İZMİR BAKIRÇAY ÜNİVERSİTESİ

MÜHENDİSLİK ve MİMARLIK FAKÜLTESİ

BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ

BİL 403 BİTİRME PROJESİ

ARA RAPORU

NotePool – Yapay Zeka Destekli Not Paylaşım Platformu

240601164

Yağız Zorlu

240601082

Erdem Temiztürk

Danışman

Dr. Öğr. Üyesi, Ozan Raşit Yürüm

Kasım, 2025

İÇİNDEKİLER

I. Projenin Durumu

1. Giriş

Bu rapor, "NotePool" adlı Yapay Zeka Destekli Not Paylaşım Platformu bitirme projesinin ara durumunu, tamamlanan kısımlarını, kullanılan mimari yapıyı ve gelecek hedeflerini detaylandırmak amacıyla hazırlanmıştır.

1.1. Projenin Amacı ve Kapsamı

NotePool projesinin temel amacı, üniversite öğrencilerinin ve akademisyenlerin ders notlarını (PDF formatında) kolayca yükleyebildiği, paylaşabildiği ve diğer kullanıcılarla etkileşime girebildiği merkezi bir sosyal platform oluşturmaktır.

Platform, geleneksel dosya paylaşım sitelerinden farklı olarak, modern bir sosyal medya akışı mantığıyla çalışacaktır. Kullanıcılar notları "post" olarak paylaşacak, bu postlara beğeni ve yorum yapabilecek, kendi koleksiyonlarına ekleyebilecek ve notları indirebilecektir.

Projenin en ayırt edici özelliği, bu sosyal platformu güçlü yapay zeka servisleri ile entegre etmektir. Bu entegrasyonlar sayesinde spam/troll içerik tespiti, PDF içeriklerinin otomatik özetlenmesi, akıllı etiketleme, semantik (anlamsal) arama ve kullanıcıya özel not önerileri gibi ileri seviye fonksiyonellikler sunulacaktır.

1.2. Projenin Yapısı ve Kullanılan Teknolojiler

Proje, Backend ve Frontend olmak üzere iki ana bileşenden oluşan modern bir "Single Page Application" (SPA) olarak geliştirilmektedir.

- Backend Teknolojisi: Microsoft .NET 8 (C#)
- Frontend Teknolojisi: Angular 17 (TypeScript)
- Veritabanı: PostgreSQL
- Altyapı ve Dağıtım (DevOps): Docker
- Mimari Yaklaşım: Onion Architecture ve CQRS (Command Query Responsibility Segregation)

1. Sistemin Genel Mimarisi

Proje, sürdürülebilir, ölçeklenebilir ve bakımı kolay bir yapı sağlamak amacıyla modern yazılım mimarisi prensipleri üzerine inşa edilmiştir.

2.1. Backend Mimarisi: Onion Architecture

NotePool'un backend'i, **Onion Architecture (Soğan Mimarisi)** prensiplerine dayanmaktadır. Bu mimari, katmanlar arasındaki bağımlılıkları **merkezden dışa doğru** yönlendirir ve **bağımlılıkların ters çevrilmesi (Dependency Inversion)** ilkesine dayanır. Amaç; iş kurallarını dış dünyadan izole etmek, test edilebilirliği artırmak ve değişimlere karşı sistemi esnek hale getirmektir. Onion Architecture yapısı dört ana katmandan oluşur:

Domain Katmanı (Core / Inner Layer):
Uygulamanın çekirdeğidir. Hiçbir dış bağımlılığı yoktur. Tüm **entity** sınıfları (örneğin: Note, User, Institution) ve bunlara ait **iş kuralları** burada tanımlanır. Bu katman tamamen bağımsızdır; ne veritabanı, ne framework, ne de API bileşenleriyle ilişkisi vardır.

Application Katmanı (Service Layer):

İş mantığının yürütüldüğü ve domain varlıklarının (entities) nasıl kullanılacağının belirlendiği katmandır. Burada **CQRS (Command Query Responsibility Segregation)** deseni uygulanır ve **MediatR** kütüphanesi ile komut/sorgu akışları yönetilir.
Domain katmanındaki arayüzleri (örneğin repository arayüzleri) kullanır ancak onların somut implementasyonlarını bilmez.

Infrastructure Katmanı (Outer Layer):

Dış dünya ile iletişimi sağlayan katmandır. Veritabanı erişimi (**Entity Framework Core**), dosya depolama (**Azure Blob Storage**), dış API çağrıları ve yapay zeka servisleri bu katmanda yer alır. Application katmanında tanımlı arayüzlerin somut karşılıklarını burada uygularız (örneğin INoteRepository → NoteRepository).

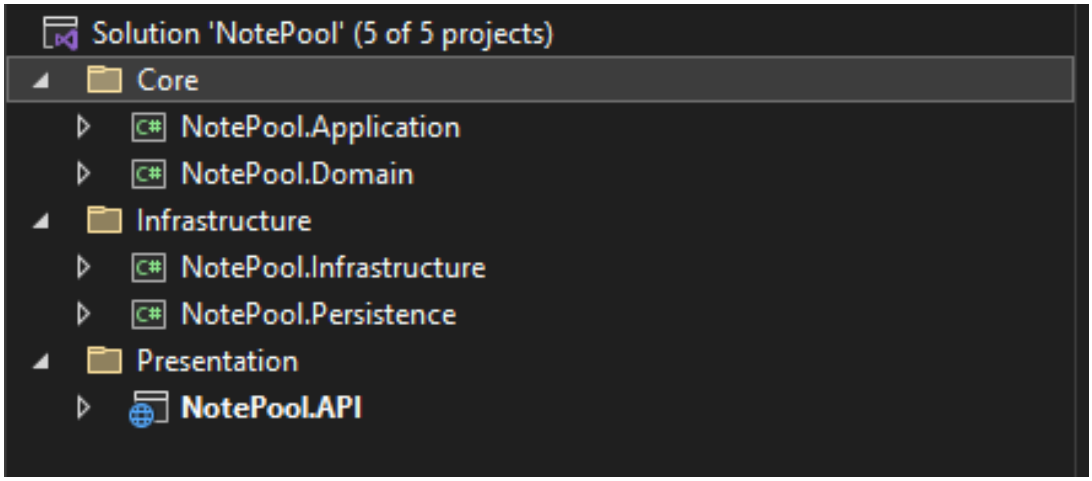
Presentation Katmanı (API / UI Layer):
Kullanıcıdan gelen isteklerin işlendiği en dış katmandır.**API Controller** sınıfları (örnek: NotesController, UsersController) burada bulunur. Controller'lar doğrudan **Application** katmanına bağımlıdır; Infrastructure detaylarını bilmezler. Bu katman sadece "iletişim yüzeyi" görevini görür.

