Servis Katmanı

Bir şehirde, şehrin enerji kaynaklarını etkili bir şekilde yöneten bir ekip bulunmaktaydı. Bu ekip, şehirdeki her binanın enerji ihtiyacını karşılamak ve kaynakları verimli bir şekilde kullanmakla sorumluydu. Bu ekip, şehrin enerji altyapısının kalbidir ve her binanın enerji talebini doğru bir şekilde anlamak, kaynakları optimize etmek ve enerji akışını düzenlemekle görevlidir.

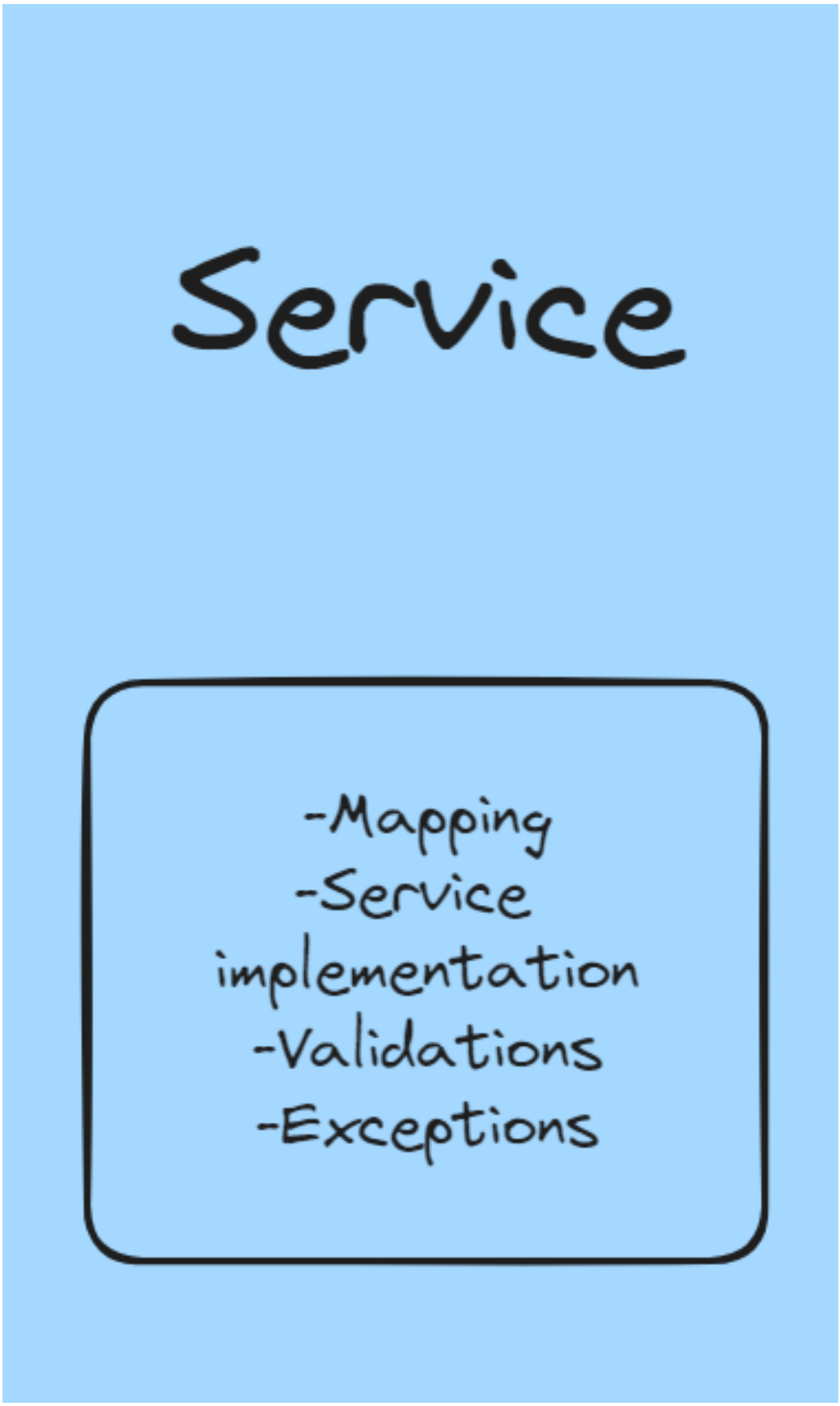
Ekip, şehrin giriş kapısındaki ölçüm cihazları aracılığıyla gelen enerji taleplerini algılar. Bu giriş katmanı, enerji taleplerini şehre giren kullanıcılardan alır ve bu talepleri şehrin enerji altyapısına ileterek İş Katmanı'na aktarır. İş Katmanı, aldığı enerji taleplerini analiz eder, enerji üretim tesislerinden gelen verilerle birleştirir ve şehirdeki enerji ihtiyacını belirler.

