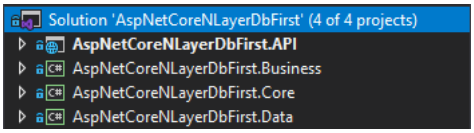
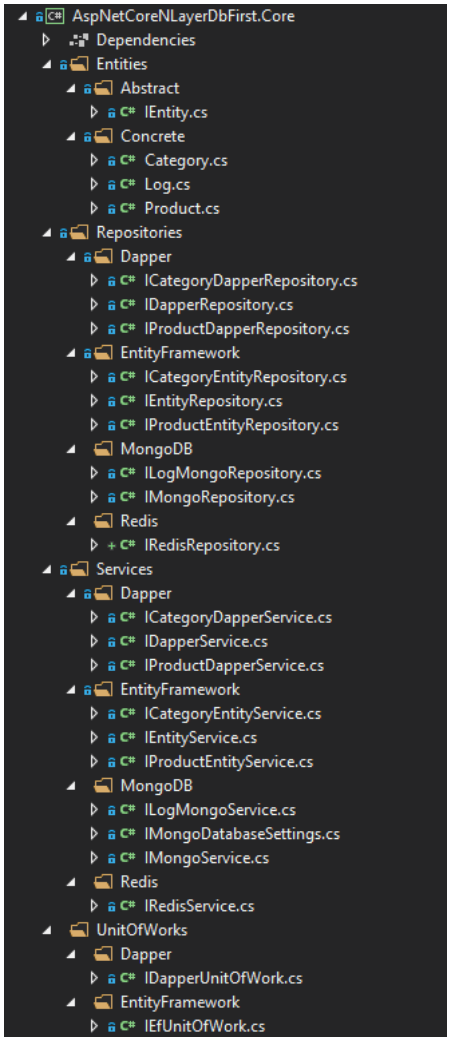
Katmanlı Mimari

Temel olarak katmanlı mimari 5 temel yapıdan oluşur. Bunlar Core, Data, Business, API, Web katmanlarıdır. Business katmanı bazı yerlerde Service olarak da geçer. Proje DbFirst olarak geliştirilmiştir. Dapper, EntityFramework, Redis ve MongoDB bir arada kullanılmıştır. Örnek projede iki adet entity (Product ve Category) bulunmaktadır.



Core katmanı, Repository ve Service arayüzlerinin (interface) ve UnitOfWork arayüzünün tutulduğu yerdir.



UnitofWork Design Pattern Nedir?

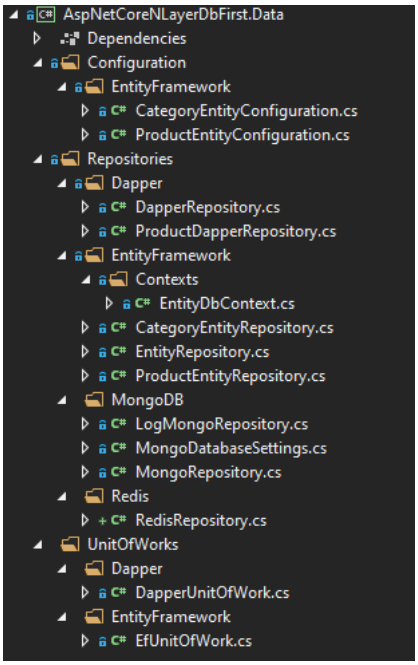
Bu kalıp (pattern), iş katmanında yapılan her değişikliğin anlık olarak veri tabanına yansıması yerine, işlemlerin toplu halde tek bir kanaldan gerçekleşmesini sağlar.

UnitofWork'ün avantajı nedir?

Bu ve bunun gibi, birden fazla işlemi tek bir işlem olarak düşünmemiz gereken yerlerde bu tasarım kalıbı (design pattern) kullanabiliriz.

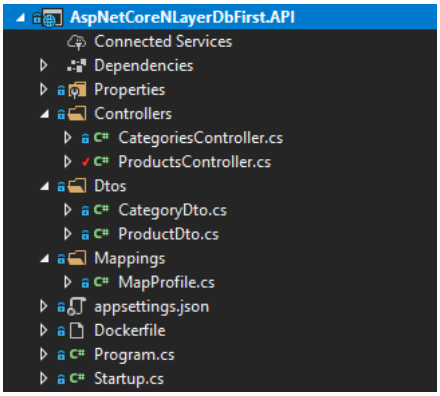
İşlemler tek bir kanaldan(tek bir transaction) toplu halde yapıldığı için performansı artı yönde etkileyecektir. Ayrıca işlemleri geri alma (rollback), hangi tabloda ne işlem yapıldı kaç kayıt eklendi gibi sorulara da cevap verebilir.

