ChatGPT

Envanter & E-Ticaret Yönetim Sistemi Projesi İncelemesi

1. Projenin amacı ve genel tanıtımı

Envanter & E-Ticaret Yönetim Sistemi, işletmelerin envanter, ürün, müşteri ve sipariş yönetimi süreçlerini kolaylaştırmak için geliştirilmiş modern bir web uygulamasıdır [1]. Bu proje sayesinde kullanıcılar stoklarındaki ürünleri takip edebilir, yeni ürünler ve müşteriler ekleyebilir, siparişleri yönetebilir ve e-ticaret operasyonlarını tek bir yerden kontrol edebilirler. Web tabanlı olması sayesinde herhangi bir tarayıcı üzerinden erişilip kullanılabilir. Uygulama, envanter takibini e-ticaret sipariş yönetimiyle birleştirerek işletmelerin **ürün kataloğu, müşteri bilgileri, sipariş durumu** gibi kritik verilerini entegre bir biçimde yönetmeyi amaçlar. Sonuç olarak proje, küçük ve orta ölçekli işletmelerin operasyonel verimliliğini artıracak kapsamlı bir yönetim sistemi sunmaktadır.

2. Ana modüller ve işlevleri

Proje, çeşitli yönetim modüllerine ayrılmıştır ve her modül belirli bir işlev grubunu yerine getirir:

Ürün Yönetimi:

Uygulamada kapsamlı bir **ürün yönetimi** modülü bulunur. Kullanıcılar sisteme yeni ürünler ekleyebilir, var olan ürün bilgilerini güncelleyebilir, gerekirse silebilir ve tüm ürünleri listeleyebilir [2]. Ürün eklerken isim, fiyat, stok seviyesi, açıklama, barkod gibi temel bilgiler girilir. Sistem, her ürün için görsel ürün kartları göstererek navigasyonu kolaylaştırır. Ürünler kategorilere ayrılabilir ve kategoriye bağlı olarak otomatik ikon ataması yapılır; bu sayede farklı ürün grupları kolayca ayırt edilir [3]. **Stok yönetimi ve takibi** de bu modülün bir parçasıdır: her ürünün stok durumu izlenir ve önceden tanımlanan minimum stok seviyesinin altına düşen ürünler tespit edilebilir. Ek olarak, ürünlere ait detaylı özellikler tanımlanabilir (ör. renk, beden gibi özellikler) ve bir ürüne birden fazla görsel eklenebilir. Tüm bu özellikler, ürün kataloğunun zengin ve düzenli bir şekilde tutulmasını sağlar.

Müşteri Yönetimi:

Müşteri yönetimi modülü, işletmenin müşterilerini kayıt altına almasını ve ilişkili bilgileri düzenlemesini sağlar. Kullanıcılar sisteme yeni müşteri ekleyebilir, mevcut müşteri kayıtlarını güncelleyebilir ve tüm müşteri listesini görüntüleyebilir [4]. Her müşteri için ad, soyad, e-posta, telefon numarası, adres gibi temel bilgiler tutulur. Uygulama, telefon numaralarını Türkçe ve uluslararası formatlara uygun şekilde biçimlendirebilir ve ülke kodu seçimi yapmaya olanak tanır [5]. Müşteri yönetimi modülü sayesinde işletme, müşteri iletişim bilgilerini ve tercihlerine dair notları tek bir yerde toplayabilir. Bu modül ayrıca müşteri segmentasyonu (ör. müşterinin **VIP** veya yeni müşteri olup olmadığı gibi) için altyapı içerir; böylece müşteriler gruplara ayrılıp özelleştirilmiş hizmet veya kampanyalar planlanabilir.

Sipariş Yönetimi:

Sipariş yönetimi modülü, gelen siparişlerin oluşturulması, takibi ve güncellenmesi işlevlerini içerir. Bir sipariş oluşturulduğunda, siparişe ait ürünler, miktarlar, müşteri bilgileri ve ödeme türü sisteme girilir.

