ASP.NET CORE-MVC

İçindekiler

KULLANICI SAYFALARININ OLUŞTURULMASI

Ürünlerin Listelenmesi

Öncelikle uygulamamızı biraz düzenleyelim. Bu konuda değişiklik yapılan dosyalar aşağıda verilmiştir. Burada ProductController yerine TSController getirildi. _Card.cshtml yerine _Product.cshtml kullanımına geçildi.

Startup.cs	**HomeController.cs**
`using Microsoft.AspNetCore.Builder;` `using Microsoft.AspNetCore.Hosting;`	`using System;` `using System.Collections.Generic;`

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using TS.Data.Abstract;
using TS.Data.Concrete.EfCore;
using TS.Business.Abstract;
using TS.Business.Concrete;


namespace TS.WebUI
{
    public class Startup
    {
        public void
ConfigureServices(IServiceCollection services)
        {
            services.AddScoped<IProductRepository,
EfCoreProductRepository>();

            services.AddScoped<IProductService,
ProductManager>();

            services.AddControllersWithViews();
        }

        public void Configure(IApplicationBuilder app,
IWebHostEnvironment env)
        {
            app.UseStaticFiles(); //wwwroot içindeki
statik dosyaların okunabilmesi için konuldu

            if (env.IsDevelopment()) //Kodlar
Geliştirme aşamasında çalıştırılırken. Yayınlanmaya
daha geçilmediyse
            {
                SeedDatabase.Seed();

                app.UseDeveloperExceptionPage();
            }

            app.UseRouting();

            app.UseEndpoints(endpoints =>
            {

                endpoints.MapControllerRoute(
                    name: "default",
                    pattern:
"{controller=home}/{action=index}/{id?}"
                    );

            });
        }
    }
}
```

```
using Microsoft.AspNetCore.Mvc;
using TS.Business.Abstract;
using TS.Data.Abstract;
using TS.WebUI.ViewModels;

namespace TS.WebUI.Controllers
{
    public class HomeController : Controller
    {
        private IProductService _productService;
        public HomeController(IProductService
productService)
        {
            this._productService = productService;
        }

        public IActionResult Index()
        {
            var productViewModel = new
ProductListViewModel()
            {
                Products = _productService.GetAll()
            };

            return View(productViewModel);
        }

        public IActionResult About()
        {
            return View();
        }

        public IActionResult Contact()
        {
            return View("MyView");
        }
    }
}
```

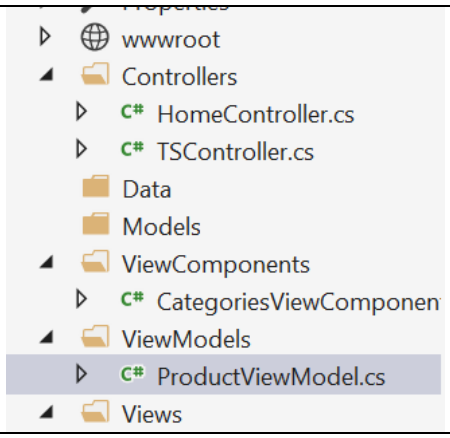
TSController.cs

```
using Microsoft.AspNetCore.Mvc;
using TS.Business.Abstract;
using TS.WebUI.ViewModels;

namespace TS.WebUI.Controllers
{
    public class TSController : Controller
    {
        private IProductService _productService;
        public TSController(IProductService
productService)
        {
            this._productService = productService;
        }

        public IActionResult List()
        {
            var productViewModel = new
ProductListViewModel()
```

ProductListViewModel.cs

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

namespace TS.WebUI.ViewModels
{
    public class ProductListViewModel
    {
        public List<Product> Products { get; set; }
    }

}
```

```
            {
                Products = _productService.GetAll()
            };

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



_Layout.cshtml

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>TurkSanayisi.com</title>
    <link href="~/css/bootstrap.min.css"
rel="stylesheet" integrity="sha384-
EVSTQN3/azprG1Anm3QDgpJLIm9Nao0Yz1ztcQTwFspd3yD65Vohh
puuCOmLASjC" crossorigin="anonymous">

</head>
<body>

    @await Html.PartialAsync("_NavBar")

    @RenderSection("HeaderSection", false)

    <main class="mt-3">
        <div class="container">
            <div class="row">
                <div class="col-md-12">
                    @RenderBody()
                </div>
            </div>
        </div>
    </main>

    <footer class="mt-3 py-5 bg-dark text-white text-
center">
        Hakkımızda | İletişim
    </footer>

    <script src="https://code.jquery.com/jquery-
3.4.1.min.js" integrity="sha384-
J6qa4849blE2+poT4WnyKhv5vZF5SrPo0iEjwBvKU7imGFAV0wwj1
yYfoRSJoZ+n" crossorigin="anonymous"></script>
    <link rel="stylesheet"
href="https://use.fontawesome.com/releases/v5.8.1/css
/all.css">

    @RenderSection("Scripts", false)
</body>
</html>
```

_NavBar.cshtml

```
@*********NAVBAR*************@
    <nav class="navbar bg-danger navbar-dark navbar-
expand-sm">
        <div class="container-fluid">

            <a href="/" class="navbar-
brand">TurkSanayisi</a>

            @*<ul class="navbar-nav me-auto mb-2 mb-lg-
0">*@
        <ul class="navbar-nav mr-auto">

            <li class="nav-item">
                <a asp-controller="TS" asp-
action="List" class="nav-link">Ürünler</a>
            </li>
            <li class="nav-item">
                <a href="" class="nav-link">Sepet</a>
            </li>

            <li class="nav-item">
                <a href="" class="nav-link">Sipariş</a>
            </li>

            <li class="nav-item">
                <a asp-controller="Admin" asp-
action="createProduct" class="nav-link">(Admin)Ürün
Ekle</a>
            </li>

            <li class="nav-item">
                <a asp-controller="Admin" asp-
action="productList" class="nav-
link">(Admin)Listele</a>
            </li>
        </ul>

            <form action="/product/list" class="d-
flex">
                <input name="q" type="text"
class="form-control me-2" placeholder="Arama yapın"
aria-label="Kelime Gir" />
                <button type="submit" class="btn btn-
outline-light mr-0">Arama</button>
            </form>

        </div>
    </nav>
```

TS>List.cshtml

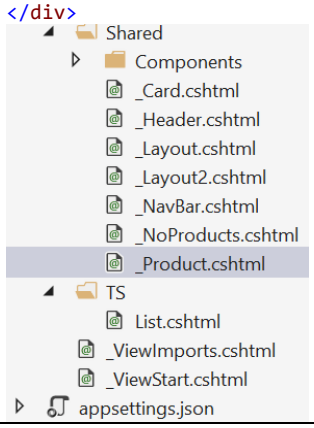
```
@model ProductListViewModel


<div class="row">
```

_Product.cshtml

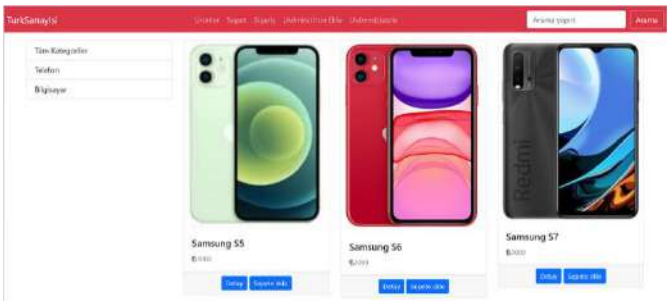
```
@model Product

<div class="card mb-2">
```

```
    <div class="col-md-3">
        @await Component.InvokeAsync("Categories")
    </div>
    <div class="col-md-9">
        <div class="row">
            @foreach (var product in Model.Products)
            {
                <div class="col-md-4">
                    @await
Html.PartialAsync("_product", product)
                </div>
            }
        </div>
    </div>
</div>


@section Scripts
{
    <script
src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js" integrity="sha384-
Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9IOYy5n3zV9zzTtmI3UksdQRVvoxMfooAo" crossorigin="anonymous"></script>
    <script
src="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/js/bootstrap.min.js" integrity="sha384-
wfSDF2E50Y2D1uUdj0O3uMBJnjuUD4Ih7YwaYd1iqfktj0Uod8GCExl3Og8ifwB6" crossorigin="anonymous"></script>
}
```

```
    <img class="card-img-top"
src="~/img/@Model.ImageUrl" alt="">
    <div class="card-body">
        <h5 class="card-title">@Model.Name</h5>
        <small class="text-muted">
            <i class="fas fa-lira-
sign"></i>@Model.Price
        </small>
    </div>
    <div class="card-footer text-center">
        <a asp-controller="TS" asp-action="Details"
asp-route-id="@Model.ProductId" class="btn btn-primary
btn-sm">Detay</a>
        <a asp-controller="TS" asp-action="AddtoCard"
asp-route-id="@Model.ProductId" class="btn btn-primary
btn-sm">Sepete ekle</a>
    </div>
</div>
```



Home/Index



TS/List

Katogori Menüsüne Bilgilerin Dinamik Olarak Veritabanından Getirilmesi

Yukarıdaki kodlarda kategori menüsüne bilgiler veritabanından getirilmiyordu. Bu iş için daha önce Product entitisi için ne yaptıysak bir benzerini burası içinde yapacağız.

Data>Abstract içinde zaten daha önceden oluşturduğumuz ICategorRepository.cs dosyamız vardı. Bu sınıf bir Interface idi. Fakat içinde kullanılacak metodlar her entity için kullandığımız IRepository içinde bulunmaktaydı. Yani IRepoisitory içindeki tüm işlemleri ICategoriRepository içinde geçerli olduğunu bileceğiz. Eğer sadece farklı bir işlem gerekiyorsa onu ICategoriRepository içinde kullanıyorduk. Burada örneğin popular kategoriler her entity için geçerli olmayabilirdi. Bunu kendi içerisine koymuştuk. Kodlarımız şu şekilde idi. Bu kodlar veritabanında yapılacak işlemleri göstermektedir.

ICategoryRepository.cs

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

namespace TS.Data.Abstract
```

IRepository.cs

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

namespace TS.Data.Abstract
{
```

```
{
    public interface ICategoryRepository :
IRepository<Category>
    {
        List<Category> GetPopularCategories();
    }
}
```

```
    public interface IRepository<TEntity>
    {
        TEntity GetById(int Id);
        List<TEntity> GetAll();
        void Create(TEntity entity);
        void Update(TEntity entity);
        void Delete(TEntity entity);
    }
}
```

Şimdi bu kodların dolu versiyonları olan EFCore içindeki sınıflarımızı oluşturalım. Burada EfCoreCategoryRepository içindeki metodları daha önce Product entitisinde de yaptığımız gibi EfCoreGenericRepository içinden alacaktır.

EfCore içindeki dosyalarımız şu şekilde oldu.

EfCoreCategoryRepository.cs

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

namespace TS.Data.Concrete.EfCore
{
    public class EfCoreCategoryRepository :
EfCoreGenericRepository<Category, TSContext>,
ICategoryRepository
    {
        public List<Category> GetPopularCategories()
        {
            throw new NotImplementedException();
        }
    }
}
```

EfCoreGenericRepository.cs

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

namespace TS.Data.Concrete.EfCore
{
    public class EfCoreGenericRepository<TEntity,
TContext> : IRepository<TEntity>
        where TEntity:class
        where TContext:DbContext,new()
    {
        public void Create(TEntity entity)
        {
            using(var context=new TContext())
            {
                context.Set<TEntity>().Add(entity);
                context.SaveChanges();
            }
        }

        public void Delete(TEntity entity)
        {
            using (var context = new TContext())
            {
                context.Set<TEntity>().Remove(entity);
                context.SaveChanges();
            }
        }

        public List<TEntity> GetAll()
        {
            using (var context = new TContext())
            {
                return
context.Set<TEntity>().ToList();
            }
        }

        public TEntity GetById(int Id)
        {
            using (var context = new TContext())
            {
                return
context.Set<TEntity>().Find(Id);
            }
        }

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

```
                context.SaveChanges();
            }
        }
    }
}
```

Şu anda Categori için veritabanından bilgileri getireceğimiz alt yapı hazırdır. Şimdi nasılki Product entitisinde bilgileri VT ye kaydetmeden önce bir kontrol yapmamız gerekiyorsa yani İş katmanına ihtiyaç duyuyorsak Category içinde iş katmanında kullanacağımız dosyaları oluşturalım. Abstract ve Concrete klasörü içinde Product entitisi için oluşturduğumuz dosyaları benzer şekilde oluşturalım. Son hali şu şekilde olacaktır.

TS.Business> Abstract> **ICategoryService.cs**

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

namespace TS.Business.Abstract
{
    public interface ICategoryService
    {
        Category GetById(int Id);
        List<Category> GetAll();
        void Create(Category entity);
        void Update(Category entity);
        void Delete(Category entity);
    }
}
```

TS.Business> Concrete> **CategoryManager.cs**

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

namespace TS.Business.Concrete
{
    public class CategoryManager : ICategoryService
    {
        private ICategoryRepository
_categoryRepository;

        public CategoryManager(ICategoryRepository
categoryRepository)
        {
            _categoryRepository = categoryRepository;
        }

        public void Create(Category entity)
        {
            //İş Kuralları buraya gelecek
            _categoryRepository.Create(entity);
        }

        public void Delete(Category entity)
        {
            _categoryRepository.Delete(entity);
        }

        public List<Category> GetAll()
        {
            return _categoryRepository.GetAll();
        }

        public Category GetById(int Id)
        {
            throw new NotImplementedException();
        }

        public void Update(Category entity)
        {
            throw new NotImplementedException();
        }
    }
}
```

Şimdi projemizin içindeki Startup.cs içine gidelim hangi Interface çağrıldığında (Abstrac lar içinde) hangi dolu versiyonu olan sınıf yapıları (Concrete lar içinde) çağrılacak ise benzer şekilde Category içinde yazalım.

Startup dosyasının içeriği şu şekilde olacaktır. Business projesine yeni eklediğimiz bu sınıfların buradan tanınması için önce Business katmanı build etmeyi ve ardından bu dosya içinde namespacelerini yukarı eklemeyi unutmayalım.

```
public void ConfigureServices(IServiceCollection services)
```

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

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

            //Bu yazı MCV deseni kullanımı için Controller Kullanımını Aktif ediyor.
            services.AddControllersWithViews();
        }
```

Artık Class yapılarımız hazırdır. Bu bilgileri kullanacağımız sayfaların tasarımlarını ayarlayalım.

