**T.C**

**İSTANBUL AYDIN ÜNİVERSİTESİ**

**ANADOLU BİL MESLEK YÜKSEKOKULU**

**Logo

Description automatically generated**

**CoreForce.NTierRestApi**

**ÖN LİSANS TEZİ/ MEZUNİYET PROJESİ**

**Emir Bahadır VERİR**

**Anadolu Bil Meslek Yüksekokulu**

**Bilgisayar Programcılığı**

**HAZİRAN / 2025**

**T.C**

**İSTANBUL AYDIN ÜNİVERSİTESİ**

**ANADOLU BİL MESLEK YÜKSEKOKULU**

**Logo

Description automatically generated**

**CoreForce.NTierRestApi**

**ÖN LİSANS TEZİ/ MEZUNİYET PROJESİ**

**Emir Bahadır VERİR B2310.032025**

**Anadolu Bil Meslek Yüksekokulu**

**Bilgisayar Programcılığı**

**HAZİRAN / 2025**

# **ÖZET**

Bu çalışma kapsamında, C# ve ASP.NET Core teknolojileri kullanılarak yalnızca sunucu tarafında çalışan bir RESTful Web API geliştirilmiştir. Geliştirilen yazılım, bir kurumun insan kaynakları ya da yönetim birimi tarafından kullanılabilecek bir yönetim panelinin arka plan sistemini temsil etmektedir. Sistem üç temel varlık olan Çalışan (Employee), Departman(Department) ve Pozisyon (Job) nesneleri üzerine kurgulanmıştır. Bu varlıklar üzerinden CRUD (Create, Read, Update, Delete) işlemleri gerçekleştirilerek temel iş süreçleri simüle edilmiştir.

Proje, yazılım geliştirme süreci boyunca 5 aşamada ilerletilmiş; her aşamada farklı konulara odaklanılarak yazılım mimarisi adım adım genişletilmiş ve derinleştirilmiştir. Uygulama geliştirilirken katmanlı mimari, Entity Framework Core, DTO yapıları, AutoMapper, asenkron programlama, hata yönetimi, custom middleware, IQueryable ve IEnumerable optimizasyonu, action filter’lar, sayfalama, filtreleme, JWT tabanlı kimlik doğrulama ve rol bazlı yetkilendirme gibi birçok modern yazılım geliştirme tekniği kullanılmıştır.

Proje süresince edinilen bilgi birikimi, hem teorik kavrayış hem de pratik yeteneklerin eş zamanlı olarak geliştirilmesini sağlamıştır. Yapılan bu çalışma; kurumsal düzeyde geliştirilen, güvenli, ölçeklenebilir ve sürdürülebilir back-end sistemlerin nasıl inşa edilebileceğine dair uygulamalı bir örnek sunmaktadır. Aynı zamanda yazılım mimarisinin parçalı şekilde ele alınması, konuların sade bir şekilde öğrenilmesine katkı sağlamıştır.

**Anahtar Kelimeler:** RestfulApi, Backend, Web servisi, Katmanlı mimari, Yönetim paneli.

# **ABSTRACT**

In this study, a RESTful Web API that operates solely on the server side has been developed using C# and ASP.NET Core technologies. The software represents the backend system of a management panel that could be used by a company’s human resources or administrative units. The system is designed around three core entities: Employee, Department, and Job. CRUD (Create, Read, Update, Delete) operations are performed on these entities to simulate essential business processes.

The project was developed through five progressive stages, with each stage focusing on different topics to gradually expand and deepen the software architecture. During development, many modern software development techniques were implemented, including layered architecture, Entity Framework Core, DTO structures, AutoMapper, asynchronous programming, exception handling, custom middleware, IQueryable optimization, action filters, pagination, filtering, JWT-based authentication, and role-based authorization.

Throughout the project, both theoretical understanding and practical skills were developed in parallel. This study presents a practical example of how secure, scalable, and maintainable back-end systems can be built at an enterprise level. Additionally, the modular approach to software architecture has contributed to a clearer understanding of complex topics.

**Keywords:** Restful API, Backend, Web service, N-Tier architecture, Management panel

# **İÇİNDEKİLER**

[ÖZET i](#_Toc200910147)

[ABSTRACT ii](#_Toc200910148)

[İÇİNDEKİLER iii](#_Toc200910149)

[I. GİRİŞ 1](#_Toc200910150)

[A. Projenin Amacı 1](#_Toc200910151)

[B. Projenin Kapsamı 2](#_Toc200910152)

[C. Projenin Önemi 3](#_Toc200910153)

[II. KULLANILAN TEKNOLOJİLER VE ARAÇLAR 4](#_Toc200910154)

[A. C# 4](#_Toc200910155)

[B. ASP.NET Core 5](#_Toc200910156)

[C. Entity Framework Core 6](#_Toc200910157)

[D. MySQL 7](#_Toc200910158)

[E. Swagger 8](#_Toc200910159)

[F. Postman 9](#_Toc200910160)

[III. YAZILIM GELİŞTİRME SÜRECİ 10](#_Toc200910161)

[A. Proje Yapısı ve Katmanlı Mimari 10](#_Toc200910162)

[1.Entities Katmanı: 10](#_Toc200910163)

[2.Repositories Katmanı: 10](#_Toc200910164)

[3.Services Katmanı: 10](#_Toc200910165)

[4.Sunum Katmanı (Presentation Layer): 11](#_Toc200910166)

[5.WebAPi Katmanı : 11](#_Toc200910167)

[B. Temel Yapı ve Altyapı Kurulumu 12](#_Toc200910168)

[1. EFcore 13](#_Toc200910169)

[2. DTO 14](#_Toc200910170)

[3. AutoMapper 15](#_Toc200910171)

[4. Extension Metod 16](#_Toc200910172)

[5. SeedData 17](#_Toc200910173)

[6. Dependency Injection 18](#_Toc200910174)

[7. FluentApi 19](#_Toc200910175)

[C. Asenkron Programlama ve Hata Yönetimi 20](#_Toc200910176)

[1. Asenkron Kod 20](#_Toc200910177)

[2. Hata Yönetimi 21](#_Toc200910178)

[3. Validasyon 22](#_Toc200910179)

[4. İçerik Pazarlığı 23](#_Toc200910180)

[5. Custom Exception Middleware 24](#_Toc200910181)

[D. Performans, Test ve Refactoring 25](#_Toc200910182)

[1. IEnumerable Ve IQueryable Optimizasyonu 25](#_Toc200910183)

[2. TrackChanges 26](#_Toc200910184)

[3. Refactoring Ve Api Testi 27](#_Toc200910185)

[E. Gelişmiş API Özellikleri 28](#_Toc200910186)

[1. Action Filters 28](#_Toc200910187)

[a. Content-Type İfadesnin Kontrolünü Gerçekleştiren ActionFilter ifadesi ve İşlevi:…………. 28](#_Toc200910188)

[b. Model Geçerliliğini Doğrulayan ActionFilter ifadesi ve İşlevi: 28](#_Toc200910189)

[2. Sayfalama 29](#_Toc200910190)

[3. Filtreleme 30](#_Toc200910191)

[4. Arama 31](#_Toc200910192)

[5. Sıralama 32](#_Toc200910193)

[6. Veri Şekillendirme 33](#_Toc200910194)

[7. Cors 34](#_Toc200910195)

[F. Güvenlik ve Dokümantasyon 35](#_Toc200910196)

[1. Options ve Head 35](#_Toc200910197)

[2. Jwt ile Rol Bazlı Güvenlik 36](#_Toc200910198)

[3. Swagger Genişletilmesi 38](#_Toc200910199)

[4. Postman Dökümantasyonu 39](#_Toc200910200)

[IV. SONUÇ VE DEĞERLENDİRME 40](#_Toc200910201)

[V. KAYNAKÇA 41](#_Toc200910202)

[VI. EKLER 43](#_Toc200910203)

1. **GİRİŞ**
2. **Projenin Amacı**

Bu projenin temel amacı, bir kurumun insan kaynakları ya da yönetim biriminin ihtiyaçlarını karşılayabilecek güvenli, ölçeklenebilir ve sürdürülebilir bir back-end sistemi geliştirmektir. Bu hedef doğrultusunda yalnızca sunucu tarafında çalışan bir RESTful Web API tasarlanmış ve gerçek dünya senaryoları göz önünde bulundurularak işlevsel bir yazılım altyapısı inşa edilmiştir. Amaç sadece teknik olarak çalışan bir sistem oluşturmak değil, aynı zamanda modern yazılım geliştirme prensiplerine uygun, geliştirilmeye açık bir yapı kurmaktır.

Proje kapsamında geliştirilen API, departman, pozisyon ve çalışan gibi temel iş birimi kavramları üzerinden şekillenmiştir. Bu yapılar üzerinden CRUD işlemleri gerçekleştirilmiş, sistemin temel işlevselliği sağlanmıştır. Ayrıca, kullanıcı rolleri ve yetkilendirme senaryoları da projeye entegre edilerek gerçekçi kullanım durumlarına uyumlu bir mimari kurulmuştur.

Yalnızca teorik bilgilerle değil, aynı zamanda gerçek uygulamalarla desteklenen bu proje, yazılım geliştirme sürecinde edinilen bilgi ve becerilerin pekiştirilmesini sağlamayı da hedeflemektedir. Yazılım dünyasında sıkça karşılaşılan sorunlar, örneğin hata yönetimi, veri tutarlılığı, performans optimizasyonu ve güvenlik gibi alanlara da çözümler üretilerek, kapsamlı bir öğrenme ve üretim süreci sağlanmıştır.

Sonuç olarak bu projenin amacı, kurumsal düzeyde kullanılabilecek nitelikte, güncel yazılım geliştirme yaklaşımlarını uygulamalı olarak içeren, örnek bir Web API çözümü sunmaktır. Bu sayede, hem geliştiricilere yol gösterebilecek bir rehber, hem de yazılım mimarisi açısından iyi yapılandırılmış bir temel oluşturulmuştur.

1. **Projenin Kapsamı**

Bu proje, bir kurumsal yapıda sıklıkla karşılaşılan yönetimsel süreçleri dijital ortama taşımayı hedefleyen bir sunucu taraflı yazılım geliştirme çalışmasıdır. Temel olarak sadece bir veri kaynağına erişen değil, aynı zamanda iş mantığını doğru şekilde yöneten, güvenlik politikaları ile korunan ve dış sistemlerle entegrasyon imkânı sunan bir RESTful Web API geliştirilmiştir.

Geliştirilen sistem üç ana varlık (departman, pozisyon, çalışan) üzerine kurulmuştur. Bu varlıklar, kurumsal bir yapıdaki departman yapılanmasını, her departmanda görev alabilecek iş tanımlarını ve bu pozisyonlarda çalışan bireyleri temsil eder. Bu yapı sayesinde kullanıcılar sisteme çalışan ekleyebilir, güncelleyebilir, silebilir ve mevcut kayıtlar üzerinde çeşitli sorgulamalar gerçekleştirebilir. Ayrıca kullanıcıların yalnızca rol ve yetkilerine uygun verilere erişmesini sağlayan JWT tabanlı güvenlik altyapısı da proje kapsamında ele alınmıştır.

Proje, beş aşamalı bir geliştirme süreci izlenerek hayata geçirilmiştir. Her aşama, yazılım geliştirme sürecinin farklı bir yönüne odaklanacak şekilde yapılandırılmıştır. İlk aşamada altyapı kurulumu ve temel yapıların oluşturulması; ikinci aşamada hata yönetimi ve asenkron programlama; üçüncü aşamada performans ve refactoring çalışmaları; dördüncü aşamada gelişmiş API özellikleri; beşinci ve son aşamada ise güvenlik ve dokümantasyon işlemleri gerçekleştirilmiştir.

Kapsam olarak yalnızca geliştirici taraflı API çözümleri sunulmuş, kullanıcı arayüzü (frontend) geliştirme bu çalışmanın dışında bırakılmıştır. Böylece proje, back-end geliştirme sürecine odaklanan, teknik anlamda sağlam temellere sahip bir eğitim ve üretim çalışması niteliği taşımaktadır. Ayrıca, ileride yapılabilecek frontend entegrasyonlarına da açık bir yapıda bırakılmıştır.

Bu yönüyle proje, hem eğitim amaçlı hem de gerçek projelerde temel alınabilecek bir örnek uygulama niteliğindedir.

1. **Projenin Önemi**

Günümüzde kurumların dijitalleşme süreci, yalnızca görsel arayüzlerin değil, aynı zamanda bu arayüzleri besleyen güçlü ve güvenilir arka plan sistemlerinin varlığı ile mümkün olmaktadır. Bu proje, tam da bu noktada, bir kurumsal yapının dijital omurgasını oluşturabilecek nitelikte bir back-end altyapısının nasıl geliştirileceğini ortaya koymaktadır.