Uygulama, her siparişin durumunu (örn. Beklemede, Kargoya Verildi, Teslim Edildi) takip edebilmek için görsel bir zaman çizelgesi (**timeline**) sunar [6]. Bu zaman çizelgesi üzerinden siparişin hazırlık, kargoya teslim, dağıtım ve teslim edilme aşamaları görsel olarak izlenebilir. Ayrıca sisteme entegre edilmiş **kargo takibi** özelliği bulunmaktadır: OpenStreetMap tabanlı bir harita entegrasyonu sayesinde, sipariş için girilen takip numarası ve kargo firması bilgilerine dayanarak kargonun durumu harita üzerinde gösterilebilir [7]. Örneğin, sipariş **kargoya verildi** durumuna geçtiğinde harita üzerinde kargonun tahmini konumu bir işaretçiyle belirtilir ve teslimata giden rota çizilir. Bu sayede yöneticiler ve müşteriler, siparişin nerede olduğu bilgisini görsel olarak takip edebilir. Uygulama, farklı kargo firmalarıyla da entegre çalışabilecek şekilde tasarlanmıştır (mevcut sürümde temel bir entegrasyon sağlanmıştır). Sipariş detay sayfasında ürün listesi, toplam tutar, kargo bilgileri ve sipariş notları gibi tüm bilgiler derli toplu sunulur.

Gelişmiş Arama ve Filtreleme:

Sistem, **gelişmiş arama ve filtreleme** özellikleriyle donatılmıştır. Kullanıcılar, ürünleri veya müşterileri çeşitli kriterlere göre hızlıca filtreleyebilir ve aradıkları bilgiye anında ulaşabilir. Örneğin, ürün listesinde belirli bir kategoriye ait ürünleri görmek, belli bir fiyat aralığındaki ürünleri süzmek veya stok durumuna göre filtrelemek mümkündür [8]. Benzer şekilde müşteri listesinde, müşterilerin şehir, ülke, e-posta veya telefon numarası gibi bilgilerine göre arama yapılıp sonuçlar daraltılabilir [9]. Arama çubuğuna yazılan kelimelere göre anlık arama önerileri ve sonuçları gösterilir, böylece kullanıcı yazdıkça ilgili kayıtları anında görür. Uygulama, aktif filtreleri arayüzde göstererek kullanıcının hangi filtrelerin uygulandığını takip etmesini sağlar ve tek tıkla filtreleri temizlemeye olanak tanır [9]. Bu modül sayesinde özellikle ürün ve müşteri sayısı arttığında dahi istenilen kaydı bulmak oldukça kolaylaşır.

3. Kod yapısı ve klasör organizasyonu

Projenin kod depoları **frontend** (ön yüz) ve **backend** (arka yüz) olmak üzere iki ana bölümden oluşur. Bu, modern web geliştirme pratiğine uygun bir **monorepo** yapısı gibidir: istemci tarafı uygulama ve sunucu tarafı API kodları ayrık fakat aynı repodadır. Kod organizasyonu, anlaşılır bir klasör yapısı ile modüler hale getirilmiştir:

- **Frontend** (Next.js uygulaması) klasör yapısı:
- `frontend/app`: Next.js'in sayfa ve yönlendirme yapısını barındıran dizin. Bu proje Next.js 13+ sürümünün **App Router** özelliğini kullanmakta; dolayısıyla sayfalar ve API rotaları bu klasörde tanımlanmış durumda.
- `frontend/components`: Uygulama genelinde tekrar kullanılabilen arayüz bileşenleri burada tutulur [10]. Örneğin butonlar, form elemanları, tablo bileşenleri veya harita bileşeni (`DeliveryMap` gibi sipariş haritası) bu klasör altındadır.
- `frontend/lib`: İstemci tarafında kullanılan yardımcı fonksiyonlar ve servis katmanı kodları. Örneğin API isteklerini merkezi bir yerden yapmak için yardımcı bir `api` fonksiyonu veya form doğrulama yardımcıları olabilir. Projede ayrıca `utils.ts` gibi yardımcı dosyalar da bu lib altında tanımlıdır.
- `frontend/types`: TypeScript için özel tip tanımlamalarının yer aldığı klasör. Veritabanı şemasına uygun tipler veya componentlerin prop tipleri burada tanımlanarak tip güvenliği sağlanmıştır.

