# MİMARİ YAKLAŞIMLAR

## API First Yaklaşımına Geçiş ve Dokümanlardaki Değişiklikler

Büyük bir mimari kararı aldıktan sonra ilk iş, bu kararı dokümanlara yansıtarak onu projenin "resmi gerçeği" haline getirmektir. Bu, geliştirme sürecindeki herkesin (şu an için sadece siz olsanız bile) aynı vizyon ve plan üzerinde ilerlemesini sağlar.

Bu yeni "API-First" yaklaşımınıza uygun olarak güncellemeniz gereken dokümanlar ve yapmanız gereken değişiklikler şunlardır:

**1. GEN-003\_Software\_Architecture\_Document\_SAD.md (Yazılım Mimarisi Dokümanı) - En Kritik Değişiklik**

Bu doküman, karardan en çok etkilenen ve güncellenmesi en öncelikli olanıdır.

* **Ne Değişecek?** WebApp projesinin artık Application katmanıyla doğrudan konuşmadığı, bunun yerine Api projesinin bir istemcisi olduğu netleştirilmelidir.
* **Aksiyonlar:**
  + **"Bağımlılık Mimarisi ve Veri Akışı"** bölümünü güncelleyin. Eduhr.Webapp'in proje referansları arasından Eduhr.Application ve Eduhr.Infrastructure'a olan doğrudan bağımlılığı kaldırılabilir. WebApp'in artık Api'ye HTTP istekleri yapacağı belirtilmelidir.
  + **"Veri Akışı Örneği"ni** revize edin. Yeni akış şu şekilde olmalıdır:
    1. **İstek:** Kullanıcı, Eduhr.Webapp'teki bir butona tıklar.
    2. **İstemci Kodu (WebApp):** WebApp'in JavaScript veya C# kodu, HttpClient kullanarak Eduhr.Api'ye bir GET /api/personnel isteği atar.
    3. **Controller (Api):** PersonnelController, isteği alır ve Application katmanına bir sorgu gönderir.
    4. **Geri Kalan Akış:** Application -> Infrastructure -> Veritabanı akışı aynı kalır.
    5. **Yanıt:** Api'den dönen JSON verisi, WebApp tarafından işlenir ve arayüz güncellenir.

**2. TEK-006\_Codes\_Planning.md (Sizin Kod Planlama Dokümanınız)**

Bu doküman, yapılacak işi doğrudan etkilediği için güncellenmelidir.

* **Ne Değişecek?** Eduhr.Webapp projesi altındaki .cshtml.cs dosyalarının sorumlulukları değişir.
* **Aksiyonlar:**
  + Eduhr.Webapp altındaki dosyaların "Amaç ve Sorumluluk" açıklamalarını güncelleyin. Örneğin, Pages/Hres/Index.cshtml.cs dosyasının amacı artık "İlgili Application servisini çağırmak" değil, "**API'ye HTTP isteği gönderip gelen JSON verisini işlemek**" olacaktır.
  + Plana yeni dosya türleri eklemeniz gerekebilir. Örneğin, API ile etkileşim kuracak **JavaScript dosyaları** (personnel-service.js gibi) veya HttpClient yapılandırmasını içeren **servis sınıfları** (ApiClientService.cs gibi) artık planın bir parçası olmalıdır.

**3. GEN-001\_Project\_Vision\_and\_Scope\_Document\_PVS.md (Proje Vizyonu ve Kapsamı)**

Bu üst seviye dokümanda küçük ama önemli bir güncelleme faydalı olacaktır.

* **Ne Değişecek?** Projenin teknik çözümünü anlatan bölüm, API merkezli yapıyı yansıtmalıdır.
* **Aksiyon:**
  + **"4. Çözüm ve Değer Önermesi"** bölümündeki teknik açıklamayı şu şekilde güncelleyebilirsiniz: "EduHR, tüm işlevselliğini merkezi bir Web API üzerinden sunan, modüler ve ölçeklenebilir bir SaaS uygulamasıdır. Bu merkezi API, hem ASP.NET Core Razor ile geliştirilmiş olan yönetim paneli (WebApp) hem de gelecekteki mobil uygulama (MobileApp) tarafından ortak olarak tüketilecektir."

**Güncelleme Gerektirmeyen Doküman**

* **GEN-002\_SRS.md (Yazılım Gereksinim Belirtimi):** Bu dokümanın büyük olasılıkla güncellenmesi **gerekmez**. Çünkü SRS, sistemin *ne yapması gerektiğini* (kullanıcı hikayeleri, fonksiyonel gereksinimler) tanımlar, *nasıl yapacağını* değil. Bir butona tıklandığında personel listesinin gelmesi gereksinimi, verinin direkt mi yoksa API üzerinden mi geldiğinden bağımsız olarak aynı kalır.

Bu doküman güncellemelerini yapmak, projenin geri kalanında tutarlı, anlaşılır ve kararlı bir şekilde ilerlemenizi sağlayacaktır.

## Çok Dillilik

Bu, projenin sadece teknik değil, aynı zamanda stratejik yönünü de derinden etkileyen bir karardır. Çok dillilik (i18n - Internationalization) desteğini projeye **bu aşamada dahil etme kararı almanız**, projenin gelecekteki büyüme potansiyeli için atılmış en doğru adımlardan biri olur.

Bu kararı projenin başında vermek, sonradan eklemeye çalışmaktan **çok daha kolay ve daha az maliyetlidir.** Çünkü çok dillilik desteği, projenin tüm katmanlarını yatay olarak kesen (cross-cutting) bir konudur.

İşte bu kararın projenize olan etkilerinin katman katman analizi:

**Etki Analizi: Katman Katman Değişiklikler**

Bu karar, SAD.md dokümanında tanımlanan tüm katmanları etkileyecektir.

1. **Sunum Katmanı (Presentation - WebApp ve Api)**

En gözle görülür değişiklikler bu katmanda olacaktır.

* **Resource Dosyaları (.resx):**
  + Arayüzdeki tüm metinler ("Personel Ekle", "Kaydet", "Ad Soyad" gibi) artık doğrudan koda yazılmayacak. Bunun yerine, her dil için ayrı kaynak dosyaları (.resx) oluşturulacak.
  + **Örnek:** Pages/Hres/Index.tr.resx ve Pages/Hres/Index.en.resx gibi dosyalarınız olacak.
* **Program.cs Yapılandırması:**
  + Hem WebApp hem de Api projelerinin Program.cs dosyalarında, uygulamanın hangi dilleri desteklediğini (tr-TR, en-US vb.), varsayılan dilin ne olduğunu ve dil seçiminin nasıl yapılacağını (URL'den, cookie'den veya tarayıcı ayarından) belirten Localization servisleri yapılandırılmalıdır.
* **Arayüz Kodları (.cshtml ve .cshtml.cs):**
  + Arayüzde metin göstermek için IViewLocalizer veya IStringLocalizer servisleri kullanılacak. Örneğin, <button>Yeni Personel Ekle</button> yerine <button>@localizer["AddNewPersonnelButton"]</button> gibi bir kullanım olacak.
* **API Yanıtları (Api):**
  + API'nin döndüğü hata ve başarı mesajları da çok dilli olmalıdır. API, isteğin Accept-Language başlığına bakarak uygun dilde mesaj döndürmelidir.

1. **Application Katmanı**

İş mantığı katmanı, özellikle hata ve doğrulama mesajları açısından etkilenir.

* **Doğrulama Mesajları (Validation):**
  + FluentValidation ile yazdığınız kuralların hata mesajları artık sabit metinler olamaz.
  + **Önce:** .NotEmpty().WithMessage("Ad alanı boş olamaz.");
  + **Şimdi:** .NotEmpty().WithMessage(x => localizer["FieldCannotBeEmpty", "FirstName"]); gibi bir yapıya dönüşerek IStringLocalizer üzerinden mesajları okumalıdır.
* **Servis Yanıtları ve DTO'lar:**
  + İşlemler sonucunda Application katmanının ürettiği herhangi bir metin tabanlı mesaj (örn: "İzin talebi başarıyla oluşturuldu.") artık kaynak dosyalarından gelmelidir.

1. **Infrastructure Katmanı**

Bu katman, kaynak dosyalarını barındırma ve dış dünya ile çok dilli iletişim kurma görevini üstlenir.

* **Kaynak Dosyalarının Merkezi:** Application katmanının kullanacağı ortak resx dosyaları (örn: ValidationMessages.tr.resx) genellikle bu katmanda veya Common projesinde merkezi bir yerde tutulur.
* **E-posta ve Bildirim Şablonları:**
  + Kullanıcılara gönderilen e-postalar (örn: "Hoş Geldin E-postası") artık her dil için ayrı şablonlar (Welcome.tr.html, Welcome.en.html) olarak tasarlanmalıdır.

1. **Domain Katmanı ve Veritabanı - En Büyük Etki!**

Bu, kararın en temel ve yapısal etkisinin olduğu yerdir. İki tür veriyi ayırmamız gerekir:

1. **Statik Veri (Arayüz Metinleri):** "Kaydet" butonu gibi. Bunlar .resx dosyaları ile çözülür ve Domain'i etkilemez.
2. **Dinamik Veri (Kullanıcının Girdiği Veri):** Bir departman adı olan "Muhasebe"nin, İngilizce arayüzde "Accounting" olarak görünmesi gerekiyorsa, bu durum doğrudan veritabanı şemanızı etkiler.
   * **Çözüm Yolu:** Genellikle "Çeviri Tablosu" (Translation Table) adı verilen bir desen kullanılır.
   * **Örnek:** departments tablonuz (id, name) yerine iki tabloya ayrılır:
     + departments (id, code) -> id=1, code="DEPT\_ACCOUNTING"
     + department\_translations (department\_id, language\_code, name) -> (1, "tr-TR", "Muhasebe"), (1, "en-US", "Accounting")
   * **Etkisi:** Bu, Department.cs gibi Domain varlıklarınızı, Repositorylerinizi ve sorgularınızı bu yeni yapıya uygun olarak tamamen değiştirmenizi gerektirir. Bu, projedeki en karmaşık değişikliktir.
3. **Önerilen Yol Haritası**

* **Kararı Kesinleştirin:** Veritabanındaki dinamik verilerin de çok dilli olup olmayacağına karar verin. (Öneri: Evet, olmalı).
* **Program.cs ile Başlayın:** Her iki projenizde de Localization servislerini ve Middleware'ini yapılandırın.
* **Kaynak Dosyalarını Oluşturun:** Önce WebApp için birkaç deneme .resx dosyası oluşturarak yapıyı kurun.
* **Veritabanı Şemasını Güncelleyin:** Domain varlıklarınızı ve Infrastructure katmanınızı çeviri tablosu desenine uygun olarak güncelleyin.
* **Arayüzleri Dönüştürün:** Adım adım tüm WebApp sayfalarındaki sabit metinleri localizer kullanımıyla değiştirin.

**Sonuç:** Çok dillilik desteği eklemek, projenin karmaşıklığını ve başlangıçtaki iş yükünü gözle görülür şekilde artırır. Ancak bu yatırımı **şimdi yapmak**, gelecekte uygulamanızın uluslararası pazarlara açılma potansiyeli olduğunda sizi haftalarca sürecek sancılı bir yeniden yapılandırma sürecinden kurtarır.

### ENUM Çok Dilliliği Nasıl Olur?

Bu, çok dilli bir uygulama geliştirirken en sık karşılaşılan ve en kritik konulardan biridir. Enum'ları İngilizce oluşturmanız **kesinlikle doğru bir başlangıçtır.**

Şimdi bu yapının çok dillilik durumunda nasıl çalıştığını, temel prensibini ve projenizde nasıl uygulamanız gerektiğini adım adım açıklayalım.

1. **A. Temel Prensip: Kod ve Görüntü Ayrımı**

En temel ve asla unutmamanız gereken kural şudur: **Enum'lar kod seviyesinde ASLA çevrilmez.**

* **Kod (Domain Katmanı):** Enum'un kendisi (LeaveStatus.Pending, Gender.Male gibi) programatik bir **tanımlayıcıdır**. Tıpkı bir veritabanı ID'si gibi, sabit, tutarlı ve dil-bağımsız olmalıdır. Kodunuz her zaman İngilizce ve sabit olan bu tanımlayıcılar üzerinden çalışmalıdır.
* **Görüntü (Presentation Katmanı):** Bu programatik tanımlayıcının kullanıcı arayüzünde hangi metne karşılık geleceği ("Onay Bekliyor", "Erkek" gibi) ise tamamen **sunum katmanının sorumluluğundadır.**

Bu ayrım, projenizin esnek, sürdürülebilir ve yeni dillere kolayca adapte edilebilir olmasını sağlar.