Geleneksel yazılım geliştirme süreçlerinden farklı olarak, bu çalışmada sadece uygulamanın işlevselliği değil, aynı zamanda kod kalitesi, genişletilebilirlik ve sürdürülebilirlik gibi yazılım mühendisliğine özgü temel ilkeler de ön planda tutulmuştur. Uygulama, geliştirici bakış açısıyla ele alınmış; modüler yapısı, servis bağımlılıklarının yönetimi, iş katmanlarının ayrıştırılması gibi mimari prensiplerle desteklenmiştir. Bu sayede, farklı ihtiyaçlara ve taleplere göre rahatlıkla genişletilebilecek bir yapı elde edilmiştir.

Projede kullanılan modern araçlar ve teknikler örneğin AutoMapper ile model-dto dönüşümleri, FluentValidation ile özelleştirilmiş doğrulamalar, IQueryable ile performans optimizasyonu, JWT ile kullanıcı yetkilendirmesi yalnızca teknik anlamda değil, aynı zamanda mesleki gelişim açısından da önemli kazanımlar sunmuştur. Öğrenilen her yapı, yazılım geliştirme sürecinin sadece bir adımı değil, aynı zamanda daha büyük sistemlerin temellerini oluşturan parçalardır.

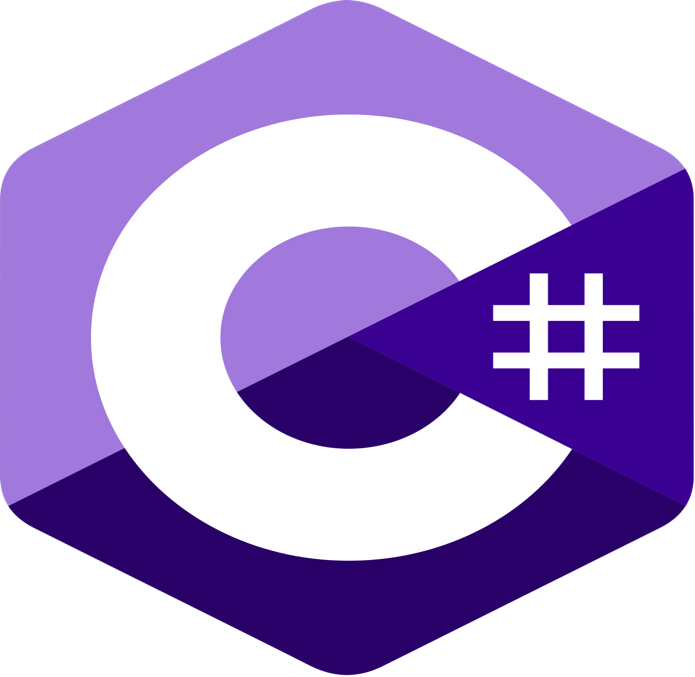
Bu proje, özellikle geliştirici olmak isteyen bireylerin, yalnızca çalışan bir uygulama geliştirmekle kalmayıp, aynı zamanda bu uygulamanın neden bu şekilde geliştirildiğini anlayabilmelerini hedeflemektedir. Böylece teorik bilgilerin pratiğe aktarılmasında bir köprü görevi görmektedir.

Sonuç olarak, bu çalışma yalnızca bir proje değil, aynı zamanda yazılım dünyasında sağlam bir temel atmak isteyenler için bir rehber niteliği taşımaktadır.

1. **KULLANILAN TEKNOLOJİLER VE ARAÇLAR**
   1. **C#**

**C-Sharp**, Microsoft tarafından geliştirilen ve .NET platformunun temelini oluşturan, nesne yönelimli ve modern bir programlama dilidir. İlk kez 2000’li yılların başında tanıtılan bu dil, özellikle kurumsal uygulamalar, masaüstü yazılımları ve web servisleri gibi geniş bir kullanım alanına sahiptir. Tip güvenliği, güçlü hata yakalama mekanizmaları ve gelişmiş IDE desteği ile yazılım geliştiricilere sağlam ve düzenli kod üretimi için verimli bir zemin sunar. Gelişim sürecinde sürekli güncellenen yapısı sayesinde hem klasik hem de çağdaş yazılım ihtiyaçlarına kolayca adapte olabilir.

Projenin ana programlama dili olarak C# seçilmiştir. Dil, statik tip sistemi sayesinde derleme anında hataları yakalayarak güvenilir kod üretimine katkı sağlar. LINQ, async‑await, extension method’lar ve record yapıları gibi modern özellikleri, kodun hem okunaklı hem de kısa tutulmasını mümkün kılar. Özellikle async ve await anahtar sözcükleri, veritabanı işlemlerinde bloklamayı önleyerek API’nın yüksek eşzamanlılık altında bile tutarlı yanıt süreleri sunmasını sağlar. Dilin .NET ekosistemiyle sıkı entegrasyonu, güçlü standart kütüphaneler ve açık kaynak topluluğu desteği sayesinde uzun vadede sürdürülebilir bir temel yaratır.

****

* 1. **ASP.NET Core**

**ASP.NET Core**, Microsoft’un açık kaynak olarak geliştirdiği, modern ve hafif bir web uygulama çatısıdır. Geleneksel ASP.NET mimarisinin evrimleşmiş hali olan bu platform, yüksek performans, esneklik ve test edilebilirlik gibi temel yazılım ihtiyaçlarına etkin çözümler sunar. Platformdan bağımsız (cross-platform) yapısı sayesinde Windows, Linux ve macOS üzerinde sorunsuz çalışabilir. Hem Web API hem de MVC mimarisini desteklemesi, küçük ölçekli servislerden kurumsal seviyedeki uygulamalara kadar geniş bir kullanım alanı sağlar. Ayrıca topluluk desteği ve sürekli güncellenen sürümleri ile modern yazılım geliştirme pratiklerini takip etmek isteyen geliştiriciler için güçlü bir tercihtir.

Bu projede ASP.NET Core’un modern yazılım geliştirme yeteneklerinden faydalanılmıştır. Yerleşik bağımlılık enjeksiyonu (Dependency Injection) mekanizması, modüler middleware altyapısı ve sade yapılandırılabilir istek işleme hattı, projede katmanlı mimarinin sürdürülebilir şekilde uygulanmasına olanak tanımıştır. Controller’lar, attribute tabanlı yönlendirme (routing) ve açık metod imzaları ile sade ve okunabilir bir biçimde yapılandırılmıştır. Projede geleneksel controller-temelli Web API yaklaşımı tercih edilerek, her bir isteğin anlaşılır ve izlenebilir şekilde yönetilmesi sağlanmıştır. Ayrıca ASP.NET Core’un sade konfigürasyon yapısı ve yerleşik hata yönetimi mekanizmaları, uygulamanın bakımını kolaylaştırarak geliştirme sürecinde kararlılığı artırmıştır. Bu sayede hem kod kalitesi korunmuş hem de sürdürülebilir bir yapı elde edilmiştir.



* 1. **Entity Framework Core**

Entity Framework Core (EF Core), Microsoft tarafından geliştirilen modern bir nesne-ilişkisel eşleme (ORM) kütüphanesidir. Veritabanı işlemlerini doğrudan SQL yazmadan gerçekleştirmeyi sağlayarak, geliştiricilere daha okunabilir ve sürdürülebilir bir veri erişim katmanı sunar. LINQ ile entegre yapısı sayesinde sorgular C# üzerinden yazılabilir ve çalışma zamanında veritabanına çevrilir. Platform bağımsız çalışabilmesi ve esnek yapılandırma seçenekleri, onu modern .NET projeleri için güçlü bir tercih haline getirir.

Veri erişim katmanında **Entity Framework Core** tercih edilmiştir. Code‑First yaklaşımı sayesinde C# sınıfları doğrudan veritabanı şemasına dönüştürülmüş, böylece model değişiklikleri migration dosyaları ile senkronize edilmiştir. EF Core’un **IQueryable** tabanlı sorgu oluşturma yeteneği, ihtiyaç duyulan veriyi veritabanı tarafında filtreleyerek aktarmak için kullanılmış; bu da ağ trafiğini ve bellek tüketimini azaltmıştır. Ayrıca ChangeTracker’ın devre dışı bırakılabildiği **No‑Tracking** sorgular ile okuma operasyonlarında ekstra performans kazanılmıştır.



* 1. **MySQL**

MySQL, dünyada en yaygın kullanılan açık kaynaklı ilişkisel veritabanı yönetim sistemlerinden biridir. Yüksek performansı, güvenilirliği ve güçlü topluluk desteği ile küçük ölçekli uygulamalardan büyük kurumsal sistemlere kadar geniş bir kullanım alanına sahiptir. MySQL, esnek yapısı sayesinde farklı işletim sistemlerinde çalışabilir ve çok sayıda programlama diliyle kolayca entegre olur.

Projede ilişkisel veritabanı olarak MySQL kullanılmıştır. Açık kaynak lisansı, yaygın barındırma seçenekleri ve replikasyon özellikleri, ölçeklenebilirlik ve maliyet avantajı sunar. EF Core’un MySQL sağlayıcısı (Pomelo) ile tam uyumlu çalışması sayesinde migration’lar sorunsuz yürütülmüş, veritabanı işlemleri C# tarafında şeffaf biçimde yönetilmiştir. MySQL Workbench ve CLI araçları, şema inceleme ve performans izleme süreçlerini kolaylaştırmıştır.



* 1. **Swagger**

API tasarımı ve geliştirmede, servislerin anlaşılır ve kolay erişilebilir olması kritik öneme sahiptir. Bu amaçla, RESTful servislerin otomatik olarak tanımlanması ve interaktif şekilde test edilebilmesi için standartlaştırılmış dokümantasyon araçlarına ihtiyaç duyulur. OpenAPI spesifikasyonu, bu ihtiyacı karşılamak üzere geliştirilmiş en yaygın çözümdür.

REST uç noktalarının belgelendirilmesi için Swagger / OpenAPI standardı kullanılmıştır. Swashbuckle.AspNetCore paketiyle entegre edilen Swagger UI, hem interaktif test ortamı hem de canlı dokümantasyon sunar. JWT şema tanımlamaları eklenerek kimlik doğrulamalı istekler de arayüz üzerinden denenmiştir. Bu yaklaşım, proje ekibi ve üçüncü taraf geliştiriciler için öğrenme eğrisini azaltır, sürüm değişikliklerinde tutarlı iletişimi garanti eder.

* 1. **Postman**

Postman, geliştiricilerin RESTful API’leri test etmesini, belgelemesini ve otomatikleştirmesini sağlayan popüler bir API istemcisidir. Web tabanlı ve masaüstü sürümleriyle kullanılabilen araç, HTTP isteklerini hızlıca yapılandırma ve yanıtlara dair detaylı incelemeler yapma imkânı sunar. Çevresel değişken tanımlama, koleksiyon yönetimi ve otomatik test scriptleri gibi özellikleriyle yazılım projelerinde hem bireysel geliştiriciler hem de ekipler için vazgeçilmez bir test ortamı oluşturur.

Manuel istek senaryoları ile yük ve regresyon testleri Postman üzerinde yürütülmüştür. Ortak koleksiyonlar sayesinde, farklı ortamlara (development, staging) özgü değişkenler tanımlanarak testler koddan bağımsız şekilde takip edilmiştir. Postman’in sunduğu otomatik test scriptleri, her endpoint’in beklenen durum kodunu ve yanıt gövdesi şemasını doğrulayarak CI/CD süreçlerine entegre edilebilir çıktılar üretmiştir. Bu sayede, yalnızca manuel değil, tekrarlanabilir ve izlenebilir test süreçleri de kurulmuştur. Ayrıca koleksiyonların dışa aktarılması, proje paydaşlarının API’yi hızlıca deneyimlemesini sağlamış, geliştirme sürecinde ekip içi iletişimi hızlandırmıştır. Test scriptleri ve otomasyon altyapısı üzerine yapılan bu çalışmalar, kod tekrarını azaltarak API’nin modülerliğine doğrudan katkı sağlamıştır.



1. **YAZILIM GELİŞTİRME SÜRECİ**
2. **Proje Yapısı ve Katmanlı Mimari**

Proje, sürdürülebilirlik ve geliştirilebilirlik hedefleri doğrultusunda **katmanlı mimari** (N-Tier architecture) prensibiyle yapılandırılmıştır. Bu yaklaşım, her bir katmanın belirli bir sorumluluğa sahip olmasını sağlar ve böylece kodun okunabilirliğini, test edilebilirliğini ve modülerliğini artırır. Katmanların birbirinden ayrılması sayesinde değişikliklerin etkisi minimize edilmiş, bağımlılıklar sadeleştirilmiş ve yeniden kullanılabilir bileşenlerin önü açılmıştır.

Proje beş ana katmandan oluşmaktadır:

**1.Entities Katmanı**:

Veritabanı ile doğrudan ilişkili sınıfları barındırır. Department, Employee, Job gibi temel modeller bu katmanda tanımlanmıştır. Bu katman sadece veri yapısını temsil eder, iş mantığı içermez. Fluent API ve veri açıklama (data annotation) öznitelikleri aracılığıyla veritabanı kuralları burada modellenmiştir.