Not: ViewComponents klasörü altındaki dosyalar sayfalardan bağımsız olarak (onların Model yapısını kullanmadan) kendi başına kendi modelini kullanarak bir kısım bilgiyi getirip sayfa içinde belli bir bölümde gösterme işlemidir. Yani sayfada bir bölgede bağımsız ayrı bir bölüm çalışacaksa ViewComponent yapısı kullanılabilir. Diğer türlü esas sayfanın Model yapısı içinde tüm Category ler taşınmak durumunda kalınırdı.

---------- 000----------

CategoriesViewComponent.cs dosyasının içerisine CategoryService eklemek için bir injection işlemi yapalım. Bunun için HomeController içerisinde yapılan injection işlemini örnek kod olarak kullanabiliriz. Son hali şu şekilde olmuş olacak. Böylece service üzerinden kontrol edilerek gelecek olan bilgiler sayfalarda kullanılabilecek. View sayfalarına GetAll() metodundan gelen categoriler gönderilebilecek. Bu satırların üst kısmındaki kodların anlamı ise şudur. Link içinde "List" şeklinde bir action kısmı var ise (ikinci seviye adres) ViewBag içerisinde seçilen kategorinin Id sini götürmektedir.

CategoriesViewComponent.cs

```
using Microsoft.AspNetCore.Mvc;
using TS.Business.Abstract;

namespace TS.WebUI.ViewComponents
{
    public class CategoriesViewComponent : ViewComponent
    {
        private ICategoryService _categoryService;
        public CategoriesViewComponent(ICategoryService categoryService)
        {
            this._categoryService = categoryService;
        }

        public IViewComponentResult Invoke()
        {
            if (RouteData.Values["action"].ToString() == "list")
                ViewBag.SelectedCategory = RouteData?.Values["id"];

            return View(_categoryService.GetAll());
        }
    }
}
```

Kategoriler sayfalarda alt bileşen olan Components sayfası türü ile gösterilmektedir. Shared>Componentes>Categories>default.cshtml dosyası aşağıdaki şekildedir.

Shared>Componentes>Categories>**default.cshtml**

```
@model List<Category>

<div class="list-group">
    <a asp-controller="Product" asp-action="List" class="list-group-item list-group-item-
action">Tüm Kategoriler</a>
```

```
    @foreach (var category in Model)
        {
            <a asp-controller="Product"
            asp-action="List"
            asp-route-id="@category.CategoryId"
            class="list-group-item list-group-item-action @(ViewBag.SelectedCategory ==
category.CategoryId.ToString()?"active":"")">
                @category.Name
            </a>
        }
</div>
```

Kategori menümüzü açılış sayfasının içine (Home/index) ve (TS/list) sayfalarının içine ekleyelim.

Home>**index.cshtml**

```
@model ProductListViewModel

    <partial name="_Header">


        <div class="row">
            <div class="col-md-3">
                @await
Component.InvokeAsync("Categories")
            </div>
            <div class="col-md-9">
                <div class="row">
                    @foreach (var product in
Model.Products)
                    {
                        <div class="col-md-
4">
                            @await
Html.PartialAsync("_Product", product)
                        </div>
                    }
                </div>
            </div>
        </div>

        @section Scripts
        {
            <script
src="https://cdn.jsdelivr.net/npm/popper.js@1
.16.0/dist/umd/popper.min.js"
integrity="sha384-
Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9IOYy5n3zV9zzTt
mI3UksdQRVvoxMfooAo"
crossorigin="anonymous"></script>
            <script
src="https://stackpath.bootstrapcdn.com/boots
trap/4.4.1/js/bootstrap.min.js"
integrity="sha384-
wfSDF2E50Y2D1uUdj0O3uMBJnjuUD4Ih7YwaYd1iqfktj
0Uod8GCExl3Og8ifwB6"
crossorigin="anonymous"></script>
        }
```

TS>**list.cshtml**

```
@model ProductListViewModel


<div class="row">
    <div class="col-md-3">
        @await
Component.InvokeAsync("Categories")
    </div>
    <div class="col-md-9">
        <div class="row">
            @foreach (var product in
Model.Products)
            {
                <div class="col-md-4">
                    @await
Html.PartialAsync("_Product", product)
                </div>
            }
        </div>
    </div>
</div>

@section Scripts
{
    <script
src="https://cdn.jsdelivr.net/npm/popper.js@1
.16.0/dist/umd/popper.min.js"
integrity="sha384-
Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9IOYy5n3zV9zzTt
mI3UksdQRVvoxMfooAo"
crossorigin="anonymous"></script>
    <script
src="https://stackpath.bootstrapcdn.com/boots
trap/4.4.1/js/bootstrap.min.js"
integrity="sha384-
wfSDF2E50Y2D1uUdj0O3uMBJnjuUD4Ih7YwaYd1iqfktj
0Uod8GCExl3Og8ifwB6"
crossorigin="anonymous"></script>
}
```

Ürün Detay Sayfasının Hazırlanması

TSController içindeki, List sayfamıza baktığımızda içinde solda kategorileri sağda ise ürünleri listelemekteyiz. Herbir ürün kartını incelediğimizde (_product.cshtml dosyası içi) içerisinde bulunan Detay butonunu Controller/Action/Id bilgileri aşağıdaki şekilde verilmiştir.

```
   <div class="card-footer text-center">
        <a asp-controller="TS" asp-action="Details" asp-route-id="@Model.ProductId" class="btn btn-primary btn-
sm">Detay</a>
        <a asp-controller="TS" asp-action="AddtoCard" asp-route-id="@Model.ProductId" class="btn btn-primary btn-
sm">Sepete ekle</a>
    </div>
```

Buna göre bizim “TS” controller içinde “Details” metodunu oluşturmamız gerekiyor. Bu metod girişinde Id bilgisini alacaktır. Yalnız bu Id bilgisi Null olabilir (nullable) manasında soru işareti (?) ile tanımlanmalıdır. Yani her zaman gelmek zorunda değildir. TSController içine eklenen kodlar aşağıdaki gibi olur.

```
       public IActionResult Details(int? Id)
        {
            //Eğer Id gönderilmedi ise bulunamadı sayfasına gidecek.
            if(Id==null)
            {
                return NotFound();
            }
            //Gönderilen Id ye göre veritabanından ilgili ürünün bilgilerini alıp product içinde tutacak.
            Product product = _productService.GetById((int)Id);

            //Eğer ürün veritabanından gelmedi ise Bulunamadı sayfasına gidecek.
            if(product == null)
            {
                return NotFound();
            }
            //Şayet ürünün içi dolu ise o zaman Views içindeki Details sayfasına gidecek.
            return View(product);
        }
```

Bundan sonra Details.cshtml sayfasını düzenlemeye geçelim. Bunun için Views>TS>Details.cshtml yolu içinde detay sayfasını oluşturacağız. Bu sayfanın içinde kategori kısmı olmasın. 3 e 9 luk bölme şeklinde görünümüz olsun. 3 lük bölmede resmi görüntüleyelim. 9 kısımda ürüne ait detay bilgiler gelsin. Açıklama kısmı daha uzun olacağından onu en alta 12 lik bir satır olarak ekleyelim. Sayfamız şu şekilde oldu.

Views>TS>Details.cshtml

```
@model Product

<div class="row">
    <div class="col-md-3">
        <img src="~/img/@Model.ImageUrl" class="img-fluid">
    </div>
    <div class="col-md-9">
        <h1 class="mb-3">
            @Model.Name
        </h1> <hr>
        <a href="#" class="btn btn-link p-0 mb-3">Telefon | Elektronik</a>
        <div class="mb-3">
            <h4 class="text-primary mb-3">
                <i class="fas fa-lira-sign"></i>@Model.Price
            </h4>
            <button type="submit" class="btn btn-primary btn-lg">Sepete Ekle</button>

        </div>
    </div>
</div>
<div class="row">
    <div class="col-md-12">
        <p class="p-3">@Model.Description</p>
    </div>
</div>
```

Bu sayfaya bilgileri Repository içindeki GetById Metodu ile getiriyorduk. Malum Repository Data katmanı içinde idi. Bu katmanın üzerine bir de Business katmanını eklemiştik. Business katmanı içindeki GetByID metodlarını da düzenlemeliyiz. Business>Concrete>ProductManager.cs içinde GetById metodu boş idi onu dolduralım.

Ürün Detay Sayfasına, Ürüne Ait Kategorilerin VT den Getirilmesi

Daha önceden hatırlayacağımız Ürün ile Kategoriler arasında çoka-çok bir yapımız vardı fakat bu yapıyı hiç kullanmamamıştık. Bu yapı TS.Entity projesinin içinde aşağıdaki şekilde oluşturulmuştu.

ProductCategory.cs

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

namespace TS.Entity
{
    public class ProductCategory
    {
        public int CategoryId { get; set; }
        public Category Category { get; set; }
        public int ProductId { get; set; }
        public Product Product { get; set; }
    }

}
```

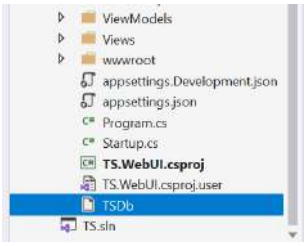
Bununla ilgili bilgileri sanal database içerisinde oluşturmak için aşağıdaki şekilde, daha önce kullandığımız TS.Data>Concrete>EfCore>**SeedDatabase** dosyası içine ekleyelim.

SeedDatabase.cs

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

namespace TS.Data.Concrete.EfCore
{
    public static class SeedDatabase
    {
        public static void Seed()
        {
            var context = new TSContext();

            if (context.Database.GetPendingMigrations().Count() == 0)
            {
                if (context.Categories.Count() == 0)
                {
                    context.Categories.AddRange(Categories);
                }

                if (context.Products.Count() == 0)
                {
                    context.Products.AddRange(Products);
                    context.AddRange(ProductCategories);
                }
            }
            context.SaveChanges();
        }

        private static Category[] Categories ={
            new Category() {Name="Telefon"},
            new Category() {Name="Bilgisayar"},
            new Category() {Name="Elektronik"}
        };
```

```
        private static Product[] Products ={
            new Product() {Name="Samsung S5", Price=1000, ImageUrl="1.jpg", Description="İyi Telefon",
IsApproved=true},
            new Product() {Name="Samsung S6", Price=2000, ImageUrl="2.jpg", Description="İyi Telefon",
IsApproved=false},
            new Product() {Name="Samsung S7", Price=3000, ImageUrl="3.jpg", Description="İyi Telefon",
IsApproved=true},
            new Product() {Name="Samsung S8", Price=4000, ImageUrl="4.jpg", Description="İyi Telefon",
IsApproved=false},
            new Product() {Name="Samsung S9", Price=5000, ImageUrl="5.jpg", Description="İyi Telefon",
IsApproved=true}
        };

        //her ürünün karşılığı olarak iki tane kategori atanmış olmaktadır.
        private static ProductCategory[] ProductCategories ={
            new ProductCategory(){Product=Products[0],Category=Categories[0]},
            new ProductCategory(){Product=Products[0],Category=Categories[2]},
            new ProductCategory(){Product=Products[1],Category=Categories[0]},
            new ProductCategory(){Product=Products[1],Category=Categories[2]},
            new ProductCategory(){Product=Products[2],Category=Categories[0]},
            new ProductCategory(){Product=Products[2],Category=Categories[2]},
            new ProductCategory(){Product=Products[3],Category=Categories[0]},
            new ProductCategory(){Product=Products[3],Category=Categories[2]}
        };

    }
}
```

Oluşturduğumuz bu verileri veritabanına aktarmak için daha önce yaptığımız işlemlerin bir benzerini yapacağız. Önce mevcut veritabanını bir silelim. Bu verilerin yeni oluşturacağımız veritabanına aktarılabilmesi için Migration yapılarının oluşması ve bunun üzerinden veritabanına verilerin aktarılması gerekecek. Yeni yapıların veritabanında oluşturmak için veritabanını silelim. Sıfırdan bir daha veritabanını oluşturalım. WebUI projesi içindeki TSDb ismini verdiğimiz veritabanını sağ tuşa tıklayıp silelim.



Veritabanını oluşturmadan önce Migrationlarımızı oluşturmuş olmamız gerekiyor. Bu işlemi daha önce yapmıştık. TS.Data içindeki migrationlarımızı incelediğimizde ProductCategory entitisine ait tablo oluşturma kodlarını daha önceki işlemlerimizde oluşturduğumuzu görürüz. Bu nedenle burada Migrationları tekrar oluşturmayacağız. Eğer ihtiyacımız olsaydı aşağıdaki şekilde migrationları oluştururduk. (c:\Users\pc1\Desktop\TS.com\TS\TS.Data\ dotnet ef migrations add InitialCreate –-startup-project ../TS.WebUI)(Dikkat daha önce oluşturulmuş ise InitialCreate den sonrası yazılmamalı)

Fakat burada migrationlarımız var olduğundan direk Veritabanını oluşturacağız. Bunun için (c:\Users\pc1\Desktop\TS.com\TS\TS.Data\) yolu içinde ilgili konuma CMD (komut satırı) ekranından geldikten sonra aşağıdaki kodları çalıştıralım. Bu komutta son kısım başlangıç projemizin WebUI olduğunu söylemiş oluyoruz.

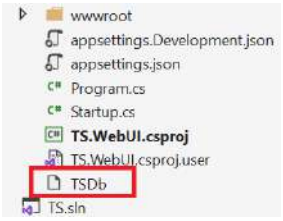
(dotnet ef database update --startup-project ../TS.WebUI)

```
C:\Users\pc1\Desktop\TS.com\TS\TS.Data>dotnet ef database update --startup-project ../TS.WebUI
```

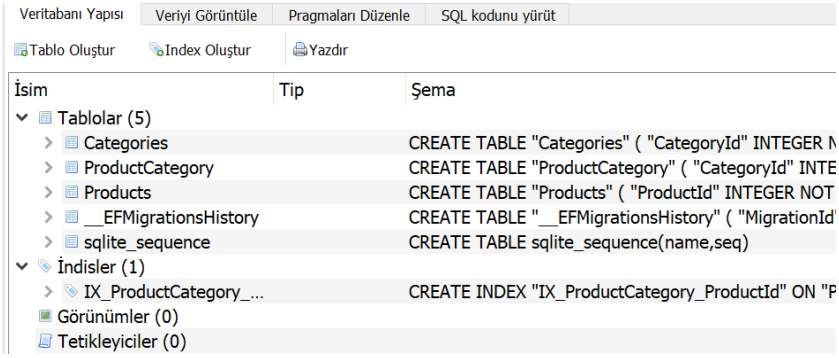
Veritabanını oluşturdu uyarısını verdi.

```
C:\Users\pc1\Desktop\TS.com\TS\TS.Data>dotnet ef database update --startup-project ../TS.WebUI
Build started...
Build succeeded.
Applying migration '20211010193002_InitialCreate'.
Done.

C:\Users\pc1\Desktop\TS.com\TS\TS.Data>
```

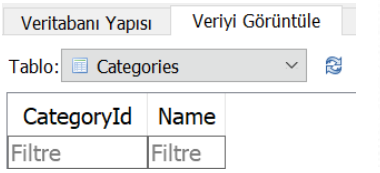
VS içindeyken Referesh yaparsak WebUI projesi içinde veritabanının oluştuğunu görürüz. Şimdi VT nını açıp içine bakalım. VT nımızın tipi SqLite olduğu için bilgisayarımızda DB Browser kurulu olması gerekir. Bunu daha önceden kurmuştuk.