1. **B. Nasıl Çalışır? (API-First Mimaride)**

Sizin API-First yaklaşımınızda süreç şu şekilde işler:

1. **Backend (Eduhr.Api):** API'niz, veritabanından LeaveRequest nesnesini çeker. Bu nesnenin içindeki Status alanı LeaveStatus.Pending'dir.
2. **API Yanıtı (JSON):** API, bu nesneyi bir DTO'ya dönüştürür ve istemciye (WebApp veya MobilUygulama) JSON olarak gönderir. Bu JSON içinde durum, programatik karşılığı ile yer alır:

JSON

{

"id": 123,

"personnelName": "Ahmet Yılmaz",

"status": "Pending" // <-- DİKKAT: "Onay Bekliyor" DEĞİL!

}

1. **Frontend (İstemci - WebApp veya MobilApp):** İstemci (WebApp veya MobilApp), "Pending" değerini alır. Kullanıcının o anki diline göre (tr-TR, en-US vb.) kendi kaynak dosyalarına (.resx, .arb vb.) bakar ve "Pending" değerinin Türkçe karşılığı olan **"Onay Bekliyor"** metnini bularak ekrana basar.

Bu sayede API'niz evrensel ve dil-bağımsız kalır; dil çevirisi sorumluluğu tamamen arayüzü yöneten istemciye bırakılır.

1. **C. Uygulama Adımları (EduHR-2025 İçin)**

Projenizde bu yapıyı kurmak için izlemeniz gereken adımlar şunlardır:

1. **Adım 1: Domain Katmanı (EduHR.Domain)**

Enum'larınızı İngilizce ve sabit olarak tanımlayın. **Mevcut yapınız doğru, hiçbir değişiklik gerekmiyor.**

C#

// src/1\_Domain/EduHR.Domain/Enums/RequestStatus.cs

public enum RequestStatus

{

Pending,

Approved,

Rejected

}

1. **Adım 2: API Katmanı (Eduhr.Api)**

API'nizin, Enum'ları JSON'a çevirirken sayısal karşılıkları (0, 1, 2) yerine metinsel karşılıklarını ("Pending", "Approved") kullanmasını sağlayın. Bu, API'yi daha okunabilir yapar.

* **Program.cs dosyanıza ekleyin:**

C#

// API projesinin Program.cs dosyasında

builder.Services.AddControllers().AddJsonOptions(options =>

{

// Enum'ları metin olarak JSON'a dönüştürür ("Pending" gibi).

options.JsonSerializerOptions.Converters.Add(new JsonStringEnumConverter());

});

1. **Adım 3: Kaynak Dosyaları (EduHR.Localization)**

Çevirilerinizi tutacak olan .resx dosyalarını oluşturun. Enum çevirileri için tutarlı bir anahtar (key) formatı belirlemek en iyi pratiktir.

* Örnek: Enums.tr.resx dosyası içeriği:

| Name | Value |

| :--- | :--- |

| Enum\_RequestStatus\_Pending | Onay Bekliyor |

| Enum\_RequestStatus\_Approved| Onaylandı |

| Enum\_RequestStatus\_Rejected| Reddedildi |

1. **Adım 4: Sunum Katmanı (Eduhr.WebApp) - Çeviriyi Ekrana Basma**

En temiz yöntem, bu işlemi yapan yeniden kullanılabilir bir **Extension Metot** oluşturmaktır.

1. WebApp projenizde Extensions adında bir klasör oluşturun.
2. İçine EnumExtensions.cs adında bir sınıf ekleyin:

C#

using Microsoft.Extensions.Localization;

using System.ComponentModel.DataAnnotations;

using System.Reflection;

public static class EnumExtensions

{

// Bu metot, bir enum değerini alır ve .resx dosyasından çevirisini bulur.

public static string ToLocalizedString(this Enum value, IStringLocalizer localizer)

{

// Örnek Anahtar: "Enum\_RequestStatus\_Pending"

var resourceKey = $"Enum\_{value.GetType().Name}\_{value}";

var localizedString = localizer[resourceKey];

// Eğer çeviri bulunamazsa, en azından enum'un kendi adını gösterir.

return localizedString.ResourceNotFound ? value.ToString() : localizedString;

}

}

1. Artık herhangi bir Razor sayfasında (.cshtml) bu metodu kolayca kullanabilirsiniz:

HTML

@using Microsoft.Extensions.Localization

@inject IStringLocalizer<SharedResources> localizer

...

<td>

@Request.Status.ToLocalizedString(localizer)

</td>

1. **D. Özet**

|  | ❌ Yanlış Yaklaşım | ✅ Doğru Yaklaşım |
| --- | --- | --- |
| **Enum Tanımı** | public enum Durum { OnayBekliyor, Onaylandi } | public enum RequestStatus { Pending, Approved } |
| **Kod Kullanımı** | if (talep.Durum == Durum.OnayBekliyor) | if (request.Status == RequestStatus.Pending) |
| **API Yanıtı** | { "durum": "OnayBekliyor" } (Hatalı) | { "status": "Pending" } (Doğru) |
| **Arayüzde Çeviri** | (Gerekmez, kod zaten Türkçe) | @Request.Status.ToLocalizedString(localizer) |
| **Sonuç** | Kod sadece Türkçe çalışır. Yeni dil eklemek imkansıza yakındır. | Kod dil-bağımsızdır. Yeni dil eklemek için sadece yeni bir .resx dosyası oluşturmak yeterlidir. |

## Özel Domain Hataları ve Domain Services

Bu eksikler yeni Entity veya Interface'ler değil, mevcut varlıklarımızı koruyacak ve karmaşık iş kurallarını yönetecek özel sınıflardır.

* **Domain Katmanındaki Eksik Kod Dosyası Türleri**

| Eksik Dosya Türü / Kavram | Örnek Dosya Adı (.cs) | Amaç ve Neden Gerekli? |
| --- | --- | --- |
| **1. Özel Domain Hataları (Custom Domain Exceptions)** | LeaveRequestDateInvalidException.cs  PersonnelAlreadyTerminatedException.cs | **Amaç:** Bir iş kuralı ihlal edildiğinde, throw new Exception("Hata oldu") gibi genel bir hata fırlatmak yerine, o duruma özel, anlamlı bir hata fırlatmaktır  **Neden Gerekli?:** Application katmanı, catch (LeaveRequestDateInvalidException ex) gibi spesifik bir hata yakalayarak kullanıcıya "Başlangıç tarihi bitiş tarihinden sonra olamaz" gibi net bir mesaj gösterebilir. Bu, hata yönetimini son derece güçlü ve anlaşılır kılar. |
| **2. Domain Servisleri (Domain Services)** | LeaveCollisionValidationService.cs  SeniorityCalculationService.cs | **Amaç:** Bazı iş kuralları veya operasyonlar, tek bir varlığın (Entity) sorumluluğuna girmez. Örneğin, yeni bir izin talebinin, personelin ekibindeki diğer izinlerle çakışıp çakışmadığını kontrol etmek, ne Personnel ne de LeaveRequest varlığının tek başına görevidir. Bu tür, birden fazla varlığı ilgilendiren mantıklar Domain Service'ler içinde yaşar.  **Neden Gerekli?:** Varlık sınıflarınızı (Entity) şişirmekten ve sorumluluklarını aşmaktan kurtarır. Karmaşık iş mantığını merkezi ve yeniden kullanılabilir bir hale getirir. |

* **Domain Servisi Nasıl Çalışır? (Somut Örnek)**

Application katmanındaki CreateLeaveRequestCommandHandler'ınız, yeni bir izin talebini kaydetmeden önce bu servisi çağırarak iş kuralını doğrular:

C#

// --- CreateLeaveRequestCommandHandler.cs içinde ---

private readonly ILeaveRequestRepository \_leaveRequestRepository;

private readonly LeaveCollisionValidationService \_leaveCollisionValidationService;

// ... (Constructor ile inject edilirler)

public async Task<int> Handle(CreateLeaveRequestCommand request, CancellationToken cancellationToken)

{

// ... (Personel bulunur)

var newLeaveRequest = new LeaveRequest(...);

// 1. Domain Servisi'ni kullanarak iş kuralını kontrol et

var isCollision = await \_leaveCollisionValidationService.IsCollisionDetectedAsync(newLeaveRequest);

if (isCollision)

{

// 2. Özel Domain Hatası fırlat

throw new LeaveRequestCollisionException("Bu tarihlerde ekipte başka bir izin bulunmaktadır.");

}

// 3. Her şey yolundaysa kaydet

await \_leaveRequestRepository.AddAsync(newLeaveRequest);

return newLeaveRequest.Id;

}

**Sonuç olarak:** Bu iki dosya türünü (Custom Exceptions ve Domain Services) de planınıza dahil ettiğinizde, Domain katmanınız sadece bir veri modeli olmaktan çıkıp, projenizin iş kurallarını aktif olarak uygulayan ve koruyan, yaşayan bir çekirdek haline gelecektir.

## Docs Klasöründe Takipler

* Adrs Klasörü
  1. Mimari değişiklikler gibi önemli değişiklik kararları, (Gerekçeleri, karar alma bilgileri, avantaj ve dezavantajları) Kayıt altına alınır.
  2. Bu kayıtlar adrs klasöründe depolanır.

## Web Uygulamasında Modal Kullanımı (Hybrid Yaklaşım)

Bu, projenizin **kullanıcı deneyimini** temelden etkileyecek, son derece yerinde ve önemli bir mimari konudur. Bu konuyu tüm yönleriyle ele alıp, projemiz için en doğru kararı verelim.

Önerdiğim iki yapı (\_ModalPartial.cshtml ve ayrı Create/Edit sayfaları) birbiriyle çelişiyor gibi görünse de, aslında modern uygulamalarda sıklıkla birlikte kullanılan ve farklı amaçlara hizmet eden iki farklı desendir.

Aşağıda her iki yaklaşımı da artıları ve eksileriyle tartışıp, **EduHR-2025** için en uygun olan **hibrit (karma)** modeli önereceğim.

**Yaklaşım 1: Tek Sayfa + Modal Kullanımı (SPA Benzeri Deneyim)**

Bu yaklaşımda, kullanıcı Index (liste) sayfasından hiç ayrılmaz. "Yeni Ekle", "Düzenle" veya "Detay Gör" gibi tüm işlemler, sayfanın üzerinde açılan modal pencereler (pop-up) içinde yapılır.

* **Avantajları:**
  + **Akıcı Kullanıcı Deneyimi:** Tam sayfa yenilemesi olmadığı için arayüz çok daha hızlı ve akıcı hissettirir. Kullanıcı, arkadaki listedeki yerini (scroll pozisyonunu) kaybetmez.
  + **Daha Az Veri Transferi:** Her işlemde tüm sayfa (\_Layout vb.) yeniden yüklenmediği için sunucuya daha az yük biner.
  + **Modern Hissiyat:** Günümüzdeki modern web uygulamalarının (SPA - Single Page Application) çalışma şekline benzer bir deneyim sunar.
* **Dezavantajları:**
  + **Artan Karmaşıklık:** Tüm modal'ların durumunu, form verilerini, validasyonlarını ve arayüz güncellemelerini yönetmek için çok daha fazla ve karmaşık JavaScript kodu yazmak gerekir.
  + **Kısıtlı Alan:** Karmaşık ve çok adımlı formları (örneğin detaylı bir personel ekleme formu) küçük bir modal pencereye sığdırmak zor olabilir ve kullanıcıyı bunaltabilir.
  + **URL ve Tarayıcı Geçmişi Sorunları:** Her işlem aynı URL üzerinde gerçekleştiği için, kullanıcı "Düzenleme" ekranını **yer imlerine ekleyemez** veya bu ekranın linkini bir başkasına **gönderemez**. Tarayıcının "Geri" butonu beklendiği gibi çalışmayabilir.

**Yaklaşım 2: Ayrı Sayfalar (Geleneksel Çok Sayfalı Uygulama)**

Bu yaklaşımda, her ana işlem için ayrı bir sayfa ve ayrı bir URL bulunur. Kullanıcı, "Yeni Ekle" butonuna tıkladığında /tenants/create gibi yeni bir sayfaya yönlendirilir.