- **Backend** (Node.js API) klasör yapısı:

- `backend/controllers`: Uygulamanın iş mantığını gerçekleştiren **kontrolör** fonksiyonları bu klasörde yer alır [11]. Her bir kaynak (ürün, müşteri, sipariş vb.) veya servis alanı için ayrı controller dosyaları bulunur. Örneğin `productController.js` ürünlerle ilgili CRUD işlemlerini, `orderController.js` sipariş işlemlerini içerir. Bu fonksiyonlar, gelen istekleri işler, veritabanı ile etkileşime geçer ve uygun yanıtı hazırlar.
- `backend/routes`: API uç noktalarının (endpoint) tanımlandığı Express router modüllerini içerir [11]. Projede farklı işlevler gruplara ayrılarak ayrı rota dosyaları oluşturulmuştur (örn. `productRoute.js`, `customerRoute.js`, `orderRoute.js` gibi). Bu dosyalarda ilgili URL patikaları belirlenir ve her bir HTTP isteği tipine karşı çağrılacak controller fonksiyonu atanır. Örneğin `/api/products` altında POST isteği `createProduct` controller'ına yönlendirilir.
- `backend/prisma`: Veritabanı şeması ve göç (migration) dosyalarını içerir [12]. Burada Prisma ORM'in kullandığı `schema.prisma` dosyası bulunur. Bu dosyada veritabanındaki tabloların (Prisma terminolojisiyle **model**lerin) tanımı yapılmıştır. Ayrıca Prisma ile oluşturulan migration dosyaları da bu klasör altındadır ve veritabanı versiyonlama tarihçesini tutar.
- `backend/middleware`: Uygulamanın Express katmanında kullanılan ara katman (middleware) fonksiyonları bu klasörde yer alır [12]. Örneğin kimlik doğrulama için `authMiddleware.js` (JWT token doğrulaması yapar), hata yakalama için global error handler, veya dosya yükleme işlemleri için `multer` konfigürasyonunu içeren middleware gibi kodlar burada bulunabilir.
- **Diğer önemli dizinler:** Back-end tarafında ayrıca `backend/utils` (JWT oluşturma, yardımcı araçlar), `backend/config` (ortam değişkenleri ve yapılandırmalar, ör. `env.js`) ve gerekli görülen yerlerde `backend/services` veya `backend/repositories` gibi alt modüller de tanımlanmıştır. Örneğin proje kodunda dosya içe/dışa aktarma işlemleri için `file-import-service` ve `file-export-service` controller'ları, ya da mağaza oluşturucu için `store-builder-service` modülleri mevcuttur. Bu modüler yapı, kodun bakımını ve geliştirilmesini kolaylaştırmakta, belirli bir özelliğin kodlarını tek bir yerde toplamaktadır.

Bu klasör organizasyonu sayesinde frontend ve backend geliştiricileri kendi alanlarında rahatça çalışabilir; aynı zamanda proje genelinde tutarlılık sağlanmıştır. İsimlendirmeler ve konumlar tutarlı olduğu için, örneğin bir ürünle ilgili kod değişikliği yapılacağı zaman ilgili controller ve route dosyaları hızlıca bulunabilir.

4. Kullanılan teknolojiler ve bağımlılıklar

Projede güncel web teknolojileri ve popüler kütüphaneler kullanılarak **modern bir teknoloji yığını (tech stack)** oluşturulmuştur. Genel olarak **React & Next.js tabanlı bir frontend** ile **Node.js tabanlı bir backend** mimarisi benimsenmiştir. Aşağıda başlıca teknoloji ve kütüphaneler özetlenmiştir:

- **Frontend Teknolojileri:** Ön yüzde **Next.js** framework'ü kullanılmıştır [13]. Next.js, React tabanlı bir uygulamanın hem sunucu taraflı render (SSR) hem de istemci taraflı dinamik özelliklerini etkin bir şekilde birleştirir. Bu sayede uygulamanın SEO dostu, hızlı ve yapılandırılmış bir arayüzü vardır. Arayüz tamamen **React** ile oluşturulmuştur ve **TypeScript** ile yazılarak kodda tip güvenliği sağlanmıştır [13]. Stil ve tasarım tarafında **Tailwind CSS** kullanılmaktadır [14]. Tailwind, yardımcı sınıflarla hızlı ve tutarlı bir biçimlendirme imkânı verirken, proje içerisinde **Shadcn UI** bileşen kütüphanesi ile entegre çalışmıştır [15]. Shadcn UI, Radix UI tabanlı etkileşimli bileşenleri Tailwind temalarıyla sunan bir UI setidir; proje bu sayede hazır tasarım bileşenlerini (modal pencereler, açılır menüler, sekmeler vb.) hızlıca kullanabilmiştir. Veri tablolarının etkin yönetimi için **TanStack Table (React Table)** kütüphanesi entegre edilmiştir [16] – bu, ürün ve müşteri listeleri gibi tablolarda sıralama, sayfalama ve gruplayabilme gibi ileri özellikleri mümkün kılar. İkon seti olarak **Lucide Icons** tercih edilmiştir [16]; Lucide, Feather Icons tabanlı modern ve özelleştirilebilir ikonlar sunar. Harita gösterimleri için **Leaflet** ve onun React sarmalayıcısı **React-**