2.2. Neden Onion Architecture?

Onion Architecture, yazılım bileşenleri arasındaki bağımlılık yönünü **içe doğru** tanımlaybilmezler. Bun hem test edilebilirliğini hem de esnekliğini artırır. Bu yapı sayesinde Domain katmanı tamamen korunur; dış katmanlardaki değişiklikler çekirdeğe etki etmez.

Avantajları:

1. **Test Edilebilirlik:**
   Domain ve Application katmanları, veritabanı veya servis entegrasyonlarından bağımsız olduğu için, tüm iş kuralları **mock repository** ve **in-memory veri** ile test edilebilir.
2. **Esneklik:**
   Örneğin NotePool'da PostgreSQL yerine MSSQL veya MongoDB'ye geçmek istediğimizde, sadece **Infrastructure** katmanındaki repository implementasyonunu değiştirmemiz yeterlidir. Diğer katmanlar bu değişiklikten etkilenmez.
3. **Bakım Kolaylığı:**
   Kodun yeri ve amacı nettir. Örneğin yeni bir özellik ("Yorum Beğenme") eklenecekse, ilgili Comment entity'si Domain katmanında, CreateCommentReactionCommand Application katmanında ve CommentReactionRepository Infrastructure katmanında düzenlenir. Her katman kendi sorumluluğu içinde kalır.



2.3. Frontend Mimarisi: Angular Modüler Yapısı

Projenin ön yüzü, Google tarafından geliştirilen **Angular** çatısı kullanılarak oluşturulmuştur. Angular, **bileşen tabanlı (component-based)** ve **tek sayfa uygulama (Single Page Application – SPA)** mimarisi sayesinde, kullanıcıya dinamik, hızlı ve modern bir arayüz sunar.

Uygulama, **modüler mimari (modular architecture)** prensiplerine uygun şekilde tasarlanmıştır. Her ana özellik kendi modülü içinde izole edilmiştir. Örneğin;

- **AuthModule** → kullanıcı giriş ve kayıt işlemlerini yönetir.
- **NoteModule** → not oluşturma, listeleme ve detay sayfalarını barındırır.
- **ProfileModule** → kullanıcı profili ve istatistik bileşenlerini içerir.

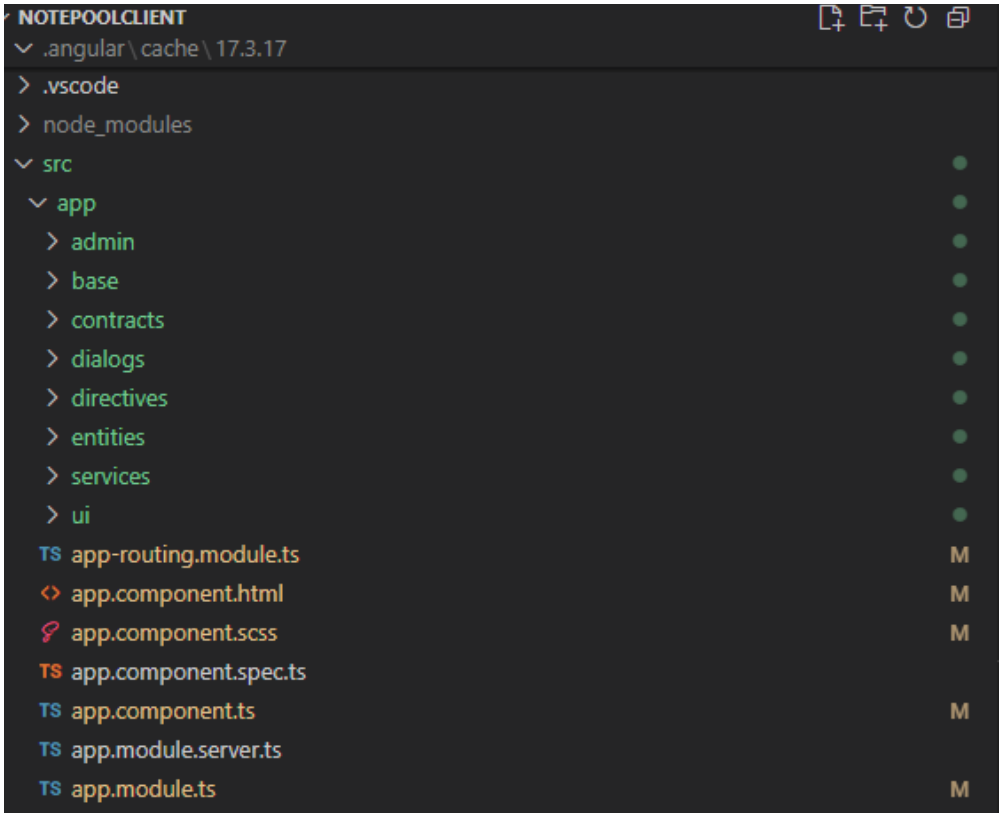
Bu yapı, projenin okunabilirliğini ve sürdürülebilirliğini artırmış, modüller arasında bağımsız geliştirme imkânı sağlamıştır.

Uygulamada **Lazy Loading (Tembel Yükleme)** yöntemi kullanılmıştır. Bu sayede, kullanıcı yalnızca ihtiyaç duyduğu modüller yüklendiğinde uygulamanın ilk açılış süresi kısalmış ve performans artışı elde edilmiştir.

Routing yapısı **Angular Router** üzerinden yönetilmekte, sayfa yenilenmeden hızlı geçişler sağlanmaktadır. Ortak bileşenler ve servisler (örneğin `AuthService`, `HttpInterceptor`, `SpinnerComponent`) tek bir merkezde toplanarak kod tekrarı azaltılmıştır.

Ayrıca, **Angular Material UI** kütüphanesi ile sade ve tutarlı bir kullanıcı arayüzü tasarlanmıştır. **Reactive Forms** yapısı sayesinde formlar üzerinde anlık doğrulama yapılmakta, **RxJS** ile asenkron veri akışları performanslı biçimde yönetilmektedir.

Sonuç olarak, NotePool'un Angular tabanlı frontend mimarisi; **modüler, performanslı, esnek ve kullanıcı dostu** bir yapıya sahiptir. Bu yapı, sistemin uzun vadede kolayca geliştirilebilir ve genişletilebilir olmasını sağlamaktadır.