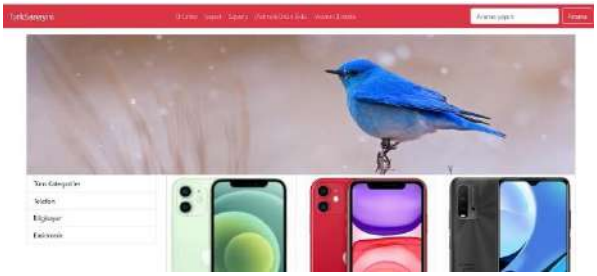
Veritabanını açtığımızda içinde tablolarımızın oluştuğunu görürüz.



Fakat tabloların içine baktığımızda içinin boş olduğunu görürüz. Bunun sebebi içerisine bilgileri TS.Data>Concrete>EfCore>SeedDatabse.cs sınıfının çalışmamasıdır. Bu sınıfın çalışması için WeUI içindeki Startup.cs dosyası içerisine aşağıdaki starları yazmıştık. Bu satır ise WebUI projesi derlendiğinde çalışmış olacağından devamında içinde çalıştırılan SeedDatabase.Seed() şeklindeki sınıfı ve metodu çalışacaktır.




```
public void Configure(IApplicationBuilder
{
    app.UseStaticFiles(); //wwwroot içinde

    if (env.IsDevelopment()) //Kodlar Geli
    {
        SeedDatabase.Seed();

        app.UseDeveloperExceptionPage();
    }
```



Data katmanındaki Repository bilgilerini düzenleme

Buraya kadar veritabanı yapısını hazır ettik. Şimdide Repository yapılarını hazır edelim.

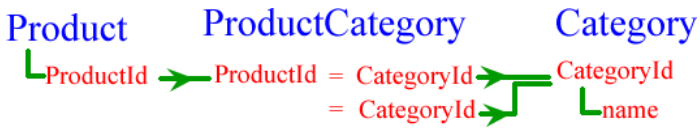
Detay sayfası görüntülenirken ürün bilgilerinin yanında kategori bilgilerinin de gelmesini istiyoruz. Bu işlem için IProductRepository içinde değişiklik yapmalıyız. Tabii bu repository içinde tüm metodlarımız bulunmuyor. Bu repository diğer genel olarak kullanılan metodları IRepository içinden yüklüyordu. IProductRepository içine aşağıdaki gibi yeni bir metod daha ekleyelim.

IProductRepository.cs

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

namespace TS.Data.Abstract
{
    public interface IProductRepository: IRepository<Product>
    {
        Product GetProductDetails(int Id);

        List<Product> GetPopularProducts();
    }
}
```

Concrete versiyonuna gelelim. Oradaki güncellemeleride yapalım. Bunun için EfCore içindeki EfCoreProductRepository.cs dosyasında güncellemeleri yapalım. Dosya içinde `IProductRepository ifadesi` hata gösterecektir. Bunun üzerine gelip "Implement Interface" (arabirimi uygula) ifadesini çalıştıralım. Bu durumda dosya içerisine sonradan eklenen GetPopularProducts() metoduna ait satırlar gelmiş olacaktır.

Bu metodun içerisinde bilgileri getirmek için TSContext sınıfını kullanacağız. Burada ürüne ait bilgileri getirirken oluşturulan sorguya dikkat etmemiz lazım. Ürün bilgilerini getirirken ürüne ait kategori bilgilerinide getireceğiz. Categori bilgilerine geçiş içinse arada ProductCategory sınıfını kullanacağız. Bunun için karmaşık bir sorgu yapısı oluşturmamız gerekecek.



`.Where(i=>i.ProductId==Id)` (Bu satır ile veritabanından getirmeye çalıştığımız ürünün ProductId si ile dışarıdan aldığımız Id bilgisini eşitlemeye çalıştık.

`.Include(i => i.ProductCategories)` Bu satır ile sorgumuzun içine bir başka tabloyu include (dahil) ediyoruz. İ.ProductCategories ifadesiyle ProductCategory tablosuna geçmiş oluyoruz. Eğer Product entitysinin tanımlandığı ilk sınıfın içine bakarsak (TS.Entity>Product.cs içerisinde) şu satırı görürüz. Burası geçiş için kullandığımız satır olmuş oluyor. `public List<ProductCategory> ProductCategories { get; set; }`

`.ThenInclude(i => i.Category)` satırında ise yapılan; Artık ProductCategory tablosuna geçtiğimiz için bu tablo içindeki Category bilgilerini almış oluyoruz. Böylece elmizde CategoryId ve Name bilgileri olmuş oluyor.

`.FirstOrDefault();` bu ifade ile Id bilgisine uyan bir Product varsa bulduğun ilk kaydı bana getir diyoruz. Ürün bilgilerini getir,rken yaptığımız Join işlemleri ile ekstradan o ürüne ait Category bilgileri de gelmiş oluyor.

Kodların son hali şu şekilde olmuş oldu.

EfCoreProductRepository.cs

```
using Microsoft.EntityFrameworkCore;
using System;
```

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

namespace TS.Data.Concrete.EfCore
{
    public class EfCoreProductRepository :
        EfCoreGenericRepository<Product, TSContext>, IProductRepository
    {
        public Product GetProductDetails(int Id)
        {
            using (var context = new TSContext())
            {
                return context.Products
                    .Where(i => i.ProductId == Id)
                    .Include(i => i.ProductCategories)
                    .ThenInclude(i => i.Category)
                    .FirstOrDefault();
            }
        }
        public List<Product> GetPopularProducts()
        {
            throw new NotImplementedException();
        }
    }
}
```

Servis katmanındaki (Business katmanı) bilgileri düzenleme

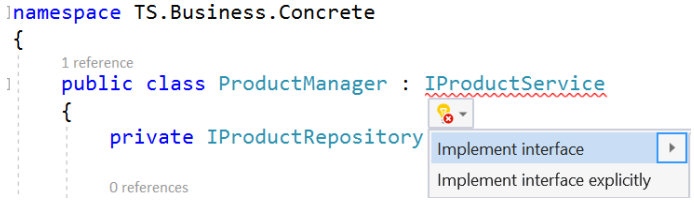
Data katmanından sonra Service katmanına (Businees katmanına) geçiyorduk. Oradaki işlemleride tamamlayalım.

TS.Business>Abstract klasörü içindeki IproductService isimli dosyamızın içine eksra bir metod daha ekleyelim. (Product GetProductDetails(int Id);)

IProductService.cs

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

namespace TS.Business.Abstract
{
    public interface IProductService
    {
        Product GetById(int Id);
        Product GetProductDetails(int Id);
        List<Product> GetAll();
        void Create(Product entity);
        void Update(Product entity);
        void Delete(Product entity);
    }
}
```

Bunun Concrete versiyonuna geçelim. Burada hata gösteren interface üzerinde “Implement interface” çalıştırılan ve yeni eklediğimiz metodun dolu hali bu sınıfın içine gelsin.



Kodlarımızın son hali aşağıdaki şekilde olacaktır. (önceki metodlara ait kodlar çıkarıldı, gösterilmedi)

ProductManager.cs

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

namespace TS.Business.Concrete
{
    public class ProductManager : IProductService
    {
        private IProductRepository _productRepository;
        public ProductManager(IProductRepository productRepository)
        {
            _productRepository = productRepository;
        }
//***
        public Product GetProductDetails(int Id)
        {
            //Bilgileri getirmeden önce eksra iş kuralları buraya yazılabilir
            return _productRepository.GetProductDetails(Id);
        }
//***
    }

}
```

WebUI Projesine GetProductDetails() Metoduna ait kodların Eklenmesi

Bundan sonra artık WebUI projesi içerisinde bilgileri görüntüleme sayfalarına eklenecek kodları düzenleyelim.

Önce TS.WebUI>TSControllers.cs dosyası içindeki Details metodu içindeki aşağıdaki satırı değiştirelim.

```
Product product = _productService.GetProductDetails((int)Id);
```

Burada product bilgisini direk sayfaya taşımak yerine ekstra bir model tanıtalım. Bu yazacağımız modeli TS.WebUI>Models içine de yazabiliriz yada TS.WebUI>ViewModels içine de yazabiliriz. Burada model klasörünü projedeki Entity ler için (product, category, order gibi) kullanıyorduk. Ama bu sınıfları TS.Entity katmanına aldığımızdan burası artık boştur.

Şimdi biz ekstra model tanımlamamızı ProductDetailModel.cs ismiyle Model klasörü içinde oluşturalım. Bu yapı bize hem bir ürün hemde bu ürüne ait kategorileri taşıyacak olan bir yapı olmuş olacak.

ProductDetailModel.cs

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

namespace TS.WebUI.Models
{
    public class ProductDetailModel
    {
        public Product Product { get; set; }
        public List<Category> Categories { get; set; }
    }
}
```

Modelimizi hazırladıktan sonra tekrar TSControllers.cs dosyası içine geçelim. Buradan View'e bilgileri gönderirken aşağıdaki şekilde kodlarımızı düzenleyelim.

```
return View(new ProductDetailModel
  {
      Product = product,
      Categories =product.ProductCategories.Select(i=>i.Category).ToList()
  });
```

Bu ifaded herbir gelen kategoriyi i değişkeni içine atıp içindeki Kategori bilgisini listeye dönüştürmekte ve bunları model yapımız içindeki Categories içine atmaktadır. Biz burada ekra bir sorgu göndermiş olmuyoruz. Zaten veritabanından bilgiler Include ederek sorgusu yapılmıştı. Burada yaptımız sadece Model yapısı içine sorgusu yapılmış Categorileri paketleyip atıyoruz. TSController.cs içindeki Details metodunun son hali aşağıdaki şekilde oldu.

TSController.cs

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using TS.Business.Abstract;
using TS.Entity;
using TS.WebUI.Models;
using TS.WebUI.ViewModels;

namespace TS.WebUI.Controllers
{
    public class TSController : Controller
    {
        private IProductService _productService;
        public TSController(IProductService productService)
        {
            this._productService = productService;
        }
//********* Diğer Metodlar Gösterilmedi ***********
        public IActionResult Details(int? Id)
        {
            //Eğer Id gönderilmedi ise bulunamadı sayfasına gidecek.
            if(Id==null)
            {
                return NotFound();
            }

            Product product = _productService.GetProductDetails((int)Id);

            //Eğer ürün veritabanından gelmedi ise Bulunamadı sayfasına gidecek.
            if(product == null)
            {
                return NotFound();
            }
            return View(new ProductDetailModel
            {
                Product = product,
                Categories =product.ProductCategories.Select(i=>i.Category).ToList()
            });
        }
    }
}
```

Artık paketlemiş olduğumuz bilgileri View>TS>Details.cshtml altında sayfamızda rahatlıkla kullanabiliriz. Dosyamızda önce `@model ProductDetailModel` şeklinde model tanımızı değiştirilim. (Dikkat dosya bu model adını tanımıyorsa _ViewImports.cshtml dosyasının içerisine `@using TS.WebUI.Models;` satırının eklenmesi gerekir)

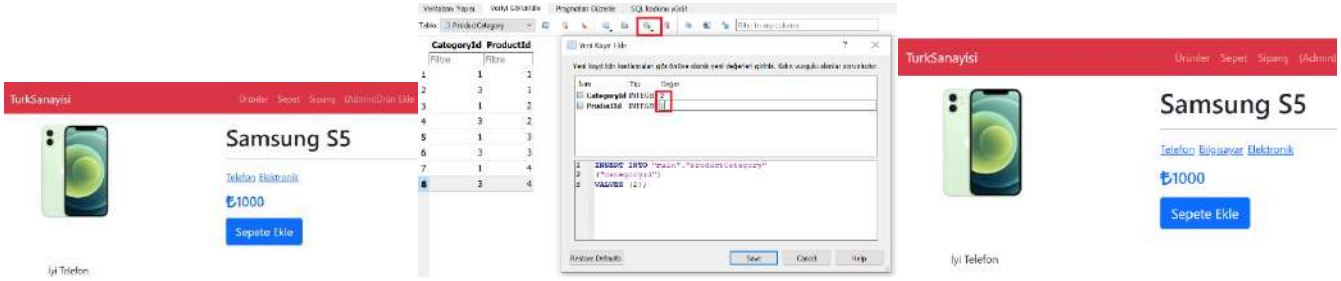
Artık ünümüze ait bilgileri sayfa içinde @Model.Product. .. şeklinde yazarak gösterebiliriz. Ayrıca kategorileri döngü şeklinde çekerek gösterelim. Kodların son hali aşağıdaki şekilde olur.

TS.WebUI>View>TS> **Details.cshtml**

```
@model ProductDetailModel

<div class="row">
    <div class="col-md-3">
        <img src="~/img/@Model.Product.ImageUrl" class="img-fluid">
    </div>
    <div class="col-md-9">
        <h1 class="mb-3">
            @Model.Product.Name
        </h1> <hr>
        @foreach (var Item in Model.Categories)
        {
            <a href="#" class="btn btn-link p-0 mb-3">@Item.Name</a>
        }
```

```
        <div class="mb-3">
            <h4 class="text-primary mb-3">
                <i class="fas fa-lira-sign"></i>@Model.Product.Price
            </h4>
            <button type="submit" class="btn btn-primary btn-lg">Sepete Ekle</button>

        </div>
    </div>
</div>
<div class="row">
    <div class="col-md-12">
        <p class="p-3">@Model.Product.Description</p>
    </div>
</div>
```

Artık uygulamamızı test edelim. Ürünün detay sayfasına kategori bilgileri veritabanından gelmiş olmaktadır. Veritabanından 1 numaralı ürüne bir kategori daha eklersek (ProductCategory tablosu içinden) sayfamızda 3 tane kategori gözükecektir. (Veritabanındayken değişiklikleri kaydet dememiz gerekir)



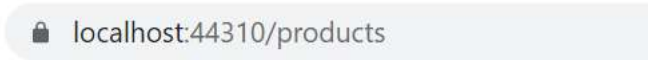
Kategoriye Göre Ürünleri Filtreleme

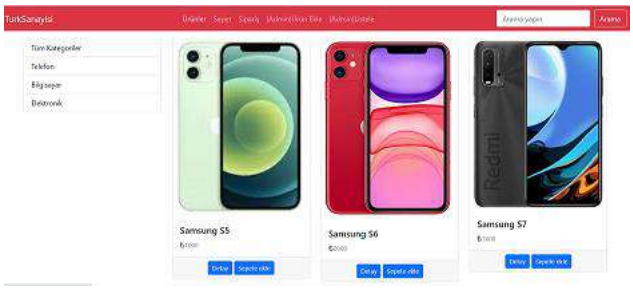
Biz ürünleri listelerken adres satırına "https://localhost:44310/**TS/List**" şeklinde adresimizi yazıyorduk. İçeridende linklerimizi bu şekilde vermiştik. Eğer linkten listeleme adreslerini kısaltmak istersek örneğin "../products" şeklinde yazıldığında yine aynı controller içindeki "list" metodunu çalıştırabiliriz. Bunun için Startup.cs içine link desenlemizi tanımlayabiliriz. Bunun için Configure metodunun içerisine aşağıdaki desenleri ekleyebiliriz.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "home",
        pattern: "home",
        defaults: new { controller = "Home",
action = "Index" }
    );
});
```

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "products",
        pattern: "products",
        defaults: new { controller = "TS",
action = "List"}
    );
});
```

Bu tanımlamalardan sonra https://localhost:44310/**TS/List** şeklinde kullandığımız bir linkin adresi tarayıcıda "../products" şeklinde gözükecektir. İçerisinde kod kısmında farklı yazıyor ama kullanıcı tarafından linkler kısa olarak gözükmüş oluyor. Böylece gerçek adresler gizlenmiş olmaktadır.





