dosyamız içinde kullanacağımız nesne tsContext şeklinde olacak. Using ifadelerini kaldırarak kodalarımıza son şeklini aşağıdaki şekilde verelim. Devam eden metodlar yazılmadı.

```
public class EfCoreProductRepository :
    EfCoreGenericRepository<Product>, IProductRepository
{
    public EfCoreProductRepository(TSContext context):base(context)
    {

    }
    private TSContext tsContext
    {
        get {
            return context as TSContext;
        }
    }

    public Product GetProductDetails(string url)
    {
        return tsContext.Products
            .Where(i => i.Url == url)
            .Include(i => i.ProductCategories)
            .ThenInclude(i => i.Category)
            .FirstOrDefault();
    }
```

Benzer şekilde diğer tüm Repository ler içinde düzeltmeleri yapalım. Dışarıdan TSContext nesnesini almayacağız. Bu nesneyi Constructor içerisinden alacağız. Ayrıca base işlemi için private bir özelliği yukarıda yaptığımız şekilde ekleyeceğiz. SeedDatabase içindede TSContext nesnesini oluşturamayız. Bunu şimdilik yorum satırı yapalım ileride bu dosyayı WebUI içine alacağız. O zaman düzenleriz.

Kodları denediğimizde çalıştığını görürüz. Artık Veribanı adresi TSContext içinden verilmiyor. WebUI içinde AppSettings dosyası içine yazıldı. Oradanda Startup dosyası bu bilgiyi alıyor ve Constructor aracılığı ile Data projesi içindeki sınıf ve yapılara bu adresi göndermiş oluyoruz.



Repository Sınıflarının Tek Bir Sınıf Altında Toplanması (UnitOfWork Pattern)

Şu ana kadar çok sayıda Entity oluşturduk. Entityler aslında veritabanında bir tabloya karşılık gelen sınıf yapısı şeklinde. Örneğin, product, category, card, order gibi isimler aldı. Her entity için yapılan veritabanı işlemlerini yada metodlarını ise Repository dediğimiz sınıf yapıları içinde tuttuk. Dolayısı ile 4 adet entity için yine 4 adet repository sınıfı oluşturduk. Oysa her oluşturduğumuz repository için startup içinde referans verdik. Ayıca her çağırdığımız sınıf için ayrı ayrı nesne oluşturma işlemler vs yaptık. **Oysa onun yerine bütün respository sınıflarını bir üst sınıf içinde toplasak ve sadece bu üst sınıfı çağırma ve nesne oluşturma işlemlerini yapsak onun altındaki diğer repository sınıflarıda direk gelecektir.** Böylece daha derli toplu bir yapı kazanmış oluruz ve kullanımı basitleşmiş olur. Bu uygulamaya "UnitOfWork Pattern"yada "iş modeli deseni" diyeceğiz. Aslında işin arka planında çalıştırılan her bir sorgu için kullanılan transaction yerine tüm işlemleri yapan tek bir transaction kullanmış oluyoruz. Buda veritabanı işlemlerini kolaylaştırıyor yada veritabanını yormamış oluyor. Biz ayrı ayrı repository sınıfı kullandığımızda ve herbir repository da SaveChanges() metodunu çalıştırıyoruz. Her seferinde bu save metodu çalıştığında ayrı bir Transaction oluşturuluyor. Oysa hepsini bir yerde toplayıp tek bir SaveChanges() kullanırsak tek bir Transaction oluşacak ve işlemler basitleşecek ve veritabanı rahatlayacak. Tabii oluştururken biraz karmaşık bir yapı kodlamış oluyoruz ama gelecek performans artışı buna değer.

(ek bilgi: https://www.gencayyildiz.com/blog/c-unit-of-work-design-patternunit-of-work-tasarim-deseni/)

Unit Of Work, toplu veritabanı işlemlerini tek seferde bir kereye mahsus execute eden ve böylece bu toplu işlem neticesinde kaç kayıtın etkilendiğini rapor olarak sunabilen bir tasarım desenidir. Genellikle Repository Design Pattern ile birlikte kullanılması tercih edilen Unit Of Work, ayrıca (genellikle) transaction kontrolüyle beraber kullanılarak tek bir merkezden tüm sorgu süreçlerini kontrol edebilmektedir. Tüm bunların yanında, kullanıcı tarafından adım adım(zincirleme) yapılan operasyonlarda süreç tam teferruatlı sonlandırılmadan vazgeçildiği taktirde o noktaya kadar yapılan tüm değişikliklerin geriye alınması gerekmektedir. İşte buradaki iş maliyetini Unit Of Work ortadan kaldırmakta ve zincir sonlanmaksızın hiçbir entitynin değerini veritabanında fiziksel olarak değiştirmemektedir.

Sınıf yapılarında yapılan değişiklik sadece Repository yapılarında olacaktır. **Yani Data katmanı içindeki Abstract ve Concrete klasörleri içinde değişiklik yapılacaktır. Servis katmanın daki (business) yapılarda bir değişiklik olmayacaktır**.

Öncelikle Startup.cs içindeki repository sınıflarına yönlendiren satırları iptal edelim.

```
//services.AddScoped<IProductRepository, EfCoreProductRepository>();
//services.AddScoped<ICategoryRepository, EfCoreCategoryRepository>();
//services.AddScoped<ICardRepository, EfCoreCardRepository>();
//services.AddScoped<IOrderRepository, EfCoreOrderRepository>();
```

Data>Abstract içinde yeni bir Interface sınıfı oluşturalım (IUnitOfWork.cs şeklinde). Bu interface dışırıdan IDispossible interfaceni kullanacak. Bunu başlığın sonuna yazarak implement edelim. Tanımlanmış olan bütün repository sınıflarını bu interface içinde gruplayacağız. IProductRepository tipinde Products tanımlayıp bunun get

versiyonunu çağıracağız. Diğer repository sınıflarını da bu şekilde burada tanımlayacağız. Bir tane de kaydetme işlemleri için kullanacağımız geri dönüşü olmayan bir Save() isminde metodumuz olsun.

TS.Data.Abstract> **IUnitOfWork.cs**

```
using System;

namespace TS.Data.Abstract
{
  public interface IUnitOfWork:IDisposable
  {
    ICardRepository Cards { get; }
    ICategoryRepository Categories { get; }
    IOrderRepository Orders { get; }
    IProductRepository Products { get;}
    void Save();
  }
}
```

Abstract içinde oluşturduğumuz bu Interface sınıfının dolu versiyonunu Concrete içinde oluşturalım. Adı yine UnitOfWork.cs şeklinde olsun. Bu sınıf IUnitOfWork arabiriminden (interface) imlement edilecek. Arabirimi uygula deyip (Implement Interface) abstract sınıfı içindeki yapı buraya geldikten sonra sınıfın en üstünde constructor yapısını oluşturalım. Burada bize bir context lazım olacak. Bunu sınıf içine constructor aracılığı ile çağıralım.

Otomatik olarak oluşturulan Dispose() metodu ve Save() metodarı içinde _context sınıfının savechanges() ve dispose() metodlarını kullanacağız.

Dışarıda ayrı dosyalar halinde public olarak tanımlanan tüm sınıf yapılarının bir versiyonunu burada Private olarak tanımlayıp kullanacağız.

```
private EfCoreCardRepository _cardRepository;
```

Interfaceden gelen Cards içine bakacağız. (=> işaretine dikkat edin). Eğer buradan gelen Cards içi dolu ise _cardRespository geri göndereceğiz. Şayet boş ise (?? Eğer null sa anlamında) o zaman `EfCoreCardRepository` den yeni bir nesne oluşturulur fakat bu işlem içinde _context ihtiyaç duyulur. Yani burada Cards varsa mevcut card gönderilir yoksa null kontrolü yapıldıktan sonra yeni bir nesne oluşturulur bunun içinde _context ihtiyaç duyulur.

```
public ICardRepository Cards => _cardRepository = _cardRepository ?? new
EfCoreCardRepository(_context);
```

Diğer repository ler içinde benzer kod yapısını oluşturalım.

TS.Data.Concrete.EfCore> **UnitOfWork.cs**

```
using TS.Data.Abstract;

namespace TS.Data.Concrete.EfCore
{
  public class UnitOfWork : IUnitOfWork
  {
    private readonly TSContext _context;
    public UnitOfWork(TSContext context)
    {
      _context = context;
    }

    private EfCoreCardRepository _cardRepository;
    private EfCoreCategoryRepository _categoryRepository;
    private EfCoreOrderRepository _orderRepository;
    private EfCoreProductRepository _productRepository;

    public ICardRepository Cards =>
      _cardRepository = _cardRepository ?? new EfCoreCardRepository(_context);
```

```
    public ICategoryRepository Categories =>
      _categoryRepository = _categoryRepository ?? new EfCoreCategoryRepository(_context);

    public IOrderRepository Orders =>
      _orderRepository = _orderRepository ?? new EfCoreOrderRepository(_context);

    public IProductRepository Products =>
      _productRepository = _productRepository ?? new EfCoreProductRepository(_context);

    public void Dispose()
    {
      _context.Dispose();
    }

    public void Save()
    {
      _context.SaveChanges();
    }
  }
}
```

Data katmanında ihtiyacımız olan sınıfları oluşturduk. Artık Business katmanında Manager sınıfları içinde çağırdığımız Repositorylerin yerine UnitOfWork sınıfı üzerinden ilgili repositoryleri çağıracağız.

Card manager içerisindeki düzenlemeleri yapalım.

TS.Business.Concrete> **CardManager.cs**

```
using TS.Business.Abstract;
using TS.Data.Abstract;
using TS.Entity;

namespace TS.Business.Concrete
{
  public class CardManager : ICardService
  {
    private IUnitOfWork _unitofwork;
    public CardManager(IUnitOfWork unitofwork)
    {
      _unitofwork = unitofwork;
    }

    public void AddToCard(string userId, int productId, int quantity)
    {
      var card = GetCardByUserId(userId);

      if (card != null)
      {
        var index = card.CardItems.FindIndex(i => i.ProductId == productId);
        if (index < 0)
        {
          card.CardItems.Add(new CardItem()
          {
            ProductId = productId,
            Quantity = quantity,
            CardId = card.Id
          });
        }
        else
        {
          card.CardItems[index].Quantity += quantity;
        }

        _unitofwork.Cards.Update(card);
        _unitofwork.Save();

      }
    }

    public void ClearCard(int cardId)
    {
      _unitofwork.Cards.ClearCard(cardId);
```

```
    }

    public void DeleteFromCard(string userId, int productId)
    {
      var card = GetCardByUserId(userId);
      if (card != null)
      {
        _unitofwork.Cards.DeleteFromCard(card.Id, productId);
      }
    }

    public Card GetCardByUserId(string userId)
    {
      return _unitofwork.Cards.GetByUserId(userId);
    }

    public void InitializeCard(string userId)
    {
      _unitofwork.Cards.Create(new Card() { UserId = userId });
    }
  }
}
```

_unitofwork.Save(); işlemini buraya eklediğimizden dolayı artık EfCoreCardRepository içindeki ve diğer repository sınıflarındaki SaveChanges işlemilerini oradan kaldıracağız. Bu işlem artık daha üst sınıf olan UnifOfWork içinde bir kez çalıştırılacak.

```
public class EfCoreCardRepository : EfCoreGenericRepository<Card>, ICardRepository
{  public override void Update(Card entity)
  {
    tsContext.Cards.Update(entity);
    //tsContext.SaveChanges();
  }
```

Manager sınıfları içinde tüm kullandığımız metodlar eğer kaydetme, güncelleme ve silme ile alakalı bir işlem ise (yani veritabanında değişiklik yapan bir işlem ise, sadece okuma gibi bir işlem değilse) bu metodların içinde Save() metodunu kullanmalıyız.

Artık yapmamız gereken startup.cs içinde IUnitOfWork interfaceni çağırınca UnitOfWork sınıfını çalıştıracağı bilgisini vermek. Bunları ayrı repositoryler için tek tek yapıyorduk. Şu anda daha üst sınıfı çağırarak hepsi için işlemi yaPmış oluyoruz.

```
services.AddScoped<IUnitOfWork, UnitOfWork>();
```

Kodlarımızı denediğimizde sitenin tüm sayfalarının çalıştığını görürüz. Böylece Repository sınıflarımızı tek bir sınıf altında toplamış olduk. İstediğimiz zamana istediğimiz Repositorye kolayca geçiş yapabiliyoruz.

Veritabanı Ayar Sınıflarını Ayrı bir Klasörde Toplama (Model Configurations - Fluent Api)

Bu dersimizde Veritabanı yapısını biraz daha düzenli yapabilmek için Fluent Api denilen yapıları biraz daha düzenli olarak görelim. (Fluent Api: Veritabanı sınıflarını ve ilişkilerini code tarafında oluşturma. Ek bilgi: https://abdurrahman.github.io/2017/01/fluent-api-nedir/)

Biz herhangi bir Entity sınıfı için (yani TS.Entitiy projesi içindeki veritabanındaki tablolar için) ekstra bir ayarlar yapmak istediğimizde oluşturduğumuz Context sınıfı içinde bu entityler ile ilgili ayarlar yapabiliyoruz. Daha önce böyle bir ayar yapmıştık. Bu ayar yapma işine **Fluent Api** denir. Eğer biz Entity sınıfları içinde bazı ayarlar yapıyorsak bunlara **Data Annotation** demiştik. Fakat proje büyüdükçe Data Annotation yapısı ile yapılan ayarlar işi zorlaştırır. Bu nedenle daha sonrasında gelen ve ezici bir ayar olan Context içinden ayar yapma işini tercih ediyoruz. Yani Fluent Api kullanarak ayar yapmayı tercih ediyoruz.

