DFA Teknoloji Yaz Dönemi Staj Programı Teknik Değerlendirme Raporu

Proje Adı: Dosya Yükleme ve Yönetimi Uygulaması (Full-Stack)

Aday Adı Soyadı: HALİL DAYI

E-posta: halildayi.013@gmail.com

Teslim Tarihi: 24 Mayıs 2025

1. Giriş: Dosya Yükleme ve Yönetimi Uygulamasına Genel Bakış

Bu teknik rapor, DFA Teknoloji 2025 Yaz Dönemi Staj Programı kapsamında verilen "Dosya Yükleme ve Yönetimi Uygulaması (Full-stack)" görevini yerine getirmek amacıyla hazırlanmıştır. Projenin temel amacı, kullanıcıların dijital dosyalarını (özellikle PDF, PNG ve JPG formatlarını) kolayca yönetebilecekleri, güvenli ve kullanıcı dostu bir web uygulaması geliştirmektir.

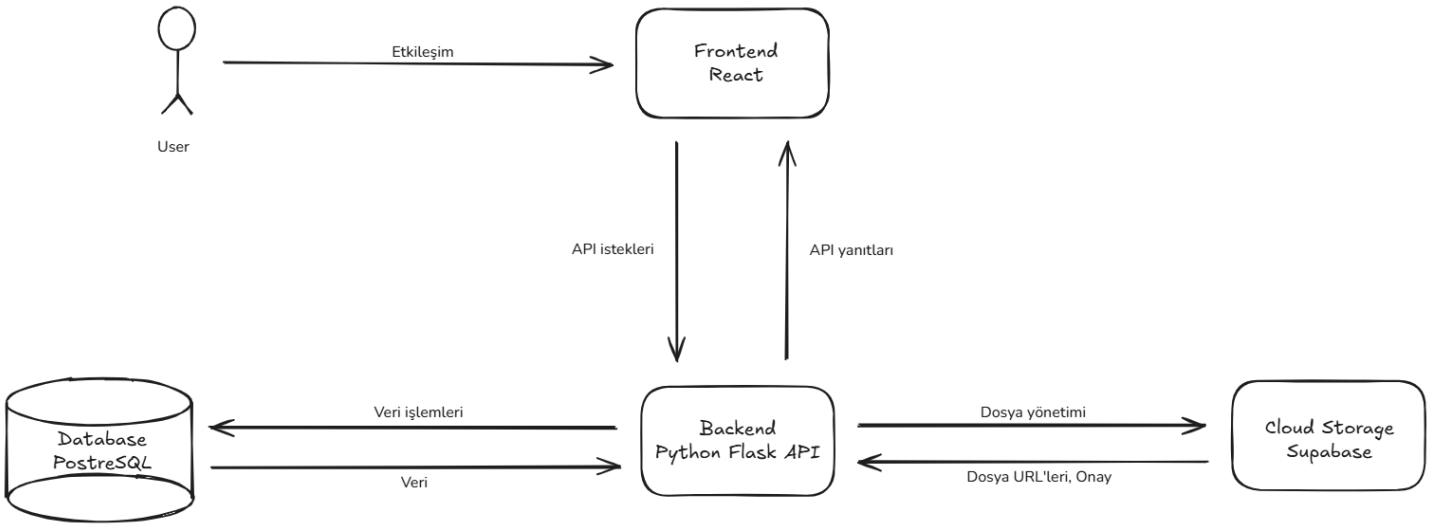
Uygulama, kullanıcıların sisteme dosya yükleme, mevcut dosyalarını listeleme, dosya adına göre arama yapma ve ihtiyaç duymadıkları dosyaları silme gibi temel dosya yönetimi işlevlerini sağlamaktadır. Ayrıca, kullanıcı deneyimini artırmak amacıyla giriş ve kayıt ekranları ile güvenli erişim sunulmuştur. Proje, web geliştirme becerilerini, frontend-backend entegrasyon yeteneğini ve dosya yönetimi mekanizmalarını somut bir örnek üzerinden sergilemektedir. Özellikle, dosyaların sunucuda tutulması yerine modern bir bulut depolama çözümü (Supabase Storage) entegre edilerek, uygulamanın ölçeklenebilirliği ve performansı optimize edilmiştir.

Bu rapor, geliştirilen uygulamanın teknik mimarisini, kullanılan teknolojileri, implementasyon detaylarını ve geliştirme sürecinde karşılaşılan zorluklara getirilen çözümleri detaylı bir şekilde açıklamaktadır.

2. Uygulamanın Genel Mimarisi

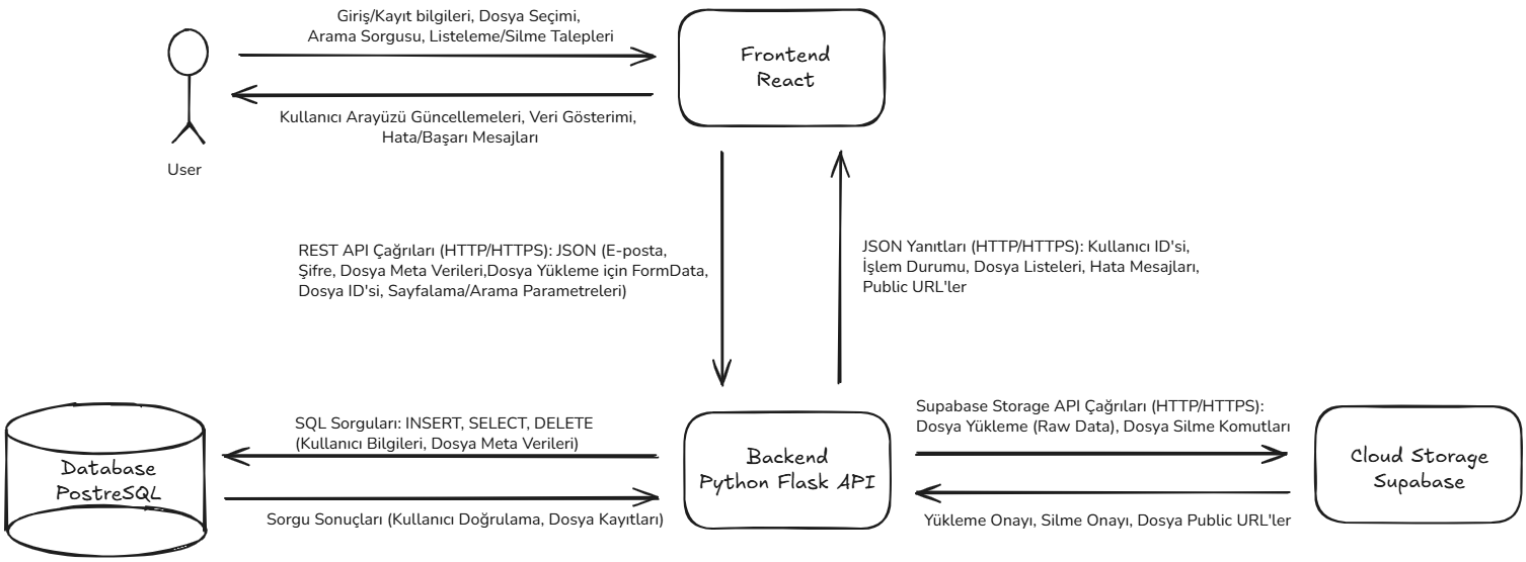
2.1. Uygulamanın Basit Genel Mimarisi

Bu diyagram, uygulamanın temel bileşenlerini ve aralarındaki ana veri akışını gösterir.



2.2. Uygulamanın Detaylı Genel Mimarisi

Aşağıdaki diyagram, uygulamanın katmanları arasındaki iletişimi ve veri akışını daha detaylı bir perspektiften sunmaktadır.



Uygulama, modern bir web mimarisinin temel prensiplerini yansıtan üç ana katman üzerine inşa edilmiştir: Frontend (Kullanıcı Arayüzü), Backend (API Servisi) ve Veri Katmanı (Veritabanı ve Dosya Depolama). Bu katmanlar, birbirinden bağımsız çalışacak şekilde tasarlanmış olup, iletişimleri RESTful API prensiplerine uygun olarak HTTP/HTTPS istekleri aracılığıyla sağlanır.

Bu katmanlı mimari, her bir bileşenin kendi özel sorumluluk alanına odaklanmasını sağlayarak geliştirme esnekliğini, uygulamanın ölçeklenebilirliğini ve bakım kolaylığını önemli ölçüde artırmıştır.

2.2.1. Frontend Katmanı:

Kullanıcıların doğrudan etkileşimde bulunduğu arayüzü sunar. Tüm kullanıcı girdileri (giriş bilgileri, dosya seçimi, arama sorguları) bu katmanda toplanır ve HTTP/HTTPS isteklerini POST veya GET istekleri aracılığıyla Backend'e iletilir. Basit oturum yönetimi için userId gibi bilgileri yerel depolamada (localStorage) tutar. Backend'den gelen yanıtlar (dosya listeleri, işlem onayları, hata mesajları) bu katmanda işlenerek kullanıcıya dinamik olarak sunulur.

2.2.2. Backend (API Servisi) Katmanı:

Uygulamanın temel iş mantığını ve veri akışını yöneten merkezi kontrol noktasıdır. Frontend'den gelen HTTP/HTTPS isteklerini karşılar, bu istekleri doğrular, gerekli işlemleri (kullanıcı doğrulama, dosya metaveri yönetimi, dosya depolama işlemleri) gerçekleştirir ve uygun HTTP yanıt kodları (örneğin 200 OK, 400 Bad Request, 401 Unauthorized) ile JSON formatında veri döndürür. Backend, hem veritabanı hem de dosya depolama servisi ile doğrudan iletişim kurar.

2.2.3. Veri Katmanı:

Uygulamanın persistent (kalıcı) verilerini barındırır. Bu katman iki ana bileşenden oluşur:

İlişkisel Veritabanı (PostgreSQL):

Kullanıcı hesap bilgileri (e-posta, şifre hash'i) ve yüklü dosyalara ait meta veriler (dosya ID'si, adı, boyutu, türü, kullanıcısı, Supabase public URL'si gibi) güvenli bir şekilde saklanır. Backend, bu veritabanı ile SQL sorguları aracılığıyla etkileşim kurar.