2.4. Veritabanı ve Altyapı (PostgreSQL & Docker)

Proje, güçlü ve açık kaynaklı bir ilişkisel veritabanı olan **PostgreSQL** üzerinde çalışmaktadır. Yüksek performansı, güvenilirliği ve **ACID** standartlarına tam uyumu sayesinde NotePool'un veri bütünlüğü güvence altına alınmıştır.

Veritabanı yapısı, **Entity Framework Core (EF Core)** kullanılarak **Code-First** yaklaşımıyla tasarlanmıştır. Bu sayede tablo yapıları ve ilişkiler C# sınıflarından otomatik olarak oluşturulmakta, değişiklikler **migration** yapısıyla yönetilmektedir. Böylece hem geliştirme hem bakım süreçleri kolaylaşmıştır.

Tüm geliştirme ve test süreçleri **Docker konteynerları** üzerinden yürütülmektedir. `docker-compose.yml` dosyası sayesinde PostgreSQL veritabanı, backend API ve Angular frontend servisleri tek bir komutla (`docker-compose up`) ayağa kaldırılabilmektedir. Bu yapı, her geliştiricide aynı ortamın oluşmasını sağlayarak "**benim makinemde çalışıyordu**" problemini ortadan kaldırmıştır.

Ayrıca Docker kullanımı, sistemin **taşınabilirliğini (portability)** ve **dağıtım kolaylığını** artırmıştır. Proje hem Windows hem de Linux ortamlarında sorunsuz çalışabilmekte, ilerleyen aşamalarda CI/CD entegrasyonuna hazır bir altyapı sunmaktadır.

II. Bitirme Projesinin Mevcut Durumu

3. Backend Geliştirmeleri (Veri ve İş Mantığı)

Projenin bu aşamasında, **backend mimarisi** büyük oranda tamamlanmış ve tüm **API uç noktaları** test edilebilir duruma getirilmiştir. Veri katmanı, **Entity Framework Core (EF Core)** kullanılarak **Code-First** yaklaşımıyla tasarlanmıştır. Böylece, tüm entity modelleri doğrudan C# sınıfları üzerinden tanımlanmış ve **migration** yapısı ile veritabanına otomatik olarak yansıtılmıştır.

Veri erişimi katmanında, **Repository Pattern** kullanılarak **soyutlama ve yeniden kullanılabilirlik** sağlanmıştır. Her entity için bir repository sınıfı oluşturulmuş, bu sayede CRUD işlemleri tek bir merkezden yönetilmiştir. EF Core'un **LINQ (Language Integrated Query)** yapısı, sorguların okunabilirliğini artırmış ve tip güvenli veri işlemleri mümkün hale getirmiştir. Örneğin, notların filtrelenmesi, sıralanması veya kullanıcı bazlı sorgulanması gibi işlemler LINQ ifadeleriyle optimize edilmiştir.

Uygulamanın iş mantığı, **CQRS (Command Query Responsibility Segregation)** prensibine göre yapılandırılmıştır. Bu desen sayesinde, **okuma (Query)** ve **yazma (Command)** işlemleri birbirinden ayrılmıştır:

- Her **Command** işlemi (örneğin CreateNoteCommand, UpdateUserCommand) için ayrı bir **Handler** sınıfı yazılmıştır.
- **Handler** sınıfları, **MediatR kütüphanesi** aracılığıyla otomatik olarak ilgili **Request**'lerle eşleştirilmiştir.
- Bu yapı, uygulamanın genişlemesini kolaylaştırmış ve **bağımlılıkları minimize ederek test edilebilirliği artırmıştır.**

Ek olarak, **Validation** işlemleri, handler'lar içerisinde veya servis düzeyinde yapılmış; böylece her istek, iş mantığına girmeden önce doğrulanmıştır. **Transaction** yönetimi EF Core üzerinden otomatik yürütülmüş ve hata durumlarında rollback sağlanmıştır.

API katmanında, her endpoint açık ve anlaşılır bir şekilde tanımlanmış; **Swagger UI** entegrasyonu sayesinde geliştiriciler, API'leri doğrudan tarayıcı üzerinden görüntüleyip test edebilmiştir. Swagger dokümantasyonu, projenin dış geliştiriciler tarafından da kolayca anlaşılmasını sağlamaktadır.

Son olarak, tüm veri işlemleri **asenkron (async/await)** olarak yürütülmekte, böylece uygulamanın performansı artırılmış ve eşzamanlı kullanıcı işlemlerinde verimlilik sağlanmıştır.

3.2. Entity Sınıfları

Tüm entity'ler Id (Primary Key), CreatedDate ve UpdatedDate alanlarını sağlayan bir BaseEntity sınıfından miras alır. Entity'ler Bookmark, Course, Comment, Institution, Note, User, NoteDownload, Department, File, NotePdfFile sınıflarından oluşmaktadır.

3.2.1. Entity Sınıflarının Detayları ve Görevleri

- Institution: Sistemin en üst düzey varlığıdır. Kullanıcıların üniversitelerini saklar. (Örn: "İzmir Bakırçay Üniversitesi").

```
public class Institution : BaseEntity
{
    1 reference
    public string Name { get; set; }
    0 references
    public string? City { get; set; }
    1 reference
    public ICollection<Department> Departments { get; set; }
}
```

- Department: Department sınıfı, kullanıcıların bağlı oldukları akademik birimleri temsil eder. Her Department, bir Institution ile ilişkilidir ve birçok User içerebilir. Bu yapı, filtreleme işlemlerinde önemli rol oynar; örneğin kullanıcı kendi bölümüne ait notları kolayca listeleyebilir.Gelecekte sistemin "ders–bölüm eşleşmeleri" özelliği aktif listeleyebilir. Gelecekteent entity'si bu ilişkiyi yönetecektir.

```
2 references
public class Department : BaseEntity
{
    0 references
    public string Name { get; set; }
    0 references
    public string? Code { get; set; }
    1 reference
    public Guid InstitutionId { get; set; }
    1 reference
    public Institution Institution { get; set; }
    0 references
    public ICollection<Course> Courses { get; set; }
}
```

- Course: Sistemdeki derslerin tutulduğu tablodur. Kullanıcılar notları yükleyeceği zaman veritabanından hazır şekilde gelen derslerden seçim yapacaklar ve bu da karmaşıklığı önleyecek. (Örn : "Çizge Teorisi")

```
7 references
public class Course : BaseEntity
{
    0 references
    public string Name { get; set; }
    0 references
    public string? Code { get; set; }
    0 references
    public ICollection<Note> Notes { get; set; }
}
```