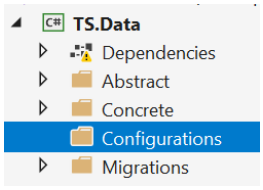
Örneğin TSContext içinde ki aşağıdaki satırlar bir Fluent Api ayarıdır. Bu FluentApi ile ProductCategory entitisi için aynı anda iki tane birincil anahtar olması için kod yazmıştık.

```
  public class TSContext : DbContext
  {
    //ProductCategory tablosu oluşturulurken kullanılacak Id bilgileri veriliyor. İki tane Id
alanı vardır.
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
      modelBuilder.Entity<ProductCategory>().HasKey(c => new {c.CategoryId, c.ProductId});
    }

  }
```

O zaman tekrar özetlersek, bir entity yi oluştururken Entity projesini kullanmıştık. Bu proje içerisinde bu güne kadar bir çok entitiy oluşturduk. (Products, Category, Card, Order, ProductCategory gibi). Bunlar veritabanındaki tablolara karşılık geliyordu. Oysa bu sınıfların içine baktığımızda veritabanı alanlarını oluştururken her hangi bir sınırlama getirmedik. Yani bir Product entity sinde Name alanı için kaç karakter olacağını vermedik. Bu durumda bu alan maksimum karaktere göre ayarlanmış oldu. Oysa bu durum veritabanındaki alanların çok yer kaplamasına yol açar. Böyle bir alan için daha az karakterde isim oluşturmamız gerekebilir. Bu sınırlamayı getirmek için Data Annotation yapısını kullanabiliriz. Bu şekilde entity sınıfı içinde örneğin Name alanı için 50 karakterden oluşacağını bildirebiliriz, yada Url alanının doldurulmasını zorunlu kılabiliriz. Bu tip uygulamaları daha önce denedik.

Fakat entity sınıfları içinde DataAnnotation kullanılması proje ilerledikçe yönetimi zorlaştırır. Bu nedenle burada tanımlama yapamazsak nerede tanımlama yapabiliriz. 2. Tanımlama yapabileceğimiz yer olarak Context sınıfı içinde Fluent Api dediğimiz yöntemle bunu yapabiliriz. Fakat her entitiy için burada böyle bir kod yazmak bu sınıfın karmaşık hale gelmesine neden olur. Bu nedenle bu yöntemi de tercih etmiyoruz. 3. bir yöntemi deneyeceğiz. Tüm entitilerin her bir alanı için yazacağımız bu tür tanımlamaları ayrı bir klasör içinde kodlayarak yapacağız.

Bunun için TS.Data içinde "Configurations" isminde bir klasör oluşturalım. Bu klasör içinde yeni bir sınıf oluşturalım. Bu sınıfın adını örneğin Product entity si için yazacağımız Fluent Api kodlarını oluşturmak üzere ProductConfiguration.cs adını verelim. Bu sınıf `IEntityTypeConfiguration<Product>` interface den türetilecek ve buna parametre olarak Product entity sini vereceğiz.



TS.Data.Configurations> **ProductConfiguration.cs**

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;
using TS.Entity;

namespace TS.Data.Configurations
{
  public class ProductConfiguration : IEntityTypeConfiguration<Product>
  {
    public void Configure(EntityTypeBuilder<Product> builder)
    {
      builder.HasKey(m => m.ProductId); //ProductId sütünu birinci sütün olarak oluşturuldu.
      builder.Property(m => m.Name).IsRequired().HasMaxLength(100); //Name alanın zorunlu ve 100 karekterden
oluşacağını bildirdi.
      //builder.Property(m => m.DateAdded).HasDefaultValueSql("getdate()"); //MSSql vt için: DateAdded alanına
kaydın oluşturulduğu andaki tarihi ekliyor.
      builder.Property(m => m.DateAdded).HasDefaultValueSql("date('now')"); //SQLite vt için: DateAdded alanına
kaydın oluşturulduğu andaki tarihi ekliyor.

      //Başlangıç (seed) ürün bilgilerini yüklüyor.
      builder.HasData(
```

```
        new Product() { ProductId = 1, Name = "Samsung S5", Url = "samsung-s5", Price = 1000, ImageUrl =
"1.jpg", Description = "İyi Telefon", IsApproved = true, IsHome = true },
        new Product() { ProductId = 2, Name = "Samsung S6", Url = "samsung-s6", Price = 2000, ImageUrl =
"2.jpg", Description = "İyi Telefon", IsApproved = false, IsHome = true },
        new Product() { ProductId = 3, Name = "Samsung S7", Url = "samsung-s7", Price = 3000, ImageUrl =
"3.jpg", Description = "İyi Telefon", IsApproved = true, IsHome = false },
        new Product() { ProductId = 4, Name = "Samsung S8", Url = "samsung-s8", Price = 4000, ImageUrl =
"4.jpg", Description = "İyi Telefon", IsApproved = false, IsHome = false },
        new Product() { ProductId = 5, Name = "Samsung S9", Url = "samsung-s9", Price = 5000, ImageUrl =
"5.jpg", Description = "İyi Telefon", IsApproved = true, IsHome = true }
      );

    }
  }
}
```

Burada Product entitisi içinde yeni bir alanı kullandık. DataAdded alanını aşağıdaki satırları TS.Entity > Product.cs içine eklemeliyiz. Daha önce oraya koymamıştık.

```
    public DateTime DateAdded { get; set; }
```

Bu konu ile ilgili başka bilgiler elde etmek için google "ef core" yazıp ilgili siteye gidip, oradan Create a model > Overview kısmından başka örneklerde incelenebilir.

Şimdi bu oluşturduğumuz Fluent Api özelliklerinin bulunduğu Class la Context sınıfımızı ilişkilendirmemiz gerekir. Bunun için TSContext içinde OnModelCreating içerisinde ApplyConfiguration metoduna biraz önce oluşturduğumuz sınıfı eklemektir. Üst kısmı ilgili Namespace de eklemek gerekir (`using TS.Data.Configurations;`).

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
        modelBuilder.ApplyConfiguration(new ProductConfiguration());
```

TSContext içinde bulunan ve daha önceden eklediğimiz `ProductCategory` entitisine ait Configuration yapımızı (Fluent Api uygulaması) oluşturalım. Yani Context içinden çıkarıp Configuration içinde o entity ye ait bir sınıfın içine alalım.

TS.Data.Configurations> **ProductCategoryConfiguration.cs**

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;
using TS.Entity;

namespace TS.Data.Configurations
{
  public class ProductCategoryConfiguration : IEntityTypeConfiguration<ProductCategory>
  {
    public void Configure(EntityTypeBuilder<ProductCategory> builder)
```

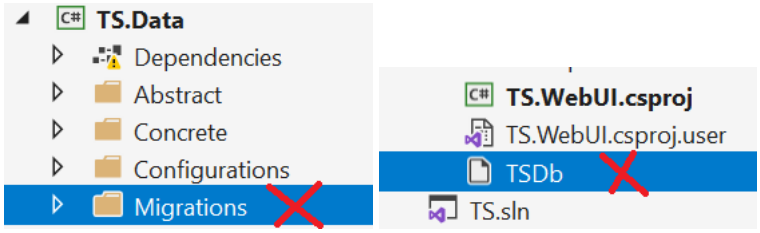
```
    {
      builder.HasKey(c => new { c.CategoryId, c.ProductId });
    }
  }
}
```

Bu sınıfı çağıran TS.Context sınıfı içindeki ilgili metodun içeriği ise şu şekilde oldu.

```
public class TSContext : DbContext
{
  protected override void OnModelCreating(ModelBuilder modelBuilder)
  {
    modelBuilder.ApplyConfiguration(new ProductConfiguration());
    modelBuilder.ApplyConfiguration(new ProductCategoryConfiguration());
  }
```

Veritabanımız yapısında epey bir değişiklik yaptık. Bu değişiklikleri veritabanına aktarmak için Migration kodlarımızı oluşturalım.

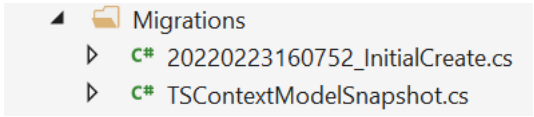
Daha önceden kullandığımız migration lar varsa bunları da kaldırsak iyi olur. Her şeyi sıfırdan oluşturuyor muşuz gibi yapalım. Normalde buna gerek te yok. Her oluşturulan migrationdan sonra yeni migrationlar oluşturulmuş ise sadece onu ilgilendiren değişiklikler Veritabanına aktarılacaktır. Önceki yapılar duracaktır. Fakat bu projede Sqlite veritabanı kullandığımızdan gelişmiş özellikleri yoktur, o nedenle önceki migrationlar sorun oluşturabiliyor. MsSQL gibi veritabanlarında bu sorun olmayacaktır. Migration klasörünü komple kaldırıyoruz. Aynı zamanda Veritabanını sıfırdan oluşturmak istediğimizden Veritabanını Explorer üzerinden direk silelim. Bu oluşturma işlemi esnasında veri kaybı olabileceğine dair sarı renkle bir uyarı verebilir. Bu bir hata değildir.



Migration NameSpace seviyesinde yazalım. Burada yazınca Startup projenin hangisi olduğu ve yapıların alınacağı Context sınıfımızın hangi projenin içinde olduğunu belirtmemiz gerekiyor. İstersek direk olarak TS.Data projesi içinde bu komutu çalıştırabilirdik. O zaman sondaki context sınıfını belirtmeye ihtiyaç kalmazdı. Burada "-p" ifadesi project manasında. "--project" şeklinde de yazabilirdik. Buda çalışırdı.

dotnet ef migrations add InitialCreate --startup-project TS.WebUI --context TSContext -p TS.Data

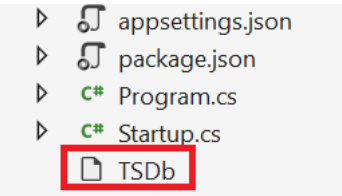
```
C:\Users\pc1\Desktop\TS.com\TS>dotnet ef migrations add InitialCreate --startup-project TS.WebUI --context TSContext -p TS.Data
Build started...
Build succeeded.
Done. To undo this action, use 'ef migrations remove'
```



Veritabanını oluşturmak ve içerisine yapıları başlangıç bilgilerini (seed) aktarmak için Migrations ları update edelim. Yine üst ana klasörde çalışalım.

dotnet ef database update --startup-project TS.WebUI --context TSContext -p TS.Data

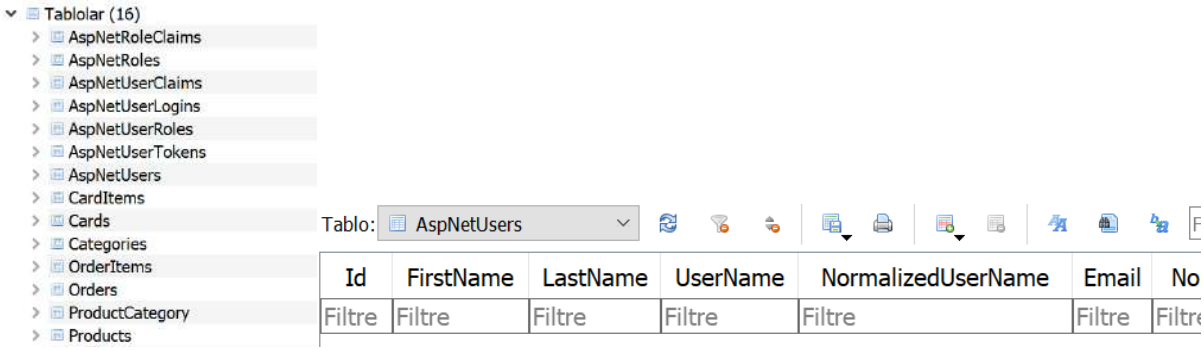
```
C:\Users\pc1\Desktop\TS.com\TS>dotnet ef database update --startup-project TS.WebUI --context TSContext -p TS.Data
Build started...
Build succeeded.
Done.
```



Projemizde WebUI altında da Context sınıfımız olduğundan (ApplicationContext) buradaki yapılarıda Veritabanına aktarmamız gerekir (user Identiy ile ilgili yapılar). Bunun için önce mevcut WebUI içindeki Migration klasörünü kaldıralım. Ardından aşağıdaki iki komutu WebUI içinde çalıştıralım. Burada startup projeyi vermemize gerek yoktur. Zaten onun içindeyiz. Fakat context sınıfımızın hangisi olduğunu vermemiz gerekir. Çünkü data içindeki context i de görür.

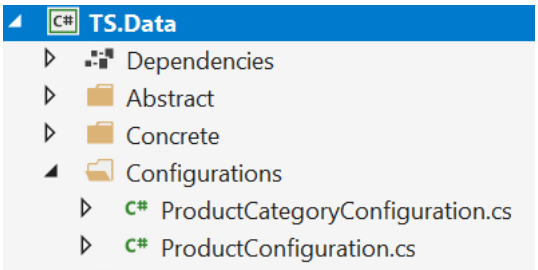
```
dotnet ef migrations add AddingItentity --context ApplicationContext