* **Avantajları:**
  + **Basitlik ve Öngörülebilirlik:** Geliştirme süreci daha basittir. Her sayfanın sorumluluğu nettir ve daha az karmaşık JavaScript gerektirir.
  + **Benzersiz ve Paylaşılabilir URL'ler:** Her işlemin (/tenants/edit/123 gibi) kendi URL'i olduğu için kullanıcılar bu linkleri kaydedebilir, paylaşabilir ve tarayıcı geçmişini sorunsuzca kullanabilir. Bu, özellikle ekip çalışması için çok önemlidir.
  + **Karmaşık Formlar İçin İdeal:** Uzun ve detaylı formlar için tam sayfa alanı sunarak daha iyi bir kullanıcı deneyimi sağlar.
  + **API-First Mimarisine Uyumlu:** Her sayfanın PageModel'ı (.cshtml.cs), kendi ihtiyacı olan API istemcisini çağırır. Bu, sorumlulukların net bir şekilde ayrıldığı temiz bir yapıdır.
* **Dezavantajları:**
  + **Daha Yavaş Deneyim:** Her tıklamada tam sayfa yenilemesi gerektiği için, modal yaklaşımına göre daha yavaş ve daha az akıcı hissedilebilir.
  + **Kullanıcı Bağlamının Kaybolması:** Kullanıcı, listeden bir kaydı düzenlemek için yeni bir sayfaya gittiğinde ve geri döndüğünde, listedeki scroll pozisyonu gibi bağlamı kaybeder.

**Öneri: EduHR-2025 İçin Hibrit (Karma) Yaklaşım**

Projenizin hem modern bir kullanıcı deneyimi sunması hem de yönetilebilir ve kullanışlı olması için en iyi çözüm, bu iki yaklaşımı birleştiren **hibrit bir model** kullanmaktır.

**Temel Kural:**

* **Karmaşık ve "hedef" olan işlemler için ayrı sayfalar kullanın.**
* **Hızlı ve "yardımcı" işlemler için modal pencereler kullanın.**

Bu durumda, Admin/Tenants klasörü için yaptığım iki öneri de geçerlidir ve birbirini tamamlar:

| İşlem | **Önerilen Yöntem** | Gerekçe |
| --- | --- | --- |
| **Listeleme** (List) | Ayrı Sayfa (Index.cshtml) | Ana giriş noktasıdır. Filtreleme, arama ve sayfalama gibi özellikler için tam sayfa en iyi deneyimi sunar. |
| **Ekleme** (Create) | Ayrı Sayfa (Create.cshtml) | Kiracı ekleme formu gelecekte detaylanabilir. Kendi URL'ine sahip olması önemlidir. |
| **Düzenleme** (Edit) | Ayrı Sayfa (Edit.cshtml) | Ekleme işlemiyle aynı gerekçelere sahiptir. |
| **Detay Görme** (Details) | **Modal Pencere** (\_ModalPartial) | Kullanıcıyı listeden ayırmadan, bir kaydın anahtar bilgilerini hızlıca görmesi için idealdir. Hızlı ve akıcı bir deneyim sunar. |
| **Silme** (Delete) | **Modal Pencere** (\_ModalPartial) | "Bu kaydı silmek istediğinizden emin misiniz?" gibi basit bir onay işlemi için tam sayfa açmak gereksizdir. Modal, bu tür hızlı onaylar için standarttır. |

**Sonuç olarak:** Projeniz için hem ayrı Index, Create, Edit sayfalarını planlamamız, hem de Shared klasöründe hızlı işlemler için bir \_ModalPartial.cshtml bulundurmamız **en doğru ve en iyi uygulama örneğidir.**

## Web Uygulamasında Çoklu Pencere Sorunu

Harika bir soru. Bu, modern web uygulamalarında sıklıkla göz ardı edilen ancak hem kullanıcı deneyimi hem de güvenlik için son derece kritik bir konudur. Sizin API-First mimariniz ve JWT Token kullanımınız bu konuyu daha da önemli hale getiriyor.

Konuyu güvenlik, etkiler, avantajlar/dezavantajlar ve projeniz için öneriler olarak dört başlıkta inceleyelim.

**1. Temel Mekanizma: Tarayıcılar Nasıl Çalışır?**

Öncelikle temel prensibi anlamalıyız: **Aynı tarayıcıda açılan aynı domaine ait tüm sekmeler, aynı "cookie jar" (çerez deposunu) paylaşır.**

Bu şu anlama gelir: Kullanıcı, A sekmesinde EduHR.WebApp'e giriş yaptığında, API'den aldığı ve kimliğini kanıtlayan JWT Token'ı tarayıcıya bir cookie olarak kaydedilir. Kullanıcı aynı tarayıcıda B sekmesini açıp EduHR.WebApp'e gittiğinde, tarayıcı bu cookie'yi otomatik olarak yeni sekmeye de gönderir. Sonuç olarak, kullanıcı **her iki sekmede de otomatik olarak giriş yapmış olur.** Bu, değiştiremeyeceğimiz, yönetmemiz gereken bir tarayıcı davranışıdır.

**2. Güvenlik Öncelikli Konular ve Önlemler**

Bu paylaşımlı oturum durumu, dikkatle yönetilmesi gereken birkaç önemli güvenlik riskini beraberinde getirir:

| Risk | Açıklama | EduHR-2025 İçin Çözüm Önerisi |
| --- | --- | --- |
| **Veri Tutarlılığı ve Kaybı** | Kullanıcı, A sekmesinde bir personelin profilini düzenlemek için açar. Aynı anda B sekmesinde de aynı personelin profilini açar. A sekmesinde personelin maaşını güncelleyip kaydeder. Sonra B sekmesine geçer ve orada gördüğü **eski veriye göre** personelin departmanını değiştirip kaydeder. **Sonuç: A sekmesinde yapılan maaş değişikliği ezilir ve kaybolur.** | **İyimser Eşzamanlılık (Optimistic Concurrency):** Bu en kritik önlemdir. API'den bir kayıt (örn: personel) çekilirken, rowVersion gibi bir versiyon numarası da alınır. Kaydı güncellemek için API'ye istek gönderildiğinde, bu versiyon numarası da geri gönderilir. API, veritabanındaki versiyon numarası ile gelen numaranın aynı olup olmadığını kontrol eder. Eğer farklıysa, "Bu kayıt siz düzenlemeye başladıktan sonra başka bir kullanıcı tarafından güncellenmiş" anlamına gelir, işlem reddedilir (HTTP 409 Conflict hatası döner) ve kullanıcıdan sayfayı yenilemesi istenir. Bu, veri kaybını kesin olarak önler. |
| **Siteler Arası İstek Sahtekarlığı (CSRF)** | Kullanıcı bir sekmede EduHR'a giriş yapmışken, başka bir sekmede kötü niyetli bir e-postadan gelen bir linke tıklar. Bu link, EduHR API'nize "bir personeli sil" gibi bir istek gönderir. Tarayıcı, kimlik doğrulama cookie'sini bu isteğe otomatik olarak ekleyeceği için, API bu isteği geçerli sanabilir ve işlemi gerçekleştirebilir. | **Anti-CSRF Token Mekanizması:** ASP.NET Core'un yerleşik Anti-Forgery Token desteği kullanılmalıdır. API, her oturum için ek bir güvenlik token'ı üretir. WebApp, POST, PUT, DELETE gibi veri değiştiren tüm isteklerinde bu token'ı da (genellikle HTTP başlığı olarak) göndermek zorunda kalır. API, hem oturum cookie'sini hem de bu ek Anti-CSRF token'ını doğrulayarak isteğin gerçekten sizin uygulamanızdan geldiğinden emin olur. |
| **Oturumun Sonlandırılması** | Kullanıcı A sekmesinde "Çıkış Yap" butonuna basar. Cookie silinir. Ancak B sekmesi hala açık ve "giriş yapılmış" gibi görünmektedir. Kullanıcı B sekmesinde bir işlem yapmaya çalıştığında ne olacak? | **API 401 Yanıtlarının Yönetimi:** En standart çözüm, B sekmesindeki kullanıcının yapacağı ilk API isteğinin 401 Unauthorized hatası almasıdır. WebApp projenizdeki merkezi api.service.js dosyası, tüm API isteklerini yönettiği için, gelen yanıtın 401 olup olmadığını kontrol etmeli ve eğer öyleyse, kullanıcıyı otomatik olarak giriş sayfasına yönlendirmelidir. Bu, kullanıcının geçersiz bir sayfada kalmasını engeller. |

**3. Etkiler, Avantajlar ve Dezavantajlar**

* **Avantaj:** En büyük avantaj, **kullanıcı deneyimidir**. Kullanıcılar, modern uygulamalarda birden fazla sekme açıp oturumlarının devam etmesini bekler. İki farklı personel kaydını karşılaştırmak veya bir listedeki kayda bakarken başka bir işlem yapmak için bu esneklik şarttır. Bu davranışı engellemek, kullanıcı için büyük bir hayal kırıklığı olurdu.
* **Dezavantaj / Etki:** En büyük dezavantaj, **geliştirici için artan karmaşıklıktır**. Yukarıda belirtilen güvenlik ve veri tutarlılığı önlemlerini (Optimistic Concurrency, Anti-CSRF) en başından itibaren planlamak ve uygulamak bir zorunluluktur. Bu durum, "sadece çalışıyor" demekten öteye geçip, "güvenli ve tutarlı çalışıyor" demeyi gerektirir.

**4. EduHR-2025 İçin Özet Yaklaşım**

Projenizde bu durumu doğru yönetmek için şu adımları en başından planlamalısınız:

1. **JWT Token'ı Güvenli ve HttpOnly Bir Cookie'de Saklayın:** Bu, token'ınızın JavaScript tarafından çalınmasını (XSS) engeller.
2. **Anti-CSRF Korumasını Aktif Edin:** Veri değiştiren tüm API endpoint'lerinizi (POST, PUT, DELETE) bu mekanizma ile koruyun.
3. **İyimser Eşzamanlılık (Optimistic Concurrency) Uygulayın:** Düzenleme ihtimali olan tüm ana varlıklarınıza (Personnel, Department vb.) bir rowVersion alanı ekleyin ve UPDATE işlemlerinde bu alanı kontrol edin.
4. **Merkezi 401 Hata Yönetimi Yapın:** İstemci tarafındaki (WebApp) JavaScript kodunuzda, API'den 401 hatası geldiğinde kullanıcıyı otomatik olarak giriş sayfasına yönlendiren bir mekanizma kurun.

## Subscriptions Süreci Nasıl Yönetilecek

### Kredi Kartı ile Online Abonelik

EduHR Projesi için başlangıçta bu yöntem seçilmemiştir. Ancak ileride seçilmesi durumda online ödeme ile ilgili entegrasyon gerekecektir.

Yurtdışı için Stripe (Türkiyede şuan itibari ile geçersiz) alternatifleri.

Yurtiçinde Iyızıco vb.

### SuperAdmin Tarafından Oluşturulan Abonelik

Bu, projenin işleyişi ve kullanıcı deneyimi açısından çok önemli bir soru. Bu süreci nasıl işleteceğimizi, en iyi uygulama örneklerini de dikkate alarak detaylı bir şekilde değerlendirelim.

**Değerlendirme ve Düşünce Süreci**

Bu manuel abonelik sürecini yönetmek için iki temel yaklaşımımız var:

1. **Ayrı Sayfalar Yaklaşımı:** Pages/Admin/Subscriptions gibi tamamen yeni bir klasör ve bu klasörün altında Index, Create, Edit gibi yeni sayfalar oluşturmak.
2. **Entegre Yaklaşım:** Yeni sayfa oluşturmak yerine, bu işlevselliği mevcut Pages/Admin/Tenants sayfalarının içine entegre etmek.