- Note: Note sınıfı sistemin çekirdek yapısını oluşturur.Kullanıcıların yüklediği PDF notlar, açıklamalar, etiketler ve oluşturulma tarihi gibi bilgiler burada tutulur.Note entity'si, User, Course, Institution gibi diğer varlıklarla çok yönlü ilişkilere sahiptir.Örneğin bir User birden fazla Note yükleyebilir, ancak her Note yalnızca bir kullanıcıya aittir.Ayrıca, notlara yapılan beğeni ve yorum işlemleri için Reaction ve Comment tablolarıyla bire-çok ilişkiler kurulmuştur. *Bu yapı, notların filtrelenmesi ve kullanıcı bazlı sıralanması gibi sorguların LINQ üzerinden optimize edilmesini sağlar.*

```
public class Note : BaseEntity
{
    2 references
    public Guid UserId { get; set; }
    4 references
    public string Title { get; set; }
    4 references
    public string? Description { get; set; }
    1 reference
    public int ViewCount { get; set; } = 0;
    4 references
    public string? Tags { get; set; }
    1 reference
    public bool IsApproved { get; set; } = true;
    2 references
    public Guid CourseId { get; set; }
    2 references
    public Guid InstitutionId { get; set; }
    0 references
    public User User { get; set; }
    0 references
    public Course Course { get; set; }
    0 references
    public Institution Institution { get; set; }
    0 references
    public ICollection<Comment> Comments { get; set; }
    0 references
    public ICollection<Reaction> Reactions { get; set; }
    0 references
    public ICollection<Bookmark> Bookmarks { get; set; }
    7 references
    public ICollection<NotePdfFile> NotePdfFiles { get; set; }
}
```

ICollection sayesinde iki tablo arasında 1-N ilişki kurabiliyoruz. 1 Notta Çok Reaksiyon olacağından dolayı bu sınıfta ICollection ile Reaction ı tanımladık.

Comment, Reaction, Bookmark sınıfları da aynı şekilde oluşturulup içlerine özellikleri girilmiştir. O sınıfların da tablo ilişkileri için gerekli diğer tablo bilgileri eklenmiştir.

3.2.2. User ve ASP.NET Core Identity

- User: Sistemin ana aktörüdür. Notları yükleyen, yorum yapan, beğenen varlıktır.
- **ASP.NET Core Identity Entegrasyonu**: User varlığımız, Microsoft'un sunduğu IdentityUser<Guid> sınıfından türetilmiştir. Sadece kullanıcı girişi ve rollendirme bazlı işlemlerin yanında projemizdeki tüm işlerden user sorumludur. Kullanıcılar hem notları siteye yükleyen kişiler olup hem de not yükleyen, yorum yapan ve reaksiyon verenler olacaktır. Bu yüzden IdentityUser olmasının yanında bir Student gibi işlev de görmektedir. Bu, bizim sıfırdan bir güvenlik sistemi yazma yükümüzü ortadan kaldırır. Identity sayesinde şu özellikler hazır olarak gelir:
  - **Güvenli Şifre Yönetimi:** PasswordHash ve PasswordSalt kullanarak şifrelerin geri döndürülemez şekilde hash'lenmesi.
  - **Rol Bazlı Yetkilendirme** : Kullanıcılara "Admin", "Student", "Moderator" gibi roller atayabilme.

- **Token Yönetimi (JWT)**: API güvenliği için JSON Web Token oluşturma ve doğrulama altyapısı.
- **Hazır Endpoint'ler:** E-posta doğrulama, şifre sıfırlama, iki faktörlü kimlik doğrulama (2FA) gibi karmaşık güvenlik akışları.

Güvenlik: Kimlik doğrulama **JWT Bearer** ile yapılır; rol tabanlı yetkilendirme (Admin, Student, Moderator) attribute seviyesinde uygulanır. Dosya yüklemede **MIME tipi** ve **maksimum dosya boyutu** kontrolü aktif olup yalnızca application/pdf kabul edilir. Kişisel veriler KVKK kapsamında işlenir; parola saklama **ASP.NET Identity** hash mekanizmasıyladır. CORS, sadece izinli origin'lere açıktır.

```
public class User : IdentityUser<Guid>
{
    1 reference
    public string FirstName { get; set; }
    1 reference
    public string LastName { get; set; }
    0 references
    public string Role { get; set; } = "Student";
    0 references
    public string? ProfileImage { get; set; }
    1 reference
    public Guid InstitutionId { get; set; }
    0 references
    public Institution Institution { get; set; }
    – references
    public Department Department { get; set; }
    0 references
    public ICollection<Bookmark> Bookmarks { get; set; }
    0 references
    public ICollection<Note> Notes { get; set; }
    0 references
    public ICollection<Reaction> Reactions { get; set; }
    0 references
    public ICollection<Comment> Comments { get; set; }
```

3.2.3 Dosya Yükleme

NotePool uygulamamızda en önemli işlevlerden biri kullanıcıların notlarını pdf olarak yükleyebilmesidir. Dosya yükleme için birden fazla tabloya ve detaylı sınıflara ihtiyacımız oldu. Öncelikle bir File adında sınıfa sahip olduk. UpdatedDate e ihtiyacımız olmadığı için NotMapped attribute kullanarak onu devredışı bıraktık. Ayrıca bir NotePdfFile sınıfı da oluşturduk ve bu bizim asıl kullandığımız sınıf oldu. Bu tablomuzu Note ile birleştirip kullanıcıların Not yüklerken dosyayı da yükleme zorunluluğunu getirmiş olduk.

```
6 references
public class File : BaseEntity
{
    4 references
    public string FileName { get; set; }
    4 references
    public string Path { get; set; }
    2 references
    public string Storage {  get; set; }

    [NotMapped]
    5 references
    public override DateTime UpdatedDate { get => base.UpdatedDate; set => base.UpdatedDate =
      value;}
}
```

```
12 references
public class NotePdfFile : File
{
    4 references
    public Guid NoteId { get; set; }
    1 reference
    public Note Note { get; set; }

}
```

Dosya yükleme süreci, kullanıcı deneyimini basitleştirmek ve sistemin güvenliğini sağlamak amacıyla detaylı şekilde tasarlanmıştır.
Yüklenen PDF dosyaları, **IFormFile** aracılığıyla backend'e iletilir ve EF Core üzerinden NotePdfFile tablosuna kaydedilir. Dosya meta verileri (isim, uzantı, boyut, MIME tipi) veritabanında saklanırken, asıl dosya içeriği fiziksel olarak sunucu dizinine veya bulut ortamına kaydedilmektedir.

Sistemde, yüklenen dosyanın **MIME türü (application/pdf)** kontrol edilmekte ve yalnızca PDF uzantılı dosyalara izin verilmektedir. Ayrıca 10 MB üzerindeki dosyalar reddedilerek depolama alanının verimli kullanımı hedeflenmiştir.