**2.Repositories Katmanı**:

Veri erişim işlemlerinin gerçekleştirildiği katmandır. Entity Framework Core kullanılarak oluşturulan bu yapı, CRUD işlemlerinin yönetimini sağlar. Ayrıca sorgu optimizasyonu, IQueryable üzerinden filtreleme ve trackChanges desteği gibi gelişmiş veri erişim stratejileri de burada uygulanır. Arayüzler üzerinden programlandığı için, kolayca test edilebilir ve gerektiğinde farklı veri kaynaklarına adapte edilebilir.

**3.Services Katmanı**:

İş mantığının uygulandığı katmandır. Repository katmanından gelen veriler burada işlenir, doğrulamalar yapılır ve dış dünyaya sunulmaya hazır hale getirilir. Ayrıca servisler, farklı işlemlerin birleştirildiği senaryoları da içerir (örneğin bir departman silindiğinde ilişkili çalışanların durumlarının güncellenmesi gibi). Bu katman, controller’ların yükünü azaltarak tek sorumluluk ilkesine uygun bir yapı oluşturur.

**4.Sunum Katmanı (Presentation Layer)**:

Kullanıcılar veya istemcilerle doğrudan etkileşim kurulan katmandır. ASP.NET Core MVC altyapısı üzerinde çalışan bu yapı, RESTful endpoint’ler aracılığıyla dış dünyaya hizmet sunar. Gelen istekler burada doğrulanır, ilgili servisler çağrılır ve sonuçlar HTTP cevapları olarak geri döner. JSON formatındaki çıktılar, içerik pazarlığı (content negotiation) prensiplerine göre biçimlendirilmiştir. Swagger UI entegrasyonu da bu katmanda yapılandırılmıştır.

**5.WebAPi Katmanı** :

Katmanlar arasında bağımlılıklar yukarıdan aşağıya doğrudur. API katmanı Service katmanına, Service katmanı Repository’ye , Repository ise Entities katmanına bağımlıdır. Böylece , ters yöndeki erişimler engellenmiş ve katmanlar arası gevşek bağlılık sağlanmıştır. Sonuç olarak , bu yapı sadece mevcut geliştirme sürecini kolaylaştırmakla kalmamış , aynı zamanda ileride yapılacak genişletme ve bakım işlemleri için de sağlam bir zemin hazırlamıştır. Özellikle ekip çalışması ve modüler geliştirme açısından bu mimari tercih , projenin başarıya ulaşmasında temel rol oynamıştır.

1. **Temel Yapı ve Altyapı Kurulumu**

Projenin temeli atılırken ilk olarak katmanlı mimarinin sorunsuz çalışmasını sağlayacak altyapı oluşturulmuştur. Bu kurulum süreci, uygulamanın ileride genişletilebilir ve sürdürülebilir olmasını garanti altına alacak şekilde planlanmıştır. Geliştirme ortamı olarak Visual Studio Code tercih edilmiş ve .NET 8.0 ile uyumlu olacak biçimde proje şablonları oluşturulmuştur.

İlk adımda, proje katmanları birbirinden bağımsız olarak oluşturulmuş ve her katman sorumluluklarına göre ayrılmıştır. Entities, Repositories, Services, WebAPI ve Presentation olmak üzere beş temel katman kurgulanmış; her biri kendi içinde ayrı bir mantıksal yapıya sahiptir. Böylelikle, bir katmandaki değişiklik diğerlerini etkilemeden yönetilebilmiştir.

Veritabanı işlemleri için kullanılan Entity Framework Core, Entities ve Repositories katmanlarına entegre edilmiştir. Burada kullanılan DbContext sınıfı, doğrudan Entities içindeki modellerle ilişkilendirilerek Code First yaklaşımıyla migration dosyalarının oluşturulmasına olanak tanımıştır. Bu sayede, veritabanı şeması C# kodu üzerinden kontrol edilebilmiş, tablo ilişkileri açık ve anlaşılır biçimde yönetilmiştir.

WebAPI katmanı, projenin tüm yapılandırmalarının merkezi haline getirilmiştir. Program.cs dosyasında yer alan servis ekleme işlemleri, genişletilebilirlik gözetilerek extension metotlar biçiminde her biri ayrı ayrı dosyalara alınmıştır. Örneğin AddApplicationServices, ConfigureRepositoryManager ve AddCustomSwagger gibi özel uzantılar, startup kodunu sade ve okunabilir kılmıştır. Aynı zamanda bu yapı, bağımlılıkların dışarıdan yönetilmesine imkân tanıyarak projenin test edilebilirliğini artırmıştır.

İçerik pazarlığı (content negotiation), özel hata filtreleri ve global hata yakalama mekanizmaları bu aşamada projeye dahil edilmiştir. Özellikle validasyon için oluşturulan özel ValidationFilter bileşeni, kullanıcıdan gelen hatalı istekleri yakalayarak tutarlı geri dönüşler sağlamıştır. Bu, sadece kullanıcı deneyimini iyileştirmekle kalmamış; aynı zamanda API’nin dış servislerle entegrasyonunu da güvenli hale getirmiştir.

Ek olarak, AutoMapper konfigürasyonları da proje başlangıcında sisteme entegre edilmiştir. Bu sayede veritabanı varlıkları ile dış dünyaya sunulan DTO’lar arasında manuel dönüştürme ihtiyacı ortadan kalkmıştır. Her şeyin modüler ve bağımsız olarak yapılandırıldığı bu altyapı kurulum aşaması, projenin tüm diğer parçaları için sağlam bir temel oluşturmuştur.

1. **EFcore**

Projemizde veri erişim katmanının temel yapı taşı olarak **Entity Framework Core (EF Core)** kullanılmıştır. EF Core, Microsoft tarafından geliştirilen açık kaynaklı, hafif ve genişletilebilir bir ORM (Object-Relational Mapper) aracıdır. Bu teknoloji, C# nesneleri ile ilişkisel veritabanları arasında köprü görevi görür ve geliştiricilerin veritabanı işlemlerini nesne odaklı bir yaklaşımla gerçekleştirmesine olanak tanır. Böylece doğrudan SQL yazmaya gerek kalmadan, daha güvenli ve bakımı kolay kodlar yazılabilir.

Projede EF Core’un önemi, veritabanı şemasının C# sınıflarından otomatik olarak oluşturulması ve yönetilmesinden gelir. Bu, **Code First** yaklaşımı sayesinde sağlanmıştır. Code First yöntemi, geliştiricilerin öncelikle model sınıflarını tanımlaması ve bu modeller üzerinden veritabanının şekillenmesini mümkün kılar. Böylece veritabanı yapısında yapılacak değişiklikler, migration dosyaları ile senkronize edilip versiyon kontrolüne dahil edilebilmiştir. Bu durum, geliştirme sürecinde veritabanı yönetimini büyük ölçüde kolaylaştırmış ve sürüm uyumsuzluğu riskini azaltmıştır.

EF Core’un sunduğu en önemli avantajlardan biri, **IQueryable** tabanlı sorgu yapısıdır. Bu yapı, sorguların veritabanı tarafında optimize edilerek çalışmasına izin verir. Ayrıca, **Change Tracker** mekanizması, EF Core’un sağladığı güçlü bir özellik olarak, nesne değişikliklerini izlemeyi sağlar. Projemizde, sadece veri güncelleme ve ekleme işlemlerinde bu izleme aktif tutulmuş, okuma işlemlerinde ise **No Tracking** sorgular tercih edilmiştir. Bu yöntem, özellikle yüksek trafik altında performansı artırmak için kritik önem taşır.

EF Core’un esnekliği ve geniş eklenti desteği, projenin diğer teknolojileri ile uyumlu çalışmasına da olanak tanımıştır. MySQL veritabanı ile kullanılan Pomelo sağlayıcısı sayesinde, EF Core migration’ları ve sorguları sorunsuz şekilde yürütülmüş; böylece platformlar arası veritabanı bağımsızlığı da sağlanmıştır.

Sonuç olarak, Entity Framework Core, bu projenin hem veri yönetimi hem de sürdürülebilirlik açısından en önemli bileşenlerinden biri olmuştur. Proje boyunca sağladığı kolaylıklar, kod kalitesine olumlu yansımış ve geliştirici deneyimini üst seviyeye çıkarmıştır.

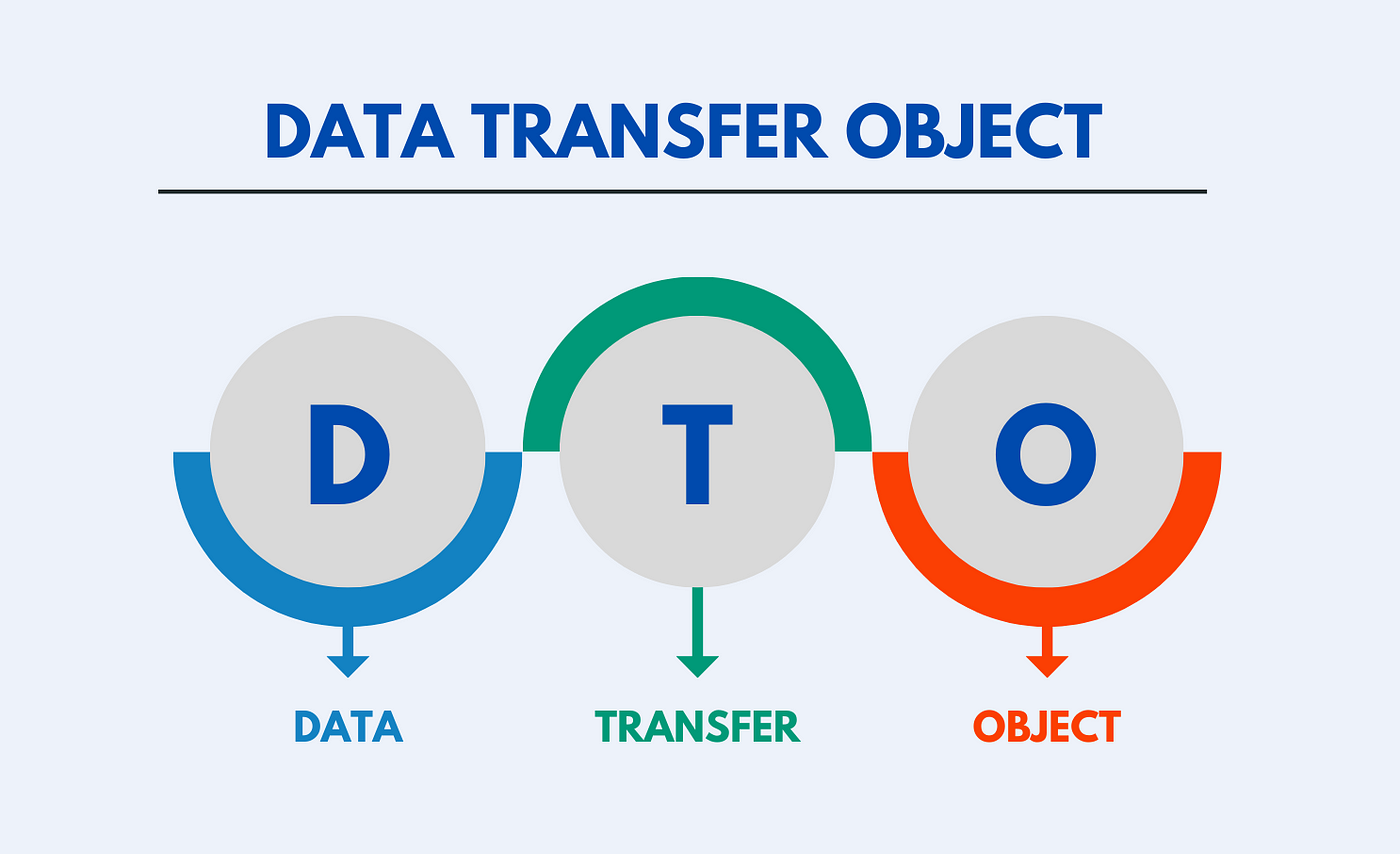
1. **DTO**

Projede veri katmanları arasında veri iletimini düzenlemek ve gereksiz veri sızıntısını önlemek amacıyla **Data Transfer Object (DTO)** yapıları kullanılmıştır. DTO’lar, özellikle API ile istemciler arasında veri taşımada kullanılan sade ve yalnızca gerekli alanları içeren nesnelerdir. Bu sayede, veri modellerinin tüm ayrıntılarını dış dünyaya açmak yerine, sadece işlem için ihtiyaç duyulan bilgiler paylaşılır.

DTO kullanımı, uygulamanın güvenliğini artırırken, veri iletimini optimize ederek performansa katkı sağlar. Özellikle karmaşık nesne grafiklerinin istemciye gönderilmesi yerine, basitleştirilmiş DTO’lar tercih edilerek ağ trafiği azaltılmıştır. Ayrıca, katmanlar arası bağımlılığı minimumda tutarak, proje bakımını kolaylaştırır ve genişletilebilirliği destekler.