Data katmanında Entity Framework ile oluşturulacak Entity'lere ait Configuration sınıfları, Core katmanındaki Repository, Service ve UnitOfWork arayüzlerinden miras (inheritance) alan sınıflar bulunmaktadır. Ayrıca veri tabanına ait sınıflar da projenin bu katmanda yer alır.



Business katmanı kullanılacak teknolojilerin (bu çalışmada Dapper, Entity Framework, MongoDB, Redis) API ile haberleştiği kısımdır. Services klasöründe bulunan sınıflar, Core katmanındaki arayüzlerden miras alarak generic (jenerik) bir yapıda oluşturulmuştur.

API katmanında Controller isimleri çoğul olarak yazılır (Products gibi).



Controller doğrudan Entity'leri kullanmak yerine DTO (Data Transfer Object) denilen ara elemanlarla haberleşir. DTO kullanılmasının sebebi Core katmanındaki Entity'lere doğrudan müdahale etmektense istenilen Entity yapısı DTO ile oluşturulmak istenmesidir. Örneğin;

```
using System.ComponentModel.DataAnnotations;

namespace AspNetCoreNLayerDbFirst.API.Dtos
{
    public class CategoryDto
    {
        public int Id { get; set; }
        [Required]
        public string Name { get; set; }
    }
}
```

Bu özel DTO'ya ek olarak o kategorinin içindeki ürünler de istenmektedir. Bu sebeple DTO yapısına gerek duyulmuştur.

```
using System.Collections.Generic;

namespace AspNetCoreNLayerDbFirst.API.Dtos
{
    public class CategoryWithProductDto : CategoryDto
    {
        public IEnumerable<ProductDto> Products { get; set; }
    }
}
```

Üstteki yapıda CategoryDto'dan inherit eden özelleştirilmiş bir DTO vardır.

Extensions klasörü API'den dönecek hataların özelleştirilmesi için kullanılır. Varsayılan hatalar yerine burada doğrudan istenilen hata döndürülebilir. Örnek bir hata yakalama kontrolü şöyledir:

```
using AspNetCoreNLayerDbFirst.API.Dtos;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Diagnostics;
using Microsoft.AspNetCore.Http;
using Newtonsoft.Json;

namespace AspNetCoreNLayerDbFirst.API.Extensions
{
    public static class UseCustomExceptionHandler
    {
        public static void UseCustomException(this IApplicationBuilder app)
        {
            app.UseExceptionHandler(config =>
            {
                config.Run(async context =>
                {
                    context.Response.StatusCode = 500;
                    context.Response.ContentType = "application/json";
                    var error = context.Features.Get<IExceptionHandlerFeature>();

                    if (error != null)
                    {
                        var ex = error.Error;

                        ErrorDto errorDTO = new ErrorDto();
                        errorDTO.Status = 500;
                        errorDTO.Errors.Add(ex.Message);

                        await context.Response.WriteAsync(JsonConvert.SerializeObject(errorDTO));
                    }
                });
            });
        }
    }
}
```

Bu hata aslında bir Middleware (ara katman) olarak yer alır. Bu sebeple ilgili projenin Startup.cs dosyasına eklenmesi gerekir.

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env, RedisService redis)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    redis.Connect();
    app.UseCustomException();
    app.UseHttpsRedirection();
```

Filters klasörü, API'den istekte bulunulduğunda API'nin uygulayacağı filtrelemedir. Controller'da Attribute olarak kullanılır.

```
using AspNetCoreNLayerDbFirst.API.Dtos;
using AspNetCoreNLayerDbFirst.Core.Services.EntityFramework;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using System.Linq;
using System.Threading.Tasks;
```

```
namespace AspNetCoreNLayerDbFirst.API.Filters
{
    public class NotFoundFilter : ActionFilterAttribute
    {
        private readonly ICategoryEntityService _categoryEntityService;

        public NotFoundFilter(ICategoryEntityService categoryEntityService)
        {
            _categoryEntityService = categoryEntityService;
        }

        public async override Task OnActionExecutionAsync(ActionExecutingContext context, ActionExecutionDelegate next)
        {
            int id = (int)context.ActionArguments.Values.FirstOrDefault();
            var product = await _categoryEntityService.GetByIdAsync(id);

            if (product != null)
            {
                await next();
            }
            else
            {
                ErrorDto errorDTO = new ErrorDto();
                errorDTO.Status = 404;
                errorDTO.Errors.Add($"id'si {id} olan ürün bulunamadı.");
                context.Result = new NotFoundObjectResult(errorDTO);
            }
        }
    }
}
```

```
[HttpGet("{id}")]
[ServiceFilter(typeof(NotFoundFilter))]
public async Task<ActionResult<Category>> GetCategory(int id)
{
    var category = await _context.Categories.FindAsync(id);

    if (category == null)
    {
        return NotFound();
    }

    return category;
}
```

Mapping klasörü Automapper aracılığıyla entity'leri dto'larla eşleştirir. Eşleştirme olmazsa hangi DTO'nun hangi Entity'e dönüştürüleceği bilinemez.