dotnet ef database update --context ApplicationContext
```

Veritabanını açtığımızda tüm tabloların oluştuğunu görebiliyoruz.



Veritabanı oluştu fakat içi boştur. İçerisinin Seed bilgileri ile doldurulması için projenin F5 ile çalıştırılması gerekir. Fakat SeedDatabase ait kodları kaldırmıştık. Bu nedenle site çalıştı fakat bilgiler gelmedi. Bir sonraki konuda Admin ve başlangıç bilgilerini yeniden ele alacağız. Bu kısımları daha kullanışlı hale getireceğiz.

Configurations dosyası içinde sadece iki adet entity (ProductCategory ve Product)için configuration sınıfı (Fluent Api) oluşturduk. Benzer şekilde tüm Entityler için bu sınıfları burada oluşturmalıyız. Böylece Veritabanı ile ilgili ayarlar bu sınıf içinde toplanmış olacaktır.



Seed Data Bilgilerinin Configurations dosyaları içine konulması (Seed Data In Configurations)

Bir önceki uygulamada Seed bilgilerini (başlangıç atanan bilgiler) iptal etmiştik. Şimdi bunları Configurations dosyaları içine yerleştirelim. İlk olarak başlangıç ürün bilgilerini ProductConfiguration.cs dosyasının içine aşağıdaki şekilde HasData() metoduyla koyalım. ProductId bilgilerini bu aşamada vermeliyiz.

TS.Data.Configurations > **ProductConfiguration.cs**

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;
using TS.Entity;

namespace TS.Data.Configurations
{
  public class ProductConfiguration : IEntityTypeConfiguration<Product>
  {
    public void Configure(EntityTypeBuilder<Product> builder)
    {
      builder.HasKey(m => m.ProductId); //ProductId sütünu birinci sütün olarak oluşturuldu.
      builder.Property(m => m.Name).IsRequired().HasMaxLength(100); //Name alanın zorunlu ve 100 karekterden
oluşacağını bildirdi.
      //builder.Property(m => m.DateAdded).HasDefaultValueSql("getdate()"); //MSSql vt için: DateAdded alanına
kaydın oluşturulduğu andaki tarihi ekliyor.
      builder.Property(m => m.DateAdded).HasDefaultValueSql("date('now')"); //SQLite vt için: DateAdded alanına
kaydın oluşturulduğu andaki tarihi ekliyor.

      //Başlangıç (seed) ürün bilgilerini yüklüyor.
      builder.HasData(
        new Product() { ProductId = 1, Name = "Samsung S5", Url = "samsung-s5", Price = 2000, ImageUrl =
"1.jpg", Description = "iyi telefon", IsApproved = true },
        new Product() { ProductId = 2, Name = "Samsung S6", Url = "samsung-s6", Price = 3000, ImageUrl =
"2.jpg", Description = "iyi telefon", IsApproved = false },
        new Product() { ProductId = 3, Name = "Samsung S7", Url = "samsung-s7", Price = 4000, ImageUrl =
"3.jpg", Description = "iyi telefon", IsApproved = true },
        new Product() { ProductId = 4, Name = "Samsung S8", Url = "samsung-s8", Price = 5000, ImageUrl =
"4.jpg", Description = "iyi telefon", IsApproved = false },
        new Product() { ProductId = 5, Name = "Samsung S9", Url = "samsung-s9", Price = 6000, ImageUrl =
"5.jpg", Description = "iyi telefon", IsApproved = true }
      );
    }
  }
}
```

Yukarıda CategoryConfiguration.cs dosyasını oluşturmamıştık. Şimdi onu oluşturalım. ProductConfiguration.cs dosyasının bir benzeri olacağı için onu kopyalalıp içerisinde gerekli değişiklikleri yapalım.

TS.Data.Configurations> **CategoryConfiguration.cs**

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;
using TS.Entity;

namespace TS.Data.Configurations
{
  public class CategoryConfiguration : IEntityTypeConfiguration<Category>
  {
    public void Configure(EntityTypeBuilder<Category> builder)
    {
      builder.HasKey(m => m.CategoryId); //CategoryId sütünu birinci sütün olarak oluşturuldu.
      builder.Property(m => m.Name).IsRequired().HasMaxLength(100); //Name alanın zorunlu ve 100 karekterden
oluşacağını bildirdi.

      //Başlangıç (seed) ürün bilgilerini yüklüyor.
      builder.HasData(
        new Category() { CategoryId = 1, Name = "Telefon", Url = "telefon" },
        new Category() { CategoryId = 2, Name = "Bilgisayar", Url = "bilgisayar" },
        new Category() { CategoryId = 3, Name = "Elektronik", Url = "elektronik" },
        new Category() { CategoryId = 4, Name = "Beyaz Eşya", Url = "beyaz-esya" }
      );

    }
  }
}
```

Product ile Category arasındaki bağlantıyı sağlayacak olan ProductCategory entitisine ait Configuration dosyasını oluşturalım.

TS.Data.Configurations > **ProductCategoryConfiguration.cs**

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;
using TS.Entity;

namespace TS.Data.Configurations
{
  public class ProductCategoryConfiguration : IEntityTypeConfiguration<ProductCategory>
  {
    public void Configure(EntityTypeBuilder<ProductCategory> builder)
    {
      builder.HasKey(c => new { c.CategoryId, c.ProductId });

      builder.HasData(
        new ProductCategory() { ProductId = 1, CategoryId = 1 },
        new ProductCategory() { ProductId = 1, CategoryId = 2 },
        new ProductCategory() { ProductId = 1, CategoryId = 3 },
        new ProductCategory() { ProductId = 2, CategoryId = 1 },
        new ProductCategory() { ProductId = 2, CategoryId = 2 },
        new ProductCategory() { ProductId = 2, CategoryId = 3 },
        new ProductCategory() { ProductId = 3, CategoryId = 4 },
        new ProductCategory() { ProductId = 4, CategoryId = 3 },
        new ProductCategory() { ProductId = 5, CategoryId = 3 },
        new ProductCategory() { ProductId = 5, CategoryId = 1 }
      );
    }
  }
}
```

Bu configuration dosyalarını TSContext içinde tanıtmalıyız.

```
public class TSContext : DbContext
{
  protected override void OnModelCreating(ModelBuilder modelBuilder)
  {
    modelBuilder.ApplyConfiguration(new ProductConfiguration());
    modelBuilder.ApplyConfiguration(new CategoryConfiguration());
    modelBuilder.ApplyConfiguration(new ProductCategoryConfiguration());
  }
}
```

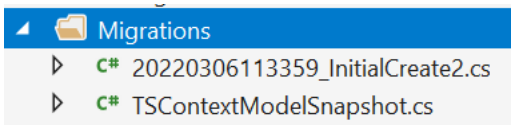
Bu bilgilerin veritabanında oluşması için Migration oluşturalım. Bunun için her şeyi sıfırdan oluşturalım. Daha önceki oluşturduğumuz Migrations klasörünü ve oluşturulmuş TSDb database ni kaldıralım. Komutumuz ana kalasörde (namespace) çalıştıralım. Öncesinde projelerimi Build etmeyi unutmayalım.

TS> dotnet ef migrations add InitialCreate --context TSContext --startup-project TS.WebUI --project TS.Data

```
C:\Users\pc1\Desktop\TS.com\TS>dotnet ef migrations add InitialCreate --context TSContext --startup-project TS.WebUI --project TS.Data
Build started...
Build succeeded.
The name 'InitialCreate' is used by an existing migration.
```

İsim çakışması verdi ama sebebi bulunamadı. Oysa böyle bir isimde migration projede yok. O yüzden sonuna 2 ekleyip tekrar deneyelim. Bu sefer çalıştı.

```
C:\Users\pc1\Desktop\TS.com\TS>dotnet ef migrations add InitialCreate2 --context TSContext --startup-project TS.WebUI --project TS.Data
Build started...
Build succeeded.
Done. To undo this action, use 'ef migrations remove'
```



Migration ların veritabanına aktarılması için (veritabanı bulunmuyor onu da oluşturacak) aynı yerde yine benzer olan update komutunu çalıştıralım.

TS> dotnet ef database update --context TSContext --startup-project TS.WebUI --project TS.Data

```
C:\Users\pc1\Desktop\TS.com\TS>dotnet ef database update --context TSContext --startup-project TS.WebUI --project TS.Data
Build started...
Build succeeded.
Done.
```

Test verilerini Configurations dosyalarının içinde yazdık. Eğer istersek bu verileri başka bir dosyanın (sınıfın) içinde gruplamak isteyebiliriz. Şimdi ona bakalım. Bunun için yine Configurations klasörü içine ModelBuilderExtensions.cs isminde bir sınıf dosyası oluşturalım. ModelBuilder sınıfını burada genişletmek istiyoruz.

TS.Data.Configurations> **ModelBuilderExtensions.cs**

```
using Microsoft.EntityFrameworkCore;
using TS.Entity;

namespace TS.Data.Configurations
{
  public static class ModelBuilderExtensions
  {
    public static void Seed(this ModelBuilder builder)
    {
      //Başlangıç (seed) ürün bilgilerini yüklüyor.
      builder.Entity<Product>().HasData(
        new Product() { ProductId = 1, Name = "Samsung S5", Url = "samsung-s5", Price = 2000, ImageUrl =
"1.jpg", Description = "iyi telefon", IsApproved = true },
        new Product() { ProductId = 2, Name = "Samsung S6", Url = "samsung-s6", Price = 3000, ImageUrl =
"2.jpg", Description = "iyi telefon", IsApproved = false },
        new Product() { ProductId = 3, Name = "Samsung S7", Url = "samsung-s7", Price = 4000, ImageUrl =
"3.jpg", Description = "iyi telefon", IsApproved = true },
        new Product() { ProductId = 4, Name = "Samsung S8", Url = "samsung-s8", Price = 5000, ImageUrl =
"4.jpg", Description = "iyi telefon", IsApproved = false },
        new Product() { ProductId = 5, Name = "Samsung S9", Url = "samsung-s9", Price = 6000, ImageUrl =
"5.jpg", Description = "iyi telefon", IsApproved = true }
      );

      //Başlangıç (seed) ürün bilgilerini yüklüyor.
      builder.Entity<Category>().HasData(
        new Category() { CategoryId = 1, Name = "Telefon", Url = "telefon" },
        new Category() { CategoryId = 2, Name = "Bilgisayar", Url = "bilgisayar" },
        new Category() { CategoryId = 3, Name = "Elektronik", Url = "elektronik" },
        new Category() { CategoryId = 4, Name = "Beyaz Eşya", Url = "beyaz-esya" }
      );

      builder.Entity<ProductCategory>().HasData(
        new ProductCategory() { ProductId = 1, CategoryId = 1 },
        new ProductCategory() { ProductId = 1, CategoryId = 2 },
        new ProductCategory() { ProductId = 1, CategoryId = 3 },
        new ProductCategory() { ProductId = 2, CategoryId = 1 },
        new ProductCategory() { ProductId = 2, CategoryId = 2 },
        new ProductCategory() { ProductId = 2, CategoryId = 3 },
        new ProductCategory() { ProductId = 3, CategoryId = 4 },
        new ProductCategory() { ProductId = 4, CategoryId = 3 },
        new ProductCategory() { ProductId = 5, CategoryId = 3 },
        new ProductCategory() { ProductId = 5, CategoryId = 1 }
      );
    }
  }
}
```

Bilgileri ayrı bir dosya için aldık. Daha sonra `ModelBuilderExtensions` Sınıfı üzerinden bu metodu (Seed) TSContext sınıfı içerisinde çağırdığımızda bu bilgiler veritabanına eklenmiş olacaktır.

TS.Data.Concrete.EfCore > **TSContext.cs**

```
using Microsoft.EntityFrameworkCore;
using TS.Data.Configurations;
using TS.Entity;
```

```
namespace TS.Data.Concrete.EfCore
{
  public class TSContext : DbContext
  {
    public TSContext(DbContextOptions options) : base(options)
    {

    }

    //Aşağıdaki DbSet ler veritabanında oluşturulacak olan tabloları ifade eder.
    public DbSet<Product> Products { get; set; }
    public DbSet<Category> Categories { get; set; }
    public DbSet<Card> Cards { get; set; }
    public DbSet<CardItem> CardItems { get; set; }
    public DbSet<Order> Orders { get; set; }
    public DbSet<OrderItem> OrderItems { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
      modelBuilder.ApplyConfiguration(new ProductConfiguration()); //Product ile ilgili Vt de gerekli olacak
sabit ayarları buradan okuyor.
      modelBuilder.ApplyConfiguration(new CategoryConfiguration());
      modelBuilder.ApplyConfiguration(new ProductCategoryConfiguration());

      modelBuilder.Seed(); //Veritabanına kaydedilecek başlangıç sabit bilgilerini bu sınıf içerisinden okuyor.
    }

  }
}
```

DİKKAT: Bazen Migration oluştururken hata alınabiliyor. Önceki migrationları kaldırsak bile onları aktif görebiliyor. Bu tür durumlar için tüm projeyi VS içinden kapatıp, öyle oluşturursak hata alınmıyor.