Bulut Dosya Depolama Servisi (Supabase Storage):

Kullanıcılar tarafından yüklenen PDF, PNG ve JPG gibi asıl dosya içerikleri bu serviste barındırılır. Bu yaklaşım, uygulama sunucusunun dosya depolama yükünü azaltır, yüksek erişilebilirlik ve ölçeklenebilirlik sağlar. Backend, Supabase'in sunduğu API'ler aracılığıyla dosya yükleme, silme ve görüntüleme linki oluşturma işlemlerini yönetir.

Bu katmanlı mimari, her bir bileşenin bağımsız olarak geliştirilmesine, test edilmesine ve ölçeklenmesine olanak tanırken, standart RESTful API ve HTTP/HTTPS protokolleri sayesinde sağlam ve güvenilir bir iletişim altyapısı sunar.

3. Kullanılan Teknolojiler

Bu proje, modern web geliştirme pratiklerini yansıtan ve her bir katmanın kendi sorumluluğunu en iyi şekilde yerine getirmesini sağlayan bir teknoloji yığını ile geliştirilmiştir. Aşağıda, projemizde kullandığımız ana teknolojiler ve tercih nedenleri detaylandırılmıştır:

3.1. Frontend Katmanı

Frontend katmanı, kullanıcıların uygulamamızla doğrudan etkileşim kurduğu arayüzü sunar. Bu katman için aşağıdaki teknolojiler tercih edilmiştir:

3.1.1. React.js:

Tercih Nedeni: Tek sayfa uygulama (SPA) geliştirme için popüler ve güçlü bir kütüphane olması. Bileşen tabanlı yapısı sayesinde UI elemanlarının modüler bir şekilde geliştirilmesini ve yeniden kullanılabilirliğini kolaylaştırması. Sanal DOM (Virtual DOM) kullanımıyla performanslı güncellemeler sunması.

3.1.2. React Router DOM:

Tercih Nedeni: React tabanlı tek sayfa uygulamalarında sayfa yönlendirme (navigasyon) işlemlerini dinamik ve kullanıcı dostu bir şekilde yönetmek için standart bir çözüm olması. Kullanıcıların farklı URL'ler arasında sorunsuz geçiş yapmasını sağlar.

3.1.3. Axios:

Tercih Nedeni: Backend API'leri ile HTTP istekleri yapmak için kullanılan Promise tabanlı ve kullanımı kolay bir HTTP istemcisi olması. Hem tarayıcıda hem de Node.js ortamında çalışabilmesi, istek ve yanıtları yakalayıp dönüştürebilmesi gibi esnek özellikler sunması. Backend'den gelen JSON yanıtlarını otomatik olarak ayrıştırması, hata yönetimini kolaylaştırması önemli avantajlarıdır.

3.2. Backend Katmanı

Backend katmanı, uygulamanın tüm iş mantığını ve veri akışını yöneten merkezi noktadır. Seçilen teknolojiler şunlardır:

3.2.1. Flask:

Tercih Nedeni: Python ekosisteminde hafif, esnek ve hızlı prototipleme için ideal bir mikro-framework olması. RESTful API'ler geliştirmek için gerekli temel araçları sunması ve büyük projelere kolayca ölçeklenebilmesi. Bağımsız kütüphanelerle kolayca entegre olabilmesi (örneğin Psycopg2, Supabase Python Client).

3.2.2. Psycopg2:

Tercih Nedeni: Python uygulamaları ile PostgreSQL veritabanları arasında güvenilir ve performanslı bir bağlantı kurmak için kullanılan bir adaptör olması. Düşük seviyeli ve yüksek performanslı bir arayüz sunarak veritabanı işlemlerini etkin bir şekilde yönetmeyi sağlar.

3.2.3. Flask-CORS:

Tercih Nedeni: Frontend ve Backend arasındaki çapraz kaynak (CORS) politikalarını yönetmek için Flask uygulamalarına kolay entegrasyon sağlaması. Güvenlikten ödün vermeden tarayıcıdan gelen farklı kaynak isteklerinin sorunsuz bir şekilde karşılanmasını garanti eder.

3.2.4. werkzeug.security (generate_password_hash, check_password_hash):

Tercih Nedeni: Kullanıcı şifrelerini güvenli bir şekilde hash'lemek ve doğrulamak için endüstri standardı yöntemler sunması. Bu fonksiyonlar, şifrelerin düz metin

olarak saklanmasını engelleyerek olası veri sızıntılarına karşı yüksek güvenlik sağlar.

3.2.5. Supabase Python Client (supabase):

Tercih Nedeni: Backend uygulamasının Supabase servisleri (özellikle Storage) ile kolay ve programatik olarak etkileşim kurmasını sağlayan resmi Python istemcisi. Dosya yükleme ve silme gibi işlemleri basit API çağrılarına dönüştürerek geliştirme sürecini hızlandırır.

3.2.6. uuid:

Tercih Nedeni: Yüklenen her dosyaya benzersiz bir kimlik (UUID) atamak için kullanılması. Bu, dosya adlarında çakışmaları önler ve dosya yönetimini daha robust hale getirir.

3.3. Veri Katmanı

Veri katmanı, uygulamanın kalıcı verilerini barındırır ve iki ana bileşenden oluşur:

3.3.1. İlişkisel Veritabanı (PostgreSQL):

Tercih Nedeni: Sağlam, güvenilir, ACID uyumlu ve yüksek performanslı bir ilişkisel veritabanı yönetim sistemi olması. Özellikle yapılandırılmış verilerin (kullanıcı bilgileri, dosya meta verileri gibi) tutarlı ve güvenli bir şekilde saklanması için idealdir. Geniş topluluk desteği ve esnek sorgu yetenekleri sunar.

3.3.2. Bulut Dosya Depolama Servisi (Supabase Storage):

Tercih Nedeni: Kullanıcılar tarafından yüklenen asıl dosya içeriklerini (PDF, PNG, JPG vb.) güvenli, ölçeklenebilir ve yüksek erişilebilir bir şekilde barındırmak için tercih edilmiştir. Bu hizmetin kullanılması, uygulama sunucusunun dosya depolama yükünü azaltır, kendi sunucumuzda karmaşık depolama altyapısı kurma ihtiyacını ortadan kaldırır ve kolayca genel erişim URL'leri oluşturulmasına olanak tanır.

3.4. Kimlik Doğrulama ve Oturum Yönetimi

3.4.1. Basit Oturum Yönetimi (localStorage üzerinde userId):

Tercih Nedeni: Uygulamanın temel işlevselliğini hızlıca devreye almak ve kullanıcı oturumlarını basit bir şekilde takip etmek amacıyla localStorage tercih edilmiştir. Kullanıcı başarılı bir şekilde giriş yaptıktan sonra, Backend'den dönen benzersiz userId değeri tarayıcının yerel depolama alanına (localStorage) kaydedilir. Bu userId, kullanıcının sonraki isteklerinde kimliğini doğrulamak için Backend'e gönderilir. Bu yaklaşım, karmaşık altyapılar gerektirmeyen hızlı ve pratik bir oturum takibi çözümü sunar.

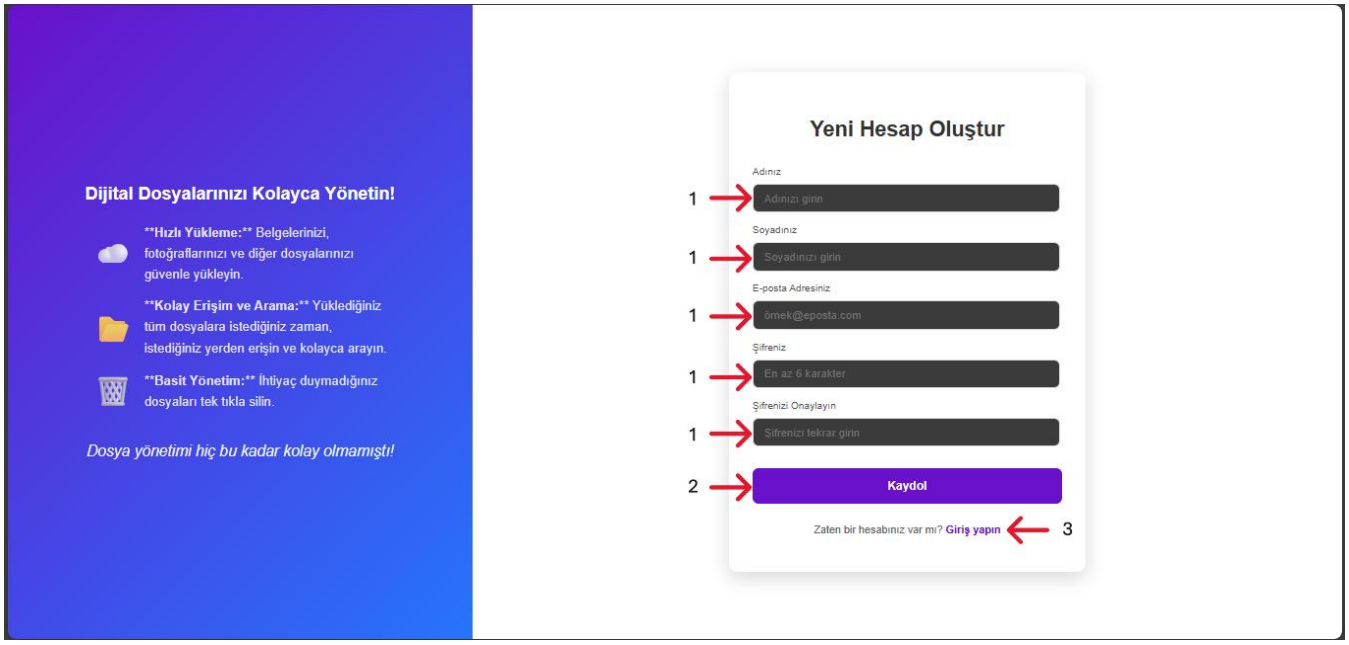
4. Implementasyon Detayları ve Fonksiyonel Açıklamalar

Bu bölüm, uygulamamızın temel işlevselliklerinin nasıl tasarlandığını ve uygulandığını detaylandırmaktadır. Kullanıcı arayüzü ile Backend ve Veri Katmanı arasındaki etkileşimler, adım adım açıklanmaktadır. Ekran görüntüleri üzerindeki numaralar, metindeki ilgili fonksiyonel alanlara işaret etmektedir.