İş katmanının belirlediği enerji ihtiyaçları, Veri Katmanı'na iletilir. Veri Katmanı, geçmiş enerji tüketim verilerini depolar ve bu verileri kullanarak gelecekteki enerji taleplerini tahmin eder. Aynı zamanda, enerji üretim tesislerinden gelen verileri de depolar ve günceller.

Enerji talepleri ve veri analizi sonucunda ortaya çıkan enerji ihtiyacı, son katman olan Servis Katmanı'na iletilir. Servis Katmanı, bu bilgileri kullanarak enerji üretim tesislerine yönergeler gönderir, enerji akışını düzenler ve her binanın ihtiyacına uygun şekilde enerji sağlar. Ayrıca, şehirdeki enerji kullanımını optimize etmek için enerji tasarrufu stratejilerini de yönetir.

Servis Katmanı, şehirdeki enerji yönetim sisteminin kritik bir parçasıdır. Her an, enerji taleplerini izler, veri analizi yapar, enerji akışını düzenler ve şehrin enerji verimliliğini artırmak için sürekli olarak çalışır. Bu sayede, şehir sakinleri her zaman enerjiye güvenebilir ve enerji kaynakları etkili bir şekilde kullanılarak şehir daha sürdürülebilir hale gelir.

Hazır mısınız? Servis Katmanı inşa etmenin sırlarını hepbirlikte keşfedelim!


Service
-Mapping
-Service
implementation
-Validations
-Exceptions

Servis katmanı, iş kodlarının etkileşime geçtiği kritik bir noktadır. Her bir varlık (entity) için genel metotları uygulamış olsak da, gerçek dünyada her varlık için veritabanında özel sorgulara ihtiyaç duyabiliriz. Bu durumda, genellikle yeni metotlar oluşturmamız gerekebilir. Örneğin, ITeamService gibi, sadece Team Entity'sine özgü Servis sınıfları oluşturabiliriz. Bu sayede, projemizi daha geniş bir kapsamda kullanabilir ve özelleştirebilir hale geliriz.

Services

Service.cs

Bu bir public sınıf olacak ve dinamik bir **"T"** tipini alarak **IService** arayüzüne yönlendirilecek. Bu arayüz, çekirdek (Core) katmanından gelmektedir. Sınıf içerisindeki implementasyonları **Service Katmanı'nda** tanımlayacağım. Bu, iş kodlarımın bulunduğu repository ile etkileşimde bulunan bir alandır. Daha önce **Repository** bölümünde **SaveChanges** metodunu çağırmamıştık. Bunun yerine, daha merkezi bir noktadan çağrılabilecek olan **UnitOfWork** desenini uyguladık. Bu desen sayesinde farklı repolardaki eklemeleri ve güncellemeleri kontrol altında tutuyoruz; bu işlemler bellekte saklansın ve ne zaman **SaveChanges** çağrılırsa, **EF Core** bellekteki tüm değişiklikleri bir işlem bloğu şeklinde veritabanına yansıtır. Eğer herhangi bir repo işlemi sırasında bir hata oluşursa, diğer işlemleri otomatik olarak geri alacak şekilde bir otomatik geri alma (**rollback**) işlemi gerçekleşir.

```
public class Service<T> : IService<T> where T : class
  {
    private readonly IGenericRepository<T> _repository;
    private readonly IUnitOfWork _unitOfWork;

    public Service(IGenericRepository<T> repository, IUnitOfWork unitOfWork)
    {
      _repository = repository;
      _unitOfWork = unitOfWork;
    }
    public async Task<T> AddAsync(T entity)
    {
      await _repository.AddAsync(entity);
      await _unitOfWork.CommitAsync();
      return entity;
    }
```

```
public async Task<IEnumerable<T>> AddRangeAsync(IEnumerable<T> entities)
{
   await _repository.AddRangeAsync(entities);
   await _unitOfWork.CommitAsync();
   return entities;
}

public async Task<IEnumerable<T>> GetAllAsync()
{
   return await _repository.GetAll().ToListAsync();
}

public async Task<T> GetById(int id)
{
   return await _repository.GetByIdAsync(id);
}

public async Task RemoveAsync(T entity)
{
   _repository.Remove(entity);
   await _unitOfWork.CommitAsync();
}

public async Task RemoveRangeAsync(IEnumerable<T> entities)
{
   _repository.RemoveRange(entities);
   await _unitOfWork.CommitAsync();
}

public async Task UpdateAsync(T entity)
{
   _repository.Update(entity);
   await _unitOfWork.CommitAsync();
}

public IQueryable<T> Where(Expression<Func<T, bool>> expression)
{
   return _repository.Where(expression);
```

```
    }
  }
```