Leaflet kullanılmaktadır (OpenStreetMap entegrasyonu için) – bu sayede sipariş teslimat haritası interaktif olarak gösterilir. Tarih seçimi gibi işlemler için **React Day Picker**, bildirimler için **React Hot Toast**, form yönetimi için **React Hook Form**, animasyonlar için **Framer Motion**, sürükle-bırak işlemleri için **@dnd-kit** kütüphanesi gibi pek çok yardımcı kütüphane de kullanılmıştır. Ek olarak proje, grafiksel gösterimler ve raporlama için **Recharts** (grafik/chart çizim kütüphanesi) ve PDF çıktıları almak için **jspdf** gibi araçları da bağımlılıklarında barındırmaktadır. Bütün bu teknolojiler, frontend tarafında zengin bir kullanıcı deneyimi ve güçlü bir geliştirme altyapısı sağlamaktadır.

- **Backend Teknolojileri:** Arka planda **Node.js** çalışmakta ve web sunucusu olarak minimalist ve yaygın **Express.js** framework'ü kullanılmaktadır [17]. Express, RESTful API'ların tanımlanması ve middleware yapısıyla esnek bir sunucu ortamı sunar. Veriler, ilişkisel bir veritabanı olan **PostgreSQL**'de tutulur ve veritabanı erişimi için **Prisma ORM** kullanılmaktadır [18]. Prisma, veritabanı ile etkileşimi kolaylaştıran ve TypeScript desteğiyle tip güvenliği sunan modern bir ORM'dir; proje içerisinde `schema.prisma` ile tanımlanan modeller Prisma tarafından JavaScript/TypeScript nesneleri olarak kullanılabilmektedir. Kimlik doğrulama için **JWT (JSON Web Token)** mekanizması tercih edilmiştir; backend'de **jsonwebtoken** paketi kullanılarak giriş yapmış kullanıcılar için token üretimi ve doğrulaması yapılır [19]. Kullanıcı şifreleri, güvenlik için **bcrypt** ile hash'lenerek veritabanında saklanır [20]. API isteklerinde CORS politikalarını yönetmek için **cors** paketi ve gelen isteklerdeki çerezleri okumak için **cookie-parser** kullanılmaktadır [21]. Uygulamada dosya yükleme işlemleri (ör. ürün fotoğrafları) için **Multer** middleware entegre edilmiştir [22]. Arka uçtaki bazı işlemler için ek yardımcı kütüphaneler de projeye dahildir: Örneğin, ürün ve müşteri verilerinin dışa aktarılması/içe aktarılması amacıyla **csv-parse** (CSV dosyalarını okuma) ve **ExcelJS** (Excel dosyalarını okuma/yazma) kütüphaneleri kullanılmaktadır [23]. HTTP isteklerini sunucu tarafında da yapabilmek için **Axios** paketi hem frontend hem backend'de dahil edilmiştir (örneğin, başka bir servisten veri çekmek gerekirse kullanılabilir) [20]. Loglama ve hata ayıklama için **morgan** ve **debug** gibi küçük araçlar da bulunmaktadır. Tüm bu teknolojiler kombinasyonu, backend tarafında güvenli, ölçeklenebilir ve bakımı kolay bir altyapı oluşturur.

- **Geliştirme Araçları ve Diğer:** Proje kurulumu için Node.js ve paket yöneticisi (npm veya Yarn) gereklidir [24]. Versiyonlama ve işbirliği Git/GitHub üzerinden yürütülür. Kod kalitesini korumak için ESLint tabanlı bir linter yapılandırılmıştır (Next.js ile gelen ESLint konfigürasyonu kullanılmıştır) ve kod stilleri için Prettier dahildir [25]. Back-end kısmı JavaScript (TypeScript'e geçiş hazırlığı da gözlenebilir) kullanırken, front-end tamamen TypeScript ile yazılmıştır. Ortam değişkenleri (veritabanı URL'si, port, JWT gizli anahtarı vb.) proje kökünde `.env` dosyalarıyla yönetilir ve **dotenv** paketiyle yüklenir [23]. Testler için jest/mocha gibi bir çerçeveye dair doğrudan bir bağımlılık listelenmemiş olsa da proje yapısında test komutları tanımlanmıştır (örn. `npm test`) [26], bu da test altyapısının planlandığını gösterir.