Projemizde DTO’lar, Entity Framework Core ile tanımlanan veri modellerinden ayrılarak, AutoMapper gibi araçlarla dönüşümleri kolaylaştırılmıştır. Bu sayede veri katmanındaki değişiklikler, API yüzeyine minimum etkide bulunacak şekilde izole edilmiştir.

Sonuç olarak, DTO kullanımı, projenin mimari sağlamlığını artıran ve veri yönetiminde kontrol sağlayan kritik bir yapı taşını oluşturmuştur.

****

1. **AutoMapper**

Projedeki veri dönüşümlerini otomatikleştirmek ve manuel eşleştirme hatalarını minimize etmek amacıyla **AutoMapper** kütüphanesi kullanılmıştır. AutoMapper, kaynak nesneler ile hedef nesneler arasındaki veri eşleştirmelerini kolaylaştıran, yapılandırılmış ve güçlü bir haritalama aracıdır.

Bu araç sayesinde, Entity Framework Core ile tanımlanan veri modelleri (Entity’ler) ile Data Transfer Object (DTO) yapıları arasındaki dönüşümler hızlı ve tutarlı şekilde gerçekleştirilmiştir. Manuel kod yazmadan, tek bir profil tanımı ile karmaşık nesne yapıları arasındaki veri akışı yönetilmiş, kod tekrarları önlenmiş ve bakım süreci kolaylaştırılmıştır.

AutoMapper’ın kullanımı, proje içinde veri katmanları arasındaki bağımlılıkları azaltırken, geliştiricilerin iş yükünü hafifletmiş ve hata olasılıklarını minimize etmiştir. Ayrıca, farklı katmanlar arasında tutarlı ve standart bir veri transferi modeli sağlanması, uygulamanın sürdürülebilirliğine katkıda bulunmuştur.

Kısacası, AutoMapper, proje genelinde veri dönüşümlerini merkezileştirerek, kod kalitesini artıran ve geliştirme sürecini hızlandıran önemli bir yardımcı araç olarak konumlanmıştır.

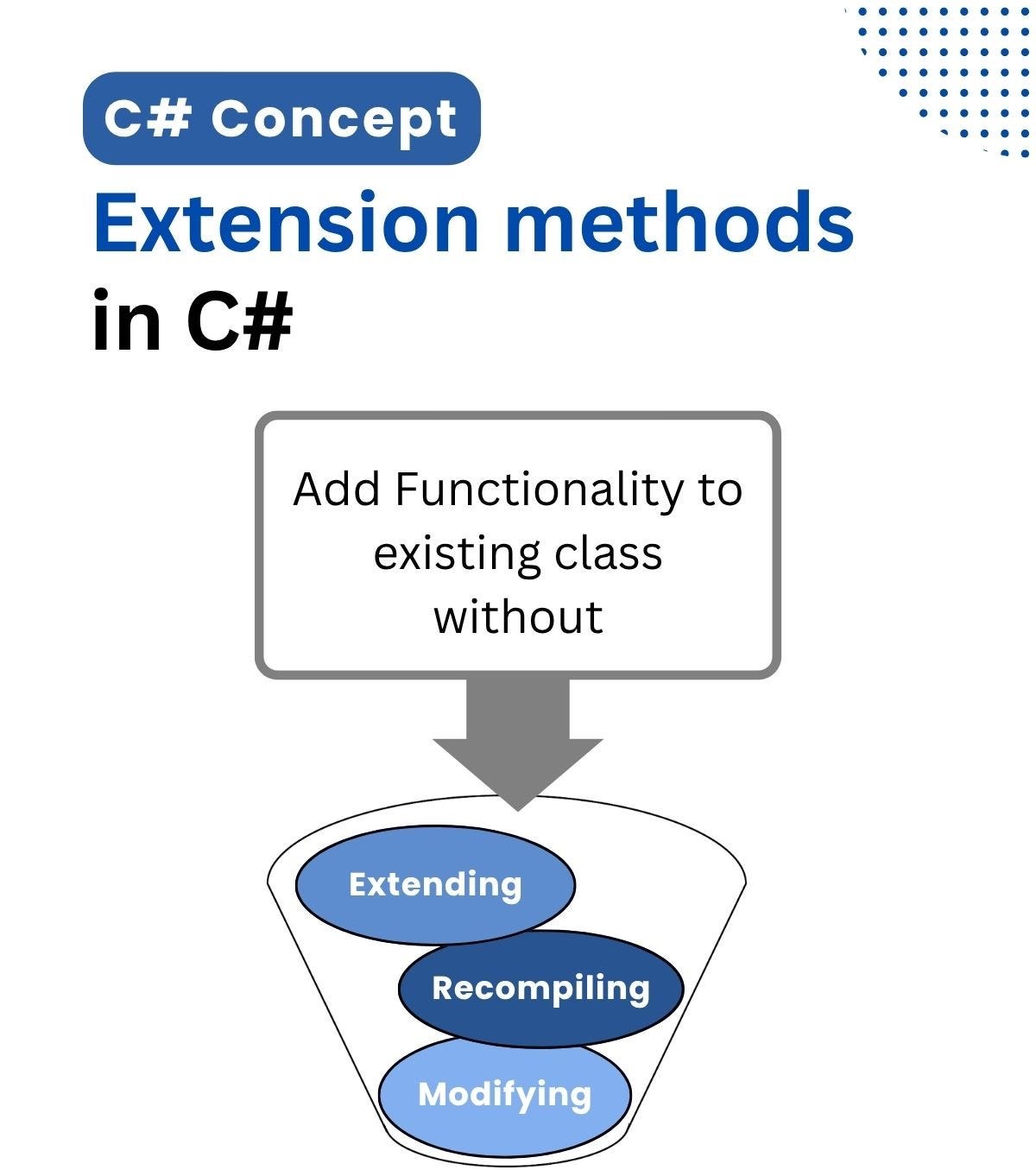
****

1. **Extension Metod**

Projede, özellikle **WebApi katmanında** yani uygulamanın kök dizininde kullanılan **extension metodlar**, kod organizasyonunu ve sürdürülebilirliğini artırmak amacıyla önemli bir rol oynamıştır. Bu metodlar, standart sınıflara ve nesnelere yeni işlevsellikler eklemeye olanak tanıyarak, program.cs dosyasındaki kod karmaşasını azaltmıştır.

Extension metodlar sayesinde, projede sıkça kullanılan yardımcı işlemler modüler bir yapıya kavuşmuş, böylece tekrarlanan kodlar tek bir yerde toplanarak kod tekrarı önlenmiş ve genel kod okunabilirliği iyileştirilmiştir. Bu yaklaşımla, proje içerisindeki yapılandırma ve konfigürasyon işlemleri hem daha derli toplu hem de yönetilebilir hale getirilmiştir.

Ayrıca, extension metodlar sayesinde, farklı katmanlarda kullanılan ortak işlevler standart bir formatta uygulanarak, geliştirme sürecinde tutarlılık sağlanmış ve yeni özelliklerin eklenmesi kolaylaştırılmıştır. Böylelikle hem bakım hem de geliştirme süreçleri için esnek ve temiz bir kod altyapısı oluşturulmuştur.

Kısaca, extension metodlar, WebApi katmanında merkezi bir yapılandırma noktası oluşturarak, projenin hem yapısal hem de işlevsel açıdan düzenli ilerlemesini mümkün kılmıştır.****

1. **SeedData**

Projede **SeedData** uygulaması, özellikle **Repositories katmanında** konumlandırılmıştır. Bu yapı, veritabanı sıfırlandığında veya yeni migration’lar uygulandığında, veritabanının otomatik olarak başlangıç verileri ile doldurulmasını sağlar. Böylece, geliştirici ve test ortamlarında hızlı bir şekilde çalışır durumdaki veri setlerine ulaşmak mümkün olur.

SeedData sayesinde, migration süreci tamamlandıktan sonra manuel veri girişi gereksinimi ortadan kalkar ve uygulamanın test aşamasına geçiş süreci ciddi anlamda kısalır. Bu yöntem, modüler ve yönetilebilir yapısıyla, farklı veri setlerinin kolayca eklenip güncellenebilmesine de imkân verir.

Böylece, projenin geliştirme sürecinde zaman tasarrufu sağlanırken, veritabanı bütünlüğü ve tutarlılığı da korunmuş olur. SeedData, özellikle otomasyon ve sürekli entegrasyon süreçlerinde, tekrar eden veri giriş işlerini minimize ederek etkin bir altyapı oluşturur.

1. **Dependency Injection**

Projede **Dependency Injection (DI),** uygulamanın gevşek bağlı ve sürdürülebilir bir mimaride geliştirilmesi için temel yapı taşlarından biri olarak kullanılmıştır. DI, nesneler arası bağımlılıkların dışarıdan sağlanmasını mümkün kılarak, sınıfların doğrudan somut bağımlılıklar yaratmasının önüne geçer. Bu sayede, test edilebilirlik artar ve modüller arası esneklik sağlanır.

ASP.NET Core’un yerleşik DI konteyneri, servislerin yaşam döngülerinin (Transient, Scoped, Singleton) kolayca yönetilmesine olanak verir. Projede, veri erişim katmanı, iş mantığı ve diğer servisler bu yapıya uygun şekilde kayıt edilmiştir. Böylece, bileşenlerin ihtiyaç duyduğu bağımlılıklar, çalışma zamanında otomatik olarak çözülerek kullanıma sunulur.

DI’nin sağladığı en önemli avantajlardan biri, kodun tekrar kullanımını artırması ve değişikliklere karşı dayanıklılığını güçlendirmesidir. Bu sayede, örneğin veritabanı sağlayıcısının değiştirilmesi veya test odaklı mock nesnelerin kullanılması gibi durumlar kolayca uygulanabilir. Ayrıca, DI mimarisi sayesinde uygulamanın farklı katmanları arasında gevşek bağlar kurulmuş olur; bu da yazılımın modülerliğini ve bakım kolaylığını artırır.

Projenin ilerleyen aşamalarında yeni servislerin eklenmesi veya mevcutların değiştirilmesi gerektiğinde, Dependency Injection altyapısı sayesinde kodda büyük çaplı değişiklikler yapmak yerine, yalnızca ilgili bağımlılıkların güncellenmesi yeterli olacaktır. Bu durum, geliştirme sürecinin hızlanmasına ve hata oranlarının azalmasına doğrudan katkı sağlar.

****

1. **FluentApi**

Entity Framework Core’da model yapılandırmalarını daha detaylı ve esnek yapmak için **Fluent API** kullanılmıştır. Fluent API, veri anotasyonlarının ötesinde, tablo ve sütun isimlendirmeleri, ilişki kuralları, kısıtlamalar ve davranışların merkezi bir şekilde tanımlanmasına olanak tanır. Bu sayede, veri tabanı şeması üzerinde tam kontrol sağlanarak projenin gereksinimlerine uygun yapılandırmalar yapılmıştır.

Projemizde, Fluent API ile tablolar arasındaki birebir, bireçok ve çokaçok ilişkiler hassas biçimde ayarlanmış; özellikle cascade delete gibi kritik davranışlar ihtiyaca göre yönetilmiştir. Ayrıca, kolon tipleri, maksimum uzunluklar ve null izinleri gibi detaylar da Fluent API kullanılarak belirlenmiştir. Bu yaklaşım, migration dosyalarının doğru ve tutarlı oluşturulmasına katkı sağlamış, veri bütünlüğünün korunmasında önemli rol oynamıştır.

Fluent API’nin sağladığı bu kapsamlı yapılandırma imkânı, kod ve veritabanı senkronizasyonunu kolaylaştırırken, projenin ilerleyen aşamalarında esnekliği ve bakım kolaylığını artırmıştır. Böylece, karmaşık veri modelleri daha güvenilir ve sürdürülebilir bir şekilde yönetilebilmiştir.

1. **Asenkron Programlama ve Hata Yönetimi**
   1. **Asenkron Kod**

Modern yazılım geliştirme süreçlerinde, özellikle web tabanlı uygulamalarda performans ve kullanıcı deneyimi kritik öneme sahiptir. Bu nedenle, **asenkron programlama** teknikleri yaygın olarak kullanılmaktadır. Projemizde de C#’ın güçlü asenkron programlama yapıları olan **async** ve **await** anahtar kelimeleri tercih edilmiştir. Asenkron kodlama, özellikle IO-bound operasyonlarda —örneğin veritabanı sorguları, dosya işlemleri veya uzak API çağrıları— ana iş parçacığının bloke olmasını engelleyerek uygulamanın tepki verme süresini iyileştirir.

Entity Framework Core ile uyumlu asenkron metotlar sayesinde, veritabanı işlemleri sırasında sistem kaynaklarının etkin kullanımı sağlanmıştır. Böylece, veri erişimi gerçekleştirilirken diğer işlemler kesintiye uğramamış, uygulamanın çoklu kullanıcı taleplerine hızlı ve stabil yanıt vermesi mümkün olmuştur. Bu durum, özellikle yüksek trafik altında çalışan API’lerin performansını artırmakta, kullanıcı deneyimini olumlu yönde etkilemektedir.