4.1. Giriş ve Kayıt İşlevselliği

Kullanıcıların uygulamaya erişimini sağlayan bu temel işlevsellik, hem yeni kullanıcı kaydını hem de mevcut kullanıcıların oturum açmasını yönetir.

4.1.1. Yeni Kullanıcı Kaydı



Kullanıcı Deneyimi: Kullanıcı kayıt ol sayfasında ad (1), soyad (1), e-posta adresini (1) ve belirlediği şifreyi (1) ilgili form alanlarına girer. Şifrenin doğruluğunu teyit etmek için şifreyi iki kez girmesi beklenir. Kaydol butonuna (2) tıkladığında kaydolabilir. Eğer kullanıcının hesabı varsa “Giriş yapınız” linkine (3) tıklayarak giriş yap sayfasına gidebilir.

Frontend İşleyişi (React - SignUp page):

- SignUpPage bileşeni, kullanıcının girdiği firstName, lastName, email, password ve confirmPassword bilgilerini yönetmek için useState hook'larını kullanır.
- Kullanıcı formu gönderdiğinde (handleSubmit fonksiyonu), event.preventDefault() çağrısı ile tarayıcının varsayılan form gönderme davranışı engellenir.
- Şifre Doğrulaması: İlk olarak, Frontend tarafında password ile confirmPassword alanlarının eşleşip eşleşmediği kontrol edilir. Eğer eşleşmiyorsa, kullanıcıya bir uyarı gösterilir ve kayıt işlemi durdurulur.
- Veri Hazırlığı: Şifreler eşleştiğinde, firstName, lastName, email ve password bilgilerini içeren bir userData JavaScript objesi oluşturulur.
- API İsteği: Bu userData objesi, axios.post metodu kullanılarak API_ENDPOINTS.signUp adresine (Backend'deki /signup endpoint'i) JSON formatında gönderilir. İsteğin Content-Type başlığı application/json olarak ayarlanır.
- Yanıt İşleme: Backend'den dönen yanıta göre (başarı veya hata), kullanıcıya uygun bir geri bildirim sağlanır.

- Başarılı bir yanıt (response.status === 201) durumunda, Backend'den gelen başarı mesajı kullanıcıya gösterilir ve react-router-dom'un Maps hook'u ile kullanıcı /signin sayfasına yönlendirilir.
- Hata durumunda (örneğin Backend'den 400 veya 409 hatası gelirse), error.response.data.message kullanılarak Backend'in döndürdüğü spesifik hata mesajı kullanıcıya sunulur. Sunucuya bağlantı sorunları gibi genel hatalar da yakalanır.

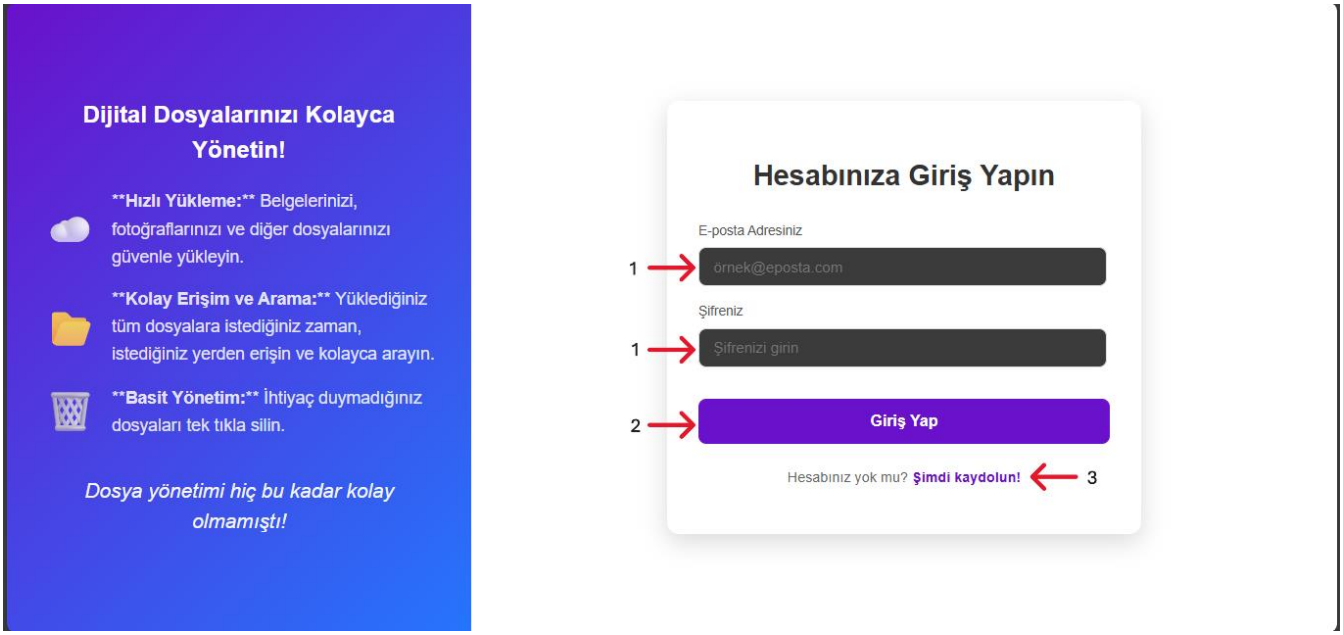
Backend İşleyişi (Flask): Flask uygulamasındaki @app.route('/signup', methods=['POST']) dekoratörü ile tanımlanan sign_up() fonksiyonu, Frontend'den gelen POST isteğini karşılar.

- request.get_json() metoduyla Frontend'den gelen JSON verisi alınır ve first_name, last_name, email, password değişkenlerine atanır.
- Tüm alanların dolu olduğu basit bir doğrulama (if not all(...)) yapılır. Eksik alan varsa 400 Bad Request yanıtı döner.
- Kullanıcının girdiği şifre, werkzeug.security modülünün generate_password_hash(password) fonksiyonu kullanılarak güvenli bir şekilde hash'lenir. Bu, şifrelerin düz metin olarak saklanmasını engeller.
- **Veritabanı Bağlantısı ve Kontroller:** get_db_connection() fonksiyonu ile PostgreSQL veritabanına bir bağlantı kurulur. Bağlantı hatası durumunda 500 Internal Server Error döner.
- **E-posta Tekrarlılığı Kontrolü:** cur.execute("SELECT id FROM users WHERE email = %s;") sorgusu ile girilen e-posta adresinin veritabanında daha önce kayıtlı olup olmadığı kontrol edilir. Eğer e-posta zaten mevcutsa, 409 Conflict HTTP durumuyla birlikte "Bu e-posta adresi zaten kayıtlı!" mesajı döndürülür.
- **Yeni Kullanıcı Kaydı:** E-posta benzersizse, INSERT INTO users sorgusu ile first_name, last_name, email ve hashed_password veritabanındaki users tablosuna eklenir. RETURNING id ifadesi, yeni oluşturulan kullanıcının ID'sinin geri döndürülmesini sağlar.
- **İşlem Tamamlama:** conn.commit() ile veritabanı işlemi onaylanır. Başarılı kayıt durumunda, "Kayıt başarılı! Lütfen giriş yapın." mesajı ve yeni userId ile birlikte 201 Created HTTP durumu döndürülür.
- **Hata Yönetimi:** Herhangi bir veritabanı hatası durumunda conn.rollback() ile işlem geri alınır ve 500 Internal Server Error yanıtı döndürülür.

Veritabanı Etkileşimi (PostgreSQL):

- psycopg2 kütüphanesi aracılığıyla users tablosuna yeni kullanıcı verilerini eklemek için INSERT INTO users (first_name, last_name, email, password_hash) VALUES (%s, %s, %s, %s) RETURNING id; sorgusu çalıştırılır.
- Kayıt işlemi öncesinde e-posta kontrolü için SELECT id FROM users WHERE email = %s; sorgusu kullanılır.

4.1.2. Mevcut Kullanıcı Girişi



Kullanıcı Deneyimi: Kullanıcı, giriş formuna kayıtlı e-posta adresini (1) ve şifresini (1) girer. Ardından "Giriş Yap" butonuna (2) tıklar. Kullanıcı üye değilse “Şimdi Kaydolun!” linkine (3) tıklayarak Hesap Oluştur sayfasına gidebilir.

Frontend İşleyişi (React - SignInPage page):

- Kullanıcı handleSubmit fonksiyonunu tetiklediğinde, girilen email ve password bilgilerini içeren bir userData objesi oluşturulur.
- Bu userData objesi, axios.post metodu kullanılarak Backend'deki /signin endpoint'ine JSON formatında (Content-Type: application/json başlığıyla) gönderilir.
- Backend'den başarılı bir yanıt (status: 200 OK) alındığında, kullanıcıya "Giriş başarılı!" mesajı gösterilir. Backend'den dönen userId değeri, localStorage.setItem('userId', response.data.userId) komutu ile tarayıcının yerel depolama alanına (localStorage)

kaydedilir. Bu userId, kullanıcının sonraki korumalı sayfalara ve API isteklerine erişimi için basit bir oturum tanımlayıcısı olarak kullanılır.

- Başarılı girişin ardından, react-router-dom'un Maps hook'u ile kullanıcı /DashboardPage'e yönlendirilir.
- API isteği sırasında bir hata oluşursa (örneğin geçersiz kimlik bilgileri veya sunucu hatası), try-catch bloğu devreye girer ve hatanın türüne göre kullanıcıya uygun bir mesaj (alert) gösterilir.