**Karşılaştırma ve Sonuç**

| Kriter | Ayrı Sayfa Yaklaşımı | **Entegre Yaklaşım (Önerilen)** |
| --- | --- | --- |
| **Kullanıcı Akışı** | **Kopuk ve Verimsiz.** Superadmin'in bir müşterinin aboneliğini yönetmek için önce "Kiracılar" listesine gitmesi, sonra ayrı bir "Abonelikler" listesine gidip o müşteriyi tekrar bulması gerekir. | **Akıcı ve Sezgisel.** Superadmin'in doğal iş akışına uyar. "A Müşterisi" detay sayfasına gider ve doğrudan o müşteriye ait abonelikleri tek bir ekranda yönetir. |
| **Geliştirme Karmaşıklığı** | **Daha Fazla Dosya.** Yeni klasörler, yeni menüler, yeni sayfalar (Index, Create, Edit) ve bu sayfalar için yeni ViewModel, ApiClient ve JavaScript dosyaları oluşturmak gerekir. | **Daha Az ve Odaklı Geliştirme.** Yeni sayfa oluşturmak gerekmez. Sadece mevcut Tenants/Details.cshtml sayfası zenginleştirilir. Yeni abonelik ekleme gibi işlemler, daha önce planladığımız \_ModalPartial.cshtml kullanılarak bir pop-up pencere ile kolayca çözülebilir. |
| **En İyi Pratik** | Veri odaklıdır (tüm abonelikleri bir arada gösterir). | **Kullanıcı ve iş akışı odaklıdır.** |

**Öneri ve Süreç Akışı**

Bu süreci yönetmek için **yeni bir klasör ve yeni sayfalara ihtiyacınız yok**. En doğru, en modern ve kullanıcı dostu yaklaşım, bu işlevselliği mevcut **Pages/Admin/Tenants** yönetimi akışına entegre etmektir.

İşletmeniz gereken süreç şu şekilde olmalıdır:

1. **Merkez Noktası: Kiracı Detay Sayfası**
   * Superadmin, ana menüden "Kiracılar" listesine (/Admin/Tenants/Index) gider.
   * İşlem yapmak istediği kiracının yanındaki "Detaylar" veya "Yönet" butonuna tıklar ve Admin/Tenants/Details.cshtml sayfasına yönlendirilir.
2. **Abonelik Yönetimi Sekmesi**
   * Kiracı Detay sayfasının içinde "Genel Bilgiler", "**Abonelikler**", "Kullanıcılar" gibi sekmeler bulunur.
   * Superadmin, "**Abonelikler**" sekmesine tıklar. Bu sekmede, o kiracıya ait geçmiş ve mevcut tüm aboneliklerin bir listesi (plan, başlangıç/bitiş tarihi, durum: *Ödendi/Ödeme Bekliyor*) yer alır.
3. **İşlemlerin Modal Pencere ile Yapılması**
   * Abonelik listesinin üzerinde **"Yeni Abonelik Ekle"** butonu bulunur. Bu butona tıklandığında, tam sayfa yenilemesi olmadan, \_ModalPartial.cshtml kullanılarak bir **modal (pop-up) pencere** açılır.
   * Bu modal penceredeki formda Plan seçimi, başlangıç/bitiş tarihi gibi bilgiler girilir ve kaydedilir. Kayıt sonrası modal kapanır ve arkadaki abonelik listesi otomatik olarak güncellenir.
   * "Ödeme Bekliyor" durumundaki bir aboneliğin yanında **"Ödendi Olarak İşaretle"** butonu bulunur. Bu butona basıldığında yine bir onay modal'ı açılır ve onay sonrası kayıt güncellenir.

**Teknik Gereksinimler**

Bu akışı desteklemek için yeni sayfalara değil, aşağıdaki yeni **API endpoint'lerine** ve **CQRS komutlarına** ihtiyacınız olacak:

* Application Katmanı:
  + CreateSubscriptionForTenantCommand.cs
  + MarkSubscriptionAsPaidCommand.cs
  + GetSubscriptionsForTenantQuery.cs
* Api Katmanı:
  + POST /api/tenants/{tenantId}/subscriptions
  + PUT /api/subscriptions/{subscriptionId}/mark-as-paid
  + GET /api/tenants/{tenantId}/subscriptions

**Sonuç olarak,** bu entegre ve modal tabanlı yaklaşım, hem Superadmin için çok daha verimli bir kullanıcı deneyimi sunar hem de gereksiz dosya ve sayfa karmaşasını önleyerek projenizi daha temiz ve yönetilebilir tutar.

# ÇÖZÜM KLASÖR YAPISI VE YAZILIM PROJELERİ

* docs
* src
* tests
* frontends
* deployment

## DOCS

## SRC (KATMANLAR ve ASP.NET CORE PROJELERİ)

### Katmanlar

1. Domain
   1. EduHR.Domain Projesi
2. Application
   1. EduHR.Application Projesi
3. Infrastructure
   1. EduHR.Infrastructure Projesi
   2. EduHR.Localization Projesi
4. Presentation
   1. EduHR.Api Projesi
   2. EduHR.Webapp Projesi
5. Common
   1. EduHR.Common Projesi

### EduHR.Domain Projesi

### EduHR.Application Projesi

* Application Katmanı Kontrol Listesi
* **[✓] İş Akışları (Motor):** Features klasörümüz, tüm kullanıcı senaryoları için Command, Query, Handler ve Response modellerini eksiksiz bir şekilde tanımlıyor.
* **[✓] Veri Kontratları (İletişim Dili):** DTOs klasörümüz, API'nin dış dünya ile konuşacağı tüm veri modellerini (okuma, yazma, güncelleme, yetkilendirme) içeriyor.
* **[✓] Dış Dünya Bağlantıları (Sözleşmeler):** Interfaces klasörümüz, Infrastructure katmanının uygulayacağı tüm dış servisler (IEmailService, IFileStorageService vb.) için gerekli sözleşmeleri tanımlıyor.
* **[✓] Girdi Güvenliği (Kapıdaki Güvenlik):** Validators klasörümüz, sisteme giren verilerin daha işlenmeden doğrulanması için gereken tüm kuralları içeriyor.
* **[✓] Veri Dönüşümü (Tercüman):** Mappings klasörümüz, Domain varlıkları ve DTO'lar arasındaki tüm dönüşümleri yönetecek merkezi MappingProfile dosyasını içeriyor.
* **[✓] Merkezi Davranışlar (Kalite Kontrol Bandı):** Behaviours klasörümüz, loglama, validasyon, performans ve güvenlik gibi tüm merkezi ve yeniden kullanılabilir kuralları barındırıyor.
* **[✓] Akış Hataları (Alarm Sistemi):** Exceptions klasörümüz, uygulama akışında oluşabilecek NotFound veya ForbiddenAccess gibi durumları yönetmek için gereken tüm hata sınıflarını içeriyor.

Özetle, Application katmanının her bir sorumluluğu için gerekli olan klasör ve dosya türlerini eksiksiz bir şekilde planlamış bulunuyoruz. Bu plan, projenizin iş mantığını barındıracak sağlam bir temel oluşturuyor.

## TESTS

## FRONTEND – (MOBILE -MARKETING WEBSITE)

## DEPLOYMENT

# SRC KLASÖRÜ

## 📁 1\_Domain

### EduHR.Domain Proje Klasörü

#### Commons Klasörü

#### Entities Klasörü

#### Interfaces Klasörü

#### Enums Klasörü

#### Events Klasörü

#### Exceptions Klasörü

#### Services Klasörü

## 📁 2\_Application

### EduHR.Application

#### Features Klasörü

Bu katmanın kalbidir. Her bir işlevsellik (personel ekleme, izin talep etme vb.), kendi alt klasöründe (örn: Features/Personnel/) gruplanır. Bu klasörlerin içinde Commands, Queries ve bunları işleyen Handlers ve Response yer alır.

##### Commands

##### Queries

##### Handlers

##### Response

#### DTOs Klasörü

~~Veri Aktarım Nesneleri. Domain katmanındaki varlıkların, Api üzerinden dış dünyaya sunulacak veya arayüzlerden alınacak olan basit veri kopyalarıdır. (Örn: PersonnelDto.cs, LeaveRequestDto.cs)~~

***Bu klasör ve içeriği kaldırıldı. Gerekçe için bakınız SRC/5\_Common/DTOs Klasörü açıklamaları***

#### Interfaces Klasörü

Dış Servis Sözleşmeleri. Infrastructure katmanında uygulanacak olan servislerin (e-posta gönderme, mevcut kullanıcı bilgisi vb.) arayüzlerini barındırır. (Örn: IEmailService.cs, ICurrentUserService.cs)

#### Validators Klasörü

Doğrulama Kuralları. Features klasöründeki Command'lara gelen verilerin geçerliliğini (örn: bir alanın boş olmaması, e-posta formatı vb.) kontrol eden FluentValidation sınıflarını içerir. (Örn: CreatePersonnelCommandValidator.cs)

FluentValidation kütüphanesini kullanarak, her bir Command için ayrı bir doğrulama sınıfı oluşturmak en iyi pratiktir. Bu, doğrulama kurallarınızı merkezi, okunabilir ve test edilebilir hale getirir.

#### Mappings Klasörü

Dönüşüm Profilleri. Domain varlıkları ile DTO'lar arasındaki veri dönüşümlerini otomatik hale getiren AutoMapper profillerini barındırır. (Örn: MappingProfile.cs)

#### Behaviours Klasörü

Davranış Hattı (Pipeline Behaviours). Tüm Command ve Query'lerin etrafına sarılabilen, kesişen (cross-cutting) endişeleri yöneten kodlardır. Gelen her isteği otomatik olarak loglamak veya doğrulamadan geçirmek için kullanılır. (Örn: ValidationBehaviour.cs)

* **Mimari İhtiyaçların Karşılanması**

Oluşturduğumuz liste, artık sadece tek tek dosyalardan ibaret değil; projenizin temel "kesişen endişelerini" (cross-cutting concerns) yönetmek için eksiksiz bir **araç takımı** haline geldi. Her bir davranış, mimarinizin farklı bir ihtiyacını karşılıyor:

* **Güvenlik ve Doğrulama (Giriş Kapısı):**
  + ValidationBehaviour: Gelen verinin formatının doğru olduğunu garanti eder.
  + AuthorizationBehaviour: İşlemi yapanın yetkisi olduğunu garanti eder.
* **Sağlamlık ve Hata Yönetimi (Güvenlik Ağı):**
  + UnhandledExceptionBehaviour: Beklenmedik hataların kaybolmamasını sağlar.
  + TransactionBehaviour: Veri bütünlüğünü korur.
* **Performans ve Verimlilik (Optimizasyon Araçları):**
  + PerformanceBehaviour: Yavaş işlemleri proaktif olarak tespit eder.
  + CachingBehaviour: Veritabanı yükünü azaltır ve yanıt sürelerini iyileştirir.
* **İzlenebilirlik ve Yönetim (Kontrol Kulesi):**
  + LoggingBehaviour: Sistemde olan biten her şeyi kaydeder.
  + TenantResolutionBehaviour: Çoklu kiracılıkta doğru verinin işlendiğinden emin olur.

#### Exceptions Klasörü

Uygulama Seviyesi Hatalar. Bu katmana özgü hata sınıflarını içerir. En yaygın örnek, bir sorgu sonucunda kayıt bulunamadığında fırlatılan NotFoundException.cs'tir.

## 📁 3\_Infrastructure

### EduHR.Infrastructure

#### Persistence

Bu, veritabanı ile ilgili tüm kodların bulunduğu ana klasördür. Kendi içinde daha organize olmak için alt klasörlere ayrılır:

##### Contexts

Entity Framework Core DbContext sınıfını (ApplicationDbContext.cs) barındırır.

Contexts klasöründe ApplicationDbContext.cs dosyasından başka bir dosyaya **ihtiyacınız yoktur**.

* Neden Tek Dosya Yeterli?

Tek bir ApplicationDbContext dosyası, projeniz için hem en doğru hem de en standart yaklaşımdır. Bunun birkaç temel nedeni vardır:

1. **Tek Veritabanı:** Projeniz, tüm verilerini tek bir PostgreSQL veritabanında saklamaktadır. Entity Framework Core'da, tek bir veritabanı ile iletişim kurmak için tek bir DbContext sınıfı yeterlidir.
2. **Unit of Work Prensibi:** ApplicationDbContext sınıfınız, projeniz için "Unit of Work" (İş Birimi) deseni görevini görür. Yani, bir işlem sırasında yapılan tüm değişikliklerin (ekleme, güncelleme, silme) bir bütün olarak ele alınmasını ve birlikte kaydedilmesini sağlar.
3. **Çoklu Kiracılık Modeli:** Seçtiğiniz "Tek Veritabanı - Ortak Şema" çoklu kiracılık modeli, tüm kiracıların verileri aynı tabloda olduğu için, bu verileri TenantId ile ayıracak olan filtreleme mantığının da yine bu tek DbContext içinde merkezi olarak yönetilmesini gerektirir.

* Ne Zaman Birden Fazla DbContext Gerekirdi?