Özetle, proje modern bir full-stack uygulaması geliştirmek için gereken tüm temel teknolojileri içermekte ve ek olarak özel amaçlı birkaç kütüphane ile zenginleştirilmiş bulunmaktadır. Bu teknoloji seçimi, geliştiricilerin üretkenliğini artırırken, son kullanıcıya da hızlı ve interaktif bir deneyim sunmayı hedefler.

5. Veritabanı modeli ve veri akışı

Veritabanı modeli (Schema): Proje, **PostgreSQL** veritabanını kullanmakta ve veritabanı şeması Prisma ORM aracılığıyla tanımlanmaktadır. Şema, e-ticaret ve envanter yönetimi alanındaki çeşitli varlıkları ve

bunların ilişkilerini kapsayacak şekilde oldukça kapsamlı tasarlanmıştır. Temel veri modelleri ve aralarındaki ilişki grupları şu şekilde özetlenebilir:

- **Ürün ve Katalog Yönetimi:** Ürünlerle ilgili bilgiler birden fazla tabloya dağıtılarak normalleştirilmiştir. **Kategori (Category)**, **Marka (Brand)** ve **Tedarikçi (Supplier)** modelleri, ürünleri sınıflandırmak için kullanılır; her biri ürünlerle bire-çok ilişkisine sahiptir (örn. bir kategori birçok üründe kullanılabilir) [27] [28]. **Ürün (Product)** modeli, envanterdeki ürünlerin ana verilerini tutar: ürün adı, stok kodu (SKU), fiyat, açıklama, barkod gibi alanlar ve ürünün ait olduğu kategori, marka, tedarikçi gibi ilişkisel alanlar bu tabloda bulunur [29]. Ürün tablosu ayrıca ürünün satış durumu (taslak/aktif), görünürlük durumu (listelenebilir/gizli) ve vergi oranı gibi e-ticaret açısından önemli alanlar içerir. Bir ürüne ait birden fazla görsel olabileceği için ayrı **Ürün Görselleri (ProductImage)** modeli tanımlanmıştır; bu tablo ürün-id ile ilişkili olarak eklenen resimlerin URL ve açıklamalarını saklar. Benzer şekilde, ürünlerin farklı varyasyonlarını (örneğin beden veya renk çeşitleri) yönetmek için **Ürün Varyantı (ProductVariant)** modeli bulunur; her varyant kendi SKU ve fiyat bilgisine sahip ayrı bir satırdır ancak ana Ürün ile ilişkilidir [30]. Ürün özelliklerini esnek şekilde tanımlayabilmek amacıyla şemada **Ürün Özelliği (ProductAttribute)** ve **Özellik Değeri (ProductAttributeValue)** tabloları da yer alır; bu yapıyla örneğin "Renk" diye bir özellik tanımlanıp alt değerleri (kırmızı, mavi vb.) tutulabilir ve **ürün-özellik değeri** ilişkisini kurmak için bir bağlantı tablosu kullanılır. Ürünlere etiketler atamak için **Tag (Etiket)** modeli tanımlanmıştır ve ürünlerle çoktan-çoğa bir ilişki içindedir [31] [32] (bir ürün birden fazla etiket taşıyabilir, bir etiket de birden fazla ürüne atanabilir). Ayrıca ürünler arasında ilişkiler kurmak için (ör. "benzer ürünler" veya "birlikte satılanlar" gibi) **ProductRelation** adında bir model bulunmaktadır; bu tablo bir ürünün diğer bir ürünle nasıl bir ilişkide olduğunu (RELATED, UPSELL, CROSS_SELL gibi) tutar. Özetle, ürünlere dair tüm bilgiler (kategori, marka, tedarikçi, etiket, varyant, görsel, özellik) ayrı tablolarla yapılandırılmış ve **Product** tablosu merkez olmak üzere birbirine bağlanmıştır [29]. Bu sayede veritabanı modeli, ürün kataloğunu son derece ayrıntılı ve ölçeklenebilir şekilde temsil etmektedir.