Asenkron programlamanın bir diğer önemli avantajı da uygulamanın kaynak tüketimini optimize etmesidir. Senkron kodlamada işlemler sıralı ve bloklayıcı olduğundan, kaynaklar gereksiz yere beklemekte ve verimsiz kullanılmaktadır. Async/await yapıları sayesinde uygulama, işlem tamamlanana kadar başka görevleri gerçekleştirebilir; böylece CPU ve bellek kullanımı dengelenir.

Bunun yanında, asenkron kod yazımı hata yönetimini de kolaylaştırır. Karmaşık callback zincirlerine ihtiyaç kalmadan, try-catch blokları asenkron metotlarda rahatlıkla kullanılabilir. Bu da kodun okunabilirliğini artırırken, hata ayıklama süreçlerini de sadeleştirir.Sonuç olarak, projemizde asenkron programlama tercih edilmesi, API’nin hem performansını yükseltmiş hem de sürdürülebilir ve temiz bir kod tabanı oluşturulmasına katkı sağlamıştır. Bu sayede uygulama, kullanıcı taleplerine daha hızlı yanıt verirken, aynı zamanda gelecekteki geliştirmeler için sağlam bir temel oluşturmuştur.

****

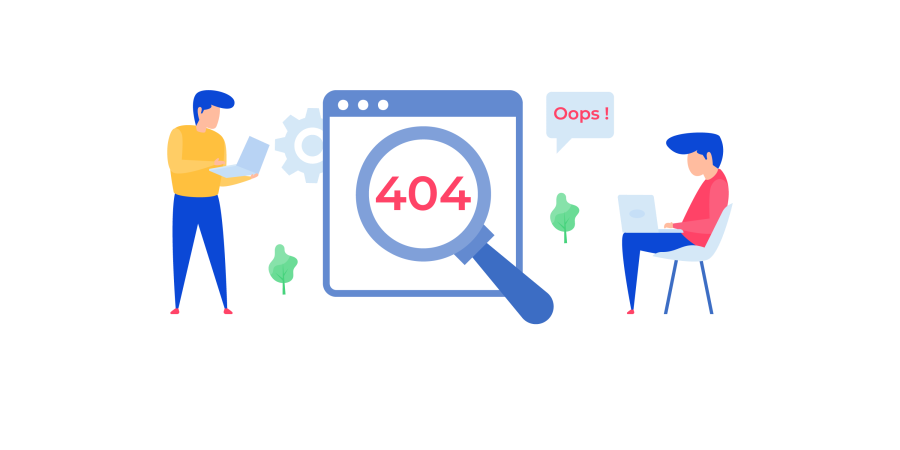
* 1. **Hata Yönetimi**

Hata yönetimi, yazılım geliştirme sürecinde uygulamanın kararlılığını ve kullanıcı deneyimini doğrudan etkileyen temel bir bileşendir. Projemizde, hata senaryolarını daha etkin ve anlamlı yönetmek amacıyla kendimize özgü özel hata sınıfları oluşturduk. Bu sınıflar, uygulamada karşılaşılan farklı hata türlerini net bir şekilde ayırt etmeye ve yönetmeye olanak tanır. Böylece, her hata durumu için spesifik bir yaklaşım geliştirilebilmekte ve kodun okunabilirliği ile bakımı kolaylaşmaktadır.

Hata meydana geldiğinde, sistem geriye standart HTTP durum kodlarının yanında, kendi tanımladığımız açıklayıcı mesajlar içeren bir yanıt gövdesi döndürür. Bu yapı sayesinde, API kullanıcıları hata hakkında daha ayrıntılı bilgi alabilir ve sorunların kaynağı daha hızlı tespit edilebilir.

Ayrıca, hata yanıtlarının tutarlı biçimde formatlanması, uygulamanın farklı bileşenlerinin ve üçüncü taraf istemcilerin hatalarla nasıl başa çıkacağını öngörülebilir kılar. Bu da hata ayıklama süreçlerini kolaylaştırmakla kalmaz, aynı zamanda uygulamanın genel güvenilirliğini artırır.

Özetle, özel hata sınıfları ve yapılandırılmış hata mesajları, projenin hata yönetim stratejisinin temel taşlarını oluşturmuş; sistemin beklenmedik durumlarla karşılaştığında bile kontrollü ve kullanıcı dostu bir şekilde yanıt vermesini sağlamıştır. Middleware yapısına ilişkin teknik detaylar ise ilerleyen bölümlerde ele alınacaktır.

****

* 1. **Validasyon**

Validasyon (doğrulama), uygulamanın veri bütünlüğünü koruması ve beklenmeyen hataların önüne geçmesi açısından vazgeçilmez bir süreçtir. Projede, gelen verilerin doğruluğunu ve tutarlılığını sağlamak için çeşitli validasyon yöntemleri uygulanmıştır. Bu sayede, istemciden gelen eksik, hatalı veya geçersiz veriler sistem tarafından erkenden tespit edilerek işlenmeden önce engellenmiştir.

Validasyon mekanizmaları, hem model bazında hem de iş mantığı katmanında kullanılmıştır. Böylece kullanıcıdan ya da dış sistemlerden gelen verilerin, tanımlı kurallara uygunluğu sağlanmıştır. Örneğin, zorunlu alanların boş bırakılmaması, veri tiplerinin ve uzunluklarının kontrolü gibi temel kurallar bu katmanda yönetilmiştir.

Ayrıca, validasyon süreçleri kullanıcıya anlamlı ve anlaşılır hata mesajları döndürecek şekilde yapılandırılmıştır. Bu yaklaşım, kullanıcı deneyimini artırırken, API’nin daha sağlam ve güvenilir çalışmasına katkı sağlamıştır. Validasyon işlemleri, gereksiz veri işleme yükünü azaltarak uygulamanın performansını da olumlu yönde etkilemiştir.

Bu sayede, sistemdeki veri kalitesi yüksek tutulmuş ve hatalı verilerin neden olabileceği problemler minimize edilmiştir. Projede kullanılan doğrulama tekniklerinin ayrıntıları ve uygulama detayları, ilerleyen bölümlerde detaylı olarak ele alınacaktır.

* 1. **İçerik Pazarlığı**

İçerik pazarlığı (Content Negotiation), API’nin istemciden gelen taleplere göre farklı formatlarda veri sunabilmesini sağlayan önemli bir özelliktir. Projemizde bu mekanizma sayesinde, varsayılan olarak JSON formatının yanı sıra, XML ve CSV gibi alternatif çıktı formatları da desteklenmiştir. Bu sayede, farklı istemci türlerine ve kullanım senaryolarına uygun esnek bir veri sunumu mümkün hale gelmiştir.

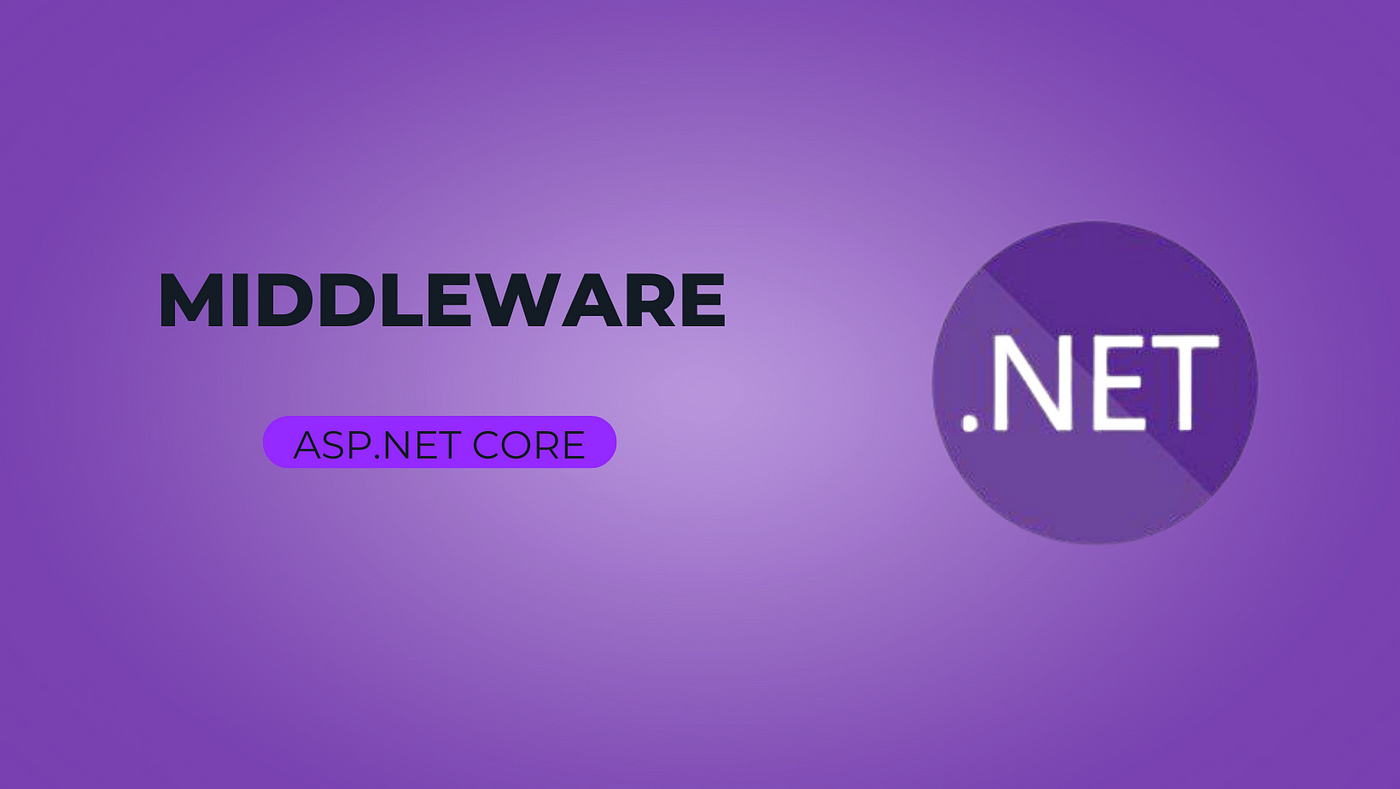
Veri girişinde ise, uygulamanın sadece JSON formatındaki verileri kabul etmesi sağlanmıştır. Böylece, istemciden gelen verilerin tutarlılığı ve standartlara uygunluğu garanti altına alınmıştır. XML formatı için yerel .NET yetenekleri kullanılırken, CSV çıktısı oluşturmak için ise popüler ve güçlü bir kütüphane olan CsvHelper tercih edilmiştir. CsvHelper, CSV verilerinin okunması ve yazılmasında kolaylık sağlayarak, proje içinde bu formatla ilgili işlemleri hızlı ve hatasız gerçekleştirmemize olanak tanımıştır.

Bu yapı, API’nin çok yönlü kullanımını artırırken, farklı uygulama ihtiyaçlarına karşılık verebilmesini sağlamıştır. Ayrıca, içerik pazarlığı sayesinde veri alışverişinde standartların dışına çıkmadan esnek ve genişletilebilir bir altyapı oluşturulmuştur.

* 1. **Custom Exception Middleware**

Projedeki hata yönetim stratejisinin çekirdeğini, **özel olarak geliştirilmiş Exception Middleware** oluşturmaktadır. Bu katman, uygulama çalışırken yakalanmamış tüm hataları merkezi biçimde ele alır ve bunları tutarlı HTTP yanıtlarına dönüştürür. Mimari açıdan bakıldığında, UseExceptionHandler uzantısı üzerinden devreye alınan bu yapı, istek–yanıt hattında ilk temas noktasına yerleşerek uygulamanın geri kalanını olası istisna zincirlerinden yalıtır.

Middleware’in temel görevi, oluşan hatayı ait olduğu özel istisna türüyle eşleştirip uygun **HTTP durum kodunu** belirlemektir. Örneğin doğrulama hataları 400 (Bad Request), bulunamayan kaynaklar 404 (Not Found) ya da veri çakışmaları 409 (Conflict) olarak geri döndürülür. Böylece istemci, hatanın niteliğini yalnızca koddan değil, aynı zamanda anlamlı bir **mesaj** içeren standart bir JSON gövdesinden de öğrenir. Yanıtlar tek tip bir “ErrorDetails” şeması kullandığı için, istemci uygulamalar hata senaryolarını öngörülebilir biçimde işleyebilir.