Eğer pattern ı (`pattern: "products/{category?}",`) şeklinde yazarsak son kısma istenirse bir kategori ismini de yazabileceğimiz bir link yapısı elde etmiş oluruz. (Örn: ../products/electronics) gibi. Adres satırında kategoriyi bu şekilde gönderdiğimizde kontroller içindeki metod aşağıdaki şekilde olması gerekir.

```
public IActionResult List(string category)
{
    var productViewModel = new ProductListViewModel()
    {
        Products = _productService.GetAll()
    };
    return View(productViewModel);
}
```

Şimdi biz veritabanından bu sorgulamayı yapacak alt yapıyı hazırlayalım. Bunun için Data projesi içindeki IProductRepository içine "GetProductsByCategory" isminde özel bir metod ekleyelim.

TS.Data> Abstract> **IProductRepository.cs**

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

namespace TS.Data.Abstract
{
    public interface IProductRepository: IRepository<Product>
    {
        Product GetProductDetails(int Id);
        List<Product> GetProductsByCategory(string name);
        List<Product> GetPopularProducts();
    }
}
```

Şimdi bunun Concrete klasörü içindeki versiyonunu düzenleyelim. Orada daha önce yaptığımız şekilde "Implement Interface" işlemini uygulayalım. Burada işlemi TSContext üzerinden yapacağız. Using şeklindeki kodları ekleyelim.

Burada yazdığımız ifadeleri tek tek açıklayalım.

`var products = context.Products;` Bu ifadenin sonuna .ToList() demediğimiz müddetçe çalışmaz. Yani veritabanına göndermeden önce başka ek sorgularda yazmak istiyoruz. Eğer bu satırın sonuna ToList() demiş olsaydık veritabanından tüm ürünleri getirirdi. Bundan sonra artık üzerine sorgu ekleyip kayıtları daraltsakta bir anlamı olmazdı. O yüzden bu ifadenin sonuna ToList() i en sonda ekleyeceğiz.

Şimdi bunun üzerine ekstra filtre ekleyelim. Metoda girişte gelen Name bilgisinin olup olmaması ile alakalı olarak ekstra bir sorgu yazalım.

`if(!string.IsNullOrEmpty(name))` satırı ile Name bilgisi boşluk yada boş değilse products dbSet üzerinden sonuna nokta işlemi ekleyerek sorgulama yapabiliriz.

`.Include(i => i.ProductCategories)` ifadesi ile ProductCategoires tablosuna gidecektir. Yani çok çokluk yapıyı oluşturan ara tablomuza geçiş yapmış oluyoruz. Böylece ilgili Product ın çoka çok talosu üzerinden Categori biglilerini almış oluyoruz.

`.ThenInclude(i => i.Category)` satırı ile ilgili product ğın kategori bilgilerini yüklemiş oluyoruz. İlk dahi edilen sorgu parçasını .Include ile sonrakileri .ThenInclude ile tanımlıyoruz.

`.Where(i => i.ProductCategories.Any(a => a.Category.Name == name))` satırı ile her ele aldığımız product ın ProductCategories lerine geçiş yapıyoruz ve .Any() ifadesi ile bir kayıt var mı yok mu bunu kontrol ediyoruz. Yani biz Any() ifadesi içerisine herhangi bir kriter yerleştireceğiz. Bu kritere uyan herhangi bir kayıt var mı yok mu ona bakacak, varsa o ürünün bilgisini getirecek. Devamındaki (`a => a.Category.Name == name`) satırı ile product kategorisindeki Name bilgisine ulaşacağız ve bu bilgi dışarıdan gönderilen name eşit mi ona bakacak.

Bu ifadenin sonunda hata işareti verdiğini görüyoruz. Üzerine gelip yardımı açtığımızda .Where den çıkan ifadenin List<> olduğunu görüyoruz ve bunun IQueryable olarak dönüştürülemediğini söylemiş oluyor.

Bu nedenle üstteki satırı aşağıdaki şekilde yazmamız gerekir. Sona eklenen AsQueryable() ifadesi ile burada Veritabanına sorguyu göndermeden önce üzerine ekstradan bir sorgu daha ekleyeceğim demektir. Bunu yazdıktan sonra aşağılarda ekstra olarak yazdığımız sorguları kabul etmiş olacak.

```
var products = context.Products.AsQueryable;
```

`return products.ToList();` veritabanına bu ifade yazılıncaya kadar gidilmiş olmuyor. Bu ifade yazıldıktan sonra veritabanına tüm oluşan sorguları götürüyor.

Özetle: ektra sorgumuzda 1. Satır ile ürün bilgilerini, 2. Satır ile Ürününün Kategorilerini, 3. Satır ile bulunan kategorilerilere ait bilgileri getirmiş oluyoruz. Sonra Where bloğu ile bir şart ekliyoruz. Şartımızı sonda ekliyoruz. Çünkü önce kayıtların bulunması gerekir. Where bloğu içerisinde `i => i.ProductCategories` kodları ile ilgili ürünün önce ProductCategorilerini yani ürüne ait kategorileri buluyoruz. `a => a.Category.Name` ifadesiyle buluna kategorilerin kategori bilgilerine geçiyoruz. Gönderdiğimiz isimde (name) herhangi bir Kategori adı varsa .Any() değeri bize true değeri gönderiyorsa bize o ürünü getir demiş oluyoruz.

```
products = products  1
    .Include(i => i.ProductCategories)  2
    .ThenInclude(i => i.Category)  3
    .Where(i => i.ProductCategories.Any(a => a.Category.Name == name));
```

Tabii burada veritabanından gelen Name ile dışarıdan gönderilen name içindeki kategori adları büyük küçük harf benzemezliği durumunda false gönderir. Bunu gözardı etmek için her ikisini de ToLower() dönüştürmemiz iyi olacaktır. Kodlarımızın son hali aşağıdaki gibi oldu.

EfCoreProductRepository.cs

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

namespace TS.Data.Concrete.EfCore
{
    public class EfCoreProductRepository :
        EfCoreGenericRepository<Product, TSContext>, IProductRepository
    {
//****çıkırılan metodlar var*****
        public List<Product> GetProductsByCategory(string name)
        {
```

```
            using (var context = new TSContext())
            {
                var products = context.Products.AsQueryable();

                if(!string.IsNullOrEmpty(name))
                {
                    products = products
                        .Include(i => i.ProductCategories)
                        .ThenInclude(i => i.Category)
                        .Where(i => i.ProductCategories.Any(a => a.Category.Name.ToLower() ==
name.ToLower()));
                }

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

Buraya kadar Data katmanındaki işleri bitirdik. Şimdi iş katmanındaki (Business) düzenlemeleri yapalım.

TS.Business.Abstracat> IProductService.cs içerisine benzer şekilde aşağıda ki gibi interface metodumuzu ekleyelim.

```
List<Product> GetProductsByCategory(string Name);
```

Bunun karşılığını Concrete içinde tanımlıyor olmamız gerekir. ProductManager içinde arabirimi uygula (imlement interface) diyerek kodları oluşturalım. Kodlarımız şu şekilde olacaktır.

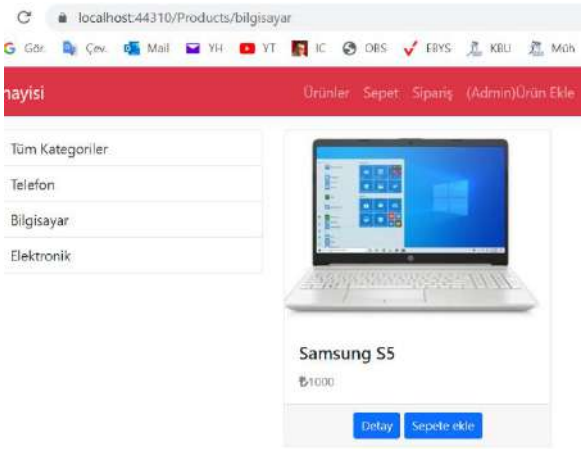
```
public List<Product> GetProductsByCategory(string name)
{
    // iş kuralları buraya yazılır
    return _productRepository.GetProductsByCategory(name);
}
```

Artık bu metodumuzu TS.WebUI>TSController içinde kullanabiliriz. Burada listeleme yaparken GetAll() metodunu kullanıyorduk. Bunun yerine yeni oluşturduğumuz metodu atayabiliriz.

```
        public IActionResult List(string category)
        {
            var productViewModel = new ProductListViewModel()
            {
                Products = _productService.GetProductsByCategory(category)
            };

            return View(productViewModel);
        }
```

Not: Buradaki düzenlemeleri yaparken projeden projeye her geçimizde bir önceki projenin Build edilmesi gerekir. Yoksa bir katmanda yapılan değişiklikler diğer katmanda gözükmeyecektir.

Kodları deneyip products/bilgisayar şeklinde linkimizi yazarsak bir tane bilgisayar gelmiş olur.



Kategori Linklerinin Düzenlenmesi

Kategorilerimizin linklerini adres satırında yazarken Id bilgisi ile yazabiliriz fakat böyle bir link kullanımı SEO (reklam ve bulunma) açısından dezevantaj oluşturur. Bu nedenle Url bilgilerimizin metinsel olması tercih edilir. Her adresin Url bilgisini veritabanı içinde tutabiliriz. Bu amaçla kategori bilgileri içine bir Url alanı ekleyelim.

Öncelikle TS.Entitiy projesi içinde sınıf yapısını değiştirelim.

TS.Entity> Category.cs

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

namespace TS.Entity
{
    public class Category
    {
        public int CategoryId { get; set; }
        public string Name { get; set; }
        public string Url { get; set; }

        //Bir kategoride birden fazla ürünü tutmak için
        public List<ProductCategory> ProductCategories { get; set; }
    }
}
```

Veritabanına kayıtlarımızı **SeedDatabase** (çekirdek database)içinden ekliyorduk. Oradaki sabit bilgilerimizide buna göre değiştirilim.

```
        private static Category[] Categories ={
            new Category() {Name="Telefon", Url="telefon"},
            new Category() {Name="Bilgisayar", Url="bilgisayar"},
            new Category() {Name="Elektronik", Url="elektronik"},
            new Category() {Name="Beyaz Eşya", Url="beyaz-esya"}
        };
```

Yönetim sayfaları hazırlandıktan sonra kayıtları alırken Url bilgilerini oradan gireceğiz. Şu aşamada bu şekilde yaptık. O kısımlar geldiği zaman Url linklerini kendisi oluşturması için bir metod yazabiliriz. Türkçe harfleri değiştirmesini, kelimele aralarındaki boşukları tire (-) koymasını otomatik yaptırıp Url linklerimizi kendimiz oluşturabiliriz.

Burada Url leri oluştururken daha önceden aynı isimde Url oluşturulmaması lazım. Bu nedenle VT ye kaydetmeden önce kontrol edilmesi gerekir. Aynı zamanda VT içinde bu sütünu Unique (tekil,benzersiz) olarak da işaretlenmiş olması gerekir.

Şimdi SeedDatabase içine eklediğimiz bilgilerin VT na aktarımı için daha önce yaptığımız gibi ya VT yi sileriz ve sıfırdan yeni bir Migration oluşturup VT yi hem oluşturma hemde bilgileri aktarma yapabiliriz.

Yada VT de bazı değişiklikler yaptıysak onların sıfırlanmaması için buradaki değişiklikleri oraya nasıl aktarırız, şimdi bu yöntemi deneyelim.

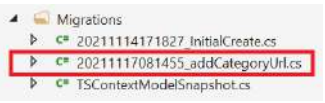
Bunun için yeni bir **ekstra migration** oluşturacağız. Tabii bu migration sadece Category tablosunun içine eklediğimiz Url alanını güncelleyecek. Sonradan eklediğimiz kayıtların içeriğini (ör. Beyaz Eşya gibi) atmaz. Biz bunları VT yapısı oluştuktan sonra kendimiz elle gireriz. Yukarıda kodların için bunu yazmamızın sebebi, bundan sonraki işlemlerde VT yi silip bir daha güncelleme yaparsak son hali yansısın diye yaptık.

Şimdi komut satırında (CMD) Data klasörüne konumlanalım. Buraya aşağıdaki kodları yazarak ekstra migrationımızı oluşturalım.

dotnet ef migrations add addCategoryUrl –-startup-project ../TS.WebUI

Ekstra migrationımız oluştu.

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

C:\Users\pc1\Desktop\TS.com\TS\TS.Data>
```



Bu migration Categories tablosu içerisine Url kolonunu ekleyecektir.

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.AddColumn<string>(
        name: "Url",
        table: "Categories",
        type: "TEXT",
        nullable: true);
}
```

Migration oluşturduktan sonra bilgilerin database aktarılması gerekir. Bunun için aşağıdaki komutu çalıştırabiliriz. Son kısmı database in oluştuğu projeyi refere etmektedir.

dotnet ef database update –-startup-project ../TS.WebUI

```
C:\Users\pc1\Desktop\TS.com\TS\TS.Data>dotnet ef database update --startup-project ../TS.WebUI
Build started...
Build succeeded.
Applying migration '20211114171827_InitialCreate'.
Failed executing DbCommand (3ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
CREATE TABLE "Categories" (
    "CategoryId" INTEGER NOT NULL CONSTRAINT "PK_Categories" PRIMARY KEY AUTOINCREMENT,
    "Name" TEXT NULL
);
```

(Not bu kod hata verdi. Veritabanı sıfırdan oluşturuldu. SeedDatabase içinde bilgiler daha önce yazılmıştı, WebUI projesi çalıştırılında bu bilgiler veritabanına eklenmiş oldu.