```
using AspNetCoreNLayerDbFirst.API.Dtos;
using AspNetCoreNLayerDbFirst.Core.Entities.Concrete;
using AutoMapper;

namespace AspNetCoreNLayerDbFirst.API.Mappings
{
    public class MapProfile : Profile
    {
        public MapProfile()
        {
            CreateMap<Category, CategoryDto>().ReverseMap();
            CreateMap<CategoryDto, Category>().ReverseMap();

            CreateMap<CategoryWithProductDto, Category>().ReverseMap();
            CreateMap<Category, CategoryWithProductDto>().ReverseMap();

            CreateMap<ProductDto, Product>().ReverseMap();
            CreateMap<Product, ProductDto>().ReverseMap();
        }
    }
}
```

appsettings.json içinde genel olarak veri tabanı ile ilgili değerler bulunur. Tabi sadece bununla kalmaz ihtiyaca göre yeni değerler de eklenebilir (bknz. login bilgileri, docker bilgileri, redis).

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
```

```
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "MSSQL": "Data Source=(localdb)\\MSSQLLocalDB;Initial Catalog=NLayerDbFirstDB;Integrated Security=True;Connect Timeout=30;E
  },
  "Redis": {
    "Host": "localhost",
    "Port": 6379
  },
  "MongoDB": {
    "CollectionName": "Logs",
    "ConnectionString": "mongodb://localhost:27017",
    "DatabaseName": "NLayerDB"
  }
}
```

CategoriesController içerisinde Redis kullanılmıştır. NoSQL veritabanı sunan Redis genellikle Cache'leme işlemleri için tercih edilmektedir. Select, Insert, Update, Delete fonksiyonlarına sahiptir. Bu Controller'da istenen cache'ler Redis'teki liste yapısında tutulmuştur. Ancak Redis'te harici olarak birçok veri tutma yöntemi de vardır (string, hashset vb.)

```
using AspNetCoreNLayerDbFirst.API.Filters;
using AspNetCoreNLayerDbFirst.Business.Services.Redis;
using AspNetCoreNLayerDbFirst.Core.Entities.Concrete;
using AspNetCoreNLayerDbFirst.Core.Services.EntityFramework;
using AspNetCoreNLayerDbFirst.Data.Repositories.EntityFramework.Contexts;
using AutoMapper;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using Newtonsoft.Json;
using StackExchange.Redis;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace AspNetCoreNLayerDbFirst.API.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class CategoriesController : ControllerBase
    {
        private readonly ICategoryEntityService _categoryService;
        private readonly IMapper _mapper;
        private readonly EntityDbContext _context;

        private readonly RedisService _redisService;
        private readonly IDatabase _db;
        private readonly string _redisKey = "categories";


        public CategoriesController(EntityDbContext context, ICategoryEntityService categoryService, IMapper mapper, RedisServi
        {
            _context = context;
            _categoryService = categoryService;
            _mapper = mapper;
            _redisService = redisService;
            _db = _redisService.GetDb(0);
        }

        // GET: api/Categories
        [HttpGet]
        public async Task<ActionResult<IEnumerable<Category>>> GetCategories()
        {
            return await _context.Categories.ToListAsync();
        }

        // GET: api/Categories/5
        [HttpGet("{id}")]
        [ServiceFilter(typeof(NotFoundFilter))]
        public async Task<ActionResult<Category>> GetCategory(int id)
        {
            var category = await _context.Categories.FindAsync(id);

            if (category == null)
            {
                return NotFound();
            }

            return category;
        }
```

```
        // PUT: api/Categories/5
        [HttpPut("{id}")]
        public async Task<IActionResult> PutCategory(int id, Category category)
        {
            if (id != category.Id)
            {
                return BadRequest();
            }

            var currentCategory = await _context.Categories.FindAsync(id);
            string jsonOld = JsonConvert.SerializeObject(currentCategory);
            await _db.ListRemoveAsync(_redisKey, jsonOld);

            try
            {
                string json = JsonConvert.SerializeObject(category);
                await _db.ListRightPushAsync(_redisKey, json);

                currentCategory.Id = category.Id;
                currentCategory.Name = category.Name;
                currentCategory.IsDeleted = category.IsDeleted;

                _context.Update<Category>(currentCategory);

                await _context.SaveChangesAsync();
            }
            catch (DbUpdateConcurrencyException)
            {
                if (!CategoryExists(id))
                {
                    return NotFound();
                }
                else
                {
                    throw;
                }
            }
            return NoContent();
        }

        // POST: api/Categories
        [HttpPost]
        public async Task<ActionResult<Category>> PostCategory(Category category)
        {

            _context.Categories.Add(category);
            await _context.SaveChangesAsync();

            string json = JsonConvert.SerializeObject(category);
            await _db.ListRightPushAsync(_redisKey, json);

            return CreatedAtAction("GetCategory", new { id = category.Id }, category);
        }

        // DELETE: api/Categories/5
        [HttpDelete("{id}")]
        public async Task<ActionResult<Category>> DeleteCategory(int id)
        {
            var category = await _context.Categories.FindAsync(id);
            if (category == null)
            {
                return NotFound();
            }

            _context.Categories.Remove(category);
            await _context.SaveChangesAsync();

            string json = JsonConvert.SerializeObject(category);
            await _db.ListRemoveAsync(_redisKey, json);

            return category;
        }

        private bool CategoryExists(int id)
        {
            return _context.Categories.Any(e => e.Id == id);
        }
    }
}
```