Backend İşleyişi (Flask - /signin endpoint'i):

- /signin endpoint'i, Frontend'den gelen POST isteğini karşılar ve request.get_json() metodu ile JSON formatındaki kullanıcı kimlik bilgilerini (email, password) ayrıştırır.
- email veya password alanlarından herhangi birinin eksik olup olmadığı kontrol edilir. Eksik bilgi varsa 400 Bad Request yanıtı döndürülür.
- Veritabanı bağlantısı (get_db_connection()) kurulur. Bağlantı hatası durumunda 500 Internal Server Error yanıtı verilir.
- Backend, users tablosundan girilen e-postaya sahip kullanıcıyı bulmak için bir SELECT sorgusu çalıştırır. Eğer kullanıcı bulunamazsa, 401 Unauthorized yanıtı döndürülür.
- Kullanıcı veritabanında bulunursa, kullanıcının girdiği şifre, werkzeug.security modülünün check_password_hash() fonksiyonu kullanılarak veritabanında kayıtlı olan hash'lenmiş şifre (password_hash) ile karşılaştırılır. Bu karşılaştırma, şifrenin kendisini açığa çıkarmadan güvenli bir doğrulama sağlar. PostgreSQL'den gelen hashed_password_from_db eğer bytes formatındaysa, check_password_hash fonksiyonunun beklediği string formatına çevrilir (decode('utf-8')).
- Şifreler eşleşirse, giriş başarılı kabul edilir ve Backend, Frontend'e "Giriş başarılı!" mesajı ile birlikte kullanıcının userId'sini JSON formatında (200 OK HTTP durumuyla) iletir.
- Şifreler eşleşmezse veya kullanıcı bulunamazsa, 401 Unauthorized HTTP durumuyla birlikte "Geçersiz e-posta veya şifre!" mesajı döndürülür.
- Herhangi bir sunucu hatası durumunda 500 Internal Server Error yanıtı döndürülür.

Veritabanı Etkileşimi (PostgreSQL):

- psycopg2 kütüphanesi aracılığıyla, kullanıcının id, email ve password_hash bilgilerini çekmek için SELECT id, email, password_hash FROM users WHERE email = %s; sorgusu çalıştırılır. Bu sorgu, kullanıcının kimliğini doğrulamak için gerekli veriyi sağlar.

4.1.3 Giriş Sonrası

Kullanıcının sisteme başarılı bir şekilde giriş yapmasının ardından, oturumunun sürdürülebilmesi ve korumalı kaynaklara erişim sağlayabilmesi için basit bir oturum yönetimi mekanizması devreye girer.

Frontend İşleyişi:

- Backend'den başarılı bir giriş yanıtıyla dönen kullanıcının benzersiz kimliği (userId), tarayıcının localStorage objesine localStorage.setItem('userId', response.data.userId) komutu ile kaydedilir.
- Bu userId, kullanıcının oturumu boyunca Frontend tarafından saklanır ve Dashboard gibi korumalı sayfalara erişim için kullanılır. Kullanıcı, tarayıcısını kapatsa veya yeniden açsa bile userId localStorage'da kalıcı olduğu için, manuel olarak çıkış yapmadığı sürece oturumu devam eder.

Güvenlik Değerlendirmeleri:

- localStorage kullanımı, projenin mevcut kapsamı ve hızlı prototipleme hedefleri doğrultusunda basit ve pratik bir çözüm sunmaktadır.
- Her korumalı API isteğinde, Frontend localStorage'dan bu userId'yi alarak Backend'e gönderir. Backend, bu userId'yi isteği işleme almadan önce doğrulama ve yetkilendirme amacıyla kullanır.
- Bu yaklaşım, JWT (JSON Web Tokens) gibi daha gelişmiş çözümlere kıyasla basitlik avantajı sunsa da, hassas veri yönetimi gerektiren ve yüksek güvenlik beklentisi olan büyük ölçekli uygulamalar için alternatif (ve daha güvenli) yöntemlerin değerlendirilmesi gerektiğini belirtmek önemlidir.

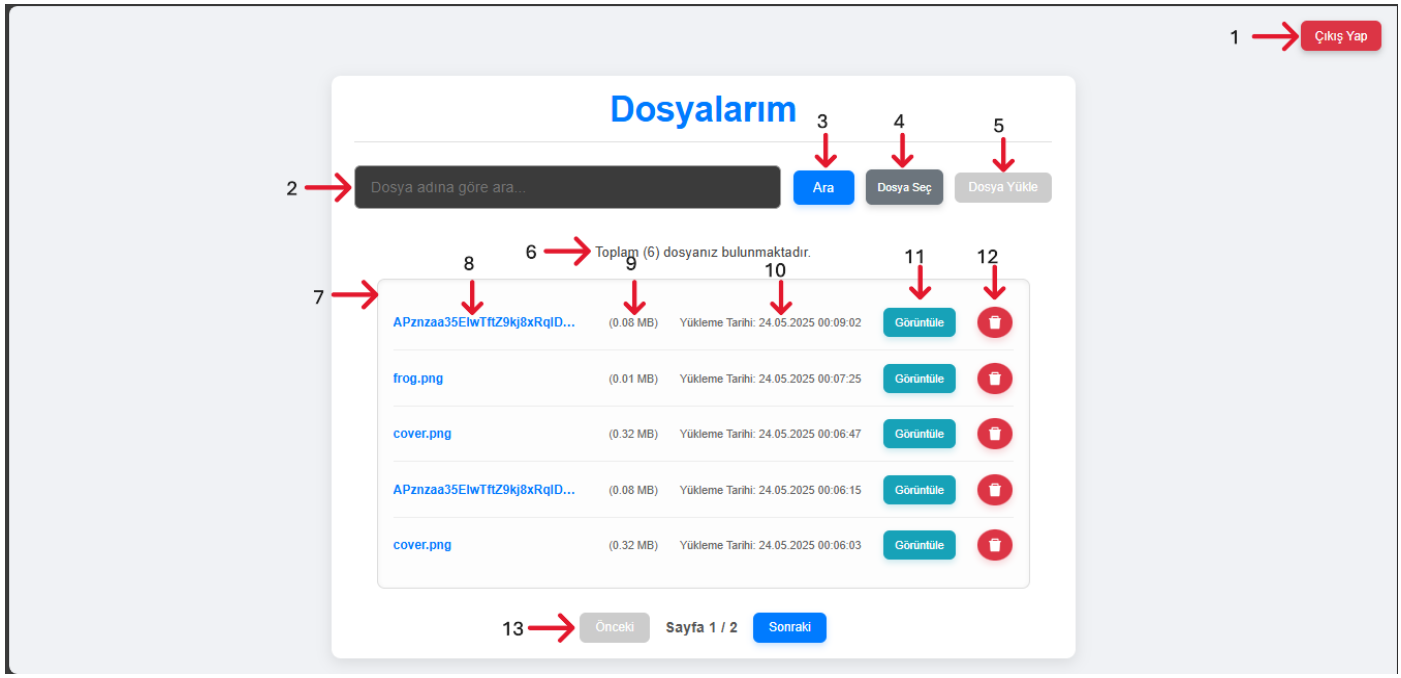
4.1.4. Kullanıcı Doğrulama ve Yetkilendirme (@validate_user_id Dekorötörü) (4Gu¨venlik Mekanizması)

Uygulamanın Backend tarafında, hassas veya kullanıcıya özel işlemleri (dosya yükleme, listeleme, silme gibi) yetkisiz erişime karşı korumak için bir kullanıcı doğrulama ve yetkilendirme mekanizması implemente edilmiştir.

Backend İşleyişi:

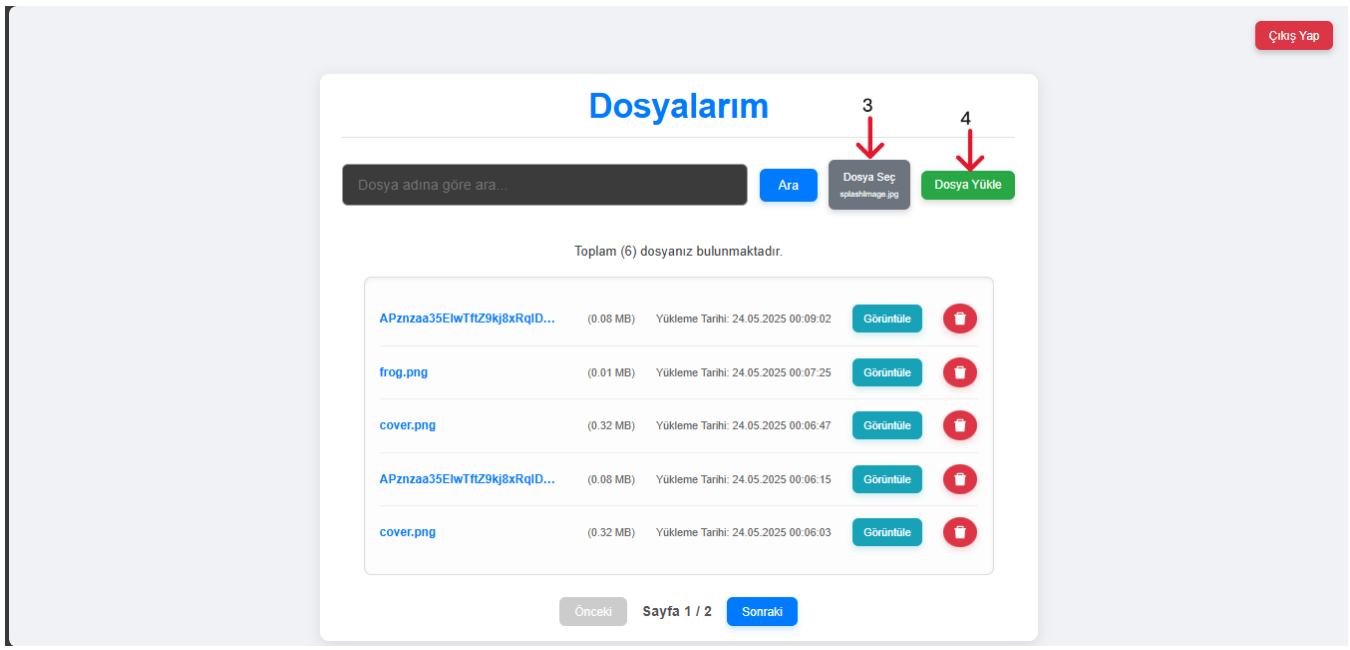
- Backend'deki korumalı tüm API endpoint'leri, özel olarak tanımlanmış bir Python dekoratörü olan @validate_user_id ile işaretlenmiştir. Bu dekoratör, bir isteğin ilgili Backend fonksiyonuna ulaşmadan önce belirli kontrolleri yapmasını sağlar.
- Bir HTTP isteği (/upload-file, /list-files, /delete-file vb.) Backend'e ulaştığında, @validate_user_id dekoratörü devreye girer.
- Dekoratör, gelen isteğin URL parametrelerinde (veya isteğin yapılandırılış şekline göre başka bir yerinde) bir userId parametresinin olup olmadığını kontrol eder.
- Eğer userId parametresi eksikse veya geçersiz bir formatta ise, dekoratör isteği hemen durdurur ve 401 Unauthorized (Yetkisiz) HTTP durumuyla birlikte bir hata mesajı döndürür. Bu, yetkisiz kullanıcıların korumalı kaynaklara erişmesini engeller.
- Eğer geçerli bir userId bulunursa, dekoratör isteği orijinal Backend fonksiyonuna (upload_file, list_files vb.) iletir. Bu fonksiyonlar daha sonra gelen userId'yi kullanarak veritabanı işlemlerini gerçekleştirir (örneğin, yalnızca o userId'ye ait dosyaları listeler).
- Bu mekanizma, uygulamanın her kullanıcının sadece kendi verileri üzerinde işlem yapabilmesini garanti altına alan basit ama etkili bir yetkilendirme katmanı sunar.