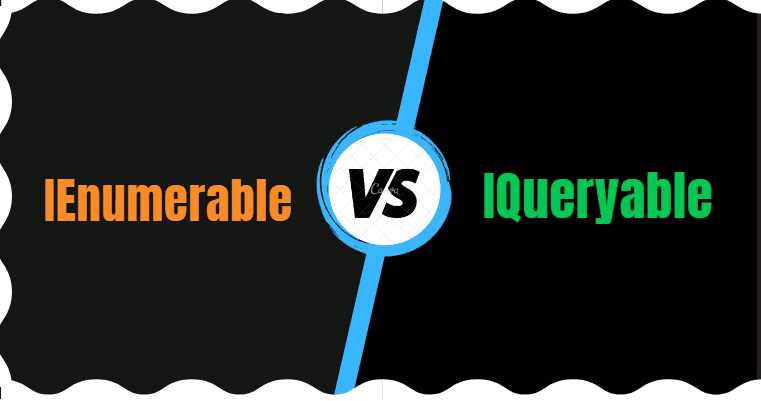
****Bu yaklaşım, farklı katmanlarda dağınık try‑catch bloklarına ihtiyaç duymadan, tüm hataları merkezi olarak yönetmeyi mümkün kılar; kod tekrarını azaltır, bakım maliyetini düşürür. Ayrıca, sınır kontrolüne dayalı bu mekanizma sayesinde loglama, izleme ve alarm sistemleri de tek noktadan beslenebilir. Sonuç olarak, özel Exception Middleware projenin dayanıklılığını ve geliştirici deneyimini ileri seviyeye taşırken, API’nin dış dünyaya karşı daima tutarlı ve anlaşılır tepkiler vermesini garanti eder.

1. **Performans, Test ve Refactoring**
2. **IEnumerable Ve IQueryable Optimizasyonu**

Veri erişiminde kullanılan koleksiyon türleri, uygulamanın performansı ve veri işleme yaklaşımı açısından büyük önem taşımaktadır. .NET platformunda yer alan IEnumerable ve IQueryable arayüzleri, veriye erişim ve işlem yapma biçimleri açısından birbirinden ayrılır. IEnumerable, belleğe alınmış veriler üzerinde işlem yaparken; IQueryable, veritabanı seviyesinde sorgular oluşturarak sadece ihtiyaç duyulan verinin çekilmesini sağlar. Bu fark, özellikle büyük veri kümelerinde performans üzerinde doğrudan etki yaratmaktadır.

Projede, bu iki yapının avantajlarını göz önünde bulundurarak dengeli bir kullanım hedeflenmiştir. Genellikle işlem sonrası tüm verinin alınması gerekiyorsa IEnumerable tercih edilmiş, ancak filtreleme, sıralama veya belirli alanların seçimi gibi durumlarda IQueryable kullanılarak sorgular veritabanı tarafında optimize edilmiştir. Bu sayede, gereksiz veri aktarımı ve bellek kullanımı önlenmiş; daha sade ve yönetilebilir kod yapıları elde edilmiştir.

Kodun daha okunabilir ve sürdürülebilir olması amacıyla, ilgili ifadeler katmanlı mimaride servis ve repository interface’lerinde tanımlanmış, somut sınıflarda ise uygun yapı tercih edilerek uygulanmıştır. Projenin büyük çoğunluğunda IEnumerable tercih edilmiş olsa da, IQueryable’ın sağladığı dinamik sorgu avantajları, performans iyileştirmesi gereken noktalarda etkili biçimde değerlendirilmiştir.

Bu iki yapının bilinçli ve yerinde kullanımı, yazılım geliştirme sürecinde sadece performansa değil, aynı zamanda kodun modülerliği ve test edilebilirliğine de önemli katkı sağlamıştır.

1. **TrackChanges**

Entity Framework Core’un sunduğu ChangeTracker mekanizması, veri üzerinde yapılan değişikliklerin izlenebilmesini sağlar. TrackChanges ifadesi bu mekanizma ile doğrudan ilişkilidir ve EF Core’un bir varlığın durumunu izleyip izlemediğini belirlemesine yardımcı olur. Bu özellik, özellikle güncelleme ve silme gibi işlemlerde, hangi alanların değiştirildiğini tespit etmek ve veritabanı ile senkron çalışmak açısından kritik rol oynar.

Projede bu özelliğin yönetimini daha kontrollü hale getirmek amacıyla, sorguların izleme davranışı ihtiyaçlara göre açık ya da kapalı olarak belirlenmiştir. Varsayılan olarak EF Core tüm varlıkları izler, ancak bazı senaryolarda bu izleme davranışı gereksiz yere performans maliyeti doğurabilir. Bu nedenle, sadece güncelleme gibi işlemlerde trackChanges: true değeri verilmiş, yalnızca veri okuma amaçlı işlemlerde ise trackChanges: false kullanılarak performans optimizasyonu sağlanmıştır.

Bu yapı repository metodlarının tasarımına da yansıtılmıştır. Generic repository üzerinden gelen metotlara trackChanges adında bir parametre eklenmiş, bu parametre üzerinden AsNoTracking() gibi EF Core fonksiyonlarıyla izleme davranışı dinamik olarak kontrol edilmiştir. Böylece hem veri tutarlılığı hem de sistem kaynaklarının verimli kullanımı açısından dengeli bir yapı kurulmuştur.

****TrackChanges yaklaşımı, kodun modülerliğini artırırken aynı zamanda yazılımın veri erişim katmanında daha esnek ve genişletilebilir bir yapı kurulmasına da olanak sağlamıştır. Özellikle ileride loglama, denetim (audit) veya soft delete gibi gelişmiş özellikler entegre edilmek istendiğinde bu yapı hazır bir zemin sunmaktadır.

1. **Refactoring Ve Api Testi**

Yazılım geliştirme sürecinde, belirli bir noktada kodun genel yapısının yeniden gözden geçirilmesi ve iyileştirilmesi kaçınılmaz hale gelir. Bu proje özelinde de tam olarak böyle bir dönüm noktasına gelinmişti. Altyapı çalışmaları büyük oranda tamamlanmış, Entity Framework, DTO yapısı, AutoMapper, Extension metodlar, hata yönetimi ve içerik pazarlığı gibi önemli yapı taşları stabil hale getirilmişti. Ancak bu bileşenlerin tamamı, uygulamanın daha çok arka planını yani altyapısını oluşturmaktaydı. Bundan sonraki adımlar ise API’nin kullanıcıya dönük, fonksiyonel yönlerini içerecek şekilde ilerleyecekti. Bu sebeple, Refactoring süreci hem geçmişin bir özeti hem de geleceğe hazırlanmanın önemli bir adımı haline geldi.

Refactoring aşamasında öncelikli olarak kod tekrarları giderilmiş, servis ve repository katmanlarındaki metotlar sadeleştirilmiş ve gereksiz bağımlılıklar kaldırılmıştır. Ayrıca, isimlendirme standartları gözden geçirilerek daha okunabilir ve sürdürülebilir bir yapı sağlanmıştır. Katmanlar arası iletişimin açık ve net olması, özellikle genişlemeye açık bir mimari için büyük önem taşımaktadır. Bu doğrultuda arayüzlerin doğru kullanımı, bağımlılıkların gevşek bağlanması ve sorumlulukların net olarak ayrılması prensiplerine bağlı kalınmıştır.

Kodun yeniden yapılandırılmasıyla eş zamanlı olarak, sistematik API testleri de yürütülmüştür. Daha önce Postman üzerinde oluşturulmuş olan koleksiyonlar ve senaryolar bu aşamada büyük önem kazanmıştır. Her endpoint’in stabil çalıştığı, beklenen veri türlerini döndürdüğü ve uygun hata yönetimini sağladığı manuel olarak test edilmiştir. Bu sayede, hem refactor işleminin yan etkileri anında fark edilmiş hem de sistemin genel dayanıklılığı test edilmiştir.

Refactoring ve test süreci, projenin altyapı ile fonksiyonellik arasında bir geçiş aşaması olarak konumlanmıştır. Bu yönüyle yalnızca teknik bir düzeltme değil, aynı zamanda yazılımın olgunlaşma evresine geçişi olarak da değerlendirilebilir. Bu aşamada elde edilen kazanımlar, ilerleyen süreçlerde API’nin işlevsel gücünü artıran sağlam bir temel oluşturmuştur.

1. **Gelişmiş API Özellikleri**
2. **Action Filters**

ASP.NET Core'da **Action Filter** yapıları, controller içindeki işlemler çalışmadan önce veya sonra devreye girerek belirli kontrolleri yapmamıza olanak tanır. Bu sayede tekrar eden kontrolleri her method içinde yazmak yerine merkezi ve yeniden kullanılabilir şekilde yönetebiliriz.

Bu projede **Presentation** katmanında iki farklı Action Filter kullanılmıştır:

1. **Content-Type İfadesnin Kontrolünü Gerçekleştiren ActionFilter ifadesi ve İşlevi:**

Bu filter, gelen isteklerin application/json formatında olup olmadığını kontrol ediyor. Eğer istek farklı bir formatta gelirse işlem yapılmadan engelleniyor. Özellikle PUT ve POST işlemlerinde kullanarak, sadece beklenen veri formatında isteklerin işlenmesi sağlanmıştır.

1. **Model Geçerliliğini Doğrulayan ActionFilter ifadesi ve İşlevi:**

Bu yapı ise gelen verilerin model kurallarına uyup uymadığını kontrol ediyor. ModelState geçerli değilse işlem daha başlamadan durduruluyor ve uygun bir hata mesajı dönülüyor. Böylece her method içinde ayrıca doğrulama kodu yazmak zorunda kalmıyorum.

Bu Action Filter yapıları sayesinde hem kod tekrarından kurtuldum hem de API’ye gelen isteklerin daha kontrollü şekilde işlenmesini sağlanıp, ayrıca ileride yeni kurallar eklenmek istendiğinde sadece filter'lara müdahale etmek yeterli olacaktır

1. **Sayfalama**

Modern API geliştirme süreçlerinde veri kümesinin kontrolsüz şekilde istemciye gönderilmesi, hem performans hem de kullanıcı deneyimi açısından çeşitli problemler doğurabilir. Bu bağlamda sayfalama (pagination) işlemi, büyük veri kümelerinin daha küçük parçalara bölünerek sunulmasını sağlar. Projede bu ihtiyacı karşılamak amacıyla özel olarak RequestParameters, PagedList ve MetaData gibi sınıflar tanımlanmıştır. Bu sınıflar aracılığıyla hem istemciden gelen sayfalama parametreleri alınmakta hem de kullanıcıya sayfa numarası, toplam sayfa sayısı, sayfa başına kayıt sayısı gibi bilgilerle zenginleştirilmiş bir veri yapısı sunulmaktadır.

Sayfalama işlemi, özellikle veri listeleme işlemlerinde performansı artırmak ve veri aktarımını kontrol altında tutmak adına kritik bir rol üstlenmiştir. Ayrıca, bu yapı sadece verinin kesilmesini değil; aynı zamanda istemcinin hangi sayfada olduğunu ve toplam kaç kayıt bulunduğunu anlamasını sağlayarak kullanıcıya daha anlamlı bir çıktı sunmaktadır. Yapının modüler olarak kurgulanması sayesinde farklı veri kümelerine kolaylıkla entegre edilebilmekte ve yeniden kullanılabilirliği artırılmaktadır.

metin, ekran görüntüsü, meneviş mavisi, yazı tipi içeren bir resim

Yapay zeka tarafından oluşturulmuş içerik yanlış olabilir.

1. **Filtreleme**

API geliştirme sürecinde kullanıcıların sadece belirli kriterlere uyan verilere erişebilmesi büyük bir esneklik ve kullanım kolaylığı sağlar. Bu ihtiyaç doğrultusunda projede filtreleme özelliği aktif hale getirilmiştir. Filtreleme işlemleri, özellikle veri yoğunluğu olan yapılarda kullanıcının işine yarayacak sonuçlara daha kısa sürede ulaşabilmesini sağlamaktadır.

Proje kapsamında filtreleme işlemleri; sorguya dâhil edilen parametreler aracılığıyla belirli koşullara uyan verilerin seçilmesi prensibine dayanmaktadır. Örneğin, belirli bir departmana veya pozisyona ait çalışanların listelenmesi gibi işlemler, filtreleme mantığıyla gerçekleştirilmektedir. Bu filtreleme altyapısı, hem kodun okunabilirliğini artırmakta hem de sorguların modüler olarak genişletilebilmesine imkân tanımaktadır.

Bu özellik sayesinde kullanıcı tarafında performanslı ve amaca yönelik sorgular oluşturulabilmiş, aynı zamanda gereksiz veri aktarımının önüne geçilerek daha verimli bir API deneyimi sunulmuştur.

metin, ekran görüntüsü, yazı tipi, diyagram içeren bir resim

Yapay zeka tarafından oluşturulmuş içerik yanlış olabilir.

1. **Arama**

Projede arama fonksiyonu, kullanıcıların geniş veri kümeleri içinde ihtiyaç duydukları bilgilere hızlı ve hedefli bir şekilde ulaşmalarını sağlamak amacıyla kritik bir özellik olarak tasarlanmıştır. Arama mekanizması, API üzerinden gönderilen sorgu parametrelerine bağlı olarak, belirlenen alanlarda—örneğin, çalışan adı, departman ismi, pozisyon veya diğer metin tabanlı alanlar—etkin bir filtreleme gerçekleştirir. Bu sayede kullanıcılar, sadece ilgilendikleri verileri görüntüleyebilir ve gereksiz bilgi kalabalığından uzak durabilirler.