Startup.cs sınıfı oldukça önemlidir çünkü uygulama ayağa kalkarken burdaki ayarlamalara göre kalkar. Kullanılacak servisler, servislerin neyle eşleştirileceği, veri tabanı bağlantı ayarlamaları, filtreleme ayarlamaları vb. burada belirtilir.

Log işlemleri için MongoDB kullanılmaktadır. Core katmanında tanımlanan Log Entity'si aşağıdaki gibidir:

```
using AspNetCoreNLayerDbFirst.Core.Entities.Abstract;
using MongoDB.Bson;
using MongoDB.Bson.Serialization.Attributes;
using System;

namespace AspNetCoreNLayerDbFirst.Core.Entities.Concrete
{
    public class Log : IEntity
    {
        [BsonId]
        [BsonRepresentation(BsonType.ObjectId)]
        public string Id { get; set; }
        public object CurrentEntity { get; set; }
        public object NewEntity { get; set; }
        public DateTime LoggingDate { get; set; }
        public string IP { get; set; }
        public string Operation { get; set; }
        public string ClassName { get; set; }
    }
}
```

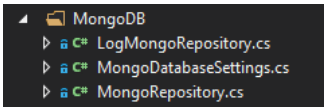
Ardından Core katmanındaki Repositories dosyasına MongoDB arayüzleri eklenir.

```
using MongoDB.Driver;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace AspNetCoreNLayerDbFirst.Core.Repositories.MongoDB
{
    public interface IMongoRepository<TEntity> where TEntity : class
    {
        Task<List<TEntity>> GetAllAsync(FilterDefinition<TEntity> filter);
        Task<TEntity> GetByIdAsync(FilterDefinition<TEntity> filter);
        Task<TEntity> AddAsync(TEntity entity);
        Task UpdateAsync(FilterDefinition<TEntity> filter, TEntity entity);
        Task DeleteAsync(FilterDefinition<TEntity> filter);
    }
}
```

ILogMongoRepository arayüzü IMongoRepository<Log> arayüzünden miras alır.

```
using AspNetCoreNLayerDbFirst.Core.Entities.Concrete;

namespace AspNetCoreNLayerDbFirst.Core.Repositories.MongoDB
{
    public interface ILogMongoRepository : IMongoRepository<Log>
    {
    }
}
```