Veritabanına kullanıcı bilgilerini WebUI içinde ApplicatonContext içinden aktarıyorduk. Bu Context ide güncelleyip veritabanına bilgileri aktaralım.

Not: Bundan sonraki kısımlarda TS projesi olarak değil Shopapp projesi olarak devam edilecek.

shopapp.webui>dotnet ef database update --context ApplicationContext

Projemiz artık çalışacaktır.

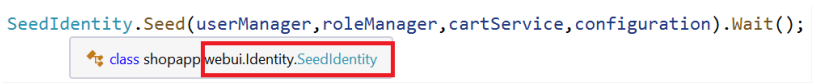
Başlangıç Kullanıcı bilgilerinin VT içine aktarılması

Daha önceden başlangıç Admin kullanıcısı bilgilerini VT içine ApplicationContext üzerinden aktarıyorduk. Fakat o uygulamımızda Admin kullanıcının Cart bilgisini aktarmadığımızdan (ki bu bilgi kullanıcı ilk oluşturulduğunda atanıyor) bu kullanıcının Cart bilgisi olmayınca alışveriş linkleri çalışmıyordu. Şimdi bu kodları biraz daha düzenleyelim.

Program ilk çalıştığında Startup.cs içindeki aşağıdaki satır çalıştırılıyor.

```
SeedIdentity.Seed(userManager,roleManager,cartService,configuration).Wait();
```

Bu kodlar SeedIdentity sınıfı içindeki Seed() metodunu çalıştırıyor. Bu nesnenin üzerine geldiğimizde kodlarımızın Webui>Identity klasörü üçinde olduğunu görüyoruz.



SeedIdentity AppSettings.json içinden atanan bilgileri çekiyordu. Bu dosya içindeki bilgileri düzenleyelim. Admin ve 2 tane de Customer ekleyelim. Burada Data sekmesi içinde iki tane liste oluşturuyoruz. Bunlar Roles[] ve Users[]

şeklinde oldu. İçerini de dolduruyoruz. Users maddelerini oluştururken onunda alt alanlarını parantezler içinde tanımlıyoruz.

WebUI>**Appsettings.json**

```
{
 "Logging": {
  "LogLevel": {
   "Default": "Information",
   "Microsoft": "Warning",
   "Microsoft.Hosting.Lifetime": "Information"
  }
 },
 "EmailSender": {
  "Host":"smtp.office365.com",
  "Port": 587,
  "EnableSSL": true,
  "UserName": "<email>",
  "Password":"<password>"
 },
 "Data":{
  "Roles": ["admin","customer"],
  "Users": [
   {
    "username": "adminuser",
    "password": "Shopapp_123",
    "email":"adminuser@shopapp.com",
    "role":"admin",
    "firstName":"Sadık",
    "lastName":"Turan"
   },
   {
    "username": "customeruser",
    "password": "Shopapp_123",
    "email":"customeruser@shopapp.com",
    "role":"customer",
    "firstName":"Çınar",
    "lastName":"Turan"
   }
   ,
   {
    "username": "customeruser2",
    "password": "Shopapp_123",
    "email":"customeruser2@shopapp.com",
    "role":"customer",
    "firstName":"ahmet",
    "lastName":"Turan"
   }
  ]
 },
 "AllowedHosts": "*",
 "ConnectionStrings": {
  "SqliteConnection" : "Data Source=shopDb",
  "MySqlConnection":"",
  "MsSqlConnection":""
 }
}
```

Buradaki bilgileri Webui>SeedIdentiy içindeki Seed() metoduyla okuyacağız. Buradaki satırları baştan bir daha düzenleyelim ve açıklayalım.

Aşağıdaki satırla data sekmesi altındaki Roles bilgilerini okuyoruz. Appsettings içinden okuma işlemini configuration nesnesi yapıyor. GetChildren ile alt özellliklerine oluşıyoruz. Bunların değerlerini select ile okuyup diziye çeviriyoruz. Yani var roles içinde artık “admin” ve “customer” ifadeleri olmuş oluyor.

```
var roles = configuration.GetSection("Data:Roles").GetChildren().Select(x=>x.Value).ToArray();
```

buradan alınan bilgilere göre aşağıdaki satırlar ile bütün role bilgileri veritabanına eklenmiş oluyor.

```
//okunan rolleri sırayla tarayacak.
```

```
foreach (var role in roles)
{
  //role bilgilsinin daha önce VT ye eklenip eklenmediğini kontrol edelim. Eğer yoksa ekleyecek.
  if (!await roleManager.RoleExistsAsync(role))
  {
    //role bilgisini veritabanında oluşturuyor. role içindeki bilgi string değil IdentityRole tipinde.
    await roleManager.CreateAsync(new IdentityRole(role));
  }
}
```

Ardından users bilgilerini alalım. Gene aynı şekilde data nın altındaki users bilgilerine ulaşalım. Ardından users bir listedir. Bunun içindeki her bir elemeanı okumak için children() metodunu kullanalım. Okuduğumuz her user bilgisini section olarak adlandırıp (tek bir user bilgisine karşılık geliyor) onunda altındaki bilgileri de alan ifadeleriyle ulaşalım.

shopapp.webui.Identity > **SeedIdentity.cs**

```
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.Configuration;
using shopapp.business.Abstract;

namespace shopapp.webui.Identity
{
  public static class SeedIdentity
  {
    public static async Task Seed(UserManager<User> userManager, RoleManager<IdentityRole>
roleManager,ICartService cartService,IConfiguration configuration)
    {
      //appsettings.json dosyası içinde verilen başlangıç rollerini okuyor. "admin" ve "customer" var.
      var roles = configuration.GetSection("Data:Roles").GetChildren().Select(x=>x.Value).ToArray();

      //okunan rolleri sırayla tarayacak.
      foreach (var role in roles)
      {
        //role bilgilsinin daha önce VT ye eklenip eklenmediğini kontrol edelim. Eğer yoksa ekleyecek.
        if (!await roleManager.RoleExistsAsync(role))
        {
          //role bilgisini veritabanında oluşturuyor. role içindeki bilgi string değil IdentityRole tipinde.
          await roleManager.CreateAsync(new IdentityRole(role));
        }
      }

      //appsetings içindeki users bilgilerini okuyor. 1 tane admin, 2 tane customer var.
      var users = configuration.GetSection("Data:Users");

      foreach (var section in users.GetChildren())
      {
        var username = section.GetValue<string>("username");
        var password = section.GetValue<string>("password");
        var email = section.GetValue<string>("email");
        var role = section.GetValue<string>("role");
        var firstName = section.GetValue<string>("firstName");
        var lastName = section.GetValue<string>("lastName");

        if(await userManager.FindByNameAsync(username)==null)
        {
          var user = new User()
          {
            UserName = username,
            Email = email,
            FirstName = firstName,
            LastName = lastName,
            EmailConfirmed = true //oluşturulan kullanıcının bilgisi başlangıçta onaylı oluyor.
          };

          //pralo bilgisi Async olduğu için şifrelenerek yukarıdaki kodlardan ayrı olarak kaydediliyor.
          var result = await userManager.CreateAsync(user,password);

          if(result.Succeeded)
          {
            //kullanıcı ile role ilişkinin kuruyor.
            await userManager.AddToRoleAsync(user,role);
            //cartservise aracılığı ile ilgili kullanıcının kart bilgisini Vt ye ekliyor.
            cartService.InitializeCart(user.Id);
```

```
            }
        }

    }
  }
 }
}
```

Burada ilgili kullanıcının cart bilgilerini oluşturmak için cartService içindeki InitalizeCart() metodu kullanılacak. Dolayısı seed() metodunun girişinde bu cartService nesnesi alındı. Bunu girişte ekledik.

```
public static async Task Seed(UserManager<User> userManager, RoleManager<IdentityRole>
roleManager,ICartService cartService,IConfiguration configuration)
```

Fakat buradaki Seed() metoduna bilgiler ilk açılışta Startup içinden geldiğinden bu sınıf içindeki configure() metoduna da bu nesnenin eklenmesi gerekir. Ayrıca oradan buraya gelirken yazılan adres kodlarının içine de yazılmalıdır. Aşağıdaki şekilde bunlarda yapıldı.

```
public class Startup
{
public void Configure(IApplicationBuilder app, IWebHostEnvironment env,IConfiguration
configuration,UserManager<User> userManager, RoleManager<IdentityRole> roleManager, ICartService
cartService)
{SeedIdentity.Seed(userManager,roleManager,cartService,configuration).Wait();
```

Kodları çalıştırmak için Migration oluşturmaya gerek yoktur. Vt deseninde bir değişiklik yapmıyoruz. Sadece ilgili kullanıcı ve roller yoksa onları ekleyecek. Kodları denediğimizde çalıştığını göreceğiz.

Bir sonraki dersimizde Vt yapısını oluşturan sınıflarımızda değişiklik yaptığımızda her seferinde migration oluşturup update ediyorduk. Bunu otomatik olarak yapmak için bir Migration Manager sınıfı oluşturalım.

Migration Manager

Biz şu anada kadar Veritabanında yaptığımız değişiklikleri Migration oluşturarak çözdük. Data ve WebUI içinde migratinlarımızı oluşturduk. Ardından güncelleme komutları ile bu değişiklikleri veritbanına aktarıyorduk. Burada migrationları yine biz oluşturalım fakat güncelleme işlemlerini otomatik olarak yaptıralım. Bu derste bunun işlemlerini görelim.

WebUI>Program.cs içindeki aşağıdaki build() metodu bize Ihost tipinde bir interface gönderiyor. Bu interface genişleterek otomatik migration işlemleri yapacağız. İşin mantığı şu şekilde. Program ilk çalıştığında Program.cs içindeki Main metodu çalışmaktadır. Bu metod çalışında Build() işlemininde devamına bir tane extension metod ekliyoruz. Yani mevcutlar üzerine eklenen ek bir işlem ekliyor muşuz gibi düşünelim. Bu işlem çalıştığında bekleyen migrationlar varsa önce onları güncelleyecek (veritabanına aktaracak) ondan sonra çalışmasına devam edecek.

```
public static void Main(string[] args)
{
    CreateHostBuilder(args).Build().Run();
}
```

1 reference

IHost IHostBuilder.Build()
Run the given actions to initialize the host. This can only be called once.

Burada yapacağımız işlem Ihost üzerinden MigrateDatabase() isminde bir extension metodu oluşturmak.

```
CreateHostBuilder(args).Build().MigrateDatabase().Run();
```

Extention metodlarımızı oluşturmak için WebUI içinde Extensions isminde bir klasör oluşturmuştuk Bu kalasörün içinde MigrationManager.cs isminde bir sınıf oluşturalım. Bunun tipi static olacak ve IHost u genişleten bir metod olacak. Dolayısı ile sınıf içerisinde yukarıda yazdığımız MigrateDatabase() isminde bir metod oluşturalım. Bu metod dışarıdan IHost u alacak ve genişlettikten sonra geri yine IHost olarak gönderecek. IHost sınıfını extend yapabilmek için (`using Microsoft.Extensions.Hosting;`) namespaceni de eklemeliyiz.

```
using Microsoft.Extensions.Hosting;

namespace shopapp.webui.Extensions
{
  public static class MigrationManager
  {
    public static IHost MigrateDatabase(this IHost host)
    {
```

Migratedatabase() metodu içinde ShopContext yada ApplicatonContext den elde edilen bir nesnenin olması gerekir. Biz daha önce bu nesneleri Startup içinde de kullanmıştık. Nasıl elde etmiştik ona bir bakalım. Startup içerisinde IServiceCollection interface ile oluşturulan services nesnesi içinde servis hizmeti verecek olan nesneleri tutuyorduk.

```
public void ConfigureServices(IServiceCollection services)
{
services.AddDbContext<ApplicationContext> (options=> options.UseSqlite(_configuration.GetConnectionString
("SqliteConnection")));
services.AddDbContext<ShopContext>(options=> options.UseSqlite (_configuration.GetConnectionString
("SqliteConnection")));
```

Madem bizim bu services konteynarı içinde hizmet veren servislerimiz var. Burada tutulan servislerden ihtiyacımız olan servisleri alıp kullanalım. Bunun için host.Services içinden scope oluşturarak servislerimizi bunun içinden çekececeğiz.

```
using (var scope = host.Services.CreateScope())
{
```

Aşağıdaki satır ile scope içerisinden ApplicationContext nesnemizi çekelim bir örneğini elde edebiliriz.

```
using (var applicationContext = scope.ServiceProvider.GetRequiredService<ApplicationContext>())
{
```

Sonra `applicationContext` nesnemiz üzerinde veritabanı migrate işlemlerini yapabiliriz. Bir hata olduğunda da kayıtlarda tutmak için loglama yapabiliriz. Burada Migrate() metodu EntityFrameworkCore namespacene ihtiyaç duyacaktır.

```
try
{
        applicationContext.Database.Migrate();
}
        catch(System.Exception)
{}
```

Aynı işlemleri ShopContext içinde yapalım.