Şimdi kategorilerimizi görüntülediğimiz sayfa kodlarına gidelim. Sayfalar içinde kategorileri Component olarak görüntülüyorduk. Shared>Components>Default.cshtml içinde bu işi yapıyorduk.

Burada tüm Kategorileri `href="/products"` linki ile getirelim. Her hangi bir kategoriye tıkladığımızda ise artık Id bilgisi ile değil Url den gelen metin bilgisi ile çağıralım. Dosyanın son hali aşağıdaki şekilde oldu. Değişiklik yapılan yerler renkli gösterilmiştir.

View>Shared>Components>**default.cshmtl**

```
@model List<Category>

<div class="list-group">
    <a href="/products" class="list-group-item list-group-item-action">Tüm Kategoriler</a>

     @foreach (var Item in Model)
        {
            <a asp-controller="TS"
            asp-action="List"
            asp-route-category="@Item.Url"
            class="list-group-item list-group-item-action @(ViewBag.SelectedCategory == Item.Url?"active":"")">
                @Item.Name
            </a>
        }
</div>
```

Bu sayfaya gelmeden önce çalışan CategoriesViewComponent.cs dosyasının içerisinde ise Action ve Id bilgileri category bilgisi olacaktır. Gelen link bilgisinin içinde kategori bilgisini taşıdığımız category değişkeni boş değilse bu kategori adını, ViewBag içindeki SelectedCategory değişkeni içine yükleyip onu gittiğimiz sayfaya götürmüş olacağız. Buna göre kodları aşağıdaki şekilde düzenleyelim.

CategoriesViewComponent.cs

```
using Microsoft.AspNetCore.Mvc;
using TS.Business.Abstract;

namespace TS.WebUI.ViewComponents
{
    public class CategoriesViewComponent : ViewComponent
    {
        private ICategoryService _categoryService;
        public CategoriesViewComponent(ICategoryService categoryService)
        {
            this._categoryService = categoryService;
        }

        public IViewComponentResult Invoke()
        {
            if (RouteData.Values["category"] !=null)
                ViewBag.SelectedCategory = RouteData?.Values["category"];

            return View(_categoryService.GetAll());
        }

    }
}
```

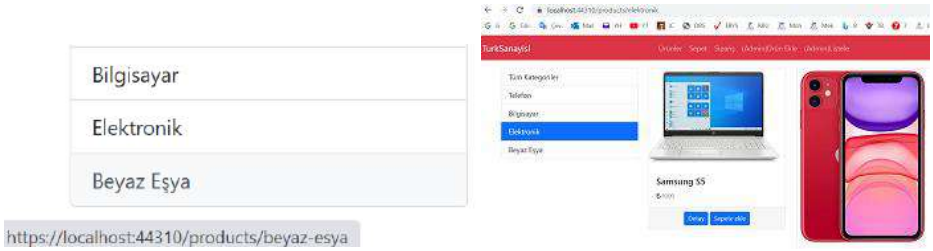
Data katmanındaki EfCoreProductRepository.cs içindede ufak bir değişiklik yapmamız gerekecek. Burada bize gelen kategori ismini artık Url olarak değiştirebiliriz. Burada büyük küçük harf dönüşümüne de gerek olmayacaktır. İlgili metodun kodları aşağıdaki şekilde olmuş olacaktır.

```
public List<Product> GetProductsByCategory(string name)
 {
     using (var context = new TSContext())
     {
         var products = context.Products.AsQueryable();

         if(!string.IsNullOrEmpty(name))
         {
             products = products
                 .Include(i => i.ProductCategories)
                 .ThenInclude(i => i.Category)
                 .Where(i => i.ProductCategories.Any(a => a.Category.Url == name));
         }

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

Son olarakda NavBar üzerinde ufak bir değişiklik yapalım. Burda ürünleri listelerken linkimizi

```
<li class="nav-item">
   <a href="/products" class="nav-link">Ürünler</a>
</li>
```

Artık kodlarımızı denediğimizde linklerin üzerine geldiğimiz zaman veritabanından atanan url linkine gittiğini ve adres çubuğunda da bu linkle gözüktüğünü görüyoruz. Bu şekilde linklerin metinsel görünümü SOE (arama motorlarının kolay indexlemesi ve sayfayı bulması için) için gereklidir.



Son olarak Details sayfasındaki kategorilerin linki yoktu. Oraya bu dinamik linklerimizi de ekleyelim.

```
@foreach (var Item in Model.Categories)
        {
            <a asp-controller="TS" asp-action="List" asp-route-category="@Item.Url" class="btn
btn-link p-0 mb-3">@Item.Name</a>
        }
```

Artık ürün detay sayfasındaki bir ürünün kategori ismine tıkladığımızda o kategoriye ait diğer ürünlerinde gösterildiği sayfaya geçebiliyoruz.

Artık bir sonraki uygulamamızda ürün detayları listelenirken Id bilgisini kullanıyorduk. Bunun yerine ürünün direk adını link içinde kullanalım. Bu uygulamayı yapalım.

Ürün Linklerinin Düzenlenmesi

Şu anda herhangi bir ürünün Detay butonuna tıkladığımızda o ürünün detay bilgilerini veren sayfa gelirken kullanılan link aşağıdaki şekilde çalışmaktadır. Yani TS controlleri altında Details action metodu ve 1 Id li ürün gelmektedir.

https://localhost:44310/TS/Details/1

Bunun yerine her bir ürünün tanımını içeren ve sadece ona ait olan bir Url hazırlayabiliriz. Aynı kategori mantığında olduğu gibi bunu yapabiliriz. Bunun için her bir ürünün Url bilgisini kendi veri tablosuna bir alan ekleyerek orada tutabiliriz.

Bunun için önce TS.Entity projesi içindeki Product.cs dosyasından ürün entitisi içine bir Url alanı ekleyelim.

TS.Entity>**Product.cs**

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

namespace TS.Entity
{
    public class Product
    {
        public int ProductId { get; set; }
        public string Name { get; set; }
        public string Url { get; set; }
        public double? Price { get; set; }
        public string Description { get; set; }
        public string ImageUrl { get; set; }
        public bool IsApproved { get; set; }

        //Bir ürünü birden fazla kategoride tutmak için
        public List<ProductCategory> ProductCategories { get; set; }
    }

}
```

Veritabanına kayıt bilgilerini eklemek içinde SeedDatabase.cs dosyası içindeki ürün bilgilerinin içine Url adreslerini ekleyelim.

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

Biz artık Id bilgisine göre değil Url bilgisine göre görüntüleme yapacağız. Bunun için Startup.cs içinde bir Route tanımlayacağız. Bu tanımlamanın pattern ine dikkat edersek {} parantezler içine yazılmıştır. Yani zorunlu değil, tercihe bağlı anlamındadır. Ayrıca başına ve sonuna herhangi bir ek adres konulmadı. Böylece domain adından sonra direk ürünün adı olan url bilgileri yazılarak çağrılacak. Yazılan bu adres TS controlleri altındaki Details metoduna gidecek. Parametre olarakda oraya ürün adını içeren url bilgisini götürecek.

```
// ../productdetails adres desenimiz
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "productdetails",
        pattern: "{url}",
        defaults: new { controller = "TS", action = "Details" }
    );
});
```

Buradaki adres desenini oluştururken `pattern: "{url}-{id}"` şeklinde tanımlasaydık Hem ürün adı hemde tireden sonra Id bilgisini vermiş olurduk.

Artık TSControllera gidelim. Buradaki Details metodu bizden bir string bilgi bekleyecek ve buradan productname route bilgisini alacak.

```
public IActionResult Details(string url)
{
    if(url == null)
    {
        return NotFound();
    }
    Product product = _productService.GetProductDetails(url);

    //Eğer ürün veritabanından gelmedi ise Bulunamadı sayfasına gidecek.
    if(product == null)
    {
```

```
            return NotFound();
        }
        //Şayet ürünün içi dolu ise o zaman Views içindeki Details sayfasına gidecek.
        return View(new ProductDetailModel
        {
            Product = product,
            Categories =product.ProductCategories.Select(i=>i.Category).ToList()
        });
    }
```

Şimdi TSData projesi içinde Abstract>IProductRepository.cs içini düzeltelim. Burada ilgili metod yine

```
Product GetProductDetails(string url);
```

Bu düzeltmeyi yaptıktan sonra EfCoreProductRepository içindeki satırları da düzenleyelim.

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

Aynı işlemleri Business katmanında da yapmalıyız. TS.Businees katmanı içindeki Abstract>IProductService içindeki satırı aşağıdaki şekilde değiştiririz.

```
Product GetProductDetails(string url);
```

Concrete kodları içindeki ProductManager içindeki kodları da aşağıdaki şekilde değiştirelim.

```
public Product GetProductDetails(string url)
{
    //Bilgileri getirmeden önce eksra iş kuralları buraya yazılabilir
    return _productRepository.GetProductDetails(url);
}
```

Ürünleri listeleyip Detay butonunadığımızda yanımızda Id bilgisi değilde url bilgisini götüreceğiz. Bunun için Views>TS>List.cshtml dosyasını ürünleri listemek için kullanıyorduk. Bu sayfa içinde Shared>_product.cshtml partial dosyası ile ürün bilgisini görüntülüyorduk. _product.cshtml dosyası içindeki Detay butonun olduğu bölüm aşağıdaki şekilde değiştirilir.

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

Sayfa ve sınıf yapılarımız hazırdır. Ama Veritabanında şu anda bir Products tablosunun içinde bir url alanı yoktur. Veribanını sıfırdan bir daha oluşturup yeni yapı ile bilgileri yüklemek istersek veritabanını silip Migrationları bir daha oluşturmalıyız ve ardından update işlemini çalıştırmalıyız. Yada bunun yerine ek bir migration olupturup bununla Products tablosunun içne url alanını ekleyebiliriz. Ama bu eklemede içi null olarak dolacaktır. Bilgileri elle eklememiz gerekecektir..

Biz database silerek işlemleri yapalım. Bunun için komut satırından aşağıdaki komutu çalıştıralım. Migrationlarımız Data projesi içinde olduğu için oradan drop komutunu çalıştıracağız. Ama Veritabanı WebUI içinde olduğundan sona onuda eklemeliyiz. Komut satırı aşağıdaki gibi olur.

```
dotnet ef database drop –-startup-project ../TS.WebUI
```

```
C:\Users\pc1\Desktop\TS.com\TS\TS.Data>dotnet ef database drop --startup-project ../TS.WebUI
Build started...
Build succeeded.
Are you sure you want to drop the database 'main' on server 'TSDb'? (y/N)
y
Dropping database 'main' on server 'TSDb'.
Successfully dropped database 'main'.
```

Bu işlemden sonra artık veritabanımız silinmiş oldu. Eğer projelerin birinde build işleminde bir hata alınırsa burada bu komut çalışmazdı. Çünkü önce build kontrolü yapıyor. Kodlarda herhangi bir hata yoksa komut çalışmış oluyor.

Burada aslında Windows tan yada proje içinden veritabanımızı direk silebilirdik. Ama onun yerine farklı bir yöntem kullanmış olduk.

Veritabanımız artık yok. Migrationlarımız eski yapıya göre hazırlanmıştı. Dolayısı ile yeni yapının migrationlara eklenmesi için ya ekstra bir migration oluşturmalıyız yada migration klasörünü kaldırıp bütün hepsini sıfırdan oluşturmalıyız. Bu ikincisini tercih edelim. Data projesi içinden Migration klasörünü siliyoruz. Yeni yapı ile migraion oluşturmak için aşağıdaki komutu çalıştıralım.

dotnet ef migrations add InitialCreate –-startup-project ../TS.WebUI

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

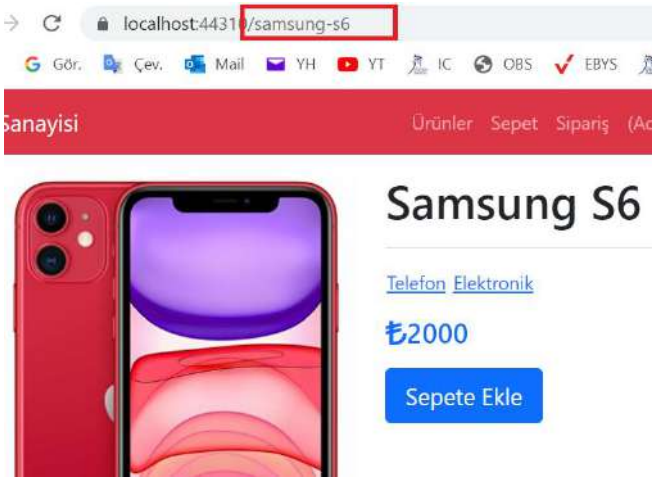
Dikkat: migration komutu çalışabilmesi için yukarıda migration klasörü silindikten sonra Build etmemiz gerekir. Yani kısaca projeler yaptığımız her değişiklikte bir build işlemi gerekir. Şimdi migration yapılarının veritabanına aktarmak ve aynı zamanda veritabanını oluşturmak için aşağıdaki komutu çalıştıralım.

dotnet ef database update --startup-project ../TS.WebUI

Bu işlem sonunda WebUI projesi içinde TSDb isminde veritabanımız oluştu fakat içinde veriler yoktur. Sadece yapı oluşturuldu. İçerine verilerin dolması için SeedDatabase dosyasının çalıştırılması gerekir. Bu işlem için ilgili satırlar Startup projesinin içine eklenmişti. Dolayısı ile projemizi f5 ile çalıştıralım ve bu bilgiler Veritabanına dolsun. VT dolduktan sonra sayfalarda bilgiler gelecektir.

Not: WebUI içinde oluşan veritabanın tabloları oluşmadı ise veritabanını Data projesinde oluşması için (dotnet ef database update) şeklinde çalıştırın ve ardından bu dosyayı WebUI içine taşıyın. Sonra F5 ile verileri aktardığımızda sayfalar dolu olarak gelecektir.

Artık ürünlerimizin detay bilgileri Id bilgisi ile değil Url bilgisi ile getirilmiş olmaktadır.



Bu şekilde link içinde ürünün adının geçmesi SOE (reklam ve bulunma) açısından önemlidir. Sadece ürünün adı değil, kategori bilgileri ve diğer önemli görülen bilgilerde bu Url içine eklenebilir.