TeamService.cs

TeamService adlı bir sınıfı tanımlanır, bu sınıf **Service<Team>** sınıfından miras alır ve **ITeamService** arayüzünü uygular. **TeamService** sınıfının Constructor'ı, genel bir veritabanı işlemlerini yöneten **IGenericRepository<Team>** ve iş birimi deseni sağlayan **IUnitOfWork** bağımlılıklarını enjekte eder. Bu tasarım, genel CRUD işlemlerini yöneten bir servis katmanını temsil eder ve Dependency Injection, Generic Repository ve iş birimi deseni gibi yazılım tasarım desenlerini içerir.

```
public class TeamService : Service<Team>, ITeamService
  {
    public TeamService(IGenericRepository<Team> repository, IUnitOfWork unitOfWork) :
base(repository, unitOfWork)
    {
    }
  }
```

UserService.cs

UserService adlı bir sınıfı tanımlanır, bu sınıf **Service<User>** sınıfından miras alır ve **IUserService** arayüzünü uygular. **UserService** sınıfının Constructor'ı, genel bir veritabanı işlemlerini yöneten **IGenericRepository<User>** ve iş birimi deseni sağlayan **IUnitOfWork** bağımlılıklarını enjekte eder. Bu tasarım, genel CRUD işlemlerini yöneten bir servis katmanını temsil eder ve Dependency Injection, Generic Repository ve iş birimi deseni gibi yazılım tasarım desenlerini içerir.

```
public class UserService : Service<User>, IUserService
  {
    public UserService(IGenericRepository<User> repository, IUnitOfWork unitOfWork) :
base(repository, unitOfWork)
    {
    }
  }
```

UserProfileService.cs

UserProfileService adlı bir sınıfı tanımlanır, bu sınıf **Service<UserProfile>** sınıfından miras alır ve **IUserProfileService** arayüzünü uygular. **UserService** sınıfının Constructor'ı, genel bir veritabanı işlemlerini yöneten **IGenericRepository<UserProfile>** ve iş birimi deseni

sağlayan **IUnitOfWork** bağımlılıklarını enjekte eder. Bu tasarım, genel CRUD işlemlerini yöneten bir servis katmanını temsil eder ve Dependency Injection, Generic Repository ve iş birimi deseni gibi yazılım tasarım desenlerini içerir.

```
public class UserProfileService : Service<UserProfile>, IUserProfileService
  {
    public UserProfileService(IGenericRepository<UserProfile> repository, IUnitOfWork unitOfWork) : base(repository, unitOfWork)
    {
    }
  }
```

DTOs

Mapping işlemlerini gerçekleştirmek için DTO sınıflarını oluşturmayı planlıyoruz. Doğrudan entity'leri dış dünyaya açmak istemiyoruz. Bu amaçla, bir ara katman olarak düşünülen bir sınıf oluşturmak gerekir. Bu sınıf API'de **DTO** olarak adlandırılırken, MVC tarafında ise genellikle **ViewModel** olarak anılır. **Core Katmanı'nda**, **Team**, **User** ve **UserProfile** için ayrı ayrı DTO'ları içeren bir "**DTOs**" klasörü oluşturmanız yeterlidir. Ayrıca, genel kullanım için bir "**BaseDto**" da eklemek gerekir. Bu temel Dto'da, müşteri tarafında güncellenmiş tarih bilgisine(**UpdatedDate**) alanına ihtiyaç duymuyoruz ve navigation propertylere de gerek yok.