Metod içinde geriye (`return host;`) diyerek host u geri göndermemiz gerekir. Girişte host aldık içinden Context nesnelerini çektik ve onları kullanarak VT için Migration işlemlerini yaptık. Bu metoddan sonra devam eden metodlarda Host un kullanımı devam ettiğinden geriye host tekrar gönderiyoruz. Tüm sınıfın içerisindeki kodlarımız şu şekilde oldu.

shopapp.webui.Extensions > **MigrationManager.cs**

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using shopapp.data.Concrete.EfCore;
using shopapp.webui.Identity;
```

```
namespace shopapp.webui.Extensions
{
  public static class MigrationManager
  {
    public static IHost MigrateDatabase(this IHost host)
    {
      using (var scope = host.Services.CreateScope())
      {
        using (var applicationContext = scope.ServiceProvider.GetRequiredService<ApplicationContext>())
        {
          try
          {
            applicationContext.Database.Migrate();
          }
          catch(System.Exception)
          {
            //Loglama için kullanılabilir.
            throw;
          }
        }

        using (var shopContext = scope.ServiceProvider.GetRequiredService<ShopContext>())
        {
          try
          {
            shopContext.Database.Migrate();
          }
          catch (System.Exception)
          {
            //Loglama için kullanılabilir.
            throw;
          }
        }
        return host;
      }
    }
  }
}
```

Program.cs nin son hali ise şu şekilde oldu.

shopapp.webui> Program.cs

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;
using shopapp.webui.Extensions;

namespace shopapp.webui
{
  public class Program
  {
    public static void Main(string[] args)
    {
      CreateHostBuilder(args).Build().MigrateDatabase().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
      Host.CreateDefaultBuilder(args)
        .ConfigureWebHostDefaults(webBuilder =>
        {
          webBuilder.UseStartup<Startup>();
        });
  }
}
```

Şu aşamada Data ve WebUI içindeki bekleyen migrationlarımız var. Veritabanını silip yaptığımız bu işlem bu migrationlarımızı otomatik güncelleyerek veritabanını oluşturup bilgileri aktarsın.

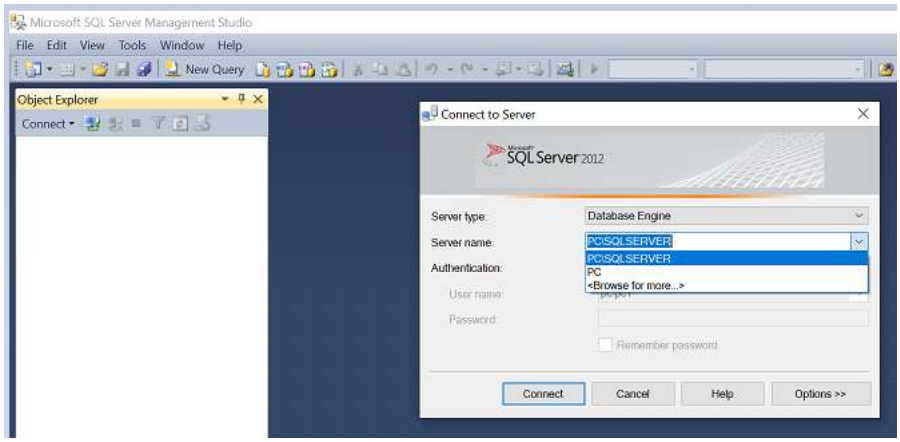
MsSQL Veritabanı Kullanımı

Şu ana kadar projemizde SqLite veritabanı kullandık. Projemizdeki ayarlarmaları MsSql veritabanına çevireceğiz. Bunun için bilgisayarımıza MsSql Express veritabanını kurabiliriz. Visual Studio kulanıyorsak zaten yerel bir MsSql de geliyor.

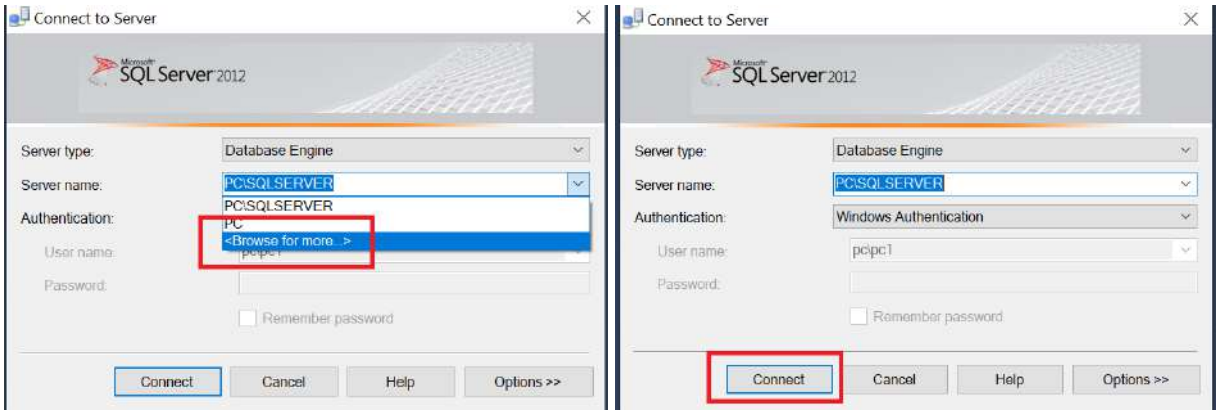
https://www.microsoft.com/tr-tr/sql-server/sql-server-downloads

Buradan Express i ücretsiz olarak kurabiliriz. Servise ulaşmak içinde “Management Studio” ihtiyacımız vardı. Sayfanın daha aşağısından Araçlardan bunu indirelim. Bu aracı kullanarak Express serverına bağlanıp yada VS içindeki SQL servera bağlanıp kolayca veritabanı oluşturabiliriz.

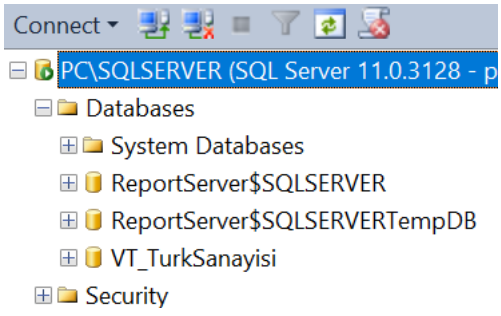
Management Studio çalıştırdığımızda bilgisayarımızdaki kuruluma bağlı olarak aşağıdaki gibi bir ekran gelecektir.



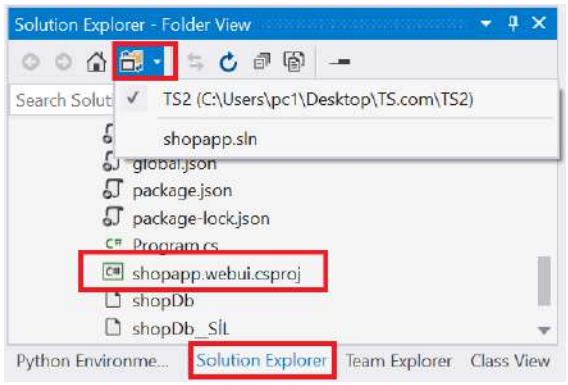
Burada “Brows for more” seçtiğimizde lokal de kurulu olan server tiplerini de seçebiliriz. Sql Server seçili iken Connect diyelim ve bir bağlantı sağlayalım.



Sol taraftaki gezgin kısmında var olan veritabanlarını görebiliriz.



Projemize dönelim. Peki biz programa Sqlite kullanacağımızı nerede bildirdik. Data projesinin içerisindeki shopapp.data.csproj isimli proje dosyasını açalım. Bu dosyayı VS içinden görebilmemiz için Solution Explorer penceresinde iken (sekmenin burada olduğuna dikkat edin) üstten Folder görünümüne geçmemiz gerekir.



Bu dosya içinde Sqlite için yüklenen paketin birde SqlServer için olan dll yükleyelim. Versiyon değişmesin.

```
<ItemGroup>
  <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="3.1.3">
   <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
   <PrivateAssets>all</PrivateAssets>
  </PackageReference>
  <PackageReference Include="Microsoft.EntityFrameworkcore.Sqlite" Version="3.1.3" />
  <PackageReference Include="Microsoft.EntityFrameworkcore.SqlServer" Version="3.1.3" />
 </ItemGroup>
```

Aynı satırı WebUI projesi içindeki csproj dosyası içine de eklemeliyiz. Bu tanımlamayı orada da yapmıştık.

```
<PackageReference Include="Microsoft.EntityFrameworkcore.SqlServer" Version="3.1.3"/>
```

Buraya kadar kullanacağımız SqlServer paketlerinin bildirimini yaptık fakat bu paketler henüz indirip projemizin içinde yoktur . Bu işlem için komut satırından WebUI projesinin içerisine girelim. Orada (dotnet restore) komutunu çalıştıralım.

```
C:\Users\pc1\Desktop\TS.com\TS2\shopapp.webui>dotnet restore
  Geri yüklenecek projeler belirleniyor...
  C:\Users\pc1\Desktop\TS.com\TS2\shopapp.webui\shopapp.webui.csproj geri yüklendi (1,66 sec içinde).
  C:\Users\pc1\Desktop\TS.com\TS2\shopapp.business\shopapp.business.csproj geri yüklendi (2,96 sec içinde).
  C:\Users\pc1\Desktop\TS.com\TS2\shopapp.data\shopapp.data.csproj geri yüklendi (2,96 sec içinde).
  Geri yükleme için 4 proje içinden 1 tanesi güncel.
```

VS içinde projemizin normal görünümüne geçince (dosya görünümü degil) Data ve WebUI projelerinin yüklenmediğini görürüz. Bu projelerin üzerinde sağ tuşa tıklayıp Reload Proje seçeniğini kullanarak yükleyelim.

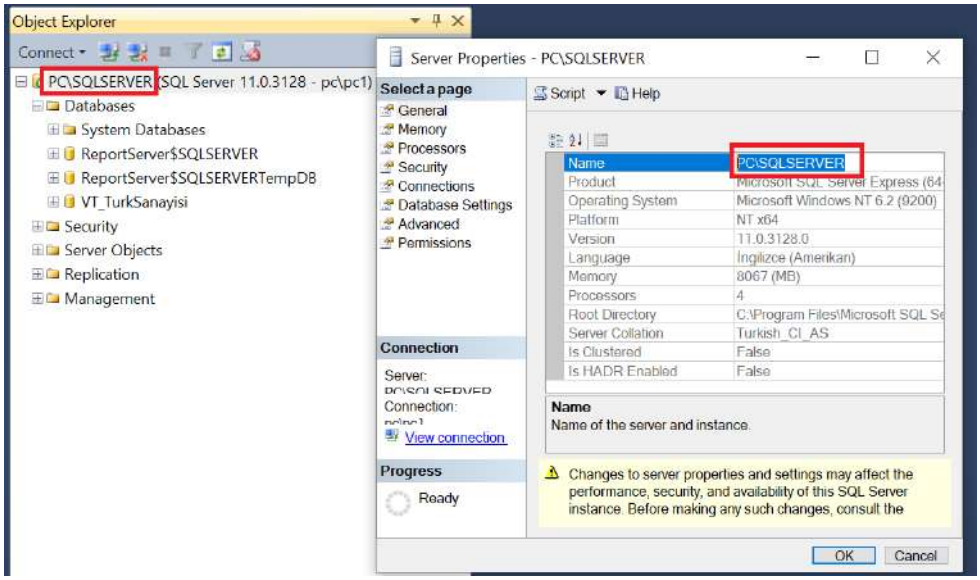
Şimdi artık Startup.cs projesi içinde Context lerimizin hangi veritabanına hangi bağlantı yolunu kullaraka bağlanacağını bildirebiliriz. Burada SqLite için yerleri SqlServer olarak değiştirebiliriz. Kütüphane yüklendiyse otomatik olarak gelmesi gerekir.

```
    public void ConfigureServices(IServiceCollection services)
    {
      //services.AddDbContext<ApplicationContext>(options=>
options.UseSqlite(_configuration.GetConnectionString("SqliteConnection")));
      //services.AddDbContext<ShopContext>(options=>
options.UseSqlite(_configuration.GetConnectionString("SqliteConnection")));

      services.AddDbContext<ApplicationContext>(options =>
options.UseSqlServer(_configuration.GetConnectionString("MsSqlConnection")));
      services.AddDbContext<ShopContext>(options =>
options.UseSqlServer(_configuration.GetConnectionString("MsSqlConnection")));
```

Burada geçen Connection ifadesinin karşılığını için Appsettings.json içinde vermiştik. Bu ifade ile kullanacağımız veritabanının adresini bildiriyoruz. Adresini vereceğimiz server Uzak server olabilir, yerel server olabilir, hiç farketmez.

```
 "AllowedHosts": "*",
 "ConnectionStrings": {
  "SqliteConnection" : "Data Source=shopDb",
  "MySqlConnection":"",
  "MsSqlConnection":""
 }
```

Serveren adını Management programından alalım. Burada ismi en üstte görüyoruz. PC\SQLServer olarak verilmiş.