Burada linklerimizi karşılayan starup.cs dosyamızın içindeki desen yapılarının iyi anlaşılması gerekir. Burada link karşılıklarını bakarken en yukarıdan taramaya başlar. Slaştan sonra bir metin bulduğunda önce onun sabit karşılığı var ise o önce gelmelidir. Örneğin: localhost/about şeklinde bir link yazdığımızda about kelimesi bir ürünün adı değildir. Sabit olarak kullandığımız bir metindir. Buna ait desen ürün desenlerinden önce gelmelidir. Eğer linkimiz localhost/samsung şeklinde olursa about deseninde karşılığı olmayacağından bir sonraki desene geçer. Burada ise pattern: "{url}" şeklinde bir desn yapısı görünce burayı çalıştırır. Çünkü bu link Lokalhost/{degişken} demektir. Yani {} içine yazılması değişebilen bir ifade olduğunu gösterir. Parantezler olmasaydı o zaman ifadeyi birebir arayacaktı. pattern: "products/{category?}" deseni ise bize şunu anlatır. Localhost/products/{degişken} şeklindedir. Products ilk bölmede sabit olarak arayacaktır. İkinci bölmede ise artık hangi ifade gelirse o bilgileri category değişkeni içine atacaktır. Bütün bunları sırasıyla aşağı doğru tarayacaktır, hiçi birini çalıştıramazsa en sondaki default linki çalıştıracaktır.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseStaticFiles(); //wwwroot içindeki statik dosyaların okunabilmesi için konuldu

    if (env.IsDevelopment()) //Kodlar Geliştirme aşamasında çalıştırılırken. Yayınlanmaya daha geçilmediyse
    {
        SeedDatabase.Seed();

        app.UseDeveloperExceptionPage();
    }

    app.UseRouting();

    // Localhost/about şeklinde sabit adres deseninde burası çalışır
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "about",
            pattern: "about",
            defaults: new { controller = "Home", action = "About" }
        );
    });


    // Localhost/samsung şeklinde değişken desen yapısında burası çalışır
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "productdetails",
            pattern: "{url}",
            defaults: new { controller = "TS", action = "Details" }
        );
    });

    // localhost/products şeklinde sabit adres deseninde burası çalışır, aynı zamanda
    // localhost/products/bilgisayar şeklinde bir desen geldiğinde yine burası çalışır. ikinci kısım değişkenlidir
ve zorunlu değildir.
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "products",
            pattern: "products/{category?}",
            defaults: new { controller = "TS",  action = "List"}
        );
    });

    // localhost/  şeklinde geldiğinde yukarıdaki hiçbir desen çalışmadığında burası çalışır.
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=home}/{action=index}/{id?}"
            );
```

```
    });
}
```

Bir sonraki dersimizde sayfalama işlemlerine geçelim. Linkimiz içinde safya numarasını göndererek o sayfaya karşılık gelen ürünleri veritabanından çekiyor olmamız gerekir. Diğer türlü tüm bilgileri VT'den çekmek mantıklı olmaz.

Sayfalama Yapısının Hazırlanması

Ürün sayımız yüzlerce binlerce olabilir. Bu ürünlerin hepsini listelemek istemeyiz. Ürünleri sayfalarına göre ve aynı zamanda kategori seçimi altında da sayfalama isteyebiliriz. "localhost/products?page=1" şeklinde bir link verdiğimizde 10 ar ürünü sayfalama yapıyorsak ilk 10 ürünü getirir. 2. Sayfayı seçersek sonraki on ürünü getirir. Buradaki mantık veritabanından bir anda tüm ürünleri çekmeden ilgili on tane ürünü gösterme işlemi olmuş olmaktadır.

Şu aşamada link içindeki sayfalamaları biz elimizle ekleyelim. Daha sonra ürün listesinin altındaki sayfa numaralarından getirelim.

Burada link desenimizi oluştururken

pattern:"products/{category?}/page=1 şeklinde üçüncü bölme içinde verebiliriz yada

pattern:"products/{category?}?page=1 şeklinde ikinci bölmenin sonuna sorgu ifadesi olarak (Querystring) ekleyebiliriz. ikinci yöntemi kullanacağız fakat link desenimiz şu şekilde kalacak

pattern:"products/{category?}"

Buna göre startup.cs içindeki desen yapısı ile kodlarımız aynı kalsın.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "products",
        pattern: "products/{category?}",
        defaults: new { controller = "TS",  action = "List"}
    );
});
```

Burada TS.Controller içindeki List() metoduna bir talep göndereceğiz. Bu talebimiz örnek olarak şu şekilde olacak.

Localhost/products/telefon?page=1

Bu şekilde bir istek gönderirsek list motodu "telefon" ifadesini "category" ifadesi içine atacaktır. ? işaretinden sonra gelen her bir parametreyide sırasıyla metod girişindeki değişkenlere atacaktır. Dolayısı ile "page=1" parametresi içindeki bilgiyide yine metod girişindeki "page" değişkeni içine atacaktır. Bu ifade farklı da yazılabilirdi.

```
public IActionResult List(string category, int page)
```

Ek olarak ikinci bir parametrede gönderebilirdik. Örneğin istek şu şekilde olursa;

"Localhost/products/telefon?page=1&color=red" bu durumda metod girişinde parametrelerin alınması

```
public IActionResult List(string category, int page, string color) şeklinde yazılır.
```

Kontroller içindeki **List** metodumuzda aşağıdaki şekilde değişiklik yapalım. Burada "int page=1" ifadesi ile hangi sayfanın görüntüleneceğini dışarıdan alıyoruz. Fakat bazen link içinde sayfalama olmayabilir. Böyle durumda içerisi tanımlanınca 0 olacağından sayfalama oluşmaz. O nedenle tanımlamanın ilk yapıldığı yerde içerisine 1 atıyoruz. Yani "int page=1" ifadesi page değişkenini int olarak tanımlar ve varsayılan ilk değerini 1 atar. Şayet dışarıdan farklı bir değer gelirse o değişkenin içine atılır.

"const int pageSize=3" ifadesinde ise tanımlamanın başına const koyuyoruzki aşağılarda bir yerde bu tanımlama bir daha değiştirilmesin (malun değişkenler içerisine yeni atanan değeri tutar. Bu değerden başka bir değer atanamasın). Sayfadaki ürün adetini 3 gibi küçük tuttuk. Çünkü şu anda fazla deneme ürünümüz yok. Gerçek kullanımda bu sayı 10 olarak değiştirilebilir. Ardından bu bilgiler artık metodumuza gönderilebilir.

```
public IActionResult List(string category, int page=1)
{
    const int pageSize = 3;

    var productViewModel = new ProductListViewModel()
    {
        Products = _productService.GetProductsByCategory(category, page, pageSize)
    };
    return View(productViewModel);
}
```

Belirlediğimiz bu parametreleri TS.Data/Abstract içindeki **IProductRepository**'de tanımlamamız gerekir.

```
List<Product> GetProductsByCategory(string name, int page, int pageSize);
```

Bu prametrelere göre gerekli düzenlemeyi Concrete versiyonu olan **EfCoreProductRepository** içinde yapmalıyız. Bu metod içinde .ToList() komutu çalıştığında sorgu çalıştırılmış oluyordu. Tam bu komut çalıştırılmadan önce sorgumuzu düzenlememiz gerekiyor. ToList() komutumuz aşağıdaki şekildeydi. Bunu düzenleyelim.

```
return products.ToList();
```

Bu komut içerisini aşağıdaki şekilde yazarsak bunun anlamı şudur. 9 ürünü atlayacak ve bundan sonra 3 tane ürünü alacak anlamındadır.

```
return products.Skip(9).Take(3).ToList();
```

Dolayısı ile buradaki ifadeyi bizim sayfada kullandığımız değişkenlerle yazarsak şu şekilde olacaktır. Yani page'in içerisi 1 gelirse, 1-1=0 olur ve dolayısı ile Skip(0) olmuş olur ve en baştan okumaya başlar. Take() metodu ile de kaçtane ürünü getireceği belirlenmiş olur.Bu durumda bizim genel uygulamamız aşağıdaki şekilde olacaktır.

```
return products.Skip((page-1)*pageSize).Take(pageSize).ToList();
```

EfCoreProductRepository.cs içindeki kodlarımızın son hali aşağıdaki şekilde oldu.

```
public List<Product> GetProductsByCategory(string name, int page, int pageSize)
{
    using (var context = new TSContext())
    {
        var products = context.Products.AsQueryable();

        if(!string.IsNullOrEmpty(name))
        {
            products = products
                .Include(i => i.ProductCategories)
                .ThenInclude(i => i.Category)
                .Where(i => i.ProductCategories.Any(a => a.Category.Url == name));
        }

        return products.Skip((page-1)*pageSize).Take(pageSize).ToList();
    }
}
```

Parametre düzelmelerini Business katmanındaki IProductService ve ProductManger içinde de düzeltelim.

IProductService

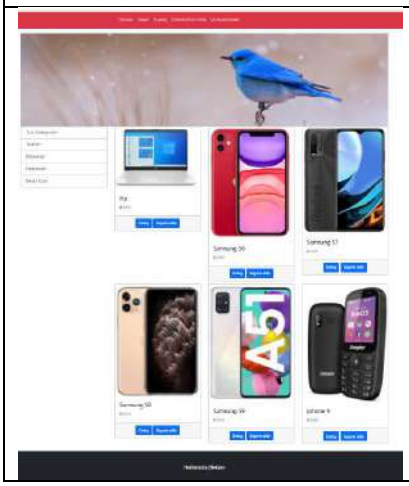
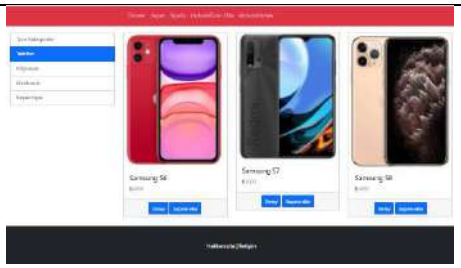
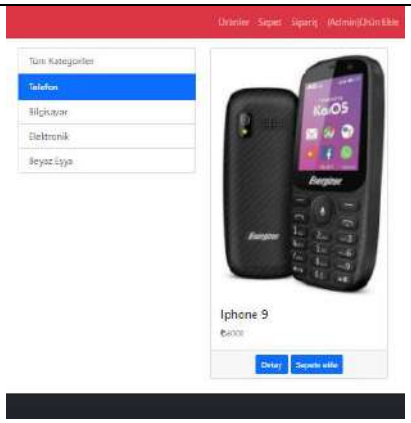
```
List<Product> GetProductsByCategory(string name, int page, int pageSize);
```

ProductManager

```
public List<Product> GetProductsByCategory(string url, int page, int pageSize)
        {
```

```
            // iş kuralları buraya yazılır
            return _productRepository.GetProductsByCategory(url, page, pageSize);
        }
```

Kodları test ettiğimizde ilk başta kategori vs belirlemediğimiz için tüm ürünler geldi. Kategorilere tıkladığımızda ise ürünler sayfada 3 adet olarak geldi.

https://localhost:44310/	https://localhost:44310/products/telefon	https://localhost:44310/products/telefon?page=2
		

Burada sayfalama linklerini elle yazdık. Bundan sonraki derste bunları listenin altındaki butonlardan çalıştıralım.

Sayfalama Link Verilerinin Hazırlanması

Ürünleri sayfalarken listenin altından butonlara tıklayarak ilgili sayfaların getirilmesi gerekir. İlk olarak Controllerdan View e aktarılacak olan verilerin hazırlanmasını yapalım.

Şu ana kadar linklerimiz aşağıdaki şekillerde çalışır hale getirildi.

localhost/products
localhost/products/?page=1
localhost/products?page=1
localhost/products/category
localhost/products/category/?page=1
localhost/products/category?page=1
Biz bu yapılara göre sayfalama linklerini oluşturuyor olmamız gerekir.

View üzerinde sayfa sayısını gösterebilmek için veritabanında kaç tane ürünün olduğunun bilinmesi gerekir ve her sayfada kaç ürün olacağını da biliyorsak kaç safya olduğu hesaplanabilir. Bu bilgilerin View gitmesi gerekir. View de kendisine gelen sayfalama bilgilerine göre dinamik olarak sayfalama linklerini oluşturacaktır.

Bu nedenle Controller içerisinde View gidecek olan bilgilerimizin önce bir paketlenme işlemini yapmamız gerekir. Daha sonraki derstede gelen bilgilere göre linkleri oluşturalım.

View gidecek olan bilgileri Model içinde tanımlayalım. Bunun için “ViewModels>ProductViewModel.cs” içerisini aşağıdaki şekilde düzenleyelim. Burada PageInfo şeklinde oluşturduğumuz class ile gidecek bilgileri paketlmiş oluyoruz.

Burada üç tane bilginin amacı bellidir ama 4 seçenek olan CurrentCategory seçeneği ise listeleme yaparken bir kategori seçimi yapılmış ise ona ait bilgileri tutmak için kullanılacak. Çünkü yukarıdaki örnek linklermizde kategori

seçimi de bulunmaktadır (localhost/products/category/?page=1). Eğer bir kategori seçimi yapılmış ise sol tarafta buna ait linkin maviye boyanmış olması da gerekir. Bu bilgi orada da kullanılacak.

Burada ayrıca sayfa sayısını hesaplamak için ufak bir alt fonksiyon yazılmıştır. Toplam sayfa sayısı bir üst integera yuvarlatılarak hesaplanmalıdır. Örneğin 10 tane ürün varsa ve her sayfada 3 ürün görüntüleniyorsa 10/3=3.3 sayfa eder. Böylece 4 tane sayfaya ihtiyaç olacaktır. Buna göre model yapımız aşağıdaki şekilde oldu.

ViewModels>**ProductViewModel.cs**

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

namespace TS.WebUI.ViewModels
{
    public class PageInfo
    {
        public int TotalItems { get; set; }

        public int ItemsPerPage { get; set; }
        public int CurrentPage { get; set; }
        public int CurrentCategory { get; set; }

        public int TotalPages()
        {
            return (int)Math.Ceiling((decimal)TotalItems / ItemsPerPage);
        }
    }
    public class ProductListViewModel
    {
        public PageInfo PageInfo { get; set; }

        public List<Product> Products { get; set; }
    }

}
```

Bu model yapımızı Controller içinden sayfaya taşıyor olmamız gerekir. Bu amaçla TSController.cs dosyası içinde List() metodu aşağıdaki şekilde değiştirildi.