Sadece aşağıdaki gibi daha karmaşık ve projenizin kapsamında olmayan senaryolarda birden fazla DbContext dosyasına ihtiyaç duyulurdu:

* **Farklı Veritabanları:** Projeniz aynı anda hem PostgreSQL'e hem de tamamen farklı bir SQL Server veritabanına bağlanmak zorunda olsaydı.

**Ayrık Bounded Context'ler (DDD):** Projeniz, birbiriyle neredeyse hiç konuşmayan çok büyük ve ayrık modüllere (örneğin, "İK Yönetimi" ve "Okul Kantin Stok Yönetimi") sahip olsaydı ve her modülün kendi ayrı veritabanı bulunsaydı.

##### Repositories

Domain katmanında tanımlanan IPersonnelRepository gibi arayüzlerin, PostgreSQL veritabanında çalışan somut kodlarını (PersonnelRepository.cs) içerir.

##### Configurations

Her bir Entity için veritabanı tablo eşleştirmelerini (mapping) yapan (PersonnelConfiguration.cs gibi) sınıfları içerir.

Her bir ...Configuration.cs dosyasının içinde, tablo ve kolon adlarını belirlemenin yanı sıra, aşağıdaki gibi kritik veritabanı kurallarını da tanımlarsınız:

| Ek Sorumluluk | Amaç ve Gerekçe | Örnek Kod (PersonnelConfiguration.cs içinde) |
| --- | --- | --- |
| **Index'ler (Indexes)** | Sık sorgulanan kolonlara index oluşturmak, veritabanı okuma performansını ciddi şekilde artırır. Bu, uygulamanızın hızını doğrudan etkiler. | builder.HasIndex(p => p.LastName); <br> *(Personelleri soyadına göre hızlı bulmak için)* |
| **Benzersiz Kısıtlamalar (Unique Constraints)** | Bir kolonun (veya birden fazla kolonun) benzersiz değerler içermesini veritabanı seviyesinde garanti altına alır. Bu, veri tutarlılığı için kritik öneme sahiptir. | builder.HasIndex(p => p.Tckn).IsUnique(); <br> *(İki personelin aynı TCKN'ye sahip olmasını engeller)* |
| **Veri Tohumlama (Data Seeding)** | Uygulama ilk kurulduğunda, bazı tabloların başlangıç verilerine ihtiyacı olabilir. Bu verileri doğrudan ilgili varlığın yapılandırma dosyası içinde tanımlayabilirsiniz. | (Örnek LeaveTypeConfiguration.cs içinde)<br>builder.HasData( new LeaveType { Id = 1, Name = "Yıllık İzin" }); |
| **Kolon Detayları** | Her bir kolonun veri tipini, uzunluğunu, varsayılan değerini veya NULL içerip içeremeyeceğini detaylı olarak belirlersiniz. | builder.Property(p => p.FirstName).HasMaxLength(100).IsRequired(); |

* Özetle:

Mevcut dosyalarımızın içini yukarıda bahsettiğimiz HasIndex, HasData, HasMaxLength gibi önemli kodlarla zenginleştirmemiz gerekiyor.

##### Migrations

EF Core tarafından otomatik olarak oluşturulan ve veritabanı şeması değişikliklerini içeren dosyaları barındırır.

#### Services

Application katmanında tanımlanan dış servis arayüzlerinin somut sınıflarını içerir. (Örn: EmailService.cs, DateTimeService.cs, FileStorageService.cs)

* **Neden Liste Bu Kadar Kısa ve Eksiksiz?**

Bunun nedeni, Infrastructure katmanının kendi içinde de sorumluluklara göre son derece düzenli bir şekilde alt klasörlere ayrılmış olmasıdır. "Servis" olarak düşünülebilecek diğer tüm kodlar, daha spesifik görevleri olan başka klasörlerde yer almaktadır.

Infrastructure katmanındaki kodların tam dağılımı şöyledir:

| Sorumluluk | Bulunduğu Klasör | Örnek Dosyalar |
| --- | --- | --- |
| **1. Veritabanı ile Konuşmak** | Persistence/Repositories/ | PersonnelRepository.cs, LeaveRequestRepository.cs ve diğer **~20+** adet repository sınıfı. |
| **2. Veritabanı Tablolarını Yapılandırmak** | Persistence/Configurations/ | PersonnelConfiguration.cs, DepartmentConfiguration.cs ve diğer **~40+** adet yapılandırma sınıfı. |
| **3. Kimlik Doğrulamayı Yönetmek** | Identity/ | IdentityService.cs, JwtTokenGenerator.cs gibi sınıflar. ICurrentUserService arayüzünün uygulaması da burada yer alır (CurrentUserService.cs). |
| **4. Dış Dünya Servislerini Uygulamak** | **Services/** | EmailService.cs, FileStorageService.cs, DateTimeService.cs ve NotificationService.cs. |

**Özetle:**

Services klasöründeki dosya sayısının az olması bir eksiklik değil, aksine Infrastructure katmanınızın sorumluluklarına göre ne kadar iyi organize edildiğinin bir göstergesidir.

* **Repository'ler** veritabanı işlemlerini,
* **Identity** sınıfları kullanıcı güvenliğini,
* **Services** klasörü ise e-posta, dosya sistemi gibi diğer tüm dış dünya operasyonlarını yönetir.

Application katmanında tanımlanan her bir dış servis arayüzü (ICurrentUserService hariç, o Identity klasöründe) için bir somut uygulama Services klasöründe mevcuttur.

#### Identity

Kullanıcı yönetimi, kimlik doğrulama ve yetkilendirme ile ilgili kodları barındırır. ASP.NET Core Identity'nin özelleştirilmesi, JWT token üretimi ve ICurrentUserService arayüzünün somut uygulaması burada yer alır.

* **Bu Dosyaların Gerektirdiği Diğer Önemli Değişiklikler**

Bu klasördeki dosyaların doğru çalışabilmesi için projenizin diğer katmanlarında da birkaç önemli düzenleme yapılması gerekir:

1. **Domain Katmanı:**
   * Entities/User.cs sınıfı, ASP.NET Core Identity'nin özelliklerinden faydalanmak için IdentityUser<int> sınıfından miras almalıdır.
   * Entities/Role.cs sınıfı, IdentityRole<int> sınıfından miras almalıdır.
2. **Infrastructure Katmanı:**
   * Persistence/Contexts/ApplicationDbContext.cs sınıfı, normal DbContext yerine IdentityDbContext<User, Role, int> sınıfından miras almalıdır. Bu, Identity tablolarının EF Core tarafından yönetilmesini sağlar.
3. **Application Katmanı:**
   * Interfaces klasörüne, Infrastructure/Identity/IdentityService.cs sınıfının sözleşmesi olan yeni bir **IIdentityService.cs** arayüzü eklenmelidir. Bu, Application katmanının somut IdentityService sınıfına doğrudan bağımlı olmasını engeller.

ASP.NET Core Identity'yi çalıştırmak için gereken kodlar sadece bu klasörde yer almaz. Bu klasör, 'uygulamayı' barındırırken, 'yapılandırma' ve 'tanımlama' kodları projenin başka yerlerindedir.

Listenin neden bu üç dosyayla tamamlandığını daha iyi anlamak için, Kimlik Yönetimi sisteminin tüm parçalarının mimariye nasıl dağıldığını görelim:

* **Kimlik Yönetimi (Identity) Kodlarının Projedeki Dağılımı**

| Görevi | İlgili Kod / Dosya | Bulunduğu Katman / Proje |
| --- | --- | --- |
| **1. Varlık Tanımları** | User.cs ve Role.cs sınıfları (IdentityUser'dan miras alan) | 📁 1\_Domain / EduHR.Domain |
| **2. Veritabanı Entegrasyonu** | ApplicationDbContext'in IdentityDbContext'ten miras alması | 📁 3\_Infrastructure / EduHR.Infrastructure |
| **3. Servislerin Yapılandırılması** | Program.cs içindeki AddIdentity(), AddAuthentication().AddJwtBearer() metot çağrıları | 📁 4\_Presentation / EduHR.Api |
| **4. Özel Uygulamalar** | IdentityService.cs, JwtTokenGenerator.cs, CurrentUserService.cs | 📁 3\_Infrastructure / EduHR.Infrastructure **(Identity Klasörü)** |

**Özetle:**

Identity klasörünüz, kimlik yönetimiyle ilgili **özel servislerinizi ve mantığınızı** barındırmak için planlanmıştır ve bu görev için listemiz tamdır.

* Kullanıcının ne olduğunu (User.cs) **Domain**'de tanımladık.
* Bu kullanıcının veritabanına nasıl kaydedileceğini **Persistence**'da (DbContext) belirledik.
* Sistemin bu servisleri nasıl kullanacağını **Presentation**'da (Program.cs) yapılandırdık.
* Kullanıcı oluşturma, token üretme gibi **fiili işlemleri yapacak olan kodları** ise bu Identity klasörüne koyduk.

Bu nedenle, mimari olarak doğru ve eksiksiz bir yapı kurmuş durumdayız. Başka bir dosyaya ihtiyacımız yok.

#### BackgroundJobs

Domain katmanında tanımlanan Domain Service'leri veya Application katmanındaki Command/Query'leri, zamanlanmış görev olarak (gece yarısı, saat başı vb.) çalıştıran somut sınıfları içerir. (Örn: LeaveBalanceUpdateJob.cs)

Bu klasör, kullanıcı etkileşimi olmadan, belirli zamanlarda (örn: her gece yarısı) veya belirli olaylar sonrasında (fire-and-forget) arka planda çalışması gereken işlemleri barındırır. Bu işlemler, PgAgent gibi bir zamanlayıcı tarafından tetiklenmek üzere tasarlanır.

* **Neden Liste Artık Tamamlandı?**

Listenin tamamlandığını şu gerekçelerle rahatlıkla söyleyebiliriz:

1. **İş Süreçleri Otomasyonu:** Projenizin temel İK süreçlerini otomatikleştiren en önemli işler planda mevcut (LeaveBalanceUpdateJob, ExpiringDocumentNotifierJob vb.).
2. **Veri Entegrasyonu:** Dış sistemlerle (PDKS) konuşacak olan veri senkronizasyon işi (PDKSDataSyncJob) planlandı.
3. **Süreç Takibi ve Hatırlatmalar:** İş akışlarının aksamasını önleyecek hatırlatma mekanizmaları (OnboardingTaskReminderJob) düşünüldü.
4. **Raporlama ve Bildirim:** Gerekli raporların (DailySummaryReportJob) ve bildirimlerin (NotificationSendingJob) arka planda çalışacak altyapısı planlandı.
5. **Sistem Sağlığı:** Veritabanı bakımı (DatabaseMaintenanceJob) gibi sistemin sağlığını koruyacak işlemler de listeye dahil edildi.

* **Bundan Sonra Yeni Bir İhtiyaç Nasıl Doğar?**

Bu aşamadan sonra eklenecek yeni bir arka plan işi, artık "unuttuğumuz" bir şey değil, projenin gelecekteki gelişimi sırasında ortaya çıkacak **yeni bir iş gereksiniminin** sonucu olacaktır.

**Örneğin:** İleride "Performans değerlendirme dönemi kapandıktan 2 hafta sonra, yöneticilere sonuçlarla ilgili bir analiz raporu gönder" gibi yeni bir kural eklerseniz, o zaman bu kural için yeni bir PerformanceAnalysisReportJob.cs dosyası oluşturmamız gerekir.

Ancak mevcut dokümanlarımıza ve iş akışlarımıza göre **planımız tamdır.**

### EduHR.Localization

Bu projenin görevi, Temiz Mimari prensipleri gereği son derece **odaklanmış ve basittir**: Projenizin "sözlüğü" olmak.

1. **Tek Bir .cs Dosyası Yeterlidir:** SharedResources.cs gibi bir "işaretleyici sınıf", IStringLocalizer<T> servisinin projedeki tüm .resx dosyalarını bulabilmesi için yeterli ve standart bir yöntemdir. Bu projede başka bir C# kod mantığına ihtiyaç yoktur.
2. **.resx Kategorileri Yeterlidir:** Oluşturduğumuz .resx dosya kategorileri (SharedResources, ValidationMessages, ModelProperties, Enums, EmailTemplates), bir iş uygulamasının ihtiyaç duyacağı tüm çeviri metni türlerini kapsamaktadır. Projeniz büyüdükçe, çok spesifik bir modül için yeni bir kaynak dosyası ekleyebilirsiniz (örn: PerformanceReviews.tr.resx), ancak bu, mevcut yapıya eklenecek yeni bir dosya *türü* değildir.
3. **Çeviriyi Kullanan Mantık Burada Yer Almaz:** En önemli nokta budur. Çevirileri kullanan mantık (örneğin daha önce konuştuğumuz EnumExtensions.cs gibi yardımcı sınıflar), bu kaynakları tüketen WebApp veya Application gibi projelerin içinde yer alır. EduHR.Localization projesi ise sadece çevirileri barındıran pasif bir kütüphane olarak kalır.