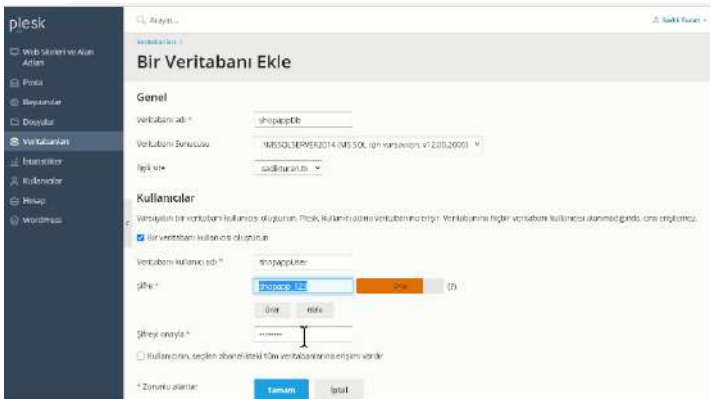
Bu şekilde gördüğümüz sever adını ve veritabanı ismini aşağıdaki şekilde verdik. Adreste ters slash lar iki tane gösterildi. Oluşacak olan veritabanının adını biz “shopappdb” olarak verdik. Herhangi bir şifre vermeyeceğimiz için security kısmını aşağıdaki şekilde yazıyoruz. Böylece SqlConnection ifademiz aşağıdaki şekilde olur.

```
"MsSqlConnection": "server=.pc\\sqlserver;database=shopappdb;integrated security=SSPI;"
```

Eğer herhangi bir yerden server hizmeti (hosting) aldıysak oradan verilen server adresi ve şifre bilgilerini kullanarak oluşturduğumuz veritabanına aşağıdaki şekilde bağlanabiliriz.

```
"MsSqlConnection":"server=mssql.sadikturan.com;database=shopappDb;User Id=shopappUser;password=shopapp_123;"
```

Böyle bir hosting hizmetini kullanırken önce veritabanını Plesk panel kullanarak oluşturmamız gerekir. Örnek bir oluşturma aşağıda verilmiştir.



Şu ana kadar SqlServer kullanmak üzere dll leri yükledik. Connection stringimizi verdik. Çalıştırmadan önce Sqlite için oluşturulmuş olan Migration ları silmemiz gerekiyor. Hem data hemde webui içindeki migrationları silelim. Sqlite ile SqlServer şemaları arasında farklılıklar var. Bu nedenle Sqlite için oluşturulmuş migrationları silmeliyiz ve yeniden Sqlserver için oluşturmalıyız.

Veritabanı ile ilgili ayarlamaları yaptığımız Data>Configuration içindeki klasörlerde ProductConfiguration.cs içinde kullandığımız bir satır SqLite ile SqlServer için farklı kullanılıyordu. O satırıda SqlServer a çevirelim. Bu satır veritabanına kayıtın oluşturulduğu tarihi atıyordu.

```
builder.Property(m=>m.DateAdded).HasDefaultValueSql("getdate()"); // mssql
//builder.Property(m=>m.DateAdded).HasDefaultValueSql ("date('now')");  // sqlite
```

Bu işlemlerden sonra Migrationları oluşturalım. Komut isteminde ana klasörde iken migration oluşturma komutlarımızı çalıştıralım.

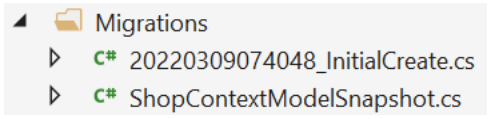
TS2> dotnet ef migrations add InitialCreate --context ShopContext --startup-project shopapp.webui --project shopapp.Data

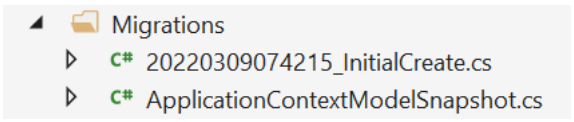
```
C:\Users\pc1\Desktop\TS.com\TS2>dotnet ef migrations add InitialCreate --context ShopContext --startup-project shopapp.webui --project shopapp.Data
Build started...
Build succeeded.
Done. To undo this action, use 'ef migrations remove'
```

TS2> dotnet ef migrations add InitialCreate --context ApplicationContext --startup-project shopapp.webui

```
C:\Users\pc1\Desktop\TS.com\TS2>dotnet ef migrations add InitialCreate --context ApplicationContext --startup-project shopapp.webui
Build started...
Build succeeded.
Done. To undo this action, use 'ef migrations remove'
```

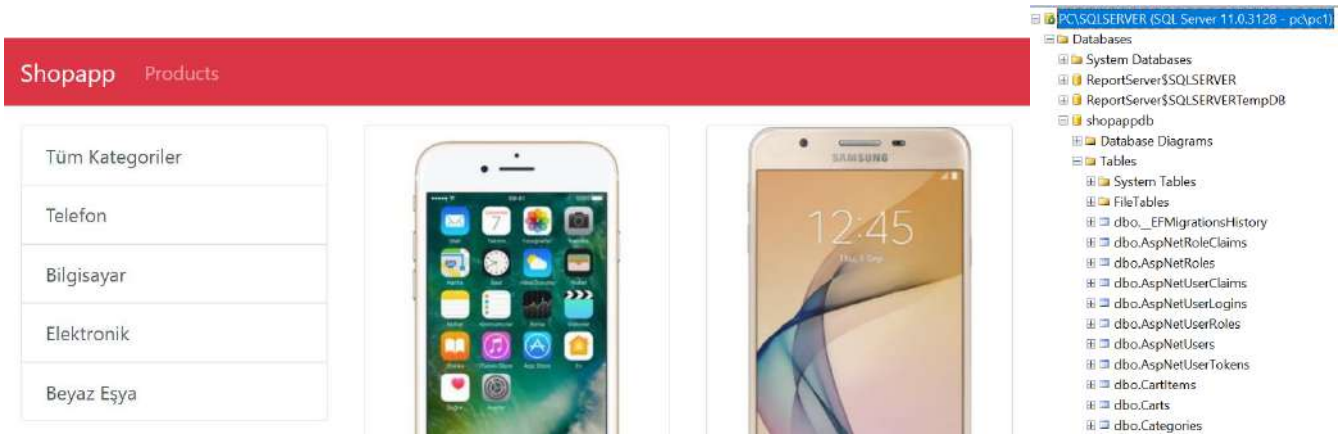
Kontrol ettiğimizde migrationlarımızın oluştuğunu görüyoruz.





Migrationları oluşturduktan sonra build yapıp bir hatamız varmı ona bakalım. Vs içinden Build>Build soluition seçebiliriz. Yada komut satırından "dotnet build" yazabiliriz. (Vs den deneyince hata alınmadı, Komut satırından denenince "Debug/Bin" çözüm yapılandırması hatası alındı)

Programı F5 ile çalıştırdığımızda SqlServer içerisinde veritabanın oluştuğunu görürüz. Ve sitemiz dolu bir şekilde karşımıza gelir.



Başka veritabanları da kullanmak istediğimizde google "ef core provider" yazıp hangi VT lerin hangi versiyonlarını Entitiy Framework ile kullanabileceğimizi görebiliriz.

https://docs.microsoft.com/tr-tr/ef/core/providers/?tabs=dotnet-core-cli

NuGet paketi	Desteklenen veritabanı motorları	Bakımcı/satıcı	Notlar/gere
Microsoft. EntityFrameworkCore. SqlServer	SQL Server 2012 sürümleri	EF Core Project (Microsoft)	
Microsoft. EntityFrameworkCore. SQLite	SQLite 3,7 sonraki sürümler	EF Core Project (Microsoft)	

Publish-Asp.Net Core Dinamik Site Yayınlama

Proje kodlarını ham olarak oluşturduğumuzda yada indirdiğimizde öncelikle içerisinde node_modules klasörü olması gerekir. Bu modül yoksa bunu kurmalıyız. Komut satırına geçip Webui içinde iken npm install diyerek kurabiliriz.

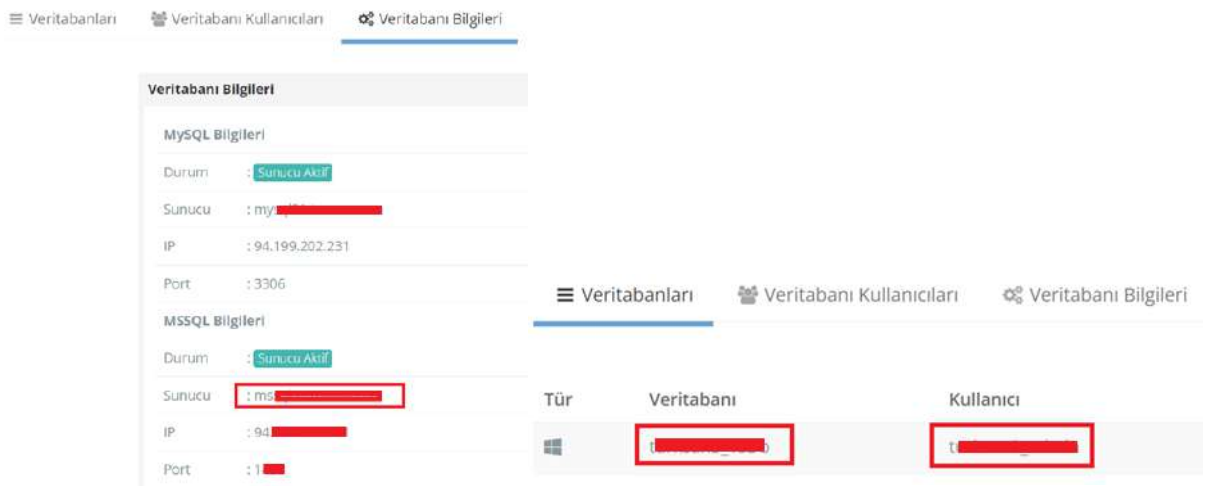
Webui içinde ayarları yaptığımız iki tane dosyamız vardır. "appsetting.json" projeyi yayınlandığımızda kullanılan dosya. "appsetting.development.json" ise geliştirme aşamasında kullanılan dosyadır. Dolayısı ile bu dosyarın içindeki ayar dosyalarını hangi kullanım aşamasında isek ona göre ayarlamalıyız. Örneğin Veritabanı adreslerini ona göre yapmalıyız.

Hosting paketini satın aldık. Bize verilen Pesk panel üzerinden bir tane veritabanı oluşturalım. MsSql (SqlServer) veritabanı oluşturalım. Vt oluşturulurken hosting şirketinin kendi panelinden değil Plesk panel üzerinden oluşturacağız.

Burada geçen TsDb şeklindeki veritabanı Hosting firmasının kullanıcı arayüzünde oluşturuldu. Detay ayarlarını burada yapacağız. Hosting firması baştaki turksan3 ifadesini kendisi ekledi. Bu isim bu domine verilmiş sabit bir isim.

Veritabanı için bir kullanıcı oluşturalım.

Veritabanını oluşturduğumuza göre artık “appsetting.json” dosyasının içine parametrelerimizi yazabiliriz. Hosting firmasının verdiği yönetim paneline geçtiğimizde, buradaki Veritabanı linkinden gerekli bilgileri okuyabiliriz. Sunucu adresini (mssql01.trwww.com)buradan alıp dosyamızın içine ekleyelim.

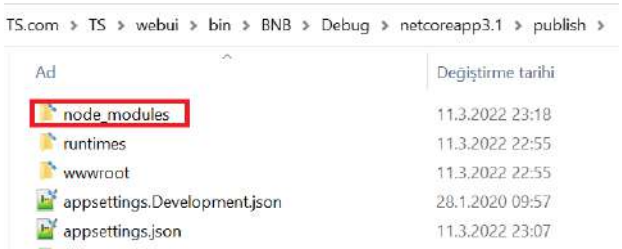


```
"MsSqlConnection": "Server=ms******m; Database=*********b; User Id=t*********n;
Password=********;"
```