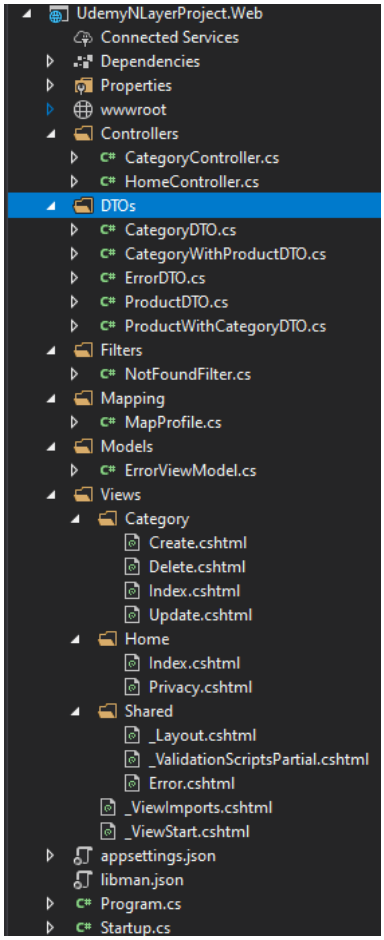
Data katmanında ise bu arayüzlerden miras alan sınıflar oluşturulur.




```
using AspNetCoreNLayerDbFirst.Core.Repositories.MongoDB;
using AspNetCoreNLayerDbFirst.Core.Services.MongoDB;
using MongoDB.Driver;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace AspNetCoreNLayerDbFirst.Data.Repositories.MongoDB
{
    public class MongoRepository<TEntity> : IMongoRepository<TEntity> where TEntity : class
    {
        private readonly IMongoCollection<TEntity> _entities;
        public MongoRepository(IMongoDatabaseSettings settings)
        {
            var client = new MongoClient(settings.ConnectionString);
            var database = client.GetDatabase(settings.DatabaseName);
            _entities = database.GetCollection<TEntity>(settings.CollectionName);
        }
        public async Task<List<TEntity>> GetAllAsync(FilterDefinition<TEntity> filter)
```

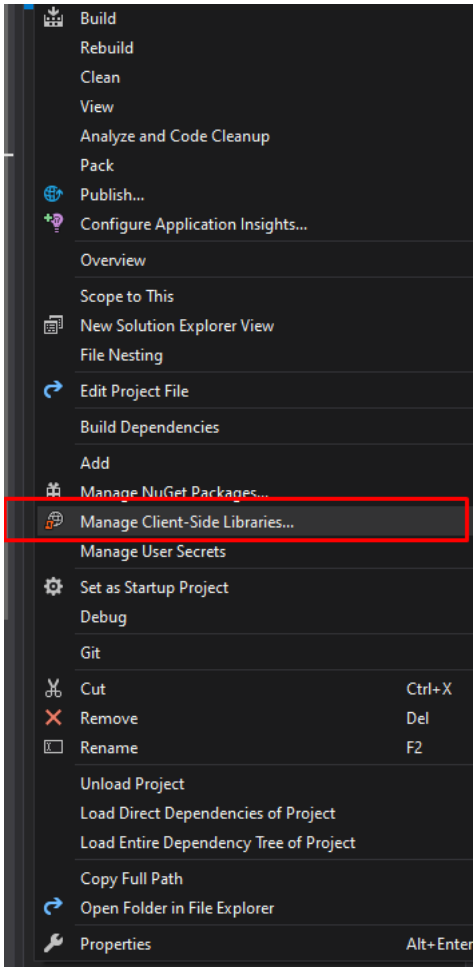
```
        {
            return await _entities.Find(filter).ToListAsync();
        }
        public async Task<TEntity> GetByIdAsync(FilterDefinition<TEntity> filter)
        {
            return await _entities.Find(filter).FirstOrDefaultAsync();
        }
        public async Task<TEntity> AddAsync(TEntity entity)
        {
            await _entities.InsertOneAsync(entity);
            return entity;
        }
        public async Task UpdateAsync(FilterDefinition<TEntity> filter, TEntity entity)
        {
            await _entities.ReplaceOneAsync(filter, entity);
        }
        public async Task DeleteAsync(FilterDefinition<TEntity> filter)
        {
            await _entities.DeleteOneAsync(filter);
        }
    }
}
```

Web katmanı ise arayüzün bulunduğu katmandır.



Bu katmanda da yine API katmanına benzer yapılar bulunur çünkü API'ye gönderilen ve gelen veriler bu katmandaki veri tipleriyle uyuşmalıdır. wwwroot klasörü içinde uygulamada kullanılacak css, js, img, font veya kütüphaneler (lib) bulunur. Kütüphaneler libman.json içerisinde belirtilir.

Projeye sağ tıklayıp aşağıdaki seçildiğinde libman.json otomatik oluşturulur.



MVC katmanından aşinâ olunan Shared klasör yapısı burada da mevcuttur. Ek olarak Startup sınıfında API katmanında olduğu gibi servis ilişkilendirmeleri, veri tabanı bağlantı ayarları uygulanır. Ayrıca Configure fonksiyonunun içinde varsayılan Middleware'lar ve URL yönlendirmeleri yapılır.