Özetle, bu proje bilinçli olarak basit tutulmuştur ve mevcut plan, projenizin tüm çok dillilik metinlerini merkezi ve yönetilebilir bir şekilde saklamak için **eksiksizdir.**

#### Resources Klasörü

## 📁 4\_Presentation

### EduHR.Api

* **Neden Plan Tamamlandı?**

Planımızın tamamlandığını teyit etmek için, bir API projesinin temel sorumluluklarını kontrol edelim:

* **[✓] Gelen İstekleri Karşılama:** Controllers klasörümüz, projenin tüm kaynakları için dış dünyaya açılan kapıları eksiksiz bir şekilde tanımlıyor.
* **[✓] Merkezi Kuralları Uygulama:** Middlewares klasörümüz, her isteğin geçtiği hata yönetimi, kiracı ve dil çözümleme gibi temel kontrol noktalarını oluşturuyor.
* **[✓] Projeyi Başlatma ve Yapılandırma:** Program.cs ve appsettings.json dosyaları, projenin hayata geçmesi ve diğer katmanlarla konuşması için gereken tüm temel yapılandırmayı içeriyor.

Ancak, bu yapısal planı tamamlarken aklınızda bulundurmanız gereken ve Program.cs içinde kodlanacak **iki önemli en iyi pratik (best practice)** daha bulunmaktadır. Bunlar yeni bir dosya gerektirmez, ancak API'nizin profesyonelliğini ve kullanılabilirliğini doğrudan etkiler:

1. **API Versiyonlama:** API'nizi en başından itibaren versiyonlamak (/api/v1/personnel gibi), gelecekte mobil uygulama gibi istemcileri bozmadan yeni değişiklikler yapmanızı sağlar. Bu, Program.cs dosyasında yapılacak bir yapılandırma ile kolayca eklenir.
2. **API Dokümantasyonu (Swagger/OpenAPI):** API'nizin kendi kendini belgelemesini sağlayan Swagger entegrasyonu, Program.cs dosyasına eklenmelidir. Bu, WebApp ve MobilApp'i geliştirirken, hangi endpoint'in hangi veriyi alıp döndürdüğünü gösteren interaktif bir arayüz (/swagger adresi gibi) sunar.

Bu iki yapılandırma adımıyla birlikte, EduHR.Api projenizin planlamasının, modern ve sürdürülebilir bir API için gereken tüm bileşenleri içerdiğini ve **tamamlandığını** söyleyebiliriz.

#### Controllers Klasörü

API'nin dış dünyaya açılan kapılarıdır. Her bir ana kaynak (Personel, Departman vb.) için bir Controller sınıfı içerir. Gelen HTTP isteklerini karşılar ve Application katmanına yönlendirir.

* **Neden Plan Artık Tamamlandı?**

Listenin tamamlandığını şundan dolayı rahatlıkla söyleyebiliriz:

1. **Kaynak Odaklı Tasarım:** Oluşturulan liste, RESTful API tasarımının en iyi pratiği olan "kaynak odaklı" yaklaşıma uymaktadır. Domain katmanında tanımladığımız her bir ana iş varlığı (Aggregate Root) için (Personnel, Department, LeaveRequest vb.) bir Controller dosyası planladık. Bu, API'nizin standart, anlaşılır ve yönetilebilir olmasını sağlar.
2. **Sorumluluk Ayrımı:** Her Controller, sadece kendi kaynağıyla ilgili işlemleri yönetir. Örneğin, DepartmentsController sadece departmanlarla ilgilenir, personel işlemleriyle ilgilenmez. Bu, kodun temiz ve odaklanmış kalmasını sağlar.
3. **Özel Durumların Yönetimi:** AuthController ve DashboardController gibi, doğrudan bir varlığa karşılık gelmeyen ancak belirli işlevleri (kimlik doğrulama, birleşik veri sunma) yerine getiren özel Controller'lar da planımıza dahildir.

* **Eksiklik Hissinizin Kaynağı Ne Olabilir?**

Belki de eksiklik hissinizin kaynağı, dosya listesinden ziyade, bu dosyaların **içine** yazılacak olan metotların (API endpoint'lerinin) henüz detaylandırılmamış olmasıdır. Dosya planımız tam, ancak bir sonraki adım bu dosyaları içerikle doldurmaktır.

Örneğin, **PersonnelController.cs** dosyasını oluşturduktan sonra, içine aşağıdaki gibi metotlar (API endpoint'leri) ekleyeceksiniz:

* **GetAll** (GET /api/personnel): Tüm personellerin özet listesini sayfalama ve filtreleme ile döner.
* **GetById** (GET /api/personnel/{id}): Tek bir personelin tüm detaylarını döner.
* **Create** (POST /api/personnel): Yeni bir personel oluşturur.
* **Update** (PUT /api/personnel/{id}): Mevcut bir personeli günceller.
* **Delete** (DELETE /api/personnel/{id}): Bir personeli siler.
* **UploadDocument** (POST /api/personnel/{id}/documents): Belirli bir personele yeni bir doküman yükler.

Özetle, Controllers klasörünüz için **dosya planınız tamdır**. Bundan sonraki adım, her bir controller dosyası için bu örnekteki gibi detaylı endpoint listelerini oluşturmaktır.

#### Middlewares Klasörü

Gelen her bir API isteğinin arasına girerek merkezi işlemler yapan kodları barındırır. (Örn: ExceptionHandlerMiddleware gibi özel hata yakalama mekanizmaları).

* **Mimari Prensip**

Listenin bu üç temel dosyadan oluşması, Temiz Mimari'deki sorumlulukların ayrıştırılması ilkesinin bir sonucudur. Bu üç ara katman, bir API isteğinin en temel ve en dış seviyedeki "kapı bekçileri" olarak görev yapar:

1. **ExceptionHandlerMiddleware:** En dıştaki **güvenlik ağıdır**. Ne olursa olsun, beklenmedik bir hata olursa istemciye asla sunucu detayı sızdırmaz.
2. **TenantResolverMiddleware:** Gelen misafirin (**isteğin**) hangi eve (**kiracıya**) ait olduğunu belirleyen ilk kontroldür. Bu olmadan, hiçbir işlem doğru veri üzerinde çalışamaz.
3. **CultureMiddleware:** Misafirin hangi dilde konuştuğunu (Accept-Language) anlayan ve tüm içerideki iletişimin o dilde olmasını sağlayan çevirmendir.

* **Peki Neden Başka Middleware Yok?**

Projenizde RequestLoggingMiddleware gibi başka ara katmanlar da olabilirdi. Ancak, biz bu tür "istek bazlı" loglama (LoggingBehaviour) ve doğrulama (ValidationBehaviour) gibi işlemleri, daha spesifik olan **MediatR Behaviours** katmanında ele aldık.

Bu, mimari olarak daha temiz bir yaklaşımdır çünkü bu işlemler, isteğin Controller'a ulaşıp anlamlı bir Command/Query'ye dönüştükten sonra daha değerli hale gelir.

Bu üç temel Middleware, API'nize gelen her isteğin **güvenli**, **kiracıya özel** ve **doğru dilde** işlenmesini garanti altına alan sağlam bir giriş kapısı oluşturur. Bu nedenle planımız bu haliyle tam ve doğrudur.

### EduHR.Webapp

#### Pages Klasörü

Kullanıcıların gördüğü her bir web sayfasını (.cshtml) ve o sayfanın mantığını (.cshtml.cs) içeren Razor Pages dosyalarını barındırır. Kendi içinde Personnel, Requests gibi modüllere ayrılır.

Uygulamanın tüm sayfalarını ve yönlendirmelerini (routes) tanımlayan Razor Pages dosyalarını barındırır:

* Her sayfa bir .cshtml (HTML) ve bir .cshtml.cs (C# kod) dosyasından oluşur.
* PageModel'ların (.cshtml.cs) ana görevi, Services klasöründeki API istemcilerini çağırarak veri almak ve bu veriyi ViewModel'ler aracılığıyla arayüze sunmaktır.
* **Shared/** alt klasörü, \_Layout.cshtml gibi ortak şablonları içerir.

##### *Shared Klasörü*

Tüm sayfalar tarafından kullanılan ortak dosyaları ve şablonları barındırır. (Örn: \_Layout.cshtml, \_LoginPartial.cshtml)

##### *Account Klasörü*

Kullanıcı girişi, çıkışı ve yetki kontrolü ile ilgili sayfaları içerir. (Örn: Login.cshtml, Logout.cshtml, AccessDenied.cshtml)

##### *Admin* *Klasörü*

Superadmin'in kullanacağı, kiracı ve abonelik planı yönetimi gibi en üst düzey yönetim sayfalarını içerir. (Örn: Tenants/Index.cshtml, Plans/Index.cshtml)

Alınan karar gereği abone yönetimi de bu klasörde sayfalar ile yapılacaktır.

##### *Settings Klasörü*

Kiracı Yöneticisi'nin (Tenant Admin) kullanacağı, şirket profili, kullanıcı yönetimi ve tatil takvimi gibi ayar sayfalarını barındırır. (Örn: CompanyProfile.cshtml, Users.cshtml)

##### *Organization Klasörü*

Kurumun departman ve pozisyon gibi hiyerarşik yapısının yönetildiği sayfaları içerir. (Örn: Index.cshtml)

*OrganizationApiClient ifadesi ile (DepartmentApiClient.cs ve PositionApiClient.cs kastediliyor)*

*En temiz ve sürdürülebilir mimari, API'deki her bir Controller için WebApp/Services klasöründe ayrı bir ApiClient oluşturmaktır. Bu nedenle, tek bir genel OrganizationApiClient yerine, iki ayrı ve odaklanmış istemci sınıfımız olmalıdır:*

1. *Biri DepartmentsController ile konuşmak için.*
2. *Diğeri ise PositionsController ile konuşmak için.*

*Bu, Pages/Organization/Index.cshtml.cs sayfasının her iki istemciyi de kullanarak ihtiyacı olan verileri toplayacağı anlamına gelir.*

##### *Personnel Klasörü*

İK Uzmanları ve yöneticiler için personel listeleme, ekleme, düzenleme ve detay görüntüleme gibi özlük işlemleri sayfalarını barındırır. (Örn: Index.cshtml, Create.cshtml, Edit.cshtml)

**Yaşam Döngüsü Kontrol Listesi**

Listenin neden tamamlanmış olduğunu, bir personelin şirket içindeki yolculuğuna göre kontrol edelim:

* **[✓] İşe Alım Sonrası (Yeni Kayıt):** Create.cshtml sayfası, yeni bir personelin sisteme dahil edilmesi için gereken tüm alanları ve adımları kapsıyor.
* **[✓] Oryantasyon (İşe Alıştırma):** Onboarding.cshtml sayfası, personelin ilk günlerindeki görevlerini takip etmek için özel bir alan sunuyor.
* **[✓] Aktif Çalışma Dönemi:**
  + Index.cshtml: Tüm personelleri listelemek ve bulmak için ana ekranımız mevcut.
  + Details.cshtml: Bir personelin tüm bilgilerine tek bir yerden bakabileceğimiz "dijital özlük dosyası" görevini görüyor.
  + Edit.cshtml: Personel bilgilerini güncellemek için gerekli formumuz var.
  + Documents.cshtml: Personele özel belgeleri (sözleşme, diploma vb.) yönetmek için özel bir sayfamız var.
  + Salary.cshtml: Maaş gibi hassas finansal bilgileri yönetmek için ayrı ve yetkilendirilmiş bir sayfamız var.
* **[✓] İşten Ayrılış (Offboarding):** Offboard.cshtml sayfası, bir personelin ayrılış sürecindeki tüm adımları (zimmet iadesi vb.) yönetmek için planlanmıştır.