Kullanıcılar dosya yüklerken oluşabilecek hatalar (örneğin bağlantı kesilmesi, geçersiz dosya türü, eksik alan) için özel hata mesajları dönülmektedir. Bu hata yönetimi, **global exception handling middleware** aracılığıyla yönetilir.

NotePdfFile tablosunun Note ile bire-bir (1–1) ilişkili olması, veri bütünlüğü açısından önemlidir. Bu sayede her notun yalnızca bir PDF dosyası olur, aynı zamanda not silindiğinde ilişkili dosya da otomatik olarak sistemden kaldırılır.

Geliştirme sürecinde, ilerleyen versiyonlarda **Azure Blob Storage veya AWS S3** entegrasyonu düşünülmektedir. Böylece yüklenen dosyalar bulutta depolanarak yedekleme ve ölçeklenebilirlik kolaylığı sağlanacaktır.

Ayrıca performans optimizasyonu amacıyla, dosya yükleme işlemleri **asenkron (async/await)** şekilde çalıştırılmış ve böylece eşzamanlı yüklemelerde sistem kaynakları daha verimli kullanılmıştır.

```
    N.Note note = await _noteReadRepository.GetByIdAsync(request.Id);
    if (note == null)
        throw new KeyNotFoundException("Note bulunamadı.");

    List<(string fileName, string pathOrContainerName)> result =
        await _storageService.UploadAsync("note-files", request.Files);

    if (result == null || result.Count == 0)
        throw new InvalidOperationException("Dosyalar yüklenemedi.");

    var pdfEntities = result.Select(r => new Domain.Entities.NotePdfFile
    {
        FileName = r.fileName,
        Path = r.pathOrContainerName,
        Storage = _storageService.StorageName,
        NoteId = note.Id
    }).ToList();

    await _notePdfFileWriteRepository.AddRangeAsync(pdfEntities);
    await _notePdfFileWriteRepository.SaveAsync();
    var response = new UploadNotePdfFileCommandResponse();

    return response;
}
```

NotesControllerda bu yükleme işleminin gerekli API kodunu yazmamızla artık projemizin en gerekli işlevini halletmiş oluyoruz.

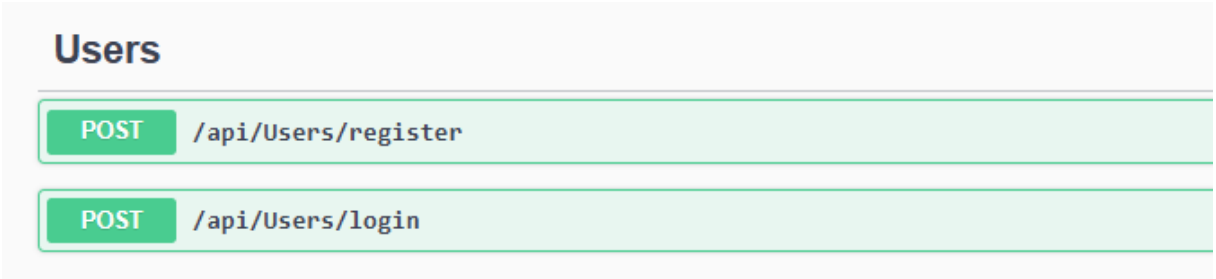
```
[HttpPost("create-with-pdf")]
[Consumes("multipart/form-data")]
0 references
public async Task<IActionResult> CreateWithPdf([FromForm] CreateNoteCommandRequest
  createNoteCommandRequest)
{
    var response = await _mediator.Send(createNoteCommandRequest);
    return Ok(response);
}
```

3.3. API Mimarisi ve API Testleri (Swagger)

Geliştirilen tüm **API endpoint'leri**, .NET'in yerleşik **Swagger / OpenAPI** arayüzü üzerinden test edilmiştir. Swagger, hem dokümantasyon hem de etkileşimli test aracı olarak kullanılmıştır.

1. **Kullanıcı Kayıt ve Giriş:**

   POST/api/Auth/Register ve POST/api/Auth/Login uç noktaları test edilmiştir. Login işleminde dönen **JWT token**, Swagger üzerinde otomatik olarak kaydedilip sonraki isteklerde **Authorization Header** olarak eklenmiştir. Böylece güvenli oturum işlemleri başarıyla doğrulanmıştır. Kayıt olan her kullanıcı veritabanında Users tablosuna eklenmektedir.

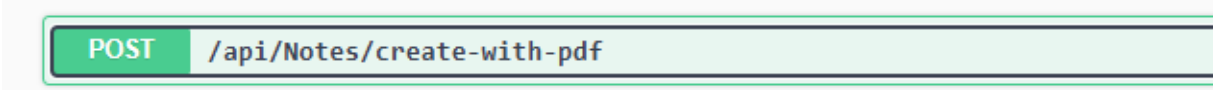


2. **Üniversite ve Kurum Doğrulaması:**

   GET /api/Institutions ve GET /api/Institutions/{id} endpoint'leriyle kurum verileri test edilmiş; Üniversitelerin hepsi veritabanından getirilir ve bu kullanıcı kaydında veya notları filtrelemede çok ciddi bir gelişme kazandırır.

3. **Not Yükleme (multipart/form-data):**

   POST /api/Notes endpoint'i üzerinden metin alanları (title, description, courseId) ve PDF dosyası birlikte gönderilmiştir. Swagger'ın "Try it out" özelliğiyle dosya yüklenmiş, EF Core aracılığıyla veritabanına kaydedilmiş ve başarılı cevapta notun **ID'si**, **oluşturulma tarihi** ve **kullanıcı bilgileri** dönmüştür.