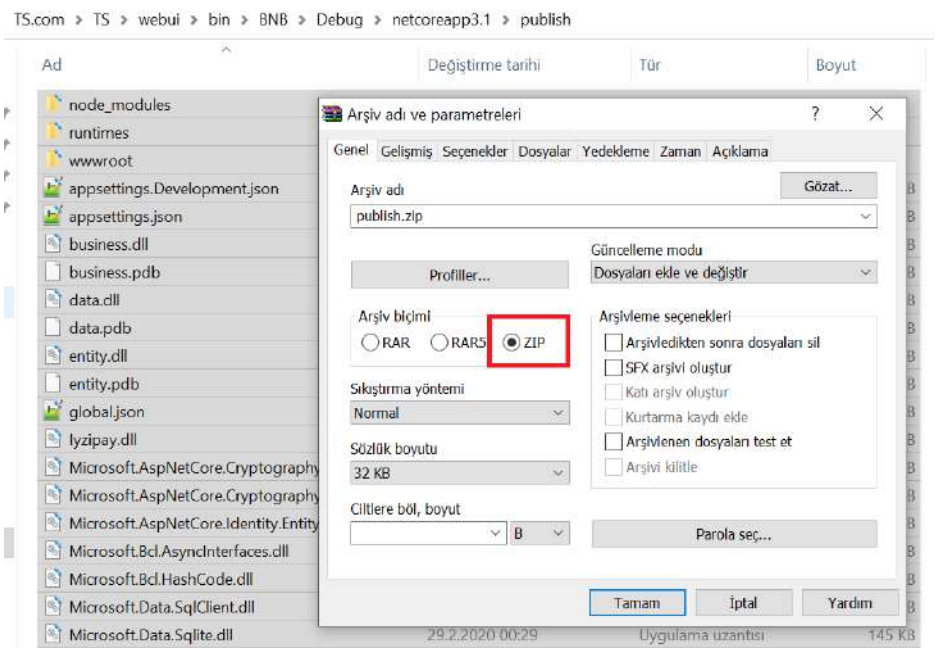
Daha sonra komut satırından WebUI projesi içinde iken “dotnet publish” diyelim ve yayınlanayacağk olduğumuz klasörü bize oluştursun. Komutu çalıştırdığımızde yayınlanacak olan klasör bin altında daha alt klasörlerden publish içinde oluşturuldu.

```
C:\Users\pc1\Desktop\TS.com\TS\webui>dotnet publish
.NET için Microsoft (R) Build Engine sürüm 16.7.2+b60ddb6f4
Telif Hakkı (C) Microsoft Corporation. Tüm hakları saklıdır.

  Geri yüklenecek projeler belirleniyor...
  C:\Users\pc1\Desktop\TS.com\TS\entity\entity.csproj geri yüklendi (439 ms içinde).
  C:\Users\pc1\Desktop\TS.com\TS\webui\webui.csproj geri yüklendi (811 ms içinde).
  C:\Users\pc1\Desktop\TS.com\TS\data\data.csproj geri yüklendi (817 ms içinde).
  C:\Users\pc1\Desktop\TS.com\TS\business\business.csproj geri yüklendi (885 ms içinde).
  entity -> C:\Users\pc1\Desktop\TS.com\TS\entity\bin\BNB\Debug\netstandard2.0\entity.dll
  data -> C:\Users\pc1\Desktop\TS.com\TS\data\bin\BNB\Debug\netstandard2.0\data.dll
  business -> C:\Users\pc1\Desktop\TS.com\TS\business\bin\BNB\Debug\netstandard2.0\business.dll
  webui -> C:\Users\pc1\Desktop\TS.com\TS\webui\bin\BNB\Debug\netcoreapp3.1\webui.dll
  webui -> C:\Users\pc1\Desktop\TS.com\TS\webui\bin\BNB\Debug\netcoreapp3.1\webui.Views.dll
  webui -> C:\Users\pc1\Desktop\TS.com\TS\webui\bin\BNB\Debug\netcoreapp3.1\publish\
```

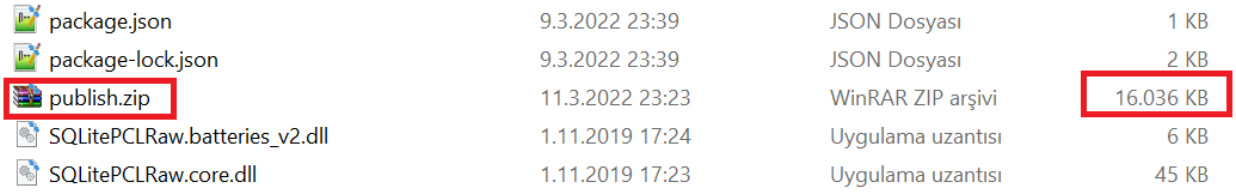
Bu klasör içerisine bir de webui altındaki node_modul klasörünü kopyalayalım.

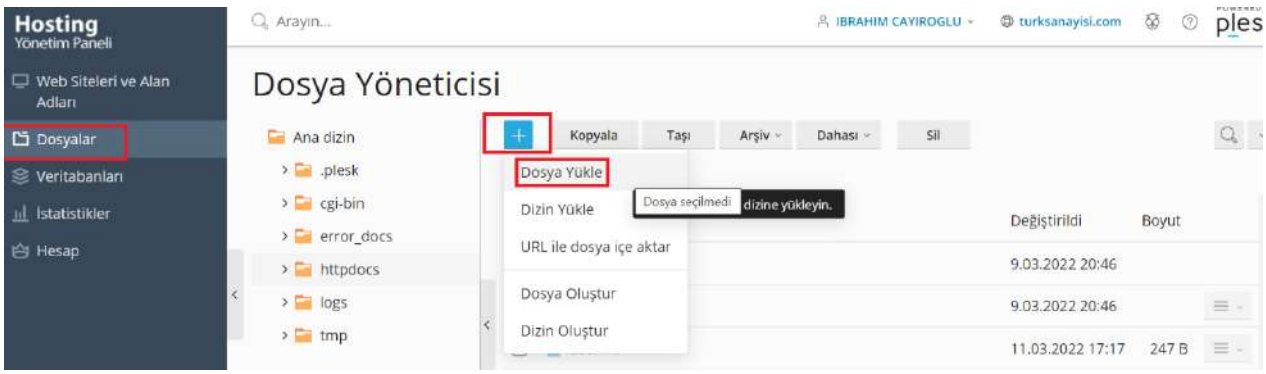


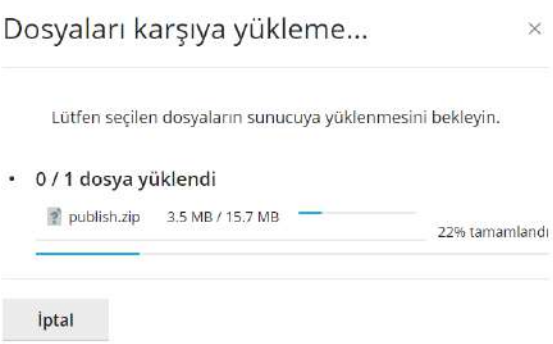
Daha sonra bütün dosyalarımızı Ctrl+A ile seçelim. Sağ tuşa tıklayıp Arşiv oluştur diyelim ve çıkan ekranda ZIP seçerek ziplenmiş dosyamızı oluşturalım. WinRar olmayacak buna dikkat edilmelidir. Plesk panel Zip tipinde dosyayı açar.



Oluşan 16 MB lık dosyamızı buradan kopyalayıp Plesk panelimizin (Hosting firmasının verdiği siteyi yönetim paneli) dosyalar kısmından yüklemeyi yapacağız.

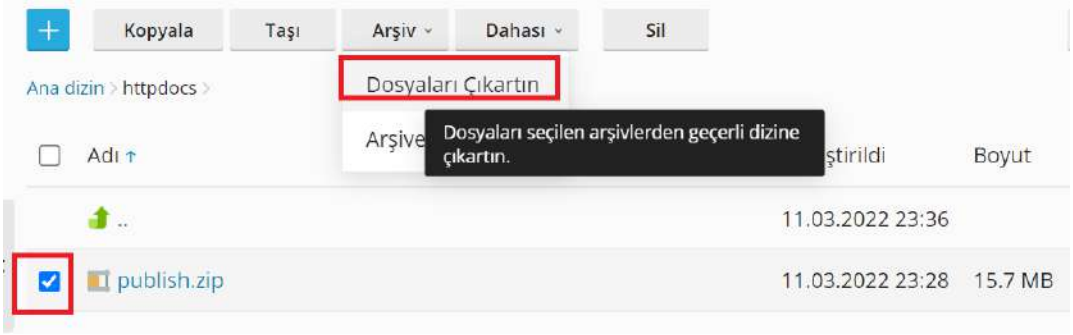




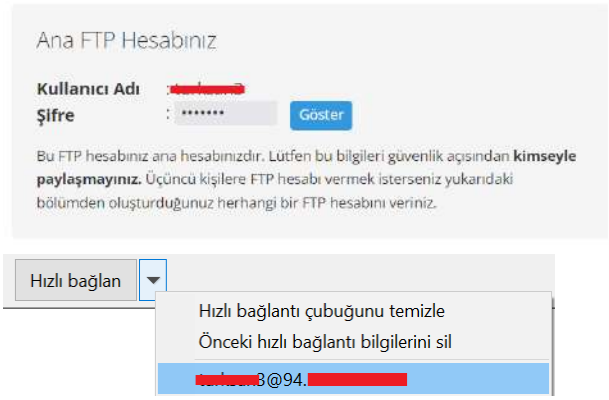


Zip Dosyamızın servera yüklendiğini görüyoruz.

Zip dosyamızı açmadan önce serverda bulunan diğer tüm dosyaları silelim. Ardından arşiv dosyamızı açalım.



Burada kullanılan yöntem çalışmadı. Alınan hosting hesabının zip li dosyaları açmadığı görüldü. Bu nedenle hosting firmasının verdiği panelden bir adet ftp şifresi oluşturuldu. Ftp ile dosya aktarama programı olarak FileZille programı kuruldu. Bu programda ftp şifresi girilerek dosyalar direk aktarıldı. FileZillede hızlı bağlan kısmından bağlanınca çalıştı.



Ek Bilgi: Türkçe karakter problemini çözmek için web.config içerisine aşağıdaki kodların ekleneceğini öğrendim fakat henüz denemesini görmedim.

Ayrıca serverda yayında Inprocess hatası alınırsa Web.config içindeki bu yeri outofprocess dönüştürmek gerekir (hostingModel="outofprocess"). Bu değişiklik yapıldığında projenin son halini publish yapıp tekrar atmamız gerekir. (bu bilgiler denenmedi !)

Web.config

```
<? xml version="1.0" encoding="utf-8"?>
<configuration>
 <location path = "." inheritInChildApplications="false">
  <system.webServer>
   <handlers>
    <add name = "aspNetCore" path="*" verb="*" modules="AspNetCoreModuleV2" resourceType="Unspecified" />
   </handlers>
   <aspNetCore processPath = "dotnet" arguments=".\webui.dll" stdoutLogEnabled="false"
stdoutLogFile=".\logs\stdout" hostingModel="inprocess" />

  </system.webServer>
 </location>

 <system.web>
  <globalization culture = "tr-TR" uiCulture="tr" requestEncoding="UTF-8" responseEncoding="UTF-8"
fileEncoding="UTF-8" responseHeaderEncoding="windows-1254" />
 </system.web>

</configuration>
<!--ProjectGuid: E2ED91AB-2E36-4A6A-B45C-7D8EC123B6D9-->
```

Ek Bilgi: Projeyi test serverda test ederken hata alırsak bize hatanın nerde olduğunu söylemeyecektir. Bunu görmek için Program.cs içine aşağıdaki satırları eklemeliyiz. Yayına geçtikten sonra bu iki satırı yorum satırına dönüştürmek gerekir, açık bırakılmamalı. (bu bilgiler denenmedi!)