- **Stok ve Depo Yönetimi:** Envanter takibinin kritik bir parçası olarak, sistemde **Depo (Warehouse)** modeli bulunmaktadır. Bir işletmenin birden fazla fiziksel deposu (örneğin farklı mağaza veya şubelerdeki depolar) olabileceği varsayılarak, her depo için adres, kapasite, yöneticisi gibi bilgiler bu tabloda saklanır. Her depo ile ürünlerin stok ilişkisini tutmak için **Stok (Stock)** modeli tanımlanmıştır; bu model, belirli bir depoda belirli bir üründen kaç adet bulunduğunu kaydeder ve ürün-depo çiftini benzersiz kabul eder [33]. Stok tablosunda ayrıca versiyon numarası gibi alanlar bulunur, böylece stok güncellemelerinde optimistik kilitleme veya değişim izi sağlanabilir. **Envanter Hareketi (InventoryTransaction)** modeli ise stok miktarındaki değişimleri tarihçesiyle kaydeder [34]. Bu tabloda bir ürünün belirli bir depodaki girişi veya çıkışı ayrı bir kayıt olarak tutulur; işlem türü (IN = giriş, OUT = çıkış) ve miktar bilgileriyle birlikte hangi kullanıcı tarafından yapıldığı ve hangi ürünü etkilediği kayıt altına alınır. Örneğin, yeni mal girişi olduğunda veya satış yapıldığında bu tabloya bir kayıt eklenir. Bu yapı sayesinde sistem, her ürünün her depodaki anlık stok seviyesini (**Stock**) bilirken, geçmişteki stok değişimlerini de (**InventoryTransaction**) takip edebilir. Warehouse modeli ile bu iki model (Stock ve Transaction) arasında bire-çok ilişkiler vardır (her depo birçok stok kaydına ve stok hareketine sahip olabilir) [35] [36]. Stok ve depo yönetimi modelleri, ürün modelleriyle entegre çalışarak tam bir envanter kontrolü sağlar.

- **Müşteri ve İletişim:** Müşteri bilgilerini tutmak için **Customer (Müşteri)** modeli kullanılır. Bu tabloda müşterinin adı, iletişim bilgileri (e-posta, telefon), adresi (şehir, ülke, posta kodu vb.) gibi alanlar bulunur [37]. Ayrıca müşteriye dair bazı ek bilgiler de tutulabilir: örneğin doğum tarihi, şirket adı (eğer kurumsal müşteri ise), vergi kimlik numarası, müşteri notları gibi alanlar mevcuttur. Müşteri kaydının önemli bir özelliği, **segmentasyon** bilgisidir – her müşteri bir segment değeri alabilir (REGULAR, VIP, NEW, vb.) [38]. Segment, pazarlama veya müşteri ilişkileri

stratejileri için müşterileri gruplamaya yarar (projede CustomerSegmentType enum'ı ile tanımlıdır, ör. VIP, PREMIUM gibi değerler içerir [39]). Müşteri modeli, sistemde işlemleri yapan kullanıcıyla ilişkilendirilmiştir; yani bir kullanıcı (yönetici) tarafından eklenen müşteriler userId ile eşleştirilir [38]. Müşteri ile **Sipariş (Order)** arasında bire-çok ilişki vardır: bir müşteri birden fazla siparişe sahip olabilir [38]. Müşterilerle yapılan önemli iletişimlerin kaydı için **CommunicationLog (İletişim Günlüğü)** modeli bulunmaktadır. Bu model, bir müşteriyle gerçekleştirilen arama, e-posta veya toplantı gibi iletişimlerin türünü, konusunu, içeriğini ve tarihini kaydeder [40]. Her iletişim kaydı ilgili müşteri ve kullanıcı ile ilişkilendirilmiştir (örn. hangi müşteriyle, hangi kullanıcı tarafından iletişim kurulmuş) [40]. Bu sayede, müşteri hizmetleri veya satış temsilcileri, müşterilerle geçmişte ne görüşüldüğünü sisteme girebilir ve daha sonra bu kayıtlar üzerinden ilişki yönetimini sürdürebilir. Müşteri modülü, işletmenin müşteri veritabanını ve etkileşim geçmişini düzenli tutarak, müşteri memnuniyetini artırmaya yardımcı olur.