```
public IActionResult List(string category, int page=1)
{
    const int pageSize = 3;

    var productViewModel = new ProductListViewModel()
    {
        PageInfo = new PageInfo()
        {
            TotalItems = _productService.GetCountByCategory(),
            CurrentPage = page,
            ItemsPerPage = pageSize,
            CurrentCategory = category
        },

        Products = _productService.GetProductsByCategory(category, page, pageSize)
    };
    return View(productViewModel);
}
```

Burada geçen GetCountByCategory() metodu IProductService içinde yoktur. Bu yüzden hata gösterir. Bu metodu ilgili TSBusiness>Abstract>IProductService.cs içine ekleyelim.

```
   public interface IProductService
    {//…
        int GetCountByCategory(string category);
    }
```

Eklenen aynı satır Data katmanı içindeki IProductRepository içinde de olması gerekir.

```
public interface IProductRepository: IRepository<Product>
{//…
    int GetCountByCategory(string category);
}
```

Daha sonra bunların Concrete versiyonlarının içinin doldurulması gerekir. Önce EfCoreProductRepository içini dolduralım. Bunun için daha önce çok kez yaptığımız şekilde bu dosya içindeki en üstteki hata gösteren ifadenin üzerine gelip ara birimi uygula diyelim (implement Interface). Bu işlemden sonra ilgili metod bu sayfa içinde oluşacaktır.

Bu metod içinde veritabanına bir sorgu göndermemiz gerekiyor. Daha önce kategoriye göre ürünleri getiren bir sorgu yazmıştık. Onu kopyalayıp üzerinde değişiklik yapalım.

```
public class EfCoreProductRepository ://..
{
public int GetCountByCategory(string category)
{
    using (var context = new TSContext())
    {
        var products = context.Products.AsQueryable();

        if (!string.IsNullOrEmpty(category))
        {
            products = products
                .Include(i => i.ProductCategories)
                .ThenInclude(i => i.Category)
                .Where(i => i.ProductCategories.Any(a => a.Category.Url == category));
        }

        return products.Count();
    }
}
}
```

Servis katmanındaki concrete versiyonun içini de dolduralım. Buradaki sınıfımız ProductManager.cs dir. Bunun içerisine yine Abstract versiyondaki metodu getirmesi için daha önce yaptığımız şekilde “implement Interface” yapalım.

```
public class ProductManager : IProductService
{
//..

public int GetCountByCategory(string category)
{
    return _productRepository.GetCountByCategory(category);
}
}
```

Bundan sonra artık TSController içine PageInfo vasıtasıyla için ihtiyaç olan bilgilerimiz gelecektir. TSController içindeki List() metodunun son hali aşağıdaki şekildedir.

```
public IActionResult List(string category, int page=1)
{
    const int pageSize = 3;

    var productViewModel = new ProductListViewModel()
    {
        PageInfo = new PageInfo()
        {
```

```
            TotalItems = _productService.GetCountByCategory(category),
            CurrentPage = page,
            ItemsPerPage = pageSize,
            CurrentCategory = category
        },

        Products = _productService.GetProductsByCategory(category, page, pageSize)
    };
    return View(productViewModel);
}
```

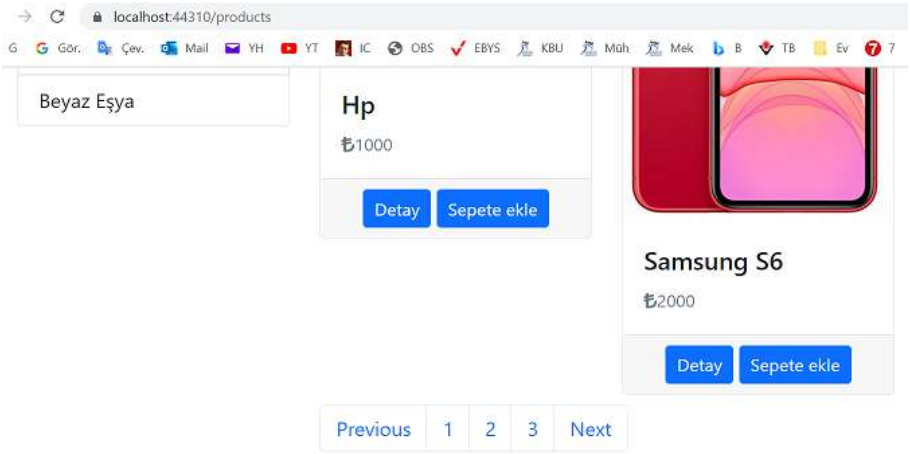
Artık listeleme yaptığımız sayfa içine model bilgisi içinde bu bilgiler taşınacaktır. Artık View>TS>List.cshtml dosyası içine bilgiler taşındığına göre burada sayfalama arayüzü için Boostrap tasarımlarını kullanalım.

Boostrap sitesine gidip Pagination yazarsak sayfalama butonları için örnekler gelecektir. Bu örneklerden bir tanesini alıp Listeleme sayfasında ürünlerin gösterildiği 9 sütunluk bölümün en altına kodları ekleyelim.

View>TS>**List.cshtml**

```
@model ProductListViewModel

<div class="row">
    <div class="col-md-3">
        @await Component.InvokeAsync("Categories")
    </div>
    <div class="col-md-9">
        <div class="row">
            @foreach (var product in Model.Products)
            {
            <div class="col-md-4">
                @await Html.PartialAsync("_product", product)

            </div>
            }
        </div>

        <div class="row">
            <div class="col">
                <nav aria-label="Page navigation example">
                    <ul class="pagination">
                        <li class="page-item"><a class="page-link" href="#">Previous</a></li>
                        <li class="page-item"><a class="page-link" href="#">1</a></li>
                        <li class="page-item"><a class="page-link" href="#">2</a></li>
                        <li class="page-item"><a class="page-link" href="#">3</a></li>
                        <li class="page-item"><a class="page-link" href="#">Next</a></li>
                    </ul>
                </nav>
            </div>
        </div>
    </div>
</div>

@section Scripts
{
    <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js" integrity="sha384-
Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9IOYy5n3zV9zzTtmI3UksdQRVvoxMfooAo" crossorigin="anonymous"></script>
    <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/js/bootstrap.min.js" integrity="sha384-
wfSDF2E50Y2D1uUdj0O3uMBJnjuUD4Ih7YwaYd1iqfktj0Uod8GCExl3Og8ifwB6" crossorigin="anonymous"></script>
}
```

Programı çalıştırınca görünüm aşağıdaki şekilde olur.



Şu anda bu görüntüleme dinamik değildir. Bir sonraki derste sayfalama bilgilerini dinamik hale getirelim.

Sayfalama Butonlarının Linklerinin Hazırlanması

Sayfalama butonlarımız html formatında hazırdır. Bunların css görünümleri Boostraptan gelmektedir. Bu butonlardan kaç tane olmasını istiyorsak butonu oluşturan <li> etiketlerini for döngüsü ile dinamik olarak oluşturacağız. Olması gereken sayfa sayısını yukarıda TotalPages metodu ile hesaplamıştık. Bu değer döngünün sınır değerini oluşturacaktır. html kodları içinde C# kodları ve değişkenleri kullanırken başına @ işaretini koyduğumuza dikkat edin. İlgili kodlar şu hale gelecektir.

```
<div class="row">
    <div class="col">
        <nav aria-label="Page navigation example">
            <ul class="pagination">
                @for (int i = 1; i <= Model.PageInfo.TotalPages(); i++)
                {
                    <li class="page-item"><a class="page-link"
                    href="/products?page=@i">@i</a></li>
                }
            </ul>
        </nav>
    </div>
</div>
```

Bu uygulamayı çalıştırdığımızda tüm ürünleri gösterirken sayfalama linklerimiz çalışmaktadır. Yani şu şekilde olan linkler çalışıyor (/products?page=2). Fakat herhangi bir kategoriye tıkladığımızda linklerimiz şu hale dönüşmektedir (/products/telefon?page=1). Bu durumda tekrar sayfalama linklerine tıklayınca önceki tüm ürünler linkine geçmektedir. Buda hata demektir. Bu nedenle kategoriye tıklandığı bilgisinin de kontrol edilmesi gerekir.

Model bilgisi içerisinden CurrentCategory bilgisini alıyorduk. Eğer bu bilginin içi boş ise tüm kategoriler seçili demektir. Yok herhangi bir kategori ismi seçili ise o zaman hangi kategorinin seçili olduğunu alabiliyoruz ve bu bilgiyi link deseni içine ekleyebiliriz.

Şimdik linklerimiz artık kategori bazında da çalıştırabiliyoruz. Fakat hangi sayfada olduğumuzun bilgiside önemlidir. Aktif sayfanın linklerde gözükmesi gerekir. Bu bilgiyi CurrentPage ifadesi ile model içinde taşıyorduk. Döngülerimizde döngü değişkeni bu değere eşit olduğunda sayfa numarasını farklı renkte gösterebiliriz. Farklı

renkte gösterim yapmak içinde içindeki class 'a "active" sınıfını ekleyeceğiz. Kodlarımızın son hali şu şekilde olacaktır. Kodlarımızın son hali şu şekilde olacaktır.

```
<div class="row">
    <div class="col">
        <nav aria-label="Page navigation example">
            <ul class="pagination">
                @if (string.IsNullOrEmpty(Model.PageInfo.CurrentCategory))
                {
                    @for (int i = 1; i <= Model.PageInfo.TotalPages(); i++)
                    {
                        <li class="page-item @(Model.PageInfo.CurrentPage==i?"active":"")">
                            <a class="page-link" href="/products?page=@i">
                                @i
                            </a>
                        </li>
                    }
                }
                else
                {
                    @for (int i = 1; i <= Model.PageInfo.TotalPages(); i++)
                    {
                        <li class="page-item @(Model.PageInfo.CurrentPage==i?"active":"")">
                            <a class="page-link"
href="/products/@Model.PageInfo.CurrentCategory?page=@i">
                                @i
                            </a>
                        </li>
                    }
                }
            </ul>
        </nav>
    </div>
</div>
```

Belirlenen Ürünlerin Ana Sayfada Listelenmesi

Şu anda anasayfada ürünlerin hepsi gelmektedir. Fakat normalde sitenin ana sayfasında görüntülenecek ürünler seçilmiş ürünler olması gerekir. Bu nedenle bu tür ürünler belirlemek için model yapımızda ve dolayısı ile Veritabanında belli bir alan oluşturmalıyız.

Bunun Product için model yapısı içinde "IsHome" isminde bir alan oluşturalım. Bu alan bilgisi true olan ürünler ana sayfada görüntülensin. Benzer şekilde aynı yerde daha önce IsApproved alanı oluşturulmuştu. Onaylı olan ürünler kullanıcı sayfalarında görüntülenecek, onaylı olmayanlar ise yönetici sayfalarında görüntülenecektir.

Product.cs model yapımız Entity projesi içinde oluşturuluyordu. Oraya gidip bu alanı ekleyelim.

TSEntity>**Product.cs**

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

namespace TS.Entity
{
    public class Product
    {
```

```
        public int ProductId { get; set; }
        public string Name { get; set; }
        public string Url { get; set; }
        public double? Price { get; set; }
        public string Description { get; set; }
        public string ImageUrl { get; set; }
        public bool IsApproved { get; set; }
        public bool IsHome { get; set; }

        public List<ProductCategory> ProductCategories { get; set; }
    }
}
```

Bu seçili ürünleri ana sayfada göstermek için GetHomePageProducts() isminde bir metod oluşturalım. Bu metodun Abstract ve Concrete versiyonlarını hem Data katmanında hemde Servis katmanında oluşturalım.

TSData>Abstract>IProductRepository.cs içine aşağıdaki kodu ekleyelim.

```
public interface IProductRepository: IRepository<Product>
{//..
       List<Product> GetHomePageProducts();
}
```

Bunun Concrete versiyonunuda aşağıdaki şekilde TSData>Concrete>EfCoreProductRepository.cs içinde oluşturalım. Burada ürünler listeye eklenirken araya bir sorgu ekliyoruz ve bu sorguda onaylı ve anasayfa gösterilecek ürün bilgisi true olanlar getiriliyor.

```
public List<Product> GetHomePageProducts()
{
    using (var context = new TSContext())
    {
        return context.Products
            .Where(i => i.IsApproved && i.IsHome).ToList();

    }
}
```

Şimdi bu metodla ilgili düzenlemeleri servis (Business) katmanına da yapalım. Abstract>IProductService içine aşağıdaki kodu ekleyelim.

```
List<Product> GetHomePageProducts();
```

Concrete versiyonunu için ise ProductManager.cs içine aşağıdaki kodları ekleyelim.

```
public List<Product> GetHomePageProducts()
{
    return _productRepository.GetHomePageProducts();
}
```

Not: Bir katmanda oluşturulan değişikliğin diğer katmanlarda gözükebilemesi için Build edilmesi gerektiğini unutmayın.

Alt yapımız hazır artık WebUI arayüz katmanı içindeki HomeController içinde ilk açılışta kullanılan Index() metodu içinde önceden yaptığımız GetAll() işlemi yerine GetHomePageProducts() işlemini ekleyelim.