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
  Host.CreateDefaultBuilder(args)
    .ConfigureWebHostDefaults(webBuilder =>
    {
      webBuilder.UseStartup<Startup>();
      //****** Aşağıdaki satırlar test ederken hata mesajlarını görmek için açılmalıdır. (;) en sonda olmalı.
      //.CaptureStartupErrors(true)
      //.UseSetting("detailedErrors", "true");
    });
```

Hosting aldığımız firmanın Plesk panelinden veritabanı içindeki tabloları görmek için bir link verilmemişse bilgisayarımıza kuracağımız SqlServer-Expression ile Host, User ve Password bilgilerini kullanarak uzak bağlantı ile servera bağlanabiliriz. Bu uygulama denendiğinde server içindeki diğer başka firmalara ait veritabanlarıda görüldü fakat içlerine girilemedi. Kendi veritabanımızın için ise tablolar gözüktü fakat içerisindeki bilgilere erişilemedi.

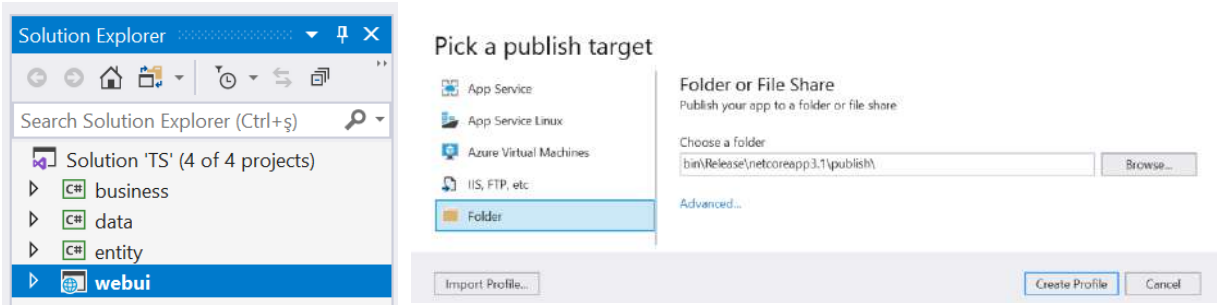
Web Sitesinin Güncellenmesi

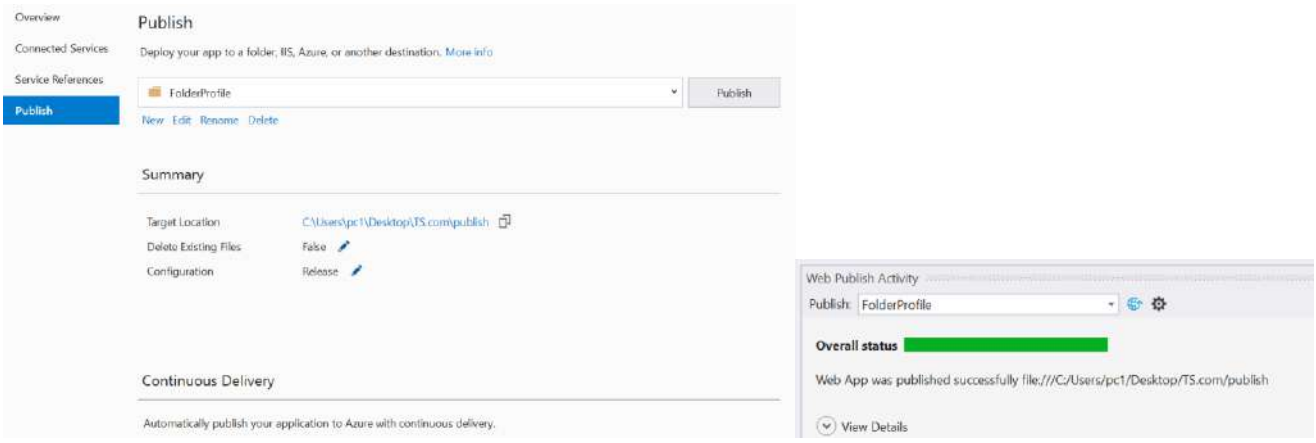
Site yayınlandıktan sonra kodlarda yada veritabanı şemasında değişiklikler olabilir. Kodlarda bir değişiklik olduğunda siteyi tekrar publish edip (exe haline getirip) tüm dosyaları atmamız gerekir. Sadece şu aşağıdaki dosyalarda güncelleme yaptıksak atalım yokda atmamıza gerek yoktur.

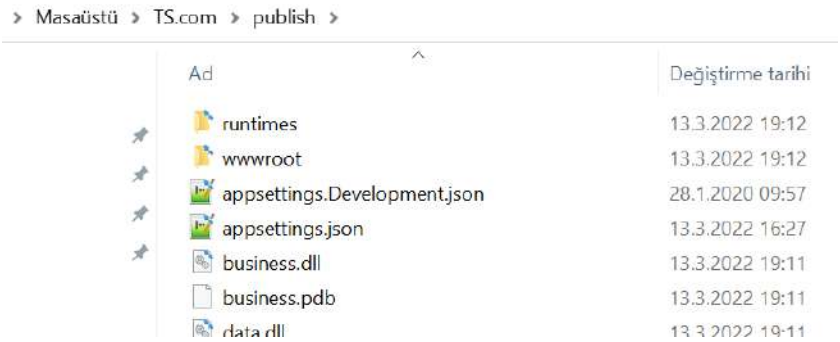
- Node Modules : bu kalasörde bir veryon değişikliği yapılmadıysa atmaya gerek yoktur.
- Wwwroot: bu klasör içinde bulunan img ve css lerde bir değişiklik yoksa atmaya gerek yoktur.
- Appsetting.json dosyasında bir değişiklik yapmadıysak atmaya gerek yoktur.

Publish İşleminin Visual Studio İçinden Yapılması

Projemizin üzerine Explorerdan webui projesi üzerine sağ tuşa tıklayıp publish seçelim. Açılan ekrandan Folder seçip kaydeceğimiz yeri belirleyelim. Publish dedikten sonra dosyalarımızın ilgili klasör içinde oluştuğunu görürüz.





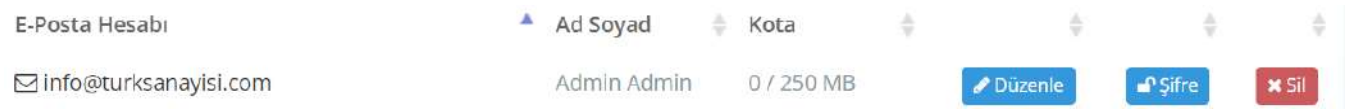


Hosting Üzerinden E-Posta Kurulumu

Hosting firmasının verdiği panel yönetiminden (plesk değil) E-posta kısmına girdiğimizde Mail servisinin aktif olmadığını görüyoruz. “Etkinleştir” butonundan bunu etkinleştirilim.

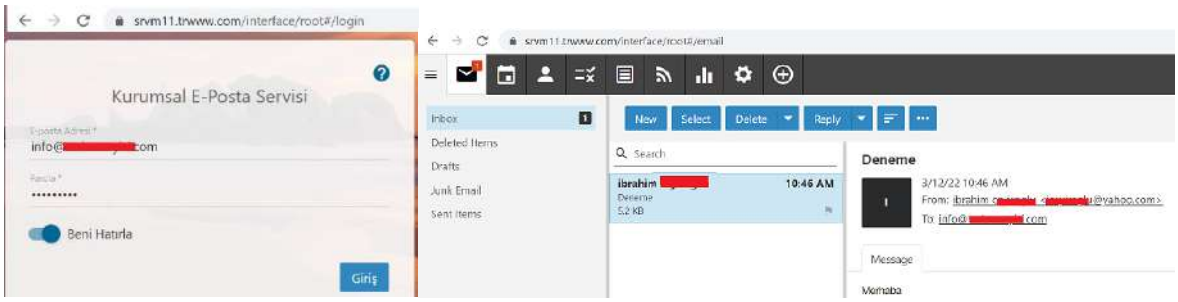
Etkinleştirdikten sonra Yönet diyerek ayarlara geçelim.

Mail adresi oluşturuldu.



Mail ayarları sekmesine girildiğinde aşağıdaki gibi mail gönderirken ve okurken kullanacağımız bilgileri görmüş oluyoruz.

Tarayıcıya (mail.ts.com) şeklinde yazdığımızda aşağıdaki sayfa açıldı. Buradan şifremizi girip bize gelen mailleri okuyabilir yada gönderebiliriz.



Uygulama içinden kodlarla mail gönderirken SMTP aracılığı ile göndereceğiz. Mail ayarları kısmında bu bilgiler verilmişti. Burada port numaraları da yazılıdır. Böyle mail gönderirken bize lazım olan iki yeri tanıtmış olduk. Normal yazışmalarda mail.ts.com adresini kullanırız. Kodla gönderirken SMTP kısmını kullanırız.

Şimdi Appsettings.json dosyası içindeki Smtp mail ayarlarını aşağıdaki şekilde yapalım.

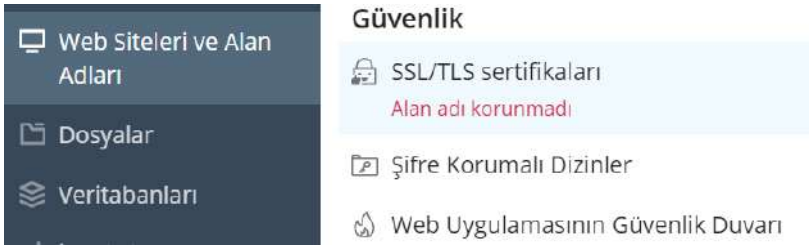
```
"EmailSender": {
  "Host":"smtp.t**s***.com",
  "Port": 4**,
  "EnableSSL": false, //ssl kurulunca true ya dönüştürülebilir
  "UserName": "info@t***s***.com",
  "Password":"*******"
 },
```

Güvenlik Sertifikası Kurulumu-SSL

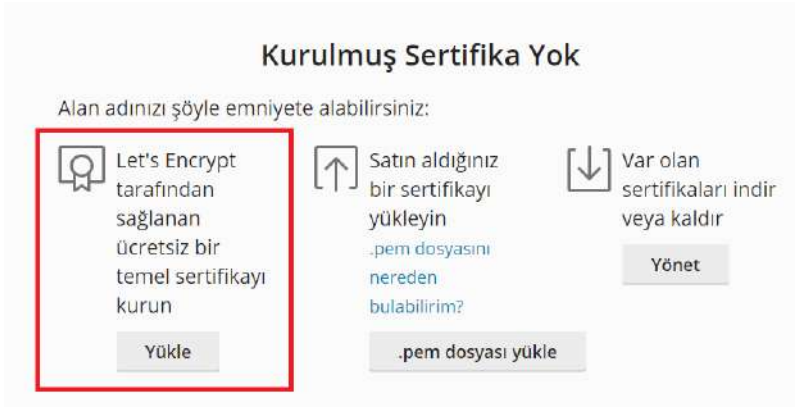
Web sitesi üzerinden gönderilen kullanıcı adı şifre, kredi kartı numarası gibi bilgilerin başkaları tarafından ele geçirilememesi için yolda şifrelenmiş olarak sayfaların gitmesi gerekir. Bu amaçla alıcı ile sunucu arasında bigilerin şifrelenerek taşınması için SSL sertifikasının kurulması gerekir. Bu sertifikanın olmadığı sitelerinde adre çubuğunda "Güvenli değil" ifadesini görürüz. Güvenli olanlarda ise bir kilit sembolü vardır. Arama motorları bu sertifikaya sahip siteleri üst sıralara çıkarırken öncelik tanır.

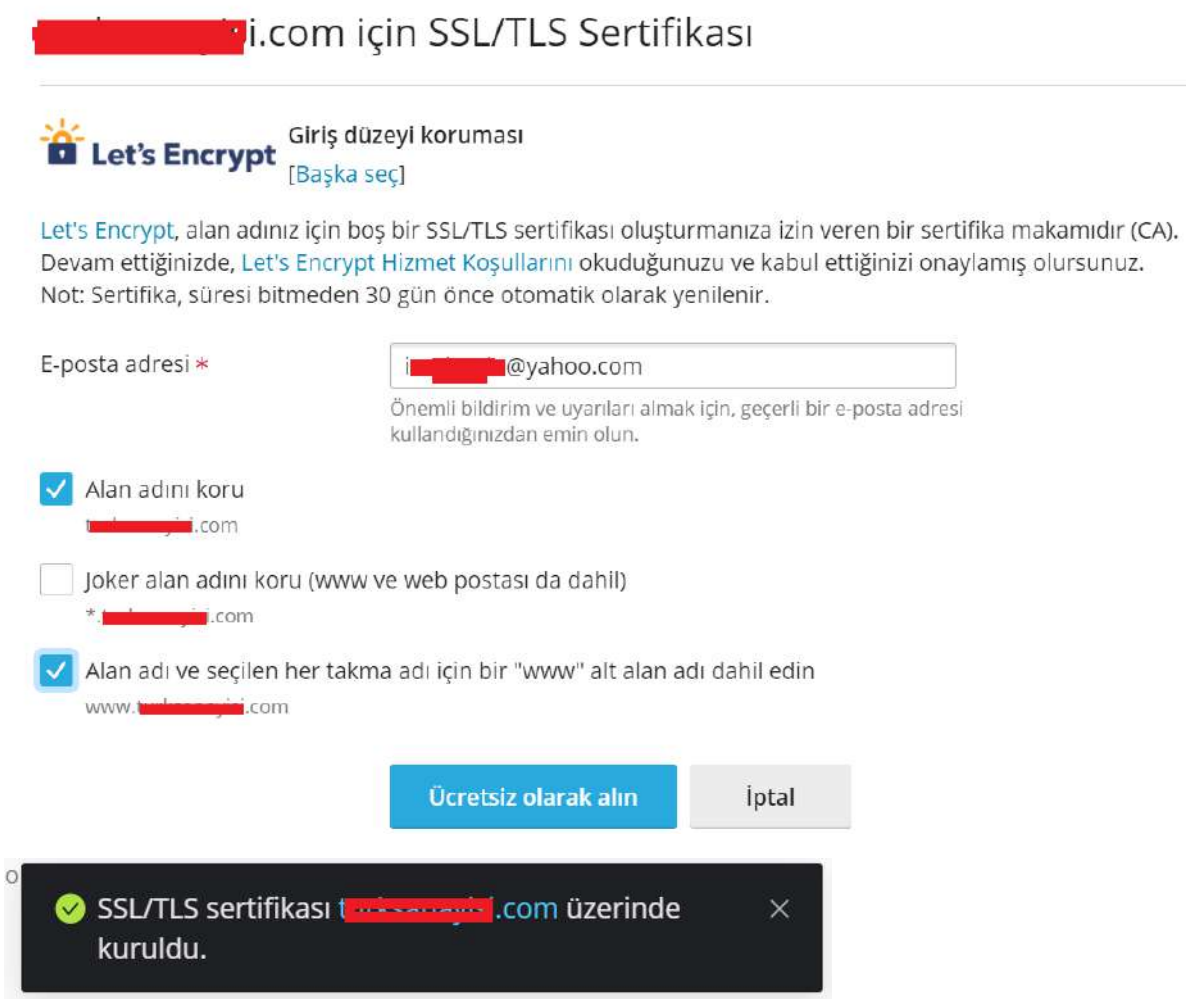
Kurulum yapmadan önce CSR kodu oluşturmalıyız (*CSR (Certificate Signing Request), web sunucunuz tarafından üretilen, içinde web sunucuya ve SSL sertifikasını kullanacak web sitesine dair bir takım veriler içeren koddur*). Bu kod SSL sertifikasının oluşturmak için gereklidir. SSL bu kod sayesinde web sitesinin verilerini otomatik olarak tespit eder.

Öncelikle Plesk panel üzerinden aşağıdaki menülerden SSL kısmını buluyoruz.



Biz hosting firmasının bize verdiği ücretsiz bir sertifikayı yükleyeceğiz.





Artık sitemizi https://www.s**********.com şeklinde yazdığımızda anahtarlı olarak çıktığını göreceğiz.

Kurs Kaynağı: Udemy-Komple Uygulamalı Web Geliştirme Kursu-Sadık Turan