4. **Not Güncelleme ve Silme:**

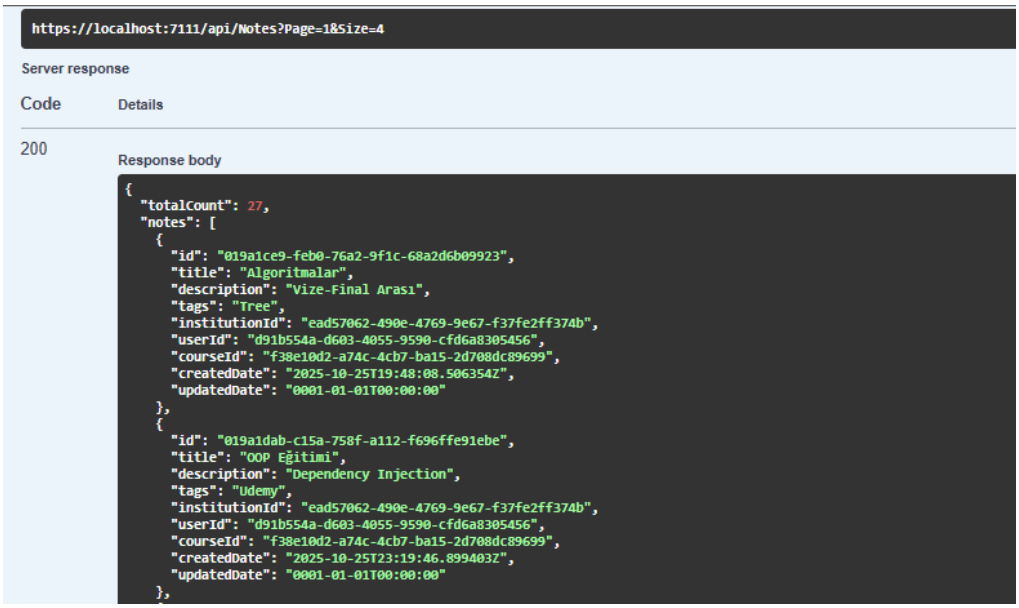
   PUT /api/Notes/{id} ve DELETE /api/Notes/{id} çağrıları test edilmiştir. Silme işlemi yalnızca notu oluşturan kullanıcıya izin vermiş, yetkisiz erişimlerde **403 Forbidden** dönmüştür.

5. **Listeleme ve Filtreleme:**

   GET /api/Notes endpoint'iyle notlar listelenmiş, **LINQ tabanlı filtreleme** (courseId, userId, keyword) ve sıralama işlemleri doğrulanmıştır.

Swagger testlerinde tüm uç noktalar için 200, 400, 401 ve 403 durum kodları kontrol edilmiş; hata mesajlarının anlamlı ve tutarlı olduğu görülmüştür.

Bu testler sonucunda, NotePool'un backend API'leri doğruluk, güvenlik ve veri bütünlüğü açısından başarılı şekilde doğrulanmıştır.



Ek olarak, tüm endpoint'ler için dönen **HTTP durum kodları** (200, 201, 400, 401, 403, 404) gözlemlenmiş ve hata mesajlarının kullanıcı dostu, açıklayıcı formatta olduğu doğrulanmıştır.

Bu testlerin sonucunda, API'nin hem işlevsel hem de güvenlik açısından stabil çalıştığı ve tüm akışların frontend entegrasyonuna hazır olduğu görülmüştür.

Swagger arayüzü, projenin ilerleyen aşamalarında geliştirici ekibin test süreçlerini kolaylaştırmak için kullanılmaya devam edecektir.

4. Frontend Geliştirmeleri (Sunum Katmanı)

Angular ile geliştirilen kullanıcı arayüzü, backend API'lerini entegre ederek modern, hızlı ve etkileşimli bir deneyim sunmaktadır. Uygulama genelinde bileşen tabanlı mimari benimsenmiş olup, Angular Router, Reactive Forms ve Material UI kütüphanesi aktif şekilde kullanılmaktadır.

4.1. Proje Yapısı ve Modülerlik

Proje, Angular'ın modüler mimarisi temel alınarak **UI** ve **Admin** olmak üzere iki ana bölümde yapılandırılmıştır. Bu sayede kullanıcı arayüzü ile yönetici paneli birbirinden bağımsız geliştirilebilir hale gelmiştir.

4.1.1. Uygulama Modülleri

- **UI Modülü:** Uygulamanın son kullanıcıya görünen kısmını oluşturur. İçerisinde aşağıdaki temel bileşenler yer alır:
  - HomeComponent → Not akışı ve önerilen içerikler
  - LoginComponent → Kullanıcı girişi
  - RegisterComponent → Yeni kullanıcı kaydı
  - ProfileComponent → Kullanıcı profili ve paylaşılan notlar
  - NotesComponent → Notların listelendiği sayfa
  - UploadNoteComponent → Yeni not yükleme ekranı

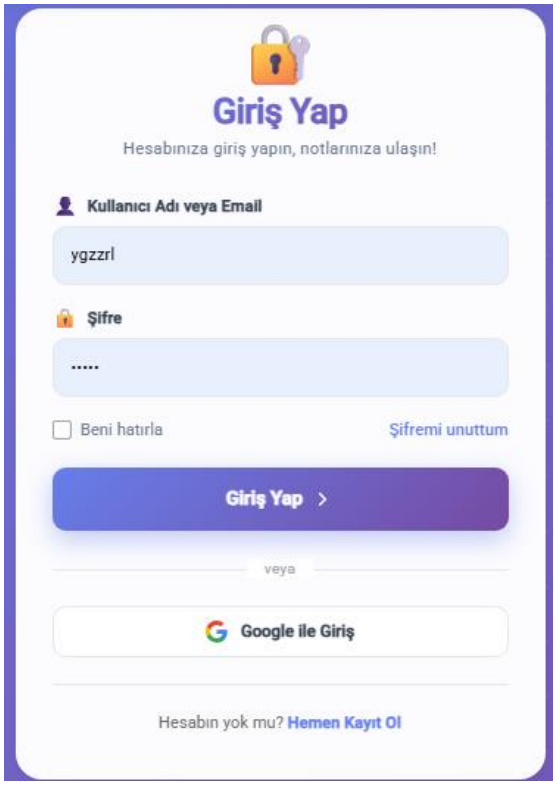
- **Admin Modülü:**
  Geliştirme aşamasında olup, sistem yöneticilerinin kullanıcıları, notları veya kurum bilgilerini yönetebilmesi için planlanmıştır. Yetkilendirme kontrolleri backend tarafındaki role alanına göre yapılmaktadır.