En iyi uygulama pratiği olarak, API'deki her endpoint için özel request ve response DTO'larını oluşturabiliriz; ancak bu biraz maliyetli olabilir. Bu nedenle, bu DTO'ları ayrı şekilde düzenleyip kullanmayı düşünebiliriz.

BaseDto

```
public abstract class BaseDto
  {
    public int Id { get; set; }
    public DateTime CreatedDate { get; set; }
  }
```

TeamDto

```
public class TeamDto:BaseDto
  {
    public string TeamName { get; set; }
  }
```

UserDto

```
public class UserDto:BaseDto
  {
    public string UserName { get; set; }
    public string Email { get; set; }
  }
```

UserProfileDto

```
public class UserProfileDto
  {
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
  }
```

Mapping İşlemleri

Servis katmanında, veri transfer işlemlerini kolaylaştırmak ve Entity nesnelerini **DTO'ya** veya **DTOları** Entity'ye dönüştürmek için **AutoMapper** kütüphanesi yaygın olarak kullanılmaktadır. **AutoMapper**, nesne eşleme (mapping) işlemlerini otomatize eden bir kütüphanedir ve bu sayede iki farklı sınıf arasında veri transferini büyük ölçüde kolaylaştırır.

AutoMapper ne ki?

AutoMapper, iki farklı sınıf arasında otomatik olarak veri transferi (mapping) sağlayan bir nesne eşleme kütüphanesidir. Bu kütüphane, bir nesnenin özelliklerini başka bir nesnenin özelliklerine uygun bir şekilde kopyalamak için kullanılır. Genellikle, Entity Framework gibi ORM (Object-Relational Mapping) araçlarıyla çalışan ve veritabanındaki nesneleri, API'lar veya kullanıcı arayüzleri gibi farklı katmanlarda kullanılan DTO'lar arasında veri transferi işlemlerinde tercih edilir. AutoMapper, manuel olarak her özelliği kopyalama ihtiyacını ortadan kaldırarak, kodun daha temiz, daha okunabilir ve daha az hata eğilimli olmasını sağlar. Ayrıca, projelerdeki değişikliklere daha esnek bir şekilde adapte olmayı ve sınıflar arasındaki uyumsuzlukları ele almayı kolaylaştırır.

Automapper kütüphanesini projeye dahil etme

AutoMapper'ı projemize eklemek için **AutoMapper.Extension.Microsoft.DependencyInjection** paketini kullanacağız. Bu paket, özellikle API veya MVC projelerinde **Dependency Injection** (Bağımlılık Enjeksiyonu) kullanımını destekler. Bu sayede AutoMapper'ın servis katmanına rahatlıkla entegre edilebilmesini sağlar.

AutoMapper'ı kullanmak için projemizin **Servis Katmanı'na** bir "**Mapping**" klasörü oluşturmak gerekir. Bu klasör içinde, farklı DTO ve Entity sınıflarını eşleştirmek için bir "**MapProfile**" adını taşıyan bir sınıf oluşturabilirsiniz. Bu sınıf genellikle public bir class olup, **AutoMapper'ın DI (Dependency Injection)** container'ına yüklenmesini sağlayacak olan API katmanında kullanılacaktır. Constructor içinde hangi nesneyi hangi nesneye nasıl maplemek istediğimizi belirteceğiz. Örnek aşağıdaki gibidir.

```
public class MapProfile:Profile
  {
    public MapProfile()
    {
      CreateMap<Team, TeamDto>().ReverseMap();
      CreateMap<User, UserDto>().ReverseMap();
      CreateMap<UserProfile, UserProfileDto>().ReverseMap();

      //dto'dan entity'e çevirmek istersem;
      CreateMap<TeamDto, Team>();
    }
  }
```

Örneğin, bir **TeamEntity** sınıfını **TeamDto** sınıfına veya tam tersi bir dönüşümü sağlamak için bu "**MapProfile**" sınıfında bir dönüşüm metodu tanımlayacağız. Bu işlemi gerçekleştirmek için AutoMapper'ın **ReverseMap** özelliğine ihtiyaç duyulmaz; çünkü AutoMapper otomatik olarak iki yönlü eşleştirmeleri gerçekleştirebilecek şekilde tasarlanmıştır. Sonuç olarak, AutoMapper kullanımı sayesinde, farklı sınıflar arasında veri transferini kolayca yönetebileceğiz ve bu da projenin daha esnek ve bakımı kolay olmasını sağlayacaktır.