Teknik açıdan arama işlemi, veritabanı sorguları seviyesinde uygulanarak performans açısından optimize edilmiştir. Filtreleme ve koşullu sorgular, veritabanının sunduğu indeksleme ve arama yeteneklerinden faydalanacak şekilde yapılandırılmıştır. Böylece, gereksiz veri transferi minimize edilmiş, API cevabı daha hızlı ve sistem kaynak kullanımı daha verimli hale getirilmiştir. Bu yaklaşım özellikle büyük veri setlerinin işlendiği projelerde kullanıcı deneyimini doğrudan olumlu etkiler.

Arama özellikleri, aynı zamanda diğer gelişmiş API işlevleri ile birlikte çalışacak biçimde tasarlanmıştır. Örneğin, arama sonuçları sayfalama, sıralama ve veri şekillendirme gibi özelliklerle kombine edilerek kullanıcılara daha düzenli ve kullanışlı bir veri sunumu sağlanmıştır. Böylece kullanıcılar, arama sonuçları içinde kolayca gezinebilir, istedikleri sıraya göre listeleyebilir ve ihtiyaç duydukları veri alanlarına odaklanabilirler.

Sonuç olarak, projedeki arama fonksiyonelliği hem kullanıcı memnuniyetini artırmak hem de sistem performansını en üst düzeye çıkarmak için kritik bir rol oynar. Bu sayede API, esnek, hızlı ve verimli bir şekilde kullanıcı taleplerine yanıt verirken, projede veri yönetimi ve erişimi konusunda önemli bir altyapı sağlamaktadır.

1. **Sıralama**

Projede sıralama fonksiyonu, kullanıcıların API üzerinden dönen veri setlerini belirli alanlara göre artan ya da azalan şekilde düzenlemelerine olanak tanır. Bu özellik, özellikle büyük veri listelerinde kullanıcıların istedikleri kriterlere göre verileri kolayca organize edebilmesini sağlar. Örneğin, çalışanların maaşına, işe başlama tarihine veya departman adına göre sıralama yapılabilir.

Teknik olarak sıralama işlemi, sorgu aşamasında dinamik olarak uygulanır. Kullanıcıdan gelen parametreler doğrultusunda, LINQ sorguları içinde sıralama kriteri belirlenir ve veritabanı sorgusuna yansıtılır. Böylece tüm sıralama işlemi, veri transferinden önce gerçekleşir; bu da hem performans açısından büyük avantaj sağlar hem de gereksiz veri işlemlerini engeller.

Sıralama özelliği, projenin diğer gelişmiş API fonksiyonlarıyla (sayfalama, filtreleme, arama) entegre çalışacak şekilde tasarlanmıştır. Bu sayede, kullanıcılar arama sonuçlarını sıralayabilir veya sayfalama yapılan listelerde istedikleri sıraya göre veri akışını kontrol edebilirler. Ayrıca, sıralama kriterlerinin esnek olması, API’nin farklı kullanım senaryolarına uyum sağlamasına katkıda bulunur.

Sonuç olarak, sıralama fonksiyonu kullanıcı deneyimini iyileştiren ve veri yönetimini kolaylaştıran önemli bir yapı taşıdır. Projede bu fonksiyonun doğru ve performanslı şekilde uygulanması, API’nin profesyonel ve kullanıcı dostu olmasına büyük katkı sağlar.

1. **Veri Şekillendirme**

Veri şekillendirme (data shaping), API’nin istemcilere sadece ihtiyaç duyduğu alanları sunmasını sağlayan oldukça verimli bir tekniktir. Bu yaklaşım, hem veri aktarımını optimize eder hem de istemci tarafında veri işleme yükünü azaltır. Projemde veri şekillendirme özelliğini uygulayarak, kullanıcıların sadece istedikleri alanları alabilmelerine olanak tanıdım. Örneğin, bir çalışan listesi istenirken yalnızca Ad, Soyad, Maaş gibi birkaç alanın döndürülmesi sağlanabilir.

Teknik olarak veri şekillendirme, gelen HTTP isteğindeki fields parametresi aracılığıyla gerçekleştirilmiştir. Kullanıcı bu parametreyi kullanarak hangi alanları almak istediğini belirtir. Ardından, bu alanlar ExpandoObject gibi dinamik veri yapıları ile işlenir. Bu sayede, gelen veri sabit bir model üzerinden değil, kullanıcının talep ettiği alanlara göre şekillenir. Böylece API çıktısı esnek hale gelir.

Bu yöntem sadece performans açısından değil, aynı zamanda veri güvenliği açısından da avantaj sağlar. Gereksiz alanların istemciye sunulmaması, hassas verilerin dışarı sızmasını önlemeye yardımcı olur. Ayrıca, veri şekillendirme işlemi projenin diğer gelişmiş özellikleri (sayfalama, filtreleme, sıralama) ile entegre çalışacak şekilde yapılandırılmıştır.

Sonuç olarak, veri şekillendirme özelliği, API’nin hem esnekliğini hem de profesyonelliğini artıran önemli bir unsurdur. Kullanıcıya sadece ihtiyacı olan veriyi sunmak, modern API tasarımının en temel prensiplerinden biri olarak projemde de başarıyla uygulanmıştır.

1. **Cors**

CORS, farklı kaynaklar (origin) arasında veri alışverişine izin vermeyi ya da bunu sınırlandırmayı amaçlayan bir güvenlik mekanizmasıdır. Projede bu yapı, özellikle istemci ile sunucu farklı adreslerde konumlandığında devreye giren erişim kısıtlamalarının önüne geçmek için yapılandırılmıştır.

CORS konfigürasyonu sayesinde sadece izin verilen kökenlerden (origin) gelen istekler kabul edilmiştir. Bu sayede hem güvenlik sağlanmış hem de dışarıdan gelen uygunsuz erişim talepleri engellenmiştir. Uygulamada kullanılan HTTP yöntemleri (GET, POST vb.) ve özel başlıklar açıkça belirtilmiş, sadece belirli domain’lere açık olacak şekilde yapılandırılmıştır.

Ayrıca CORS’un sunduğu en büyük katkılardan biri, API'nin gelişmiş fonksiyonel özelliklerinden biri olan **sayfalama, filtreleme, arama ve sıralama** gibi işlemler sonucunda oluşan özel header bilgilerinin (örneğin toplam sayfa sayısı, mevcut sayfa numarası gibi meta veriler) istemciye başarılı bir şekilde ulaşmasını sağlamasıdır. Bu bilgiler, response header içinde taşındığı için CORS üzerinden gerekli yapılandırmalar yapılmadığında tarayıcı tarafından engellenebilirdi. Ancak doğru yapılandırma ile bu sorun önlenmiş ve istemci tarafında bu meta bilgilere sorunsuz bir şekilde erişilmiştir.

Bu yönüyle CORS sadece temel erişim kontrolü sağlamakla kalmamış, aynı zamanda API’nin gelişmiş yanıtlarını destekleyen önemli bir yapı taşı hâline gelmiştir.

**metin, ekran görüntüsü, yazı tipi, tasarım içeren bir resim

Yapay zeka tarafından oluşturulmuş içerik yanlış olabilir.**

1. **Güvenlik ve Dokümantasyon**
2. **Options ve Head**

Web API geliştirirken sadece GET, POST gibi ana HTTP metodlarını desteklemek yeterli değildir. Modern istemciler, özellikle tarayıcı tabanlı uygulamalar, zaman zaman farklı HTTP metodlarına da ihtiyaç duyar. Bu noktada **OPTIONS** ve **HEAD** metodlarının desteklenmesi önem kazanır.

**OPTIONS** metodu, bir kaynağa yapılabilecek istek türlerini belirtmek amacıyla kullanılır. Bu, özellikle CORS ön uç isteklerinde (preflight request) kritik bir rol oynar. Sunucu, istemciden gelen "Ben bu kaynağa şu metodlarla erişebilir miyim?" sorusuna cevap verir. Böylece uygulama güvenliğini bozmadan kontrollü bir erişim sağlanır.

**HEAD** metodu ise, belirli bir isteğin sadece başlık bilgilerini (header) almayı sağlar. Veri içeriği gönderilmez. Bu, genellikle istemcinin bir kaynağın var olup olmadığını kontrol etmesi veya yanıtın büyüklüğünü öğrenmesi gereken senaryolarda kullanılır.

Projede bu metodların desteklenmesi, API'nin HTTP protokolüne uygun, esnek ve modern istemcilerle uyumlu hâle gelmesine katkı sağlar. Ayrıca ilerleyen aşamalarda yapılabilecek güvenlik ve performans iyileştirmeleri için sağlam bir temel oluşturur.

1. **Jwt ile Rol Bazlı Güvenlik**

Bir Web API geliştirilirken en temel önceliklerden biri de uygulamanın **yetkisiz erişimlere karşı korunması**dır. Özellikle çok kullanıcılı sistemlerde, kullanıcıların uygulama içerisindeki yetki seviyelerine göre farklı erişim haklarına sahip olmaları beklenir. Bu tür durumlar için **Rol Bazlı Yetkilendirme,** güvenliğin temel taşlarından birini oluşturur. Projede bu güvenlik yapısını sağlamak için **JWT (JSON Web Token)** tabanlı bir kimlik doğrulama ve yetkilendirme sistemi kullanıldı.

JWT, kullanıcının kimliğini doğruladıktan sonra bir token (jeton) üretir ve bu token, kullanıcının sisteme erişim sağlayabilmesi için her istekte gönderilir. Bu token’lar şifrelenmiş ve içerisine kullanıcının bilgileri (örneğin kullanıcı adı, kimlik numarası, roller gibi) gömülü olduğu için, ekstra bir kimlik denetimi ihtiyacını ortadan kaldırır. Sistem, gelen her istekte token’ı çözümleyerek kullanıcının kimliğini ve rolünü doğrular.

Bu proje kapsamında **Admin**, **Manager** ve **User** olmak üzere üç temel rol tanımı yapıldı:

* **Admin rolü**, sistemin tüm operasyonlarına erişim sağlayabilecek en yetkili kullanıcıları temsil eder. Yeni kullanıcılar tanımlayabilir, var olanları silebilir veya güncelleyebilir. Sistemin genel ayarlarında değişiklik yapma ve diğer tüm kaynaklara müdahale etme yetkisine sahiptir.
* **Manager rolü**, daha çok operasyonel süreçleri yöneten, kullanıcıları görüntüleyip bazılarını kısıtlı ölçüde güncelleyebilen bir ara yetki seviyesidir.
* **User rolü** ise, sistemin en düşük yetki seviyesindeki rolüdür. Genellikle kendi verilerini görüntüleme ve sınırlı sayıda işlem yapma hakkına sahiptir.

grafik, yazı tipi, grafik tasarım, logo içeren bir resim

Yapay zeka tarafından oluşturulmuş içerik yanlış olabilir.

Rol bazlı erişim kontrolü, controller seviyesinde [Authorize(Roles = "Admin")] gibi attribute’lar aracılığıyla gerçekleştirildi. Bu sayede hangi endpoint’in hangi roller tarafından kullanılabileceği açıkça belirtildi. Ayrıca, uygulama geliştirilirken sadece doğrulama değil, aynı zamanda **yetkilendirme** de göz önünde bulunduruldu. Yani bir kullanıcının token’ı geçerli olsa bile, erişmek istediği kaynak için uygun role sahip değilse erişim reddedildi.

Token üretimi sırasında roller, token içeriğine (claims) dahil edilerek her istekle birlikte sunucuya gönderildi. Bu roller token’dan okunarak middleware aracılığıyla denetlendi. Bu yapı sayesinde merkezi bir güvenlik mekanizması kurulmuş oldu.

metin, ekran görüntüsü, logo, grafik içeren bir resim

Yapay zeka tarafından oluşturulmuş içerik yanlış olabilir.JWT ile rol bazlı güvenlik yapısı, uygulamanın güvenliğini artırmakla kalmadı; aynı zamanda kodun okunabilirliğini ve bakımını da kolaylaştırdı. Roller sayesinde farklı kullanıcı tiplerine özel işlevler rahatlıkla geliştirildi ve erişim sınırları net olarak çizildi. Bu güvenlik modeli aynı zamanda gelecekte sistemin büyümesi durumunda yeni roller eklemeyi veya var olan rollerin yetkilerini yeniden tanımlamayı da kolaylaştıracak şekilde esnek bir yapıda kurgulandı.

1. **Swagger Genişletilmesi**

Web API projesinde, geliştirilen uç noktaların dokümantasyonu ve test edilmesini kolaylaştırmak amacıyla Swagger aracı projeye entegre edilmiştir. Ancak sadece temel Swagger kurulumu ile yetinilmemiş, yapı güvenlik ve açıklama özellikleriyle de genişletilmiştir.

Swagger konfigürasyonu SwaggerServiceExtension sınıfı aracılığıyla yapılmış ve proje içerisine merkezi bir servis olarak eklenmiştir. Bu sınıf içerisinde, Swagger’ın genel başlık bilgileri (Title, Version, Description) tanımlanmış; böylece Swagger arayüzüne gelen kullanıcılar API hakkında ilk bakışta bilgi sahibi olabilmiştir.

JWT tabanlı güvenlik sistemi kullanıldığı için Swagger, bu yapıyı destekleyecek şekilde özelleştirilmiştir. AddSecurityDefinition ve AddSecurityRequirement metotlarıyla “Bearer” tipinde bir yetkilendirme mekanizması arayüze entegre edilmiştir. Bu sayede kullanıcılar, Swagger UI üzerinden token girerek yetki gerektiren işlemleri test edebilir hale gelmiştir.

Yapılan bu geliştirmeler Swagger'ı sadece bir dökümantasyon aracı olmanın ötesine taşımış; aynı zamanda güvenli test ortamı ve geliştirici dostu bir arayüz haline getirmiştir. Farklı roller (Admin, Manager, User) için erişim testleri kolayca yapılmış, uç nokta denemeleri sade ve güvenli şekilde gerçekleştirilmiştir.

Sonuç olarak Swagger, bu proje özelinde sadece görsel bir rehber değil, aynı zamanda işlevsel bir geliştirme ve test aracına dönüştürülmüştür.

**yazı tipi, grafik, logo, kırpıntı çizim içeren bir resim

Yapay zeka tarafından oluşturulmuş içerik yanlış olabilir.**

1. **Postman Dökümantasyonu**

Projenin test süreçlerini daha verimli ve sistematik bir şekilde yürütmek amacıyla Postman aracı kapsamlı şekilde kullanılmıştır. Uygulama içerisindeki tüm uç noktalar için Postman üzerinde bir koleksiyon oluşturulmuş; bu koleksiyon, API’nin dış dünyaya sunumu ve test edilmesi açısından önemli bir araç haline gelmiştir.

Özellikle **POST** ve **PUT** işlemleri için test sürecini hızlandırmak adına **test scriptleri** geliştirilmiştir. Bu scriptler sayesinde, yapılan işlemlerin sonuçları otomatik olarak kontrol edilerek doğrulama sağlanmış ve manuel test ihtiyacı en aza indirilmiştir. Aynı zamanda hatalı isteklerin veya eksik verilerin etkili biçimde tespiti mümkün olmuştur.

Her bir istek için detaylı **açıklama metinleri** eklenmiş, böylece koleksiyonu kullanan geliştiricilerin veya test ekiplerinin ilgili isteğin ne amaçla kullanıldığını hızlıca anlayabilmesi sağlanmıştır. Bu açıklamalar, uç noktanın örnek kullanımı, gerekli header bilgileri ve gövde yapısını açık bir şekilde ifade etmiştir.

metin, logo, yazı tipi, grafik içeren bir resim

Yapay zeka tarafından oluşturulmuş içerik yanlış olabilir.Genel olarak Postman kullanımı, API'nin sürdürülebilirliği ve dış kullanıcılarla paylaşılabilirliği açısından önemli bir rol oynamış; testlerin modüler, tekrar edilebilir ve anlaşılır biçimde yürütülmesini mümkün kılmıştır.

1. **SONUÇ VE DEĞERLENDİRME**

Bu çalışma, katmanlı mimari prensiplerine uygun olarak geliştirilen bir Web API projesinin baştan sona planlanması, inşa edilmesi ve yapılandırılması sürecini kapsamlı bir şekilde ele almıştır. Geliştirme süreci yalnızca teknik gerekliliklerin karşılanmasıyla sınırlı kalmamış; yazılım geliştirme süreçlerinde önemli olan sürdürülebilirlik, okunabilirlik, test edilebilirlik ve güvenlik gibi temel yazılım kalitesi kriterlerine de özel olarak odaklanılmıştır.

Proje boyunca önce altyapı çalışmaları tamamlanmış, ardından fonksiyonel özelliklerin devreye alınmasıyla API daha yetkin hale getirilmiştir. Bu iki aşama arasında yapılan refactoring, izlenen yazılım geliştirme metodolojisinin sağlıklı ve adım adım ilerlediğini göstermektedir. Geliştirilen katmanlı yapı ile bağımlılıklar azaltılmış, esneklik ve yönetilebilirlik artırılmıştır.

IEnumerable ve IQueryable gibi yapılar bilinçli şekilde kullanılarak performans ve kaynak yönetimi gözetilmiş, TrackChanges ile veritabanı erişimi üzerinde kontrol artırılmış, action filter yapıları sayesinde ise tekrar eden kontroller merkezi hale getirilmiştir. Sayfalama, filtreleme, arama, sıralama ve veri şekillendirme gibi gelişmiş API fonksiyonları, uç noktaların esnekliğini ve kullanıcı deneyimini güçlendirmiştir. Ayrıca, CORS yapılandırması sayesinde bu fonksiyonlara ait özel header bilgileri güvenli biçimde istemciye ulaşabilmiştir.

Projede güvenlik açısından da önemli adımlar atılmış; rol bazlı yetkilendirme, token yapısı ve Swagger üzerinden yetkili test ortamı sağlanmıştır. Swagger ve Postman gibi araçlarla geliştirilen API’nin dış dünyaya açık, anlaşılır ve test edilebilir olması sağlanmış; bu araçların genişletilmesi ile geliştirici ve test ekiplerine kullanım kolaylığı sunulmuştur.

Sonuç olarak bu proje, yalnızca belirli işlevleri yerine getiren bir API olmaktan öte, modern yazılım geliştirme prensiplerine uygun olarak tasarlanmış, sürdürülebilir ve genişletilebilir bir sistem örneği sunmaktadır. Geliştirilen yapı, ilerleyen süreçlerde yeni özelliklerin kolayca eklenmesine olanak tanımakta ve gerçek dünyadaki kurumsal projelere uygulanabilirliğiyle öne çıkmaktadır.

1. **KAYNAKÇA**

Logo-URL-1 : <https://commons.wikimedia.org/wiki/File:Logo_C_sharp.svg>

Logo-URL-2 : <https://www.techmeet360.com/blog/http-repl/asp-net-core-logo/>

Logo-URL-3 : <https://github.com/devicons/devicon/issues/620>

Logo-URL-4 : <https://logos-marques.com/mysql-logo/>

Logo-URL-5 : <https://www.linkedin.com/pulse/swagger-enhancing-api-development-documentation-rizwana-khan-zvjxe/>

Logo-URL-6 : <https://logowik.com/postman-api-platform-logo-vector-svg-pdf-ai-eps-cdr-free-download-19638.html>

Logo-URL-7 : <https://mustafa-keser.medium.com/data-transfer-object-dto-nedir-ve-nas%C4%B1l-kullan%C4%B1l%C4%B1r-1cb004dc1975>

Logo-URL-8 : <https://tech.playgokids.com/auto-mapper-net6/>

Logo-URL-9 : <https://medium.com/@shreyans_padmani/extension-methods-in-c-e297108461c1>

Logo-URL-10:<https://www.linkedin.com/pulse/implementation-dependency-injection-pattern-c-nadim-attar-alk1f/>

Logo-URL-11 : <https://xuanthulab.net/ef-core-tao-quan-he-trong-entity-framework-voi-fluent-api-c-csharp.html>

Logo-URL-12 : <https://www.bytehide.com/blog/async-await-csharp>

Logo-URL-13 : <https://clearinsights.io/blog/helpful-guide-to-exception-handling-in-vb-net/>

Logo-URL-14 : <https://didourebai.medium.com/use-csv-helper-in-net-core-4053b971ea75>

Logo-URL-15 : <https://medium.com/@marcio.kgr/middleware-asp-net-core-c-a2ef8e20dc55>

Logo-URL-16 : <https://medium.com/@kamlesh90/ienumerable-vs-iqueryable-f7e56551d864>

Logo-URL-17 : <https://medium.com/@anton_tomchuk/ef-core-tracking-traps-you-should-be-aware-of-24bc75e465d7>

Logo-URL-18 : <https://medium.com/@talhaawan78654321/pagination-in-asp-net-core-with-best-practices-a-complete-guide-for-developers-with-free-code-24293ca055ed>

Logo-URL-19 : <https://wirawat-roj.medium.com/%E0%B8%A1%E0%B8%B2%E0%B8%A3%E0%B8%B9%E0%B9%89%E0%B8%88%E0%B8%B1%E0%B8%81-filters-%E0%B9%83%E0%B8%99-asp-net-core-%E0%B8%81%E0%B8%B1%E0%B8%99-34a935db400f>

Logo-URL-20 : <https://emrebener.medium.com/configuring-cors-in-asp-net-core-e985a9b4c0a1>

Logo-URL-21 : <https://seeklogo.com/vector-logo/428033/jwt>

Logo-URL-22 : <https://burakneis.com/asp-net-core-identity/>

1. **EKLER**

|  |  |
| --- | --- |
|  | **İSTANBUL AYDIN ÜNİVERSİTESİ**  **ANADOLU BİL MESLEK YÜKSEKOKULU**  BİLGİSAYAR PROGRAMCILIĞI PROGRAMI  MEZUNİYET PROJESİ BAŞVURU FORMU |

|  |  |
| --- | --- |
| **Proje Adı** | CoreForce.NTierRestAPi |
| **Amaç** | Bu projenin temel amacı, N-Katmanlı mimari (N-Tier Architecture) kullanarak modüler, sürdürülebilir ve geliştirilebilir bir RESTful API altyapısı oluşturmak ve bu yapıyı C# ve ASP.NET Core teknolojileriyle uygulamaktır. Kodların sorumluluklarına göre katmanlara ayrıldığı bu mimari yaklaşım, büyük ölçekli projelerde kodun yönetilmesini kolaylaştırır. Projede modern yazılım geliştirme prensipleri olan SOLID ilkelerine, temiz kod anlayışına ve ayrık yapıların kullanımına özen gösterilmiştir. Ayrıca proje kapsamında, hata yönetimi, model doğrulama, içerik pazarlığı ve sayfalama gibi gelişmiş API özelliklerine de yer verilmiştir. Bu çalışma aynı zamanda yazılım geliştirme sürecinde kurumsal projelere temel teşkil edecek bir altyapı sunmayı hedeflemektedir. |
| **Yöntem** | - ASP.NET Core Web API  - Entity Framework Core  - NTier Architecture  - MySQL  - C# |
| **Mezuniyet Projesi Özeti** | Bu mezuniyet projesi kapsamında geliştirilen CoreForce.NTierRestApi, N-Katmanlı mimariye uygun şekilde oluşturulmuş, genişletilebilir bir RESTful API örneğidir. Proje, yazılım mimarisi bakımından katmanların ayrılması ve sorumlulukların dağıtılması prensiplerine göre tasarlanmıştır. Veri erişimi, iş kuralları ve sunum katmanı birbirinden ayrılarak, projenin okunabilirliği ve yönetilebilirliği artırılmıştır. Veri işlemleri Entity Framework Core ile gerçekleştirilmiş, veri iletiminde ise DTO yapıları ve AutoMapper kullanılmıştır. API üzerinden yapılan işlemlerde; veri doğrulama, hata mesajlarının özelleştirilmesi, HTTP durum kodlarının anlamlı şekilde iletilmesi, içerik pazarlığı yoluyla farklı türde yanıtların döndürülebilmesi gibi birçok profesyonel özellik uygulanmıştır. Proje aynı zamanda Swagger ile dökümantasyonu yapılmış bir API olarak, test edilebilirlik açısından da İlgili Postman testleri ve Scriptleri ile kullanıcılara kolaylık sunmaktadır. Bu çalışma, kurumsal düzeyde geliştirilen projelerde kullanılan mimari ve yapısal tekniklerin bireysel düzeyde nasıl uygulanabileceğini göstermeyi, geliştiricilere web geliştirme dünyasında pratik, modüler, yönetilebilir yazılım çalışmalarının nasıl çalıştığını göstermeyi hedefler. |
| **Beklenen Sonuçlar ve Yararlar** | Projenin tamamlanmasıyla birlikte geliştirilebilir, sürdürülebilir ve modüler yapıda bir RESTful API elde edilmesi hedeflenmektedir. Proje; kod kalitesini artırma, test edilebilir yapı kurma ve sorumlulukları ayrıştırma gibi yazılımın temel ilkelerini somut şekilde öğretmeyi amaçlamaktadır. Aynı zamanda, validasyon, hata yönetimi, sayfalama ve içerik pazarlığı gibi konular sayesinde, geliştiricinin ileri düzey API tekniklerine olan hakimiyetini artırması beklenmektedir. Ayrıca, geliştirici bu çalışma sayesinde GitHub gibi platformlarda güçlü bir portföy sunma şansı da elde edecektir. |

**Projeyi Alan Öğrenciler:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Öğrenci No** | **Adı** | **Soyadı** | **Email** | **Telefon** | **İmza** |
| B2310.032025 | Bahadır | Verir | bhdrverir@gmail.com | 05438302014 |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

Proje Danışmanı

.......... / ......... / 2025