**Gelecek Fazlar İçin Geliştirme Notları**

Listenin bu haliyle tamamlanmış olması, gelecekte yeni sayfalar eklenmeyeceği anlamına gelmez. Projenizin yol haritasına uygun olarak, bu klasöre gelecekte eklenebilecek iki tip sayfa bulunmaktadır:

1. **Toplu İşlemler Sayfası (BulkImport.cshtml):** Gelecekte, personelleri tek tek eklemek yerine bir Excel/CSV şablonuyla sisteme toplu olarak aktarmak için bu sayfaya ihtiyaç duyulacaktır.
2. **Raporlama Sayfası (Reports.cshtml):** Personel devir hızı (turnover), demografik dağılım veya departman bazlı personel sayısı gibi standart personel raporlarını oluşturmak ve indirmek için bu sayfa eklenebilir.

**Özetle,** mevcut planımız personel yönetiminin tüm temel ve detaylı **bireysel işlem adımlarını** eksiksiz olarak karşılamaktadır. Gelecekteki toplu işlem ve raporlama ihtiyaçları ise bu sağlam temel üzerine kolayca eklenebilir.

##### *Leave Klasörü*

İzin talebi oluşturma gibi sadece izin modülüne özel self-servis sayfaları içerir.

##### *Advance Klasörü*

Avans talebi oluşturma gibi sadece avans modülüne özel self-servis sayfaları içerir.

**Neden Index Sayfası Yok? (Mimari Karar)**

"Eksik var" hissiyatınız muhtemelen Leave (İzin) klasöründe bir Index.cshtml sayfası planlarken, Advance (Avans) klasöründe planlamamış olmamızdan kaynaklanıyor. Bu, bilinçli ve mimari bir karardır:

1. **"Bakiye" Kavramı Yok:** İzin (Leave) modülünden farklı olarak, Avans modülünde personelin takip edeceği bir "bakiye" (kalan avans hakkı gibi) bulunmamaktadır. İzin modülündeki Index sayfasının temel amacı bu bakiyeyi göstermekti.
2. **Merkezi Talep Geçmişi:** Personelin geçmiş avans taleplerini görme ihtiyacı, tüm taleplerini (izin, avans, masraf vb.) bir arada görebileceği, daha önce planladığımız **Pages/Requests/MyRequests.cshtml** sayfasında karşılanmaktadır.

Bu yaklaşım, kod tekrarını önler ve personelin tüm geçmiş taleplerini tek bir merkezi yerden takip etmesini sağlayarak daha iyi bir kullanıcı deneyimi sunar.

Bu nedenle, Pages/Advance klasörü için mevcut planımız, bir avans talebinin tüm yaşam döngüsünü (Oluşturma, Detay Görme, Düzenleme, İptal Etme) kapsayacak şekilde **hem doğru hem de tamamlanmıştır.**

##### *Requests Klasörü*

Personelin, tüm farklı türlerdeki taleplerini (izin, avans vb.) bir arada gördüğü "Taleplerim" gibi birleşik liste sayfalarını içerir.

**Mimari Not ve Teyit**

Bu klasörde neden sadece tek bir sayfa (MyRequests) olduğuna dair bir şüpheniz olmasın. Bu, bilinçli bir mimari karardır:

* **Listeleme Burada, Yönetim Kendi Yerinde:** Bu klasör, sadece talepleri **listelemekle** sorumludur. Bir talebin detayını görmek, düzenlemek veya iptal etmek için, bu listedeki ilgili satırdan /Leave/Details/{id} veya /Advance/Edit/{id} gibi **kendi modülünün sayfasına** link verilir.
* **Kod Tekrarını Önleme:** Her talep türü için ayrı ayrı "Taleplerim" listesi yapmak yerine, tümünü tek bir yerde toplamak kod tekrarını önler ve kullanıcı için daha basit bir deneyim sunar.

Bu nedenle, Pages/Requests klasörü için oluşturulan bu plan, projenizin gereksinimlerini **tam ve en verimli şekilde** karşılamaktadır.

##### *Manager Klasörü*

Departman Yöneticileri'ne özel, ekip yönetimi ile ilgili sayfaları barındırır. (Örn: Approvals.cshtml, TeamCalendar.cshtml)

##### *TimeAttendance Klasörü*

Puantaj yönetimi ile ilgili sayfaları içerir. (Örn: Index.cshtml)

**"Muhasebe Raporu" Hakkında Not:** Kapsam dokümanınızda belirtilen "Muhasebe Raporu" (MO\_314) ihtiyacı ise, ayrı bir sayfa gerektirmez. Bu işlevsellik, Index.cshtml veya Approval.cshtml sayfasına eklenecek bir "Onaylanmış Verileri Dışa Aktar" butonu ile karşılanacaktır.

##### *Finance Klasörü*

Bordro yükleme gibi finansal işlemlere yönelik sayfaları içerir. (Örn: PayrollUpload.cshtml)

Bu klasör, doğrudan maaş hesaplamasından ziyade, maaş bordrosu gibi finansal dokümanların yönetimi gibi İK Uzmanı'nın veya Kiracı Yöneticisi'nin sorumluluğundaki idari görevlere odaklanır.

**Mimari Not ve Teyit**

Listenin bu kadar kısa olması sizi şaşırtmasın. Bu, bilinçli bir mimari karardır. Diğer finansal işlemler, daha mantıksal olan başka klasörlerde ele alınır:

* **Personel Maaş Yönetimi (MO\_611):** Bu işlevsellik, tek bir personele odaklandığı ve yetki gerektirdiği için, daha önce planladığımız **Pages/Personnel/Salary.cshtml** sayfasında yer alır.
* **Personelin Kendi Bordrosunu Görüntülemesi (A\_15\_US05):** Bu bir self-servis özelliktir. Bu nedenle, personelin kendi profilinin bir parçası olarak **Pages/Profile/Me.cshtml** içindeki bir sekmede veya Pages/Profile/MyPayslips.cshtml gibi bir sayfada ele alınır.

Bu sorumluluk ayrımı sayesinde, Pages/Finance klasörünün görevi net bir şekilde "toplu ve idari finansal işlemler" olarak tanımlanmıştır.

**Mimari Not 2**

Bu son eklemelerle birlikte, Finance klasörünüzün sorumluluk alanı netleşmiş olur:

* **Operasyon:** PayrollUpload.cshtml
* **Analiz:** SalaryReports.cshtml
* **Yapılandırma:** PayrollSettings.cshtml

Bireysel maaş yönetimi (Pages/Personnel/Salary.cshtml) ve personelin kendi bordrosunu görmesi (Pages/Profile/Me.cshtml içinde bir sekme) gibi diğer finansal konular, daha önce kararlaştırdığımız gibi kendi mantıksal klasörlerinde kalmaya devam eder.

##### *Recruitment Klasörü*

(Gelecek Faz) İşe alım modülü ile ilgili sayfaları içerir. (Örn: Postings.cshtml, Candidates.cshtml)

##### *Performance Klasörü*

(Gelecek Faz) Performans yönetimi modülü ile ilgili sayfaları içerir. (Örn: Cycles.cshtml, MyReviews.cshtml)

Planımızdaki MyReviews.cshtml ve FillReview.cshtml sayfaları, tek bir role (örn: sadece personel) hizmet etmek yerine, **değerlendirme sürecine katılan tüm roller** (Personel, Yönetici, Öğrenci, Veli) için çalışacak şekilde tasarlanmıştır.

İşte Öğrenci ve Veli'nin bu mevcut sayfaları kullanarak süreci nasıl tamamlayacağı:

1. **Adım: Görev Listesi (MyReviews.cshtml)**
   * Bir **Öğrenci** sisteme giriş yaptığında, MyReviews.cshtml sayfası onun için bir "Doldurmam Gereken Anketler" listesine dönüşür.
   * API, bu kullanıcının rolünün "Öğrenci" olduğunu anlar ve ona sadece kendisine atanmış görevleri (örn: "Öğretmen Değerlendirmesi: Ahmet Yılmaz") gönderir.
   * Aynı şekilde, bir **Veli** giriş yaptığında, bu sayfa ona çocuğu için atanmış anketleri ("[Çocuk Adı] için Öğretmen Değerlendirmesi: Zeynep Kaya") listeler.
2. **Adım: Anket Formu (FillReview.cshtml)**
   * Öğrenci veya Veli, MyReviews sayfasındaki bir göreve tıkladığında, FillReview.cshtml sayfasına yönlendirilir.
   * Bu sayfa, "dinamik bir form oluşturucu" gibi çalışır. URL'deki görev ID'sini kullanarak API'den o ankete özel soruları (örn: "Öğrenci için Öğretmen Değerlendirme Soruları") ister ve ekranda sadece o soruları gösterir.
   * Kullanıcı hikayesinde belirtilen **anonimlik** gibi kurallar, bu sayfanın arayüzünü değiştirmez. Bu, form API'ye gönderildiğinde arka planda (Application katmanında) yönetilecek bir iş kuralıdır.

Bu yapı, gelecekte "Akran Değerlendirmesi" gibi yeni değerlendirme türleri eklemeniz gerektiğinde bile yeni sayfalar oluşturmanızı gerektirmez; sadece API tarafında yeni bir form şablonu ve yeni kurallar tanımlamanız yeterli olur.

##### *Assets Klasörü*

(Gelecek Faz) Varlık yönetimi modülü ile ilgili sayfaları içerir. (Örn: Index.cshtml)

**Mimari Not (Personel Ekranı)**

Kullanıcı hikayelerinde belirtilen, personelin **"kendisine zimmetlenmiş varlıkları görme"** ihtiyacı, bu klasörde ayrı bir sayfa gerektirmez. Bu self-servis özellik, personelin kendi profilinin bir parçası olarak **Pages/Profile/Me.cshtml** içindeki bir "Zimmetlerim" sekmesinde ele alınır.

##### *Portal Klasörü*

(Gelecek Faz) Şirket duyuruları gibi portal sayfalarını barındırır. (Örn: Announcements.cshtml)

#### Services Klasörü

EduHR.Api ile iletişim kuracak olan HttpClient tabanlı istemci sınıflarını (PersonnelApiClient.cs vb.) içerir.

**Bu klasör, API-First mimarisinin en kritik parçasıdır.**

* EduHR.Api ile tüm iletişimi merkezileştiren ve soyutlayan istemci sınıflarını barındırır.
* **IPersonnelApiClient.cs**: Personel API'si için sözleşmeyi tanımlar.
* **PersonnelApiClient.cs**: HttpClient kullanarak /api/personnel uç noktalarına fiilen istekleri gönderir, JWT token'ı yönetir ve gelen JSON yanıtlarını DTO'lara dönüştürür.

#### ViewModels Klasörü

Pages klasöründeki arayüzlerin ihtiyaç duyduğu, API'den gelen verilerle arayüze özel verileri birleştiren modelleri (PersonnelListViewModel.cs vb.) barındırır.

Pages klasöründeki görünümlerin (views) ihtiyaç duyduğu, özel olarak hazırlanmış modelleri içerir. Bir ViewModel, birden fazla API çağrısından gelen veriyi veya arayüze özel (örn: seçim kutusu seçenekleri) bilgileri birleştirebilir. (Örn: PersonnelCreateViewModel.cs)

Oluşturduğumuz liste, projemizde yer alan ve veri ile çalışan her bir sayfa veya arayüz bileşeni için özel bir ViewModel tanımlamaktadır. Bu yaklaşımın temel amacı, API'den gelen genel amaçlı DTO'lar ile arayüzün spesifik ihtiyaçları arasında temiz bir köprü kurmaktır.

* **Kontrol listemiz bu görevi eksiksiz yerine getiriyor:**

[✓] Form Sayfaları: CreatePersonnelViewModel, LeaveRequestCreateViewModel gibi tüm veri giriş formları, hem API'ye gönderilecek veriyi (DTO) hem de formdaki seçim kutularını (List<DepartmentDto>) dolduracak veriyi barındıran kendi ViewModel'lerine sahiptir.

[✓] Liste Sayfaları: PersonnelListViewModel, ApprovalsViewModel gibi tüm listeleme ekranları, gösterilecek veri listesini ve sayfalama/filtreleme gibi arayüze özel seçenekleri kendi ViewModel'lerinde barındırır.

[✓] Karmaşık Görüntüleme Sayfaları: DashboardViewModel gibi birden çok farklı API kaynağından veri alıp tek bir ekranda gösteren sayfalar için de özel ViewModel'ler planlanmıştır.

Bu yaklaşım, her sayfanın veri ihtiyacını kendi içinde izole eder ve yönetilmesini kolaylaştırır. Gelecekte ekleyeceğiniz yeni bir form sayfası, kendi ViewModel'ini de beraberinde getirecektir.