4.2. DashboardPage



4.2.1. Dosya Yükleme İşlevselliği

Kullanıcıların dijital dosyalarını sisteme yüklemesini sağlayan bu işlevsellik, Frontend'deki seçim mekanizmasından, dosyanın Backend'e iletilmesine ve Supabase Storage'a güvenli bir şekilde kaydedilmesine kadar uzanan adımları içerir.



Frontend'de Dosya Seçimi ve Hazırlığı

Kullanıcı Deneyimi: Kullanıcı, Dashboard ekranındaki dosya seç butona (3) tıklayarak işletim sisteminin dosya seçme iletişim kutusunu açar. Buradan yüklemek istediği bir PDF, PNG veya JPG dosyasını seçer. Seçilen dosyanın adı, yükleme işleminden önce kullanıcının görebileceği şekilde arayüzde görüntülenir.

Frontend İşleyişi (React - DashboardPage page):

- Bir dosya seçildiğinde, input type="file" elementinin onChange olayı tetiklenir. Bu olay, seçilen dosya nesnesini (File objesi) setSelectedFile state'ine kaydederken, dosyanın adını da setSelectedFileName state'ine aktarır.
- Kullanıcı "Dosya Yükle" butonuna (4) tıkladığında handleUpload fonksiyonu çalışır. Bu fonksiyon ilk olarak bir dosyanın seçili olup olmadığını ve kullanıcının oturum açmış (userId'nin localStorage'da var olup olmadığı) olduğunu kontrol eder. Eksik bir durum varsa ilgili uyarılar gösterilir.

Dosyanın Backend'e Gönderilmesi

Kullanıcı Deneyimi: Dosya seçildikten ve gerekli kontroller yapıldıktan sonra, kullanıcı "Dosya Yükle" butonuna (4) tıklar.

Frontend İşleyişi (React - DashboardPage page):

- handleUpload fonksiyonu içinde, selectedFile ve localStorage'dan alınan userId kullanılarak yeni bir FormData objesi oluşturulur. FormData.append('file', selectedFile) ile dosya ve formData.append('userId', userId) ile kullanıcı ID'si bu objeye eklenir.
- Bu formData objesi, axios.post metoduyla Backend'deki /upload-file endpoint'ine gönderilir. HTTP isteğinin headers kısmında Content-Type: multipart/form-data olarak ayarlanır, bu da dosya yüklemeleri için standart web formatıdır.
- Yükleme başarılı olursa (response.status === 201), kullanıcıya bir başarı mesajı gösterilir, selectedFile ve selectedFileName state'leri temizlenir ve WorkspaceFiles fonksiyonu tekrar çağrılarak dosya listesi güncellenir. Aksi takdirde, hata mesajları kullanıcıya iletilir.

Backend'de Dosyanın İşlenmesi ve Depolama

Backend İşleyişi (Flask - /upload-file endpoint'i):

- /upload-file endpoint'i, @validate_user_id dekoratörü tarafından korunur; bu sayede sadece kimliği doğrulanmış kullanıcılar dosya yükleyebilir. Dekoratör, isteğin userId'sini user_id parametresi olarak fonksiyona iletir.

- Flask'ın request.files objesinden yüklenen dosya (file) alınır. Eğer file objesi yoksa veya boşsa (file.filename == ''), 400 Bad Request hatası döndürülür.
- Dosyanın original_name (orijinal adı) ve mime_type (dosya türü) bilgileri alınır. Dosyanın içeriği file.read() ile okunur ve size_bytes (boyut) bu içeriğin uzunluğundan elde edilir.
- Benzersiz Dosya Adı Oluşturma: Dosya çakışmalarını önlemek ve güvenlik için uuid.uuid4() fonksiyonu ile benzersiz bir dosya adı oluşturulur ve orijinal dosya uzantısı (file_extension) buna eklenir.
- Supabase Storage'a Yükleme: Dosya, supabase.storage.from_(SUPABASE_BUCKET_NAME).upload() metodu kullanılarak Supabase Storage'a yüklenir. Yükleme yolu, kullanıcının klasörü (user_id) ve benzersiz dosya adını (stored_name) içerecek şekilde SUPABASE_BUCKET_NAME/{user_id}/{stored_name} olarak yapılandırılır. Dosyanın içeriği (file_contents) ve MIME tipi bu metoda parametre olarak geçirilir.
- Public URL Alma: Supabase'e yükleme başarılı olduktan sonra, supabase.storage.from_(SUPABASE_BUCKET_NAME).get_public_url(storage_path) metodu ile dosyanın doğrudan erişilebilir public_url'i alınır. Bu URL, Frontend'in dosyayı görüntülemesi veya indirmesi için kullanılır. URL'nin alınamaması durumunda 500 Internal Server Error döndürülür.
- Veritabanına Meta Veri Kaydı: Dosya Supabase'e yüklendikten ve public URL alındıktan sonra, dosyanın user_id, original_name, stored_name, size_bytes, mime_type ve public_url gibi meta verileri, PostgreSQL veritabanındaki files tablosuna bir INSERT sorgusu ile kaydedilir. Ayrıca uploaded_at sütununa yükleme tarihi otomatik olarak atanır.
- İşlem başarılı olursa, conn.commit() ile veritabanı değişiklikleri kaydedilir ve Frontend'e başarı mesajı (201 Created) ile birlikte yeni dosyanın ID'si ve public URL'i döndürülür. Herhangi bir adımda hata oluşursa, veritabanı işlemi geri alınır (conn.rollback()) ve uygun bir hata mesajı (500 Internal Server Error) döndürülür.

Veritabanı Etkileşimi (PostgreSQL):

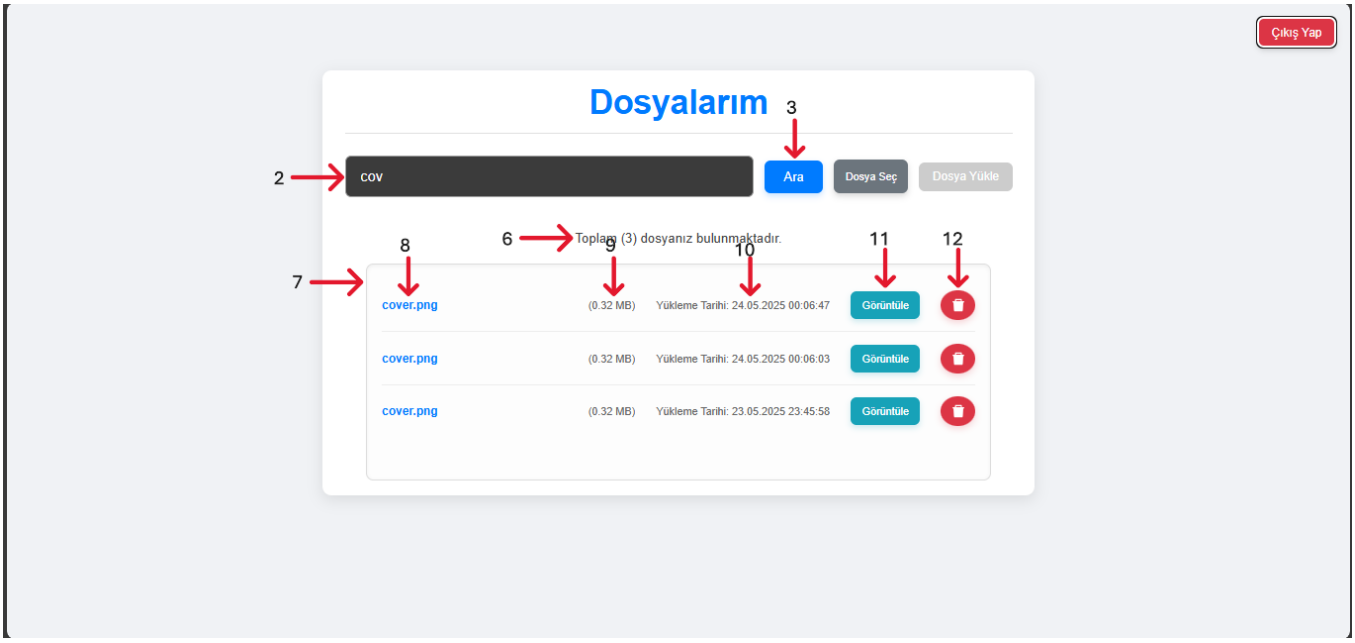
- psycopg2 aracılığıyla, files tablosuna dosya meta verilerini kaydetmek için INSERT INTO files (user_id, original_name, stored_name, size_bytes, mime_type, public_url, uploaded_at) VALUES (%s, %s, %s, %s, %s, %s, NOW()) RETURNING id; sorgusu çalıştırılır.

Bulut Dosya Depolama Etkileşimi (Supabase Storage):

- Dosya yükleme için supabase.storage.from_(SUPABASE_BUCKET_NAME).upload() kullanılır.
- Public URL almak için supabase.storage.from_(SUPABASE_BUCKET_NAME).get_public_url() kullanılır.

4.2.2 Dosya Listeleme İşlevselliği

Kullanıcının daha önce yüklediği dosyalara erişimini ve yönetmesini sağlayan bu işlevsellik, dinamik listeleme ve gelişmiş sayfalama özellikleriyle sunulmuştur.



Frontend'de Dosya Verilerinin Çekilmesi

Kullanıcı Deneyimi: Kullanıcı Dashboard sayfasına giriş yaptığında, sisteme yüklediği dosyalar otomatik olarak listelenir (7). Listelenen dosyaların adı (8), boyutu (9), yükleme tarihi (10) ve bir görüntüleme linki (11) gibi temel bilgileri sunulur.