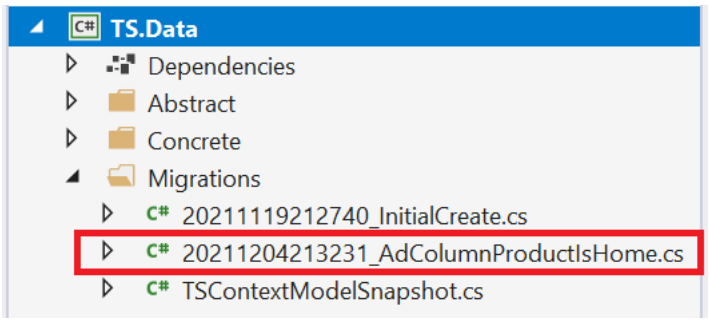
```
public IActionResult Index()
{
    var productViewModel = new ProductListViewModel()
    {
        Products = _productService.GetHomePageProducts()
    };

    return View(productViewModel);
}
```

Şu anda veritabanı yapımızda bir değişiklik yaptık. Bu değişikliğin veritabanına yansıması için ek bir migration oluşturup onun vasıtasıyla Veritabanını değiştirelim. Aşağıdaki komutu Komut istemi ekranına gidip oradan TSData klasörüne kadar gidip orada çalıştıralım.

dotnet ef migrations add AdColumnProductIsHome –-startup-project ../TS.WebUI

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

C:\Users\pc1\Desktop\TS.com\TS\TS.Data>
```

Migration oluştu.



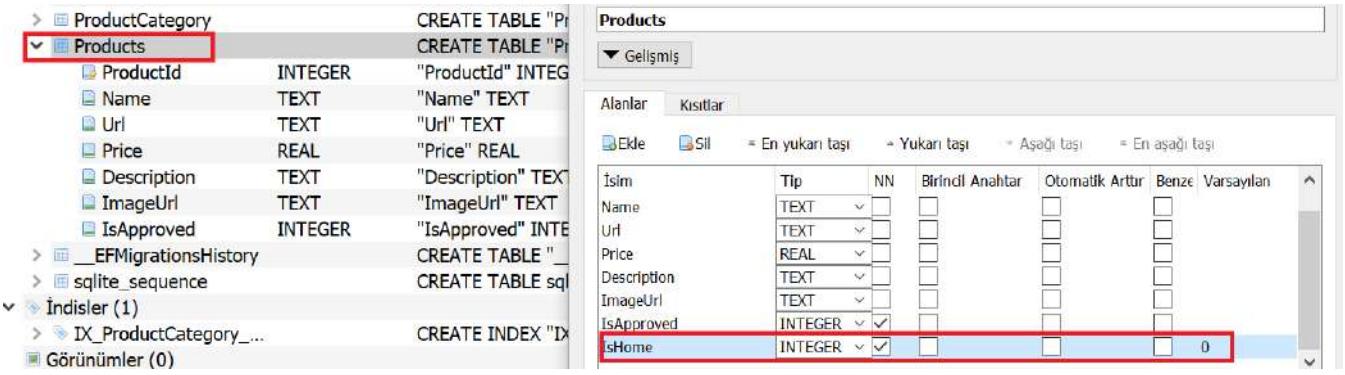
Bu migrationı aşağıdaki komutla çalıştırıp Veritabanına değişikliği yansıtalım.

dotnet ef database update --startup-project ../TS.WebUI

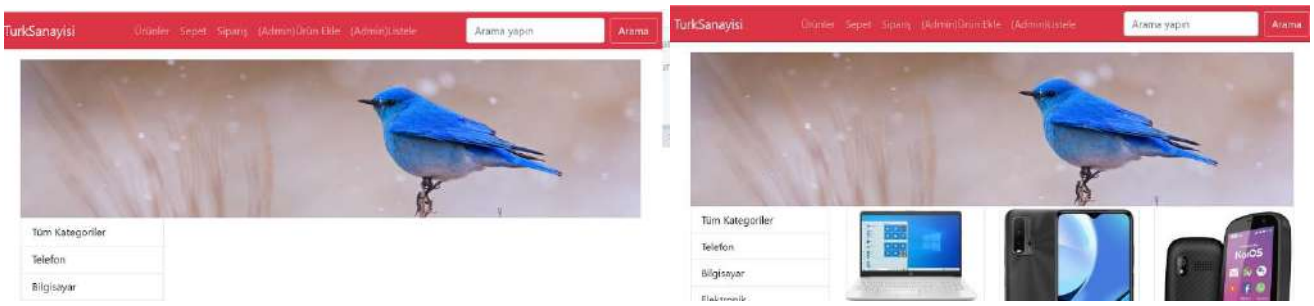
Migration aktarıldı.

```
C:\Users\pc1\Desktop\TS.com\TS\TS.Data>dotnet ef database update --startup-project ../TS.WebUI
Build started...
Build succeeded.
No migrations were applied. The database is already up to date.
Done.
```

Not bu komut çalışmadı. Database elle düzeltilerek yapıldı. Tablo düzenleme modunda açıldı. IsHome aşağıda gösterildiği şekilde eklendi.



Bazı ürünler anasayfada gözükmesi için true yapıldı. İlk açılışta anasayfa boş geldi. Değerler değiştirildikten sonra ürünler ana sayfada gözükmeye başladı.



Tabii burada görüntülenen ürünler hem onaylı olanlar (IsApproved) hemde Ana sayfa seçeneği true olan (IsHome) ürünlerdir. Bu işlemleri yukarıda geçen `Where(i => i.IsApproved && i.IsHome).ToList();` Satırı ile yapmıştık. Oysa buraya kadar olan kısımda ürünleri listelerken onaylı olup olmamasına bakmamıştık. Normal listelemede de onaylı ürünleri göstermemiz gerekmektedir.

Bunun için EfCoreProductRepository.cs içine aşağıdaki satırı eklemiş olalım. Niçin buraya ekledik. AsQueryable ifadesi üzerine ekradan sorgu ekleneceğini gösterir. Ondan önce üzerine sürekli geçerli olacak bir sorguyu eklemiş olduk. Daha sonra bunun üzerine eksradan daha sorgu ekleyeceğimizi söylemiş olduk. Zaten ilgili satırlardan sonra ToList() metodu çalıştırılmadan bir çok sorgu daha eklemiştik. Category adı var mı yokmu, sayfalamalara yönelik sorgular gibi.

İlgili motodun tamamı şu şekilde olur.

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

```
public List<Product> GetProductsByCategory(string name, int page, int pageSize)
{
    using (var context = new TSContext())
    {
        var products = context.Products
            .Where(i=>i.IsApproved)
            .AsQueryable();

        if(!string.IsNullOrEmpty(name))
        {
            products = products
                .Include(i => i.ProductCategories)
                .ThenInclude(i => i.Category)
                .Where(i => i.ProductCategories.Any(a => a.Category.Url == name));
        }

        return products.Skip((page-1)*pageSize).Take(pageSize).ToList();
    }
}
```

Burada toplam ürün sayısını tespit ederken de IsAproved kullanılması gerekir. Yoksa gelen ürün az, sayfalama sayısı fazla olacaktır. Bunun için aşağıdaki kodları ekledik.

```
public int GetCountByCategory(string category)
{
  using (var context = new TSContext())
  {
      var products = context.Products
          .Where(i=>i.IsApproved)
          .AsQueryable();
```

Kelime Bazlı Ürün Arama

Öncelikle NavBar içindeki arama kutusu ve butonu Kategorilerin altına alalım. NavBardaki ilgili yere daha çok kullanıcı ile bilgileri getirilim. İlk olarak Navbar içindeki form kısmını oradan alıp, yeni bir partial dosya (_Search.cshtml) olupturup onun içine atalım. Formun sol tarafta daha güzel durması için bir kart içine alalım. Kart görünümünün bir header ve body kısmı olsun. Butonla textbox alt alta olması için class=”form-group” ekleyelim. Görünümü düzeltmek için boostrap sınıf atamalarının içlerini ayarlayalım. Dosyanın son hali şu şekilde olacaktır.

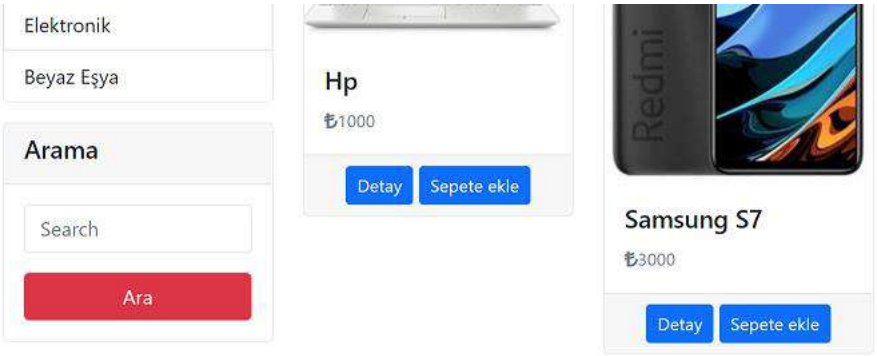
_Search.cshtml

```
<div class="card mt-3">
    <div class="card-header">
        <h5>Arama</h5>
    </div>
    <div class="card-body">
        <form action="/search">
            <div class="form-group">
                <input name="q" type="text" class="form-control" placeholder="Search">
            </div>
            <button type="submit" class="form-control btn-danger mt-3">Ara</button>
        </form>
    </div>
</div>
```

Bu kodlarımızın ana sayfada kategorilerin altında gözükmesini istiyoruz. Oraya sayfa bağlantı linkini yerleştirmeliyiz. Bunun için kategori görünümünün altına partial linkini ekleyelim. Ana sayfa Views>Home>Index.cshtml sayfasıdır.

```
@model ProductListViewModel

    <partial name="_Header">


        <div class="row">
            <div class="col-md-3">
                @await Component.InvokeAsync("Categories")
                <partial name="_Search">
            </div>
```

Görünüm şu şekil aldı.



Arama işlemi için önce arama sonucuna göre sonuçları getirecek sınıf yapılarımızı oluşturalım. Data>Abstract>IProductRepository içine ilgili metodumuzu ekleyelim. GetPopularProducs() metodlarını kullanmıyorduk. Bunları kaldıralım.

```
public interface IProductRepository: IRepository<Product>
    {
        List<Product> GetSearchResult(string searchString);
```

EfCoreProductRepository içinde içi dolu (concrete) kodlarımızı oluşturalım. Ara birimi uygula diyerek oluşturalım. Metod içinde bir arama işlemi yapacağız. Daha önceki yazılan kodlardan GetProductsByCategory den uygun

kodları alıp üzerinde düzenleme yapalım. Category araması olmasın. IsApproved sorgusu bulunsun. Şimdilik sayfalama sorgularını da kaldıralım. IsAproved sorgusu yanına && operatörü ile ekleme yaparak ya Name kısmında yada Description kısmında ilgili string varsa o ürünleri getirsin. Metod içeriği şu şekilde olacaktır.

```
public class EfCoreProductRepository :
        EfCoreGenericRepository<Product, TSContext>, IProductRepository

       public List<Product> GetSearchResult(string searchString)
       {
           using (var context = new TSContext())
           {
               var products = context.Products
                     Where(i => i.IsApproved &&
                     (i.Name.ToLower().Contains(searchString.ToLower()) ||
                     i.Description.ToLower().Contains(searchString.ToLower())))
                   .AsQueryable();


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

Service (business) katmanındaki düzenlemeleri de yapalım.

```
public interface IProductService
{

       List<Product> GetSearchResult(string searchString);
```

Bunun concrete versiyon ProductManager içinde oluşturuluyordu.

```
public class ProductManager : IProductService
{

       public List<Product> GetSearchResult(string searchString)
       {
           // iş kuralları buraya yazılır
           return _productRepository.GetSearchResult(searchString);
       }
```

Burada arama kelimesi bir çok kelimeden oluşabilir. Bu durumda aranan kelimeler dizi içine alınıp ona göre arama yaptırılabilir. Şu aşamada bu detaylandırmayı yapmayalım. Şu durumda kullanıcı birden fazla kelime yazarsa bu kelimeler olduğu gibi aranacaktır.

Artık TSController sayfasındaki düzenlemeye geçelim. Buraya ekstra bir metod ekleyeceğiz. Bu motoda bilgiler formdan geleceği için (içinde textbox ve buton var) hangi metodla geldiğini de bilmeliyiz. Varsayılan GET metodu olduğu için (bilgiler açıktan görülecek şekilde gelir) metod üzerine GET metoduna dair bilgiyi yazmamız gerekmez. Serch() metodumuzun içeriği aşağıdaki şekilde olabilir. Sayfalama kullanmadık.

```
public class TSController : Controller
{

       public IActionResult Search(string q)
       {
           var productViewModel = new ProductListViewModel()
           {
               Products = _productService.GetSearchResult(q)
           };
           return View(productViewModel);
       }
```

Bu bilgiler View>TS içinde Search.cshtml sayfasına taşınmaya çalışılacaktır. Bu sayfayı oluşturalım. (Dikkat önceki oluşturduğumuz _Search.cshtml sayfası partial sayfaydı. Bu ondan farklı tek başına bir sayfa olacak. Bu nedenle partial sayfaların ilk harflerini küçük yaparsak iyi olacaktır.)

Search.cshmtl sayfasının içeriği List sayfası ile aynı olduğu için buradaki içeriği alıp kullanalım. Sadece alt kısımda sayfalama olmayacak. Bu kısmı çıkaralım. Sayfamız şu şekilde olacaktır.

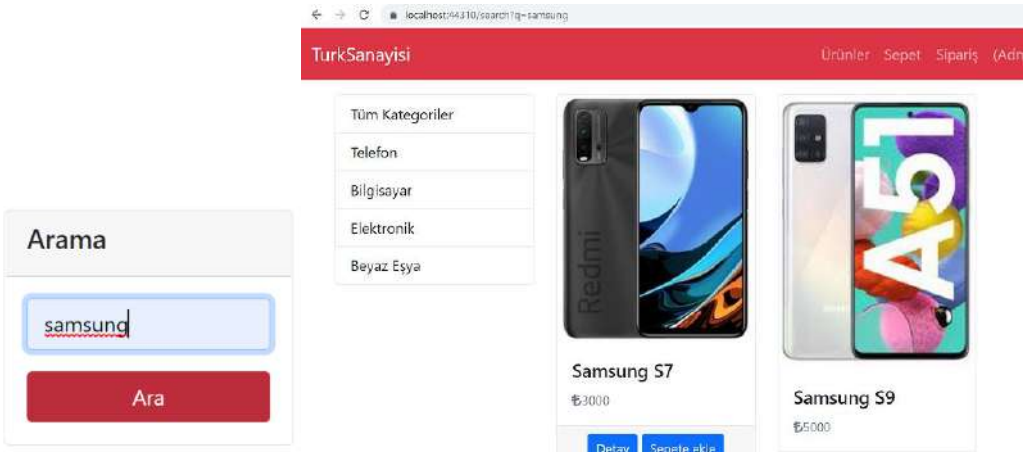
Views>TS>**Search.cshtml**

```
@model ProductListViewModel


<div class="row">
    <div class="col-md-3">
        @await Component.InvokeAsync("Categories")
    </div>
    <div class="col-md-9">
        <div class="row">
            @foreach (var product in Model.Products)
            {
                <div class="col-md-4">
                    @await Html.PartialAsync("_product", product)
                </div>
            }
        </div>

    </div>
</div>

@section Scripts
{
    <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js" integrity="sha384-
Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9IOYy5n3zV9zzTtmI3UksdQRVvoxMfooAo" crossorigin="anonymous"></script>
    <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/js/bootstrap.min.js" integrity="sha384-
wfSDF2E50Y2D1uUdj0O3uMBJnjuUD4Ih7YwaYd1iqfktj0Uod8GCExl3Og8ifwB6" crossorigin="anonymous"></script>
}
```

Şimdi linklerimiz içinde Route kısmında search kelimesi geçtiğinde hangi kontrolıra gideceğine dair yönlendirmeyi yapmak için Startup.cs dosyamızın içine link desenimizi oluşturalım.

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "search",
        pattern: "search",
        defaults: new { controller = "TS", action = "Search" }
    );
});
```

Kodları denersek, samsung yazdığımızda bu isimdeki onaylı ürünler gelecektir.



Burada arama kısmının altına ürünlerin çeşitli özelliklerini getirip oradan seçilen özelliklere göre arama detaylandırılabilir.

Artık kullanıcı sayfalarımız hazırdır. Bir sonraki bölümde yönetici sayfalarını hazırlayalım.

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