4.2. Yetkilendirme (Login / Register)

Kullanıcıların platforma erişimi için **Angular Reactive Forms** altyapısı kullanılmıştır. Login ve Register ekranlarında:

- Gerçek zamanlı **validasyon** (örneğin e-posta formatı, minimum parola uzunluğu, boş alan kontrolü),
- Başarılı girişte alınan **JWT token'ın** localStorage üzerinde saklanması,
- Token'ın AuthInterceptor aracılığıyla her API isteğine otomatik eklenmesi

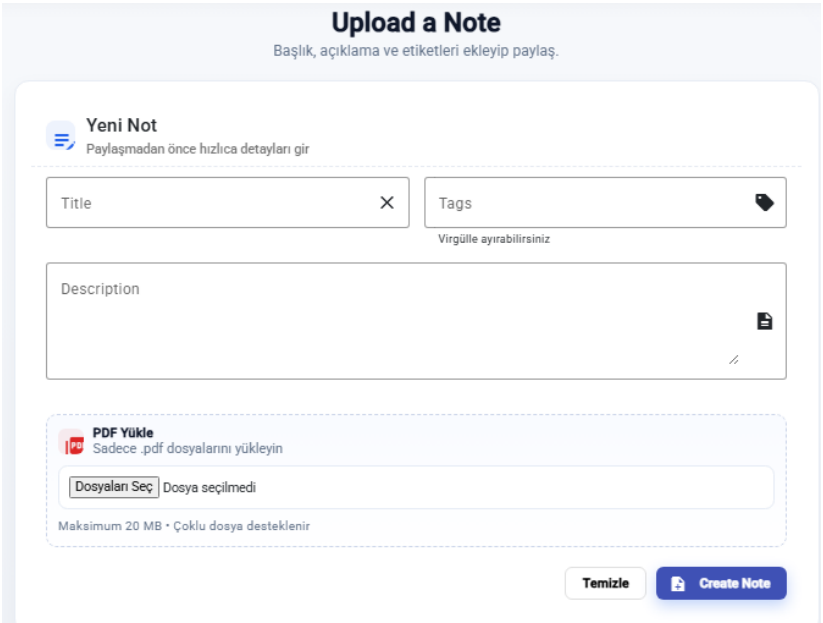
işlevleri uygulanmıştır.



4.2.2. Not Yükleme (Upload Note)

UploadNoteComponent, kullanıcıların yeni not oluşturup PDF yükleyebildiği form bileşenidir.

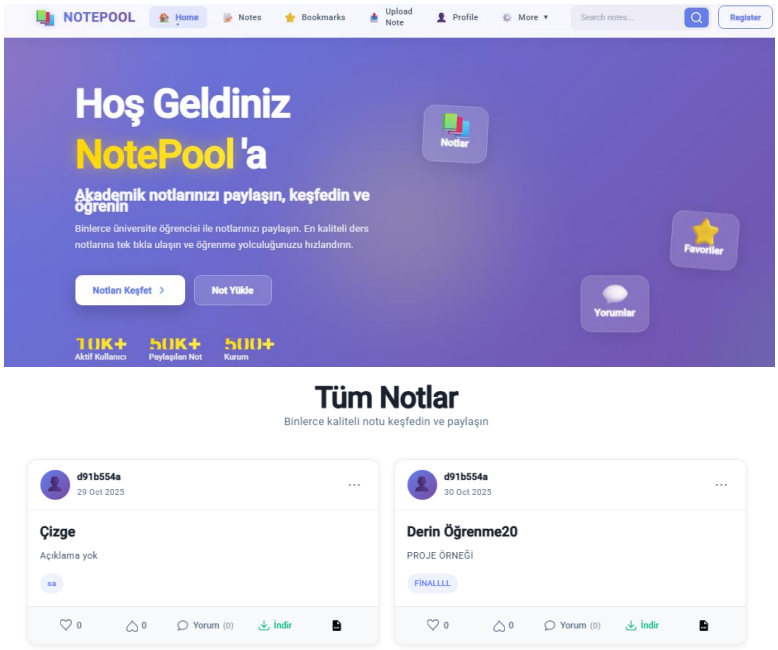
- Form; başlık, açıklama, ders seçimi, etiket ve PDF dosyasını içerir.
- Dosya yükleme **multipart/form-data** formatında gerçekleşir.
- Başarılı yükleme sonrası kullanıcıya toast bildirimi gösterilir ve not listesi güncellenir.



4.2.3. Ana Akış (Home Feed) ve Not Detayı

HomeComponent, kullanıcıların platformdaki notları sosyal medya benzeri bir akışta görebildiği bölümdür.

- Not kartları Angular Material MatCard bileşeniyle oluşturulmuştur.
- Veriler backend'deki /api/Notes endpoint'inden çekilir.
- Liste, **sonsuz kaydırma (infinite scroll)** özelliğiyle dinamik olarak genişler.



4.2.4. Profil Yönetimi

ProfileComponent, kullanıcının kendi notlarını, kaydettiği içerikleri (bookmarks) ve istatistiklerini görüntüleyebildiği sayfadır.
Sayfa, userId parametresine göre dinamik olarak yüklenir ve veriler /api/Notes ve /api/Users endpoint'lerinden çekilir.

III. Planlanan Proje Görevleri

5.1. Spam ve Troll İçerik Tespiti

Bu aşamada, kullanıcıların platforma yüklediği notların veya yorumların içerik bakımından uygunluğunu denetleyecek bir yapay zekâ modeli geliştirilecektir. Modelin temel amacı, platformun akademik amacına zarar veren **spam, alakasız, kötü niyetli veya tekrarlayan içerikleri** otomatik olarak tespit edip işaretlemektir.

Bu sistem, **doğal dil işleme** yöntemleriyle çalışacaktır. Not başlıkları, açıklamaları ve yorum metinleri üzerinde analiz yapılacak; toksik kelime analizi, anahtar kelime yoğunluğu, tekrar oranı ve anlamsal benzerlik ölçümleri değerlendirilecektir.

Modelin eğitimi için örnek olarak:

- "Temiz",
- "Spam",
- "Troll"

etiketlerine sahip örnek veri setleri kullanılacaktır.

Bu veri seti, Python ortamında **scikit-learn**, **spaCy** veya **transformers** kütüphaneleriyle ön işleme alınacak, ardından sınıflandırma modeli olarak **Logistic Regression**, **Random Forest** veya derin öğrenme tabanlı bir **Transformer Classifier** kullanılacaktır.

Model çıktısı, not yükleme veya yorum ekleme sırasında **gerçek zamanlı (real-time)** olarak çalışacak, olası spam içerikler için "uyarı" veya "onay bekliyor" durumu gösterecektir. Bu sayede platformun bilgi kalitesi korunacaktır.

5.2. Akıllı (Semantik) Arama Sisteminin Geliştirilmesi

NotePool'un yapay zekâ destekli çekirdek özelliklerinden biri de **semantik arama** fonksiyonudur. Bu görevde, kullanıcıların yalnızca kelime eşleşmesiyle değil, **anlam benzerliğine göre** not arayabilmesi sağlanacaktır.

Sistem, klasik SQL LIKE aramalarının ötesine geçerek, not başlıkları ve açıklamalarını **vektör uzayına** dönüştürecektir. Bunun için önceden eğitilmiş dil modelleri kullanılacaktır.