Frontend İşleyişi (React - DashboardPage page):

- DashboardPage bileşeni yüklendiğinde veya currentPage ya da filesPerPage (sayfa başına dosya sayısı) state'lerinde bir değişiklik olduğunda, useEffect hook'u tetiklenir ve WorkspaceFiles fonksiyonunu çağırır.
- WorkspaceFiles fonksiyonu, localStorage'dan alınan userId'yi kontrol ederek kullanıcının oturumunun açık olduğundan emin olur. Aksi halde kullanıcıyı giriş sayfasına yönlendirir.
- Fonksiyon, Backend'deki /list-files endpoint'ine userId, page (mevcut sayfa) ve limit (sayfa başına dosya sayısı) parametreleri ile bir axios.get isteği gönderir.
- Backend'den gelen yanıt (response.data) işlenir. Yanıtın geçerli bir files dizisi içerip içermediği kontrol edilir. Başarılı yanıtta, files (dosya listesi), totalPages (toplam sayfa sayısı), currentPage (mevcut sayfa numarası) ve totalFilesCount (toplam dosya sayısı) state'leri güncellenir. Bu state güncellemeleri, React'in kullanıcı arayüzünü (UI) otomatik olarak yeniden render etmesini sağlar. Hata durumlarında ise ilgili state'ler sıfırlanır ve kullanıcıya bilgilendirici bir mesaj gösterilir.

Backend'de Dosya Sorgulama ve Sayfalama

Backend İşleyişi (Flask - /list-files endpoint'i):

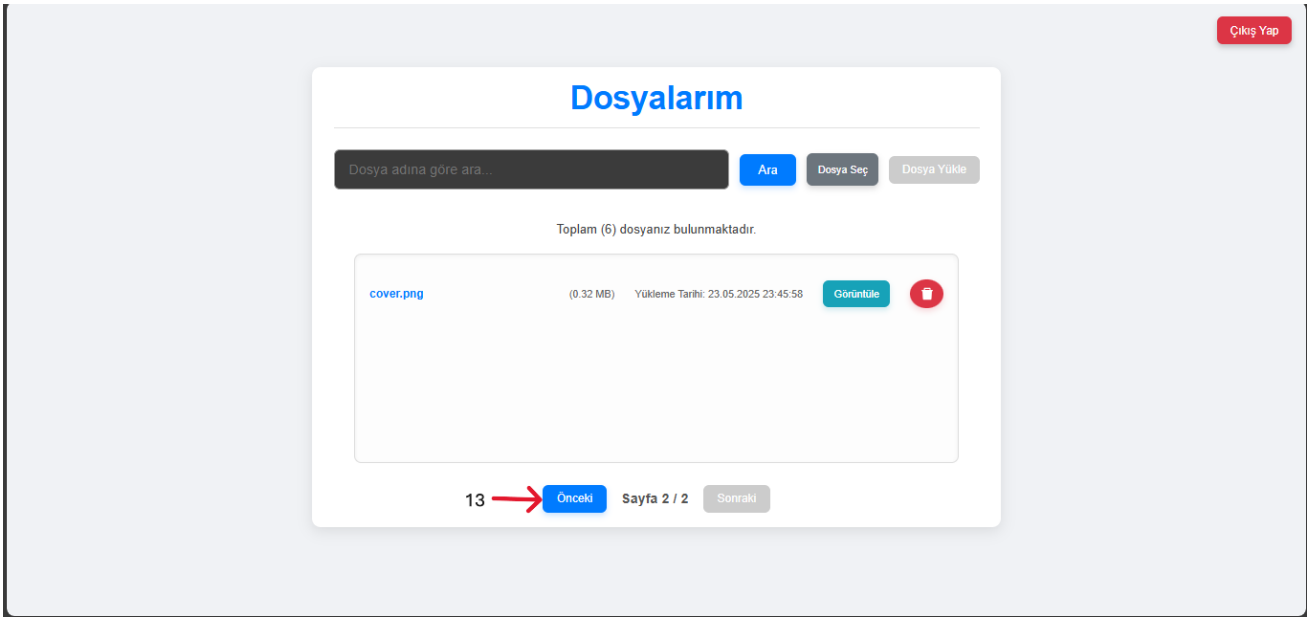
- /list-files endpoint'i, @validate_user_id dekoratörü tarafından korunur, bu sayede yalnızca kimliği doğrulanmış kullanıcıların dosya listesine erişimi sağlanır. Dekoratör, isteğin userId'sini user_id parametresi olarak fonksiyona iletir.
- Gelen HTTP isteğinden page ve limit (sayfa başına dosya sayısı) parametreleri alınır. Veritabanından kaç kayıt atlanacağını belirten offset değeri (page - 1) * limit formülüyle hesaplanır.
- get_db_connection() fonksiyonu aracılığıyla PostgreSQL veritabanına bağlantı kurulur.
- Toplam Dosya Sayısı: İlk olarak, mevcut user_id'ye ait toplam dosya sayısını elde etmek için files tablosunda bir COUNT(*) sorgusu çalıştırılır. Bu bilgi, Frontend'deki "Toplam (X) dosyanız bulunmaktadır" metni ve sayfalama kontrollerinin hesaplanması için kritiktir.
- Dosya Verilerini Çekme: Belirtilen user_id'ye ait dosyalar, id, original_name (dosya adı), size_bytes (boyut), uploaded_at (yükleme tarihi), mime_type (dosya tipi) ve public_url gibi meta verileriyle files tablosundan çekilir. Son yüklenen dosyaların en üstte görünmesi için ORDER BY uploaded_at DESC sıralaması kullanılır. Hesaplanan LIMIT ve OFFSET değerleri, veritabanı sorgusuna uygulanarak sayfalama işlemi gerçekleştirilir.
- Yanıt Hazırlama: Veritabanından alınan dosya verileri, her dosya için gerekli bilgileri (ID, dosya adı, boyut, public URL, yükleme tarihi, içerik tipi) içeren bir JSON listesi (files_list) haline dönüştürülür.

- Frontend'e Yanıt: Toplam dosya sayısı ve sayfa limiti kullanılarak total_pages hesaplanır. Backend, Frontend'e "Dosyalar başarıyla listelendi." mesajı, files_list, currentPage, totalPages ve totalFiles bilgilerini içeren bir JSON yanıtı (200 OK HTTP durumuyla) döndürür.

Veritabanı Etkileşimi (PostgreSQL):

psycopg2 kütüphanesi kullanılarak files tablosunda SELECT COUNT(*) ve sayfalama uygulanan SELECT sorguları çalıştırılır.

Frontend'de Dosya Bilgilerinin Gösterilmesi ve Sayfalama Kontrolleri



Kullanıcı Deneyimi: Kullanıcı, sayfa altında bulunan "Önceki" ve "Sonraki" butonlarını kullanarak dosya listesi sayfaları arasında gezinir. Hangi sayfada olduğu ve toplam kaç sayfa olduğu bilgisi kendisine sunulur.

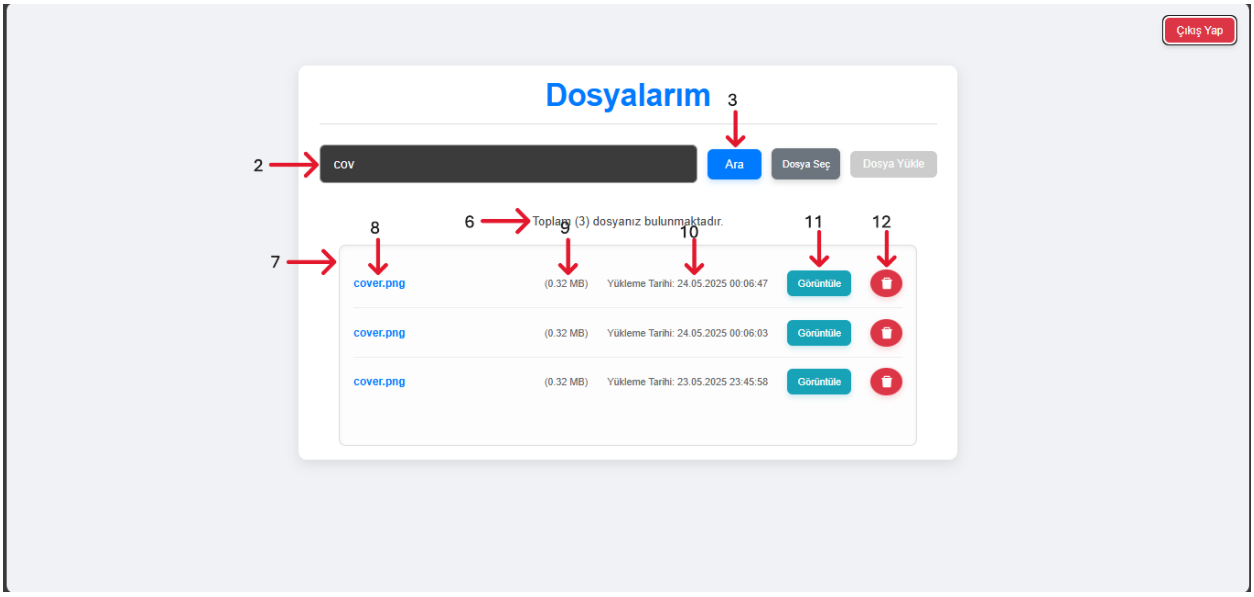
Frontend İşleyişi (React - DashboardPage page):