#### wwwroot Klasörü

Tarayıcıya doğrudan gönderilen tüm statik dosyaları içerir. <br>• css/: Siteye özel stil dosyaları (site.css).

* js/: Siteye özel JavaScript dosyaları. API ile dinamik olarak konuşacak pages/ ve services/ gibi alt klasörler burada yer alır
* images/: Logolar, ikonlar gibi resim dosyaları
* lib/: Bootstrap, jQuery gibi üçüncü parti kütüphaneler.

##### wwwroot/js/services Klasörü

* **Bu Servisler Nasıl Birlikte Çalışır? (Örnek)**

Bu yapı, pages/ klasöründeki kodları son derece temiz tutar:

JavaScript

// --- Dosya: wwwroot/js/pages/personnel-index.js ---

// Sadece ilgili servisi içeri aktarır.

import { personnelService } from '../services/personnel.service.js';

// Servis üzerinden temiz bir şekilde veriyi çeker.

const allPersonnel = await personnelService.getAll();

// Gelen veri ile arayüzü günceller.

renderTable(allPersonnel);

// --- Dosya: wwwroot/js/services/personnel.service.js ---

// Temel api servisini içeri aktarır.

import { api } from './api.service.js';

// Dışarıya temiz fonksiyonlar sunar.

export const personnelService = {

getAll: () => api.get('/personnel'), // Karmaşık fetch ve header işlemleri burada gizlenir.

create: (data) => api.post('/personnel', data)

};

* **Listenin tamamlandığını şu mimari prensibe dayanarak teyit edebiliriz:**

js/services/ klasörünün temel amacı, EduHR.Api projesindeki her bir Controller'ın yönettiği kaynaklar ile konuşmaktır. Planlamamızda, her bir ana iş kaynağı için bir Controller oluşturduk. Bu nedenle, js/services klasöründe de bu Controller'lara karşılık gelen bir JavaScript servis dosyası bulunmaktadır.

Bu birebir eşleşme, yapının eksiksiz olduğunu gösterir:

* PersonnelController.cs <--> personnel.service.js
* LeaveController.cs <--> leave.service.js
* AuthController.cs <--> auth.service.js
* DepartmentsController.cs <--> department.service.js
* ... ve diğer tüm kaynaklar için bu yapı devam eder.

**Temel servis olan api.service.js** ise, bu dosyaların tümünün ortak kullandığı, merkezi istek (fetch) ve yetkilendirme (token) mantığını barındırarak kod tekrarını önler.

* **Geleceğe Yönelik Not**

Gelecekte API'nize yeni bir Controller (örn: SurveysController.cs) eklediğinizde, bu yapı sayesinde WebApp tarafında ne yapmanız gerektiğini tam olarak bileceksiniz: wwwroot/js/services/ klasörüne yeni bir survey.service.js dosyası eklemek.

##### wwwroot/js/pages Klasörü

* **Önemli Uygulama Notu**

Bu dosyaların her biri, sadece ilgili olduğu Razor sayfasında (.cshtml) sayfanın sonundaki @section Scripts { ... } bloğu kullanılarak yüklenmelidir. Bu, performansı optimize eder ve kodların birbirine karışmasını engeller.

Bu sayfa-spesifik .js dosyaları, tüm sayfalarda yüklenen site.js'in aksine, sadece ilgili olduğu Razor sayfasında (.cshtml) yüklenmelidir. Bu, genellikle sayfanın sonundaki @section Scripts { ... } bloğu kullanılarak yapılır:

**Örnek: /Pages/Personnel/Index.cshtml dosyasının sonu**

HTML

@section Scripts {

<script type="module" src="~/js/pages/personnel-index.js"></script>

}

Bu yaklaşım, uygulamanızın performansını artırır ve kodunuzun temiz kalmasını sağlar.

* **Geleceğe Yönelik Not**

Gelecekte projeye yeni bir dinamik sayfa (örn: /Surveys/Index.cshtml) eklediğinizde, bu yapı sayesinde ne yapmanız gerektiğini tam olarak bileceksiniz: wwwroot/js/pages/ klasörüne yeni bir surveys-index.js dosyası eklemek.

#### TagHelpers Klasörü

Razor sayfalarında (.cshtml) tekrar eden HTML yapılarını basitleştirmek için oluşturulan, C# ile yazılmış yeniden kullanılabilir arayüz bileşenlerini içerir. (Örn: ProfileImageTagHelper.cs)

## 📁 5\_Common

### DTOs Klasörü

**Mimari Not: DTO'ların Yeni Yeri**

API-First mimarimizde, hem EduHR.Api projesinin (DTO'ları üretmek için) hem de EduHR.WebApp projesinin (API'den gelen DTO'ları anlamak için) aynı DTO sınıflarını bilmesi gerekir. Bu ortak ihtiyacı karşılamak için, **daha önce Application katmanında planladığımız** DTOs klasörünün, en doğru mimari pratik olarak bu Common projesine taşınması gerekir.

**Veri Kontratı (DTO) Kontrol Listesi**

Listenin neden tamamlanmış olduğunu, bir veri katmanının karşılaması gereken tüm temel ihtiyaçları kontrol ederek teyit edebiliriz:

* **[✓] Veri Okuma (GET):** Her bir ana varlık için, hem tüm detayları içeren (PersonnelDetailsDto) hem de listeler için daha hafif olan özet (PersonnelSummaryDto) DTO'larımız mevcut.
* **[✓] Veri Yaratma (POST):** Her bir "yeni kayıt oluşturma" işlemi için, API'ye gönderilecek veriyi net bir şekilde tanımlayan Create...Dto modelleri planlandı (CreatePersonnelDto, CreateDepartmentDto vb.).
* **[✓] Veri Güncelleme (PUT/PATCH):** Her bir "güncelleme" işlemi için, sadece güncellenecek alanları içeren özel Update...Dto modelleri planlandı (UpdateUserDto vb.).
* **[✓] Kimlik Doğrulama:** API'nin en temel ihtiyacı olan login işlemi için LoginRequestDto ve LoginResponseDto çifti ile kontrat tanımlandı.
* **[✓] Arayüz Desteği:** WebApp arayüzündeki seçim kutularını (dropdown) doldurmak gibi ihtiyaçlar için RoleDto, LeaveTypeDto, DepartmentDto gibi basit ve odaklanmış DTO'lar listeye dahil edildi.
* **[✓] Yapısal Tutarlılık:** PagedResultDto<T> gibi jenerik bir yapı ile, tüm listeleme işlemlerinden dönecek olan verinin standart bir formata sahip olması garanti altına alındı.

Bu DTO seti, Application katmanının iş mantığı ile dış dünya (API istemcileri) arasında tam bir "ortak dil" oluşturur. Planımız, projenin mevcut ve gelecek fazlarındaki tüm veri transferi senaryolarını kapsamaktadır.

### Wrappers Klasörü

**Bu Sınıflar Nasıl Kullanılır? (Örnek)**

**PersonnelController'ınızın içinde:**

C#

// GET /api/personnel/123

// Başarılı bir veri yanıtı

var personnelDto = await \_mediator.Send(new GetPersonnelByIdQuery(123));

return Ok(new ApiResponse<PersonnelDetailsDto>(personnelDto, "Personel bilgileri başarıyla getirildi."));

// Dönen JSON: {"success": true, "data": { ...personel bilgileri... }, "message": "..."}

// DELETE /api/personnel/123

// Başarılı bir işlem yanıtı (veri dönmüyor)

await \_mediator.Send(new DeletePersonnelCommand(123));

return Ok(new ApiResponse("Personel başarıyla silindi."));

// Dönen JSON: {"success": true, "data": null, "message": "Personel başarıyla silindi."}

### Exceptions Klasörü

**Mimari Not ve Teyit**

Daha önceki planımızda ValidationException.cs'i Application katmanında düşünmüştük. Ancak API-First mimarisinde, bu hatanın hem sunucu (Api) hem de istemci (WebApp) tarafından bilinmesi gerektiği için, en doğru ve temiz mimari pratik, onu tüm projelerin erişebileceği **Common** katmanına taşımaktır.

Bu iki hata sınıfı, Api ve WebApp arasındaki tüm öngörülebilir hata senaryolarını (genel hatalar ve form doğrulama hataları) yönetmek için **yeterli ve eksiksizdir.**

# TEST KLASÖRÜ

## EduHR.Domain.UnitTests

### Entities Klasörü

İçerisinde karmaşık iş kuralları barındıran varlık (Entity) sınıflarınızın metotlarını test eden dosyaları içerir. (Örn: LeaveRequestTests.cs dosyası, bir izin talebinin Approve() metodunun durumu doğru değiştirip değiştirmediğini test eder.)

### Services Klasörü

Domain/Services klasöründeki servislerin testlerini içerir. (Örn: LeaveCollisionValidationServiceTests.cs dosyası, izin çakışma algoritmasının doğru çalışıp çalışmadığını test eder.)

## EduHR.Application.UnitTests

### Features Klasörü

Application projesindeki Features klasör yapısını birebir yansıtır. Her bir Command ve Query'nin Handler'ı için testler burada bulunur.

* **Personnel/Commands/**: CreatePersonnelCommandHandlerTests.cs
* **Personnel/Queries/**: GetPersonnelByIdQueryHandlerTests.cs
* **Leave/Commands/**: SubmitLeaveRequestCommandHandlerTests.cs

### Common Klasörü

Testler sırasında tekrar tekrar kullanılan sahte (mock) nesneleri veya yardımcı metotları oluşturmak için kullanılır. (Örn: MockRepositoryGenerator.cs)

## EduHR.Api.IntegrationTests

### Controllers Klasörü

Api projesindeki Controller yapısını yansıtır.

Her bir Controller için testler burada yer alır

* PersonnelControllerTests.cs: /api/personnel uç noktasına GET, POST gibi istekler göndererek, personelin gerçekten oluşturulup oluşturulmadığını veya listenin doğru gelip gelmediğini test eder.
* AuthControllerTests.cs: /api/auth/login uç noktasına istek göndererek token alınıp alınamadığını test eder.

### Common Klasörü

Test ortamını hazırlayan temel sınıfları içerir. Özellikle, testler için uygulamanızı bellekte ayağa kaldıran CustomWebApplicationFactory.cs gibi sınıflar burada yer alır.

**tests Klasörü Tamamlama Kontrolü**

Projenizin kalitesini güvence altına almak için planladığımız test stratejisinin her bir parçasını tamamladık:

* **[✓] Domain Katmanı Testleri (EduHR.Domain.UnitTests):** Projenizin en temel iş kurallarının ve varlık (Entity) mantığının, dış dünyadan tamamen izole bir şekilde doğru çalıştığını doğrulayacak testleri planladık.
* **[✓] Application Katmanı Testleri (EduHR.Application.UnitTests):** Projenizin beyni olan tüm iş akışlarının (CQRS Handler), veri doğrulama (Validator), dönüşüm (Mapping) ve merkezi davranış (Behaviour) kurallarının, bağımlılıkları taklit edilerek (mocking) beklendiği gibi çalıştığını garanti altına alacak testleri eksiksiz listeledik.
* **[✓] Uçtan Uca API Testleri (EduHR.Api.IntegrationTests):** Projenizin dış dünyaya açılan kapısı olan API'nin, bir bütün olarak (controller -> application -> infrastructure -> test veritabanı) doğru çalıştığını doğrulayan, gerçek HTTP istekleri gönderen entegrasyon testlerini ve altyapı kodlarını planladık.

# FRONTENDS KLASÖRÜ

### flutter\_app

**Mobil Uygulama.** Flutter ve Dart programlama dili kullanılarak geliştirilecek olan, hem iOS hem de Android platformlarında çalışacak olan mobil uygulamanın tüm kaynak kodlarını içerir. Bu uygulama, tüm veri ihtiyaçları için EduHR.Api ile iletişim kuracaktır.

### marketing\_website

**Pazarlama ve Satış Sitesi.** (Opsiyonel - Gelecek Faz) Potansiyel müşterileri bilgilendirmek, ürün özelliklerini sergilemek ve "Demo Talebi" toplamak amacıyla kullanılacak olan, halka açık pazarlama web sitesinin kaynak kodlarını barındırır. Teknik yığınınızda belirtildiği gibi bu, React gibi bir teknoloji ile geliştirilebilir.

# DEPLOYMENT KLASÖRÜ