- **Sipariş ve Ödeme:** Satış işlemleri, **Sipariş (Order)** modeli ile temsil edilir. Sipariş tablosu, bir siparişe ait tüm temel bilgileri içerir: benzersiz sipariş numarası, siparişi oluşturan kullanıcı ve ilgili müşteri referansı, siparişin durumu (durum alanı bir enum olup PENDING, PROCESSING, SHIPPED, DELIVERED gibi değerler alabilir), ödeme tipi (nakit, kredi kartı, havale vb. şeklinde PaymentType enum'ı) ve ödeme durumu (ödendi/ödenmedi şeklinde bir bayrak) gibi alanlar bulunur [41]. Ayrıca siparişe uygulanan kargo ücreti, vergi tutarı, indirim tutarı ve siparişin toplam tutarı gibi finansal alanlar da yer alır. Siparişle ilgili müşteri notları, teslimat adresi, kargo takip numarası ve kargo firması adı gibi bilgiler de sipariş tablosunda tutulur [42]. Bu sayede, örneğin kargo entegrasyonu kapsamında siparişin hangi numara ile takip edileceği ve hangi taşıyıcı (kargo firması) ile gönderildiği kayıtlıdır. Bir siparişin içeriğindeki ürünler, ayrı bir **Sipariş Kalemi (OrderItem)** modeli ile tutulur. OrderItem, bir siparişe ait tek bir ürünü temsil eder; içinde ilgili ürünün ID'si, adı, SKU kodu, sipariş edilen miktar, birim fiyatı, vergi oranı, indirim ve o kalemin toplam fiyatı gibi alanlar vardır [43]. Her siparişin bir veya daha fazla kalemi olabilir ve OrderItem kayıtları sipariş ile ilişkilidir (orderId ile). Sipariş ilerleyişi esnasında yapılan durum değişiklikleri ve notlar **Sipariş Günlüğü (OrderLog)** modeli ile kaydedilir. OrderLog, bir siparişin durumu değiştiğinde (örn. kargoya verildi, teslim edildi, iptal edildi vs.) veya siparişle ilgili önemli bir olay gerçekleştiğinde (ör. müşteri arandı, not eklendi) bunun zaman damgasıyla birlikte kayıt altına alınmasını sağlar [44]. OrderLog, siparişin hangi yeni duruma geçtiğini, bu değişikliği yapan kullanıcıyı ve varsa bir mesaj/detay içerir. Böylece siparişin yaşam döngüsü geçmişe dönük olarak izlenebilir (bu veriler arayüzde timeline olarak gösterilmektedir). Ödeme tarafında ise **Payment (Ödeme)** modeli mevcuttur. Bu model, bir siparişe ait ödeme girişlerini tutar [45]. İçinde ödemenin tutarı, para birimi, ödeme yöntemi (PaymentMethod alanı, ör. Kredi Kartı, Nakit), ilgili ödeme işlem numarası (örneğin banka işlemi veya kredi kartı işlemi ID'si) ve ödemenin durumu (Pending, Completed, Failed gibi PaymentStatus enum'ı) yer alır [45]. Bu sayede eğer bir siparişe dair birden fazla ödeme hareketi (ön ödeme, kalan ödeme, iade vb.) varsa hepsi kayıt altına alınabilir. Payment tablosu da sipariş ile bire-çok bağlıdır. Tüm sipariş ve ödeme modelleri, kullanıcı (yönetici) hesapları ile ilişkilendirilmiştir; böylece hangi işlemin hangi yönetici tarafından yapıldığı kaydedilir. Genel olarak sipariş yönetimi veritabanı modeli, gerçek bir e-ticaret sipariş sürecinin tüm adımlarını (sepet -> sipariş -> ödeme -> teslimat -> iade vs.) kapsayacak esneklikte tasarlanmıştır.

- **Kullanıcı (User) ve Yetkilendirme: Kullanıcı** modeli, sistemi kullanan admin/yönetici hesaplarını temsil eder. Her kullanıcının benzersiz bir e-posta adresi, kullanıcı adı ve hash'lenmiş şifresi bu tabloda saklanır [46]. Kullanıcı modeli, sistemdeki hemen hemen tüm diğer modellere referans verir: Örneğin **Product**, **Customer**, **Order** gibi tablolarda userId sütunu bulunur ve ilgili kaydı oluşturan kullanıcıyı işaret eder [47]. Bu sayede, bir işlemi kimin yaptığını denetlemek veya kullanıcı bazlı filtrelemeler yapmak mümkün olur. User tablosu ile diğer tablolar arasında bire-çok ilişkiler tanımlanmıştır (bir kullanıcı birçok ürün, müşteri, sipariş vs. kaydı oluşturabilir) [47].