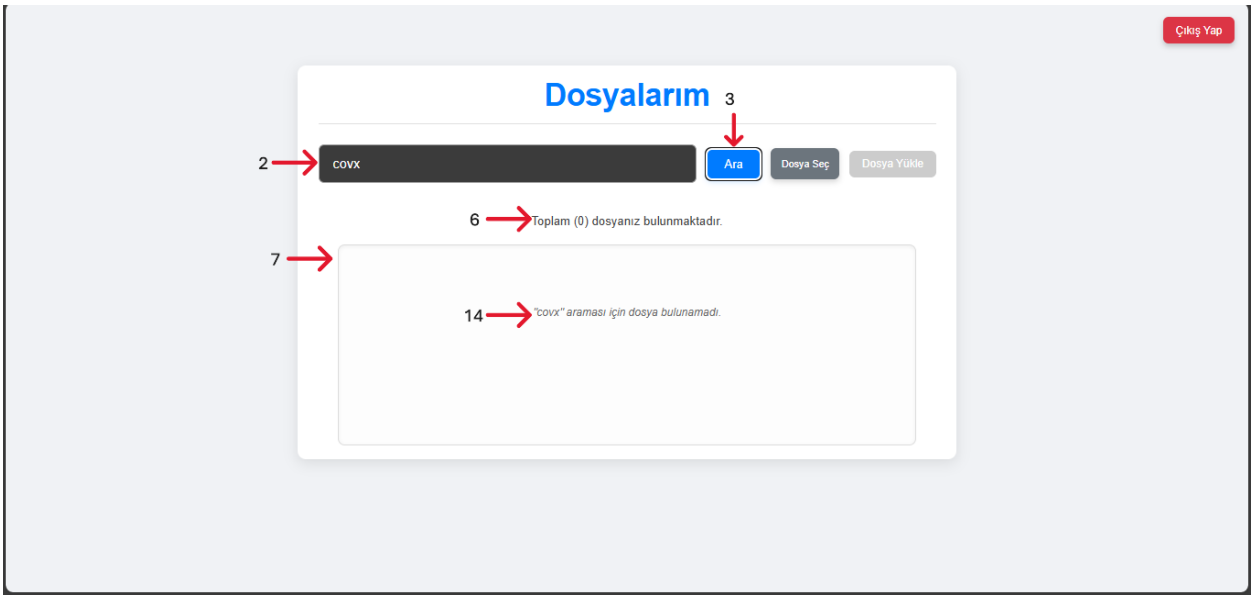
- Dosya Listesi Render Edilmesi: Backend'den gelen files dizisi, React'in map fonksiyonu kullanılarak her bir dosya için bir <li> öğesine dönüştürülür. Her <li> öğesi içinde:

    - Dosya Adı: file.file_name.
    - Dosya Boyutu: file.file_size değeri MB'a dönüştürülerek iki ondalık basamaklı olarak gösterilir (örneğin "1.25 MB").
    - Yükleme Tarihi: file.uploaded_at değeri new Date().toLocaleString() kullanılarak okunabilir bir tarih ve saat formatına dönüştürülür.
    - Görüntüle Linki: file.public_url kullanılarak bir <a> etiketi oluşturulur. Bu link, kullanıcının tarayıcısının doğrudan Supabase Storage'daki dosyayı açmasını veya indirmesini sağlar. target="_blank" özelliği ile yeni sekmede açılır.
    - Sil Butonu: DeleteFilled ikonu içeren bir buton bulunur. Bu butona tıklandığında handleDelete fonksiyonu çağrılarak ilgili dosyanın silme işlemi başlatılır.
- Toplam Dosya Sayısı Bilgisi: totalFilesCount state'i kullanılarak, "Toplam (X) dosyanız bulunmaktadır." şeklinde genel bir özet sunulur.
- Sayfalama Kontrolleri: totalPages değeri 1'den büyükse, sayfalama butonları (Önceki ve Sonraki) ve sayfa numarası bilgisi (Sayfa X / Y) görüntülenir.
    - handlePreviousPage ve handleNextPage fonksiyonları, currentPage state'ini güncelleyerek yeni sayfayı Backend'den çekmeyi tetikler.
    - currentPage === 1 ve currentPage === totalPages durumlarına göre "Önceki" ve "Sonraki" butonları otomatik olarak devre dışı (disabled) bırakılır, bu da kullanıcıların geçersiz sayfalara gitmesini engeller.
- Boş Liste Mesajı: Eğer files dizisi boşsa, kullanıcıya "Henüz yüklediğiniz bir dosya bulunmamaktadır. İlk dosyanızı yüklemek için dosyanızı 'Dosya Seç' butonuyla seçiniz ve 'Dosya Yükle' butonunu kullanın." şeklinde bilgilendirici bir mesaj gösterilir.

4.2.3 Dosya Arama İşlevselliği

Uygulama, kullanıcıların yükledikleri dosyalar arasında belirli anahtar kelimelere göre arama yapmalarına olanak tanıyarak dosya bulma sürecini kolaylaştırır. Bu özellik, özellikle çok sayıda dosya barındıran kullanıcılar için büyük bir kolaylık sağlar.





Frontend'de Arama Sorgusunun Başlatılması

Kullanıcı Deneyimi: Kullanıcı, Dashboard ekranındaki "Dosya adına göre ara..." metin kutusuna (2) aramak istediği dosya adının bir kısmını veya tamamını yazar. Ardından "Ara" butonuna (3) tıklar veya klavyeden Enter tuşuna basar.

Frontend İşleyişi (React - DashboardPage page):

- Kullanıcının metin kutusuna girdiği değer, searchQuery state'inde saklanır ve handleSearchChange fonksiyonuyla güncellenir.

- Kullanıcı "Ara" butonuna (2) tıkladığında (onClick={handleSearchSubmit}) veya Enter tuşuna bastığında (onKeyDown={handleKeyDown}), handleSearchSubmit fonksiyonu tetiklenir.
- handleSearchSubmit fonksiyonu, searchQuery değerini activeSearchQuery state'ine atar ve currentPage'i 1'e sıfırlar. Bu state değişiklikleri, bir useEffect hook'unu tetikler.
- useEffect hook'u içinde çağrılan fetchFiles fonksiyonu, activeSearchQuery değeri dolu olduğu için, Backend'deki /files/search endpoint'ine bir axios.get isteği gönderir. Bu istek, userId, page, limit ve query (arama anahtar kelimesi) parametrelerini içerir.
- Backend'den gelen arama sonuçları, setFiles ile güncellenir ve dosya listesi arama sonuçlarına göre yeniden render edilir. Eğer arama sonucunda dosya bulunamazsa, kullanıcıya "araması için dosya bulunamadı." şeklinde bilgilendirici bir mesaj gösterilir.

Backend'de Arama Sorgusunun İşlenmesi

Backend İşleyişi (Flask - /files/search endpoint'i):

- /files/search endpoint'i, @validate_user_id dekoratörü tarafından korunur ve user_id'yi parametre olarak alır.
- Gelen HTTP isteğinden query (arama anahtar kelimesi), page ve limit parametreleri alınır. offset değeri sayfalama için hesaplanır.
- Arama sorgusu (query), PostgreSQL'de kısmi ve büyük/küçük harf duyarsız arama yapmak için ILIKE %{query}% formatında bir search_pattern'a dönüştürülür.
- get_db_connection() ile veritabanı bağlantısı kurulur.
- Toplam Arama Sonucu Sayısı: İlk olarak, files tablosunda user_id ve original_name ILIKE %s koşuluna uyan toplam dosya sayısını almak için bir COUNT(*) sorgusu çalıştırılır. Bu, arama sonuçları için sayfalama bilgilerinin doğru hesaplanmasını sağlar.
- Filtrelenmiş Dosya Verilerini Çekme: files tablosundan, belirtilen user_id'ye ait ve original_name'i arama deseniyle eşleşen dosyalar çekilir. Sorgu, id, original_name, size_bytes, uploaded_at, mime_type ve public_url sütunlarını içerir. Son yüklenenlerin önce gelmesi için ORDER BY uploaded_at DESC sıralaması ve sayfalama için LIMIT ve OFFSET kullanılır.
- Yanıt Hazırlama: Veritabanından çekilen filtrelenmiş dosya verileri, Frontend'e gönderilmek üzere JSON formatında bir liste (files_list) haline getirilir. Her dosya için gerekli tüm bilgiler (ID, dosya adı, boyut, public URL, yükleme tarihi, içerik tipi) eklenir.
- Frontend'e Yanıt: Hesaplanan total_pages ile birlikte, "Arama sonuçları başarıyla listelendi." mesajı, files_list, currentPage, totalPages ve totalFiles bilgilerini içeren bir JSON yanıtı (200 OK HTTP durumuyla) döndürülür. Hata durumlarında, uygun bir hata mesajı ve durumu döndürülür.

Veritabanı Etkileşimi (PostgreSQL):

- psycopg2 kütüphanesi aracılığıyla, files tablosunda arama sorgusunu içeren SELECT COUNT(*) ve SELECT sorguları çalıştırılır. ILIKE operatörü, PostgreSQL'in güçlü metin arama yeteneklerinden faydalanır.

4.2.4. Dosya Silme İşlevselliği

Uygulama, kullanıcıların yükledikleri dosyaları sistemden güvenli ve kalıcı olarak kaldırmalarına olanak tanır. Bu işlevsellik, hem bulut depolama alanındaki dosyanın hem de veritabanındaki ilgili meta verisinin senkronize bir şekilde silinmesini garanti eder.

Frontend'de Silme İsteğinin Başlatılması

Kullanıcı Deneyimi: Kullanıcı, Dashboard'daki dosya listesinde silmek istediği dosyanın yanındaki çöp kutusu ikonuna (12) tıklar. Bu eylem üzerine, yanlışlıkla silmeleri önlemek amacıyla bir onay penceresi (window.confirm) açılır ve kullanıcıdan işlemi teyit etmesi istenir.

Frontend İşleyişi (React - DashboardPage page):

- Çöp kutusu ikonuna (12) tıklandığında, handleDelete fonksiyonu, silinecek dosyanın benzersiz kimliği olan fileId parametresi ile çağrılır.
- Fonksiyon, ilk olarak window.confirm ile kullanıcıdan onay alır. Onay verilmezse işlem iptal edilir.
- Ardından, kullanıcının oturum açmış olup olmadığını (userId kontrolü) doğrular. Oturum yoksa kullanıcı giriş sayfasına yönlendirilir.
- Onay ve oturum kontrolü başarılı olursa, localStorage'dan alınan userId ile birlikte Backend'deki /delete-file/{fileId} endpoint'ine bir axios.delete isteği gönderilir. fileId doğrudan URL yoluna dahil edilirken, userId bir sorgu parametresi olarak eklenir (örneğin: /delete-file/12345?userId=abcde).
- Backend'den başarılı bir yanıt (response.status === 200 OK) alındığında, kullanıcıya "Dosya başarıyla silindi!" mesajı gösterilir ve fetchFiles fonksiyonu tekrar çağrılarak dosya listesi güncellenir. Bu, silinen dosyanın listeden anında kaybolmasını sağlar. Hata durumlarında ise kullanıcıya uygun bir mesaj (alert) iletilir.