Program.cs üzerinde nasıl bir ekleme yapmalıyız?

.NET Core veya ASP.NET Core uygulamasında **AutoMapper** kütüphanesini kullanıma eklemek üzere yapılan konfigürasyon işlemi aşağıdaki gibidir.

```
builder.Services.AddAutoMapper(typeof(MapProfile));
```

Validasyon İşlemleri

.NET Core'da validasyonlar, gelen verilerin doğruluğunu sağlamak, güvenliği artırmak, veri bütünlüğünü korumak ve iş kurallarını uygulamak amacıyla kullanılır. Kullanıcı tarafından sağlanan verilerin güvenilir ve beklenen formatta olmasını sağlamak, siber saldırı risklerini azaltır. Model bağlama doğrulaması ve açık doğrulama gibi yöntemlerle uygulanan validasyonlar, hem uygulama güvenliği hem de kullanıcı deneyimini iyileştirmeye yardımcı

olur. İş kurallarına uygunluğu kontrol etmek ve hatalı veri girişlerini önlemek, uygulamanın güvenilirliğini artırmak için kritiktir.

FluentValidation

Fluent Validation, .NET platformunda kullanılan bir doğrulama (validation) kütüphanesidir. Bu kütüphane, özellikle karmaşık doğrulama senaryolarını yönetmek ve temiz, esnek doğrulama kuralları oluşturmak için tasarlanmıştır. Fluent Validation, özellikle .NET Core projelerinde ve ASP.NET Core MVC uygulamalarında sıklıkla tercih edilmektedir. Fluent Validation, .NET Core ve ASP.NET Core ile uyumludur. Dependency Injection sistemine uygun olarak tasarlanmıştır, bu da projelerinize kolayca entegre edilebileceği anlamına gelir.Fluent Validation, doğrulama kurallarını açık ve deklaratif bir şekilde ifade etmeyi sağlar. Bu sayede karmaşık doğrulama senaryolarını daha okunabilir ve bakımı kolay bir şekilde tanımlayabilirsiniz.

FluentValidation Entegrasyonu

FluentValidation.AspNetCore paketini .NET Core projesine komut satırından yüklemek için aşağıdaki adımları takip edebilirsiniz. Bu adımlar, .NET CLI (Command Line Interface) kullanılarak gerçekleştirilir.

dotnet add package FluentValidation.AspNetCore

FluentValidation Örnekleri

Service katmanında “**Validations**” diye bir klasör açılarak **TeamDtoValidator**,**UserDtoValidator** ve **UserProfileValidator** classları oluşturulur ve validasyonlar bu classların içlerine eklenir.

TeamDtoValidator

```
public class TeamDtoValidator:AbstractValidator<TeamDto>
  {
     public TeamDtoValidator()
     {
        RuleFor(x => x.TeamName).NotNull().WithMessage("{PropertyName} null
geçilemez.").NotEmpty().WithMessage("{PropertyName} boş geçilemez");
     }
  }
```

UserDtoValidator

```
public class UserDtoValidator: AbstractValidator<UserDto>
  {
```

```
    public UserDtoValidator()
    {
      RuleFor(x => x.UserName).NotEmpty().WithMessage("Kullanıcı adı boş olamaz.")
        .NotNull().WithMessage("Kullanıcı adı null olamaz!")
        .MaximumLength(50).WithMessage("Kullanıcı adı en fazla 50 karakter olabilir. ");
      RuleFor(x => x.Email)
        .NotEmpty().WithMessage("Email boş geçilemez.")
        .NotNull().WithMessage("Email null olamaz.")
        .EmailAddress().WithMessage("Geçeri bir e-posta adresi giriniz.");
    }
  }
```

UserProfileValidator

```
public class UserProfileValidator: AbstractValidator<UserProfileDto>
  {
    public UserProfileValidator()
    {
      RuleFor(x => x.FirstName).NotEmpty().WithMessage("Kullanıcının adı boş olamaz.")
        .NotNull().WithMessage("Kullanıcının adı null olamaz.");
      RuleFor(x => x.LastName).NotEmpty().WithMessage("Kullanıcının soyadı boş
olamaz.")
        .NotNull().WithMessage("Kullanıcının soyadı null olamaz.");
    }
  }
```

FluentValidation Program.cs eklenmesi

ASP.NET Core uygulamasında FluentValidation'ın **AddFluentValidation** metodu aracılığıyla doğrulama sınıflarını eklemektedir. **RegisterValidatorsFromAssemblyContaining** metodu ise belirli bir sınıfın bulunduğu assembly içindeki tüm **FluentValidation** doğrulama sınıflarını otomatik olarak kaydetmek için kullanılır.