İleride rol tabanlı yetki ayrımları eklenecek olursa, User tablosuna bir rol alanı eklenerek farklı yetki seviyelerine sahip kullanıcılar tanımlanabilir; ancak mevcut şemada böyle bir alan yok, dolayısıyla proje tek tip kullanıcı (yönetici) varsayımıyla çalışmaktadır. Kullanıcı modeli aynı zamanda sistemde mağaza oluşturma özelliğiyle de ilişkilidir (her kullanıcının bir veya daha fazla mağaza sayfası olabilir, aşağıda değinilmiştir).

- **Mağaza Sayfası Oluşturucu (Ek Özellikler):** Veritabanında, ana envanter yönetimi fonksiyonlarından bağımsız olarak, basit bir **mağaza sayfası oluşturma** özelliğine dair tablolar yer alır. Bu kısım, kullanıcıların sürükle-bırak bileşenler ile kendi e-ticaret vitrin sayfalarını tasarlayıp yayınlayabilmesini hedefleyen ek bir modül gibidir. **Store (Mağaza)** modeli, kullanıcının oluşturduğu çevrimiçi mağaza sitelerini temsil eder; her mağazanın bir adı, açıklaması ve durumu bulunur ve ilgili kullanıcı hesabıyla ilişkilidir [48]. **StorePage** modeli, her mağaza için oluşturulmuş tekil sayfaları ifade eder (ör. ana sayfa, hakkında sayfası vb.), ve Store ile ilişkilidir [49]. Bu sayfalar içerisinde kullanılacak bileşenler için **Component** ve **ComponentVersion** modelleri tanımlanmıştır; bunlar önceden tanımlı arayüz bileşenlerinin şablonlarını ve versiyonlarını saklar. **PageSection** modeli ise bir StorePage içindeki her bir bileşenin yerini, özelliklerini ve hiyerarşisini tutar [50] [51] (bir sayfa içinde birden fazla bileşen ve alt bileşen yer alabilir). Bu mağaza oluşturucu modelleri, projeye temel envanter ve sipariş yönetimine ek olarak basit bir CMS (içerik yönetimi) kabiliyeti kazandırmaktadır. Ancak bu özellik, projenin merkezî işlevi olmayıp destekleyici bir eklenti gibidir. Kod tabanında mağaza oluşturma ile ilgili route'lar ve controller'lar bulunmaktadır, ancak kullanımı isteğe bağlıdır ve son kullanıcı arayüzünde bu modülün tam olarak entegre edilip edilmediği proje durumuna bağlıdır.

Yukarıda özetlenen veritabanı modeli sayesinde uygulama, gerçek bir envanter ve satış yönetimi sisteminde ihtiyaç duyulabilecek hemen her varlığı ve ilişkiyi temsil edebilmektedir. Bu **kapsamlı şema**, gelecekte yeni özelliklerin eklenmesi (ör. iade yönetimi, kampanyalar, kullanıcı rollerinin eklenmesi vb.) için de sağlam bir temel sunar.

Veri akışı (Uygulama mimarisi): Sistemde veri akışı, istemci (frontend) ile sunucu (backend) arasındaki REST API etkileşimine dayanmaktadır. Next.js tabanlı istemci uygulaması, kullanıcı etkileşimleri sonucunda uygun backend API uç noktalarına HTTP istekleri gönderir. Proje yapılandırmasında frontend uygulaması, istekleri yapacağı API adresini ortam değişkeninden almaktadır (örn. `NEXT_PUBLIC_API_URL=http://localhost:3001/api` şeklinde) [52]. Bu sayede, tarayıcıda çalışan arayüz kodu, doğrudan backend'in `/api` ile başlayan REST servislerine istek atar.

Sunucu tarafında Express.js, bu istekleri karşılar. `backend/routes` altında tanımlanmış rotalar, gelen istek URL'sine ve HTTP metoduna göre ilgili **controller** fonksiyonuna yönlendirme yapar. Uygulamada güvenlik için tüm kritik API rotaları **kimlik doğrulama** ile korunmaktadır: Express tarafında yazılmış `requireAuth` adlı middleware, JWT token doğrulaması yaparak sadece geçerli oturum sahibi kullanıcıların verilere erişmesine izin verir [53]. Örneğin, ürünleri listeleme, ekleme, silme gibi tüm `/api