Backend'de Silme İşleminin Yönetilmesi

Backend İşleyişi (Flask - /delete-file/<uuid:file_id> endpoint'i):

- /delete-file/<uuid:file_id> endpoint'i, @validate_user_id dekoratörü tarafından korunur; bu, yalnızca kimliği doğrulanmış kullanıcıların dosya silebilmesini sağlar. Dekoratör, URL'den alınan file_id'yi ve isteğin userId'sini fonksiyona iletir.
- Veritabanı bağlantısı (get_db_connection()) kurulur.
- Backend, silinecek dosyanın stored_name bilgisini (Supabase Storage'daki benzersiz dosya adı) ve kullanıcının bu dosyayı silme yetkisi olup olmadığını doğrulamak için PostgreSQL veritabanındaki files tablosundan bir SELECT sorgusu çalıştırır. Sorgu, gelen file_id ve user_id'yi kullanarak eşleşen kaydı arar.
- Eğer dosya bulunamazsa veya belirtilen userId'ye ait değilse (yetkisiz erişim), 404 Not Found yanıtı döndürülür.
- Dosya bilgisi başarıyla alındığında, Backend, stored_name bilgisini kullanarak Supabase Python Client aracılığıyla dosyayı Supabase Storage'dan siler. Bu işlem supabase.storage.from_(SUPABASE_BUCKET_NAME).remove([storage_path_to_delete]) metodu ile gerçekleştirilir.
- Supabase Storage'dan dosya başarıyla silindikten sonra, PostgreSQL veritabanındaki files tablosundan ilgili file_id'ye ait meta veri kaydı bir DELETE sorgusu aracılığıyla silinir. Bu iki aşamalı silme, hem dosya içeriğinin hem de veritabanı kaydının senkronize kalmasını sağlar.
- Tüm işlemler başarılı olursa, veritabanı değişiklikleri conn.commit() ile kalıcı hale getirilir ve Frontend'e "Dosya başarıyla silindi!" mesajı (200 OK HTTP durumuyla) döndürülür.
- Herhangi bir adımda hata oluşursa (örneğin veritabanı hatası, Supabase silme hatası), conn.rollback() ile veritabanı işlemi geri alınır ve uygun bir hata mesajı (500 Internal Server Error) döndürülür.

Veritabanı ve Bulut Depolama Etkileşimleri

- Veritabanı Etkileşimi (PostgreSQL): psycopg2 kütüphanesi aracılığıyla, silinecek dosyanın stored_name'ini ve yetkiyi doğrulamak için SELECT stored_name FROM files WHERE id = %s AND user_id = %s; sorgusu, ardından dosya kaydını silmek için DELETE FROM files WHERE id = %s AND user_id = %s; sorgusu çalıştırılır.
- Bulut Dosya Depolama Etkileşimi (Supabase Storage): Dosya içeriğini silmek için supabase.storage.from_(SUPABASE_BUCKET_NAME).remove() metodu kullanılır.

5. Karşılaşılan Zorluklar ve Çözümleri

Her projede olduğu gibi, bu uygulamanın geliştirilmesi sırasında da çeşitli teknik zorluklarla karşılaştık. Bu zorlukları aşarken edindiğimiz deneyimler, projenin sağlamlığını artırmıştır.

5.1. Geliştirme Ortamı ve Backend Erişimi

Zorluk: Başlangıçta Backend geliştirmesi için Google Colab ortamı ve dışarıdan erişim sağlamak amacıyla Ngrok kullanıldı. Ancak, Ngrok'un istikrarsız bağlantı sorunları ve oturum süre kısıtlamaları (özellikle ücretsiz planında) Frontend ile Backend arasındaki iletişimi olumsuz etkiledi ve geliştirme sürecini yavaşlattı.

Çözüm: Geliştirme verimliliğini ve kararlılığı artırmak amacıyla Google Colab / Ngrok ikilisinden vazgeçilerek, Backend tamamen yerel bir Flask sunucusu üzerine taşındı. Bu geçiş, hem daha güvenilir bir geliştirme ortamı sağladı hem de olası bağlantı kesintilerini ortadan kaldırarak akıcı bir geliştirme deneyimi sundu.

5.2. Dosya Depolama Stratejisi

Zorluk: Kullanıcı dosyalarını güvenli ve ölçeklenebilir bir şekilde depolama ihtiyacı vardı. Sunucu üzerinde lokal depolama; güvenlik, yedekleme ve gelecekteki ölçeklenebilirlik açısından potansiyel zorluklar içeriyordu. Ayrıca, bulut depolama çözümlerinde ücretsiz kullanım katmanlarının getirdiği depolama alanı kısıtları da değerlendirilmesi gereken bir faktördü.

Çözüm: Supabase Storage, kolay entegrasyonu, geliştirici dostu API'leri ve genel olarak hızlı kurulum imkanı nedeniyle tercih edildi. Bu, projenin hızlı bir şekilde hayata geçirilmesine olanak tanıdı ve dosya depolama yükünü kendi sunucumuzdan aldı. Mevcut proje kapsamında kullanıcı sayısı ve depolanacak dosya büyüklükleri göz önüne alındığında, Supabase'in ücretsiz katmanının sunduğu depolama alanı yeterli görülmüştür. Ancak, daha fazla kullanıcı veya daha büyük dosya hacimleri için Supabase'in ücretli planlarına geçiş yapılması veya farklı bir ölçeklenebilir depolama çözümünün (örneğin AWS S3 gibi) entegre edilmesi gerekebileceği de bu aşamada tespit edilmiştir.

5.3. Kullanıcı Kimlik Doğrulama ve Yetkilendirme

Zorluk: Kullanıcı bilgilerinin güvenli bir şekilde saklanması ve yetkisiz erişimlerin engellenmesi gerekiyordu.

Çözüm: Kullanıcı şifreleri veritabanına kaydedilirken werkzeug.security ile hash'lenerek güvenliği sağlandı. Oturum yönetimi için ise başarılı girişte dönen userId'nin localStorage'da saklanması gibi basit bir oturum takibi yöntemi benimsendi. Ayrıca, Backend'deki hassas endpoint'ler için @validate_user_id dekoratörü geliştirilerek, her isteğin geçerli bir userId içerip içermediği kontrol edilerek yetkilendirme sağlandı.

6. Gelecekteki Geliştirmeler / İyileştirmeler

Uygulamamız şu an temel dosya yönetimi ihtiyaçlarını karşılasa da, bir full-stack geliştiricinin vizyonunun sürekli iyileştirme doğrultusunda olması gerektiğini ve gelecekte eklenebilecek veya geliştirilebilecek önemli alanlar bulunduğunu düşünüyorum.

6.1. Kimlik Doğrulama ve Oturum Güvenliği

Mevcut basit oturum yönetimi yerine, uygulamanın güvenlik seviyesini artırmak için JWT (JSON Web Tokens) tabanlı kimlik doğrulama sistemine geçiş önceliklidir. Bu, tokenların imzalanabilirliği ve son kullanma tarihine sahip olması sayesinde yetkisiz erişim riskini azaltacak ve sunucunun ölçeklenebilirliğini destekleyecektir.

6.2. Dosya Depolama Yönetimi ve Ölçeklenebilirlik

Supabase Storage'ın kolay entegrasyonu projenin hızlı başlamasını sağladı. Ancak, ücretsiz katmanın depolama alanı kısıtları, uygulamanın kullanıcı sayısı ve dosya hacmi arttıkça zorluk çıkarabilir. Gelecekte, daha büyük ölçekler için Supabase'in ücretli planlarına geçiş veya AWS S3 gibi daha esnek bulut depolama çözümlerinin entegrasyonu değerlendirilmelidir.

6.3. Kullanıcı Deneyimi ve Fonksiyonel Genişletmeler

Kullanıcı deneyimini zenginleştirmek için aşağıdaki özellikler düşünülebilir:

- **Gelişmiş Dosya Yönetimi:** Dosyalar için versiyonlama (eski sürümlere dönme) ve klasörleme sistemi (dosyaları düzenleme) gibi yetenekler eklemek.

- **Dosya Paylaşım Özellikleri:** Kullanıcıların dosyalarını güvenli linklerle paylaşabilmesi.

6.4. Uygulama Kalitesi ve Bakım

Uygulamanın uzun vadeli sürdürülebilirliği ve kararlılığı için şu teknik iyileştirmeler hedeflenmektedir:

- **Detaylı Hata Loglama:** Uygulama hatalarını daha kapsamlı bir şekilde kaydetmek ve hızlıca tespit edebilmek için bir izleme sistemi entegrasyonu.
- **Test Katmanı Oluşturma:** Uygulamanın güvenilirliğini artırmak ve gelecek geliştirmelerde olası hataları önlemek adına unit ve entegrasyon testlerinin yazılması.
- **Veritabanı Bağlantı Havuzu:** Yüksek yük altındaki performans için veritabanı bağlantı havuzu (connection pooling) kullanımı.

7. Sonuç

Bu proje, kullanıcıların dosyalarını güvenli ve etkin bir şekilde yönetmelerine olanak tanıyan, modern ve tam işlevli bir web uygulamasının geliştirilmesini başarıyla tamamlamıştır. Belirlenen tüm fonksiyonel gereksinimler—kullanıcı kaydı ve girişi, dosya yükleme, listeleme, arama ve silme—başarıyla implemente edilmiştir.

React.js ile dinamik ve kullanıcı dostu bir arayüz sunulurken, Python Flask tabanlı Backend, sağlam bir API servisi olarak iş mantığını ve veri akışını yönetmiştir. PostgreSQL veritabanı kullanıcı ve dosya meta verilerini güvenle saklarken, Supabase Storage ise dosya içerikleri için ölçeklenebilir ve güvenilir bir bulut depolama çözümü sağlamıştır. Frontend ile Backend arasındaki entegrasyon, RESTful API prensipleri ve HTTP/HTTPS protokolleri aracılığıyla sorunsuz bir şekilde gerçekleştirilmiştir.

Geliştirme sürecinde karşılaşılan Ngrok bağlantı sorunları ve CORS politikaları gibi teknik zorluklar başarıyla aşıldı. Genel olarak, bu proje, modern bir web uygulamasının uçtan uca geliştirilmesi konusundaki yetkinliğimi ve verilen hedeflere ulaşma kapasitemi sergilemektedir.