Bu kod parçasında, **TeamDtoValidator**, **UserDtoValidator** ve **UserProfileValidator** sınıfları içeren assembly içindeki tüm doğrulama sınıflarını kaydetmekte ve bunları **AddFluentValidation** metoduyla eklemektedir. Bu sayede, bu sınıfların kuralları otomatik olarak uygulanacaktır.

```
builder.Services.AddControllers()
  .AddFluentValidation(x =>
  {
```

```
    x.RegisterValidatorsFromAssemblyContaining<TeamDtoValidator>();
    x.RegisterValidatorsFromAssemblyContaining<UserDtoValidator>();
    x.RegisterValidatorsFromAssemblyContaining<UserProfileValidator>();
  });
```

Exceptions

Bu tür özel istisna sınıfları, genellikle uygulama içinde belirli bir durumu veya hatayı temsil etmek ve yönetmek için kullanılır. Bu durumda, **ClientSideException ve NotFoundException**sınıfı, istemci tarafında (client-side) gerçekleşen özel durumları temsil etmek üzere oluşturulmuş gibi görünmektedir. Bu Exception'u servis tarafına tanımladık çünkü bir sonraki yazıda bu sınıfı API katmanında kullanacağız.

Bu sınıf, **ClientSideException** adını taşır ve **Exception** sınıfından türemiştir. Bu sınıf, özel durumlarla ilgili istisnaları temsil etmek için kullanılabilir.

- **ClientSideException** sınıfı, **Exception** sınıfından türemiştir.
  Bu, **ClientSideException**'ın genel bir istisna olduğunu ve C# dilindeki istisna mekanizmasını kullanarak programın normal akışını durdurabileceğini gösterir.
- **[Serializable]** özelliği, bu sınıfın nesne durumlarının serileştirilebilir olduğunu belirtir. Serileştirme, nesne durumlarını ikincil depolama veya ağ üzerinden taşıma gibi senaryolarda kullanışlıdır.
- **ClientSideException** sınıfı, bir parametre olarak bir **string** alarak başlatılabilir. Bu, istisna oluşturulurken bir hata iletisi iletmek için kullanılır.
- **protected ClientSideException(SerializationInfo info, StreamingContext context)** adında bir özel kurucu metot bulunmaktadır. Bu, serileştirilmiş bir **ClientSideException** nesnesini oluştururken çağrılabilir. Bu metot, **info** ve **context** parametrelerini kullanarak serileştirme sırasında bilgi alabilir.

ClientSideException Sınıfı

```
using System;
using System.Runtime.Serialization;
namespace ZeyrekAgentPanelBE.Service.Exceptions
{
  [Serializable]
  public class ClientSideException : Exception
  {
    public ClientSideException(string message) : base(message)
    {
```

```
    }
    protected ClientSideException(SerializationInfo info, StreamingContext context) :
base(info, context)
    {
    }
  }
}
```

NotFound Exception Sınıfı

```
using System;
using System.Runtime.Serialization;

namespace ZeyrekAgentPanelBE.Service.Exceptions
{
  [Serializable]
  public class NotFoundException : Exception
  {
    public NotFoundException(string message) : base(message)
    {
    }
    protected NotFoundException(SerializationInfo info, StreamingContext context) :
base(info, context)
    {
    }
  }
}
```

Bugünlük benim anlatacaklarım bu kadar!

Bir sonraki yazıda görüşmek üzere :)

KAYNAKÇA

- https://www.youtube.com/watch?v=r-RUY2caw3s
- https://www.youtube.com/watch?v=xJC7ItRoEbw
- https://www.youtube.com/watch?v=Srp1iyZu-ww&pp=ygUNbi1sYXllciBwcm9qZQ%3D%3D
- https://www.udemy.com/course/asp-net-core-api-web-cok-katmanli-mimari-api-best-practices/

- https://www.gencayyildiz.com/blog/c-ta-n-tier-architecturecokn-katmanli-mimari/