Arama sorgusu kullanıcıdan geldiğinde, aynı modelle vektöre dönüştürülüp veritabanındaki notların embedding'leriyle **koşu (cosine similarity)** hesaplanacaktır.
En yüksek benzerliğe sahip notlar kullanıcıya sıralı biçimde gösterilecektir.

Bu yapı sayesinde:

- "Ders notu algoritma karmaşıklığı" araması yapıldığında "Algorithm Analysis" veya "Big-O Complexity" içeren notlar da listelenecektir.
- Türkçe ve İngilizce karışık sorgular da anlam tabanlı karşılaştırmayla bulunabilecektir.

Bu sistem, NotePool'un bilgi erişimini hızlandıracak ve kullanıcıların gerçekten aradığı içeriklere ulaşmasını kolaylaştıracaktır.

5.3. Kullanıcıya Öneri Sistemi

Bu aşamada, NotePool kullanıcılarının ilgi alanlarına göre kişiselleştirilmiş not önerileri alabilmesi için bir **öneri sistemi** geliştirilecektir.

Model, kullanıcıların sistemdeki davranışlarını (beğenilen notlar, görüntülenen dersler, indirme geçmişi, arama sorguları) analiz ederek dinamik bir profil çıkaracaktır.
Örneğin:

- Bir kullanıcı sık sık "veri bilimi" ders notlarını görüntülüyorsa, model benzer konulardaki notları önerir.
- Kullanıcının kurum, bölüm ve kayıtlı ders bilgileri de öneri ağırlığında etkili olacaktır.

Model, başlangıçta **içerik tabanlı filtreleme** yöntemiyle çalışacak, daha sonra yeterli veri biriktiğinde **işbirlikçi filtreleme** yöntemine geçilecektir.

Öneri motoru, Python tarafında **pandas + scikit-learn** veya doğrudan **.NET ML.NET Recommendation API** ile entegre edilecektir.

Sonuçlar frontend'de "Sana Önerilen Notlar" bölümü olarak kullanıcıya sunulacaktır.

5.4. Otomatik Yazım Düzeltme ve Dil Tutarlılığı Analizi

Kullanıcıların yüklediği not başlıklarında, açıklamalarda veya yorumlarda yazım hataları olması doğal bir durumdur. Bu görevde, sisteme entegre edilecek **yazım düzeltme** ve **gramer denetimi** modülü sayesinde metin kalitesi artırılacaktır.

Model, **Türkçe Doğal Dil İşleme (Turkish NLP)** araçlarını kullanarak:

- Yazım hatalarını otomatik olarak algılayacak,
- Öneri veya otomatik düzeltme sunacaktır.

Bu modül, **Zemberek NLP**, **LanguageTool**, veya transformer tabanlı **T5 Türkçe modelleri** ile çalışabilir.
Ek olarak, İngilizce içerikler için de çok dilli destek planlanmaktadır.

Yapay zekâ desteği sayesinde platformdaki not açıklamaları daha okunabilir, arama sonuçları daha tutarlı hale gelecektir.

5.5. Yapay Zekâ Bileşenlerinin Entegrasyonu, Testleri ve Son Dokümantasyon

Son aşamada, geliştirilen tüm yapay zekâ modülleri sistemin mevcut backend ve frontend yapısına entegre edilecektir.
Bu entegrasyon süreci aşağıdaki adımlardan oluşur:

1. **Servis Entegrasyonu:** AI modülleri mikroservis yapısında çalışacak ve backend API'leriyle HTTP üzerinden iletişim kuracaktır.

2. **Veri Akışı Testleri:** PDF yükleme → metin çıkarımı → özetleme → etiketleme zinciri uçtan uca test edilecektir.
3. **Performans Ölçümleri:** Modellerin cevap süreleri (latency) ölçülerek sistem yanıt süresine olumsuz etkisi en aza indirilecektir.
4. **Doğruluk (Accuracy) Testleri:** Özetleme, etiketleme ve arama sonuçlarının doğruluğu kullanıcı testleriyle değerlendirilecektir.
5. **Güvenlik ve Gizlilik:** Kullanıcı verileri anonimleştirilecek, hiçbir metin üçüncü taraflara paylaşılmadan yerel olarak işlenecektir.

Proje tesliminden önce, her modül için ayrı test senaryoları, doğruluk raporları ve kullanım kılavuzları hazırlanacaktır.
Ayrıca sistemin genel mimarisi, yapay zekâ bileşenlerinin akış diyagramları ve API entegrasyon detayları **proje dokümantasyonuna** eklenecektir.

KAYNAKÇA

Angular Team. (2024). *Angular Material UI components*. https://material.angular.io/

Angular Team. (2024). *Angular Router: Navigation & Routing guide*. Angular. https://angular.dev/guide/router

Angular Team. (2024). *HTTP Client and Interceptors*. Angular. https://angular.dev/guide/http

Angular Team. (2024). *Reactive Forms in Angular*. Angular. https://angular.dev/guide/forms-overview

AutoMapper. (2024). *AutoMapper Documentation*. https://docs.automapper.org/en/stable/

FluentValidation. (2024). *FluentValidation for .NET*. https://docs.fluentvalidation.net/en/latest/

Gençay, Y. (2021, 3 Eylül). *Nedir bu Onion Architecture? Tam teferruatlı inceleyelim*. Gençay Yıldız Blog. https://www.gencayyildiz.com/blog/nedir-bu-onion-architecture-tam-teferruatli-inceleyelim/

Gençay, Y. (2022). *Entity Framework Core*. Gençay Yıldız Blog. https://www.gencayyildiz.com/blog/category/entity-framework-core/

Google. (2025). *Gemini – AI Chat and Reasoning Model*. https://gemini.google.com/app?hl=

Google Developers. (2024). *Angular – Web Framework Documentation*. https://angular.dev/

Microsoft. (2024). *ASP.NET Core documentation*. Microsoft Learn. https://learn.microsoft.com/en-us/aspnet/core/?view=aspnetcore-8.0

Microsoft. (2024). *ASP.NET Core Identity: Authentication and Authorization*. Microsoft Learn. https://learn.microsoft.com/en-us/aspnet/core/security/authentication/identity

Microsoft. (2024). *Entity Framework Core documentation*. Microsoft Learn. https://learn.microsoft.com/en-us/ef/core/

Microsoft. (2024). *MediatR Library for .NET*. NuGet Gallery. https://www.nuget.org/packages/MediatR/

Microsoft. (2024). *Swagger / Swashbuckle ASP.NET Core integration*. Microsoft Learn. https://learn.microsoft.com/en-us/aspnet/core/tutorials/getting-started-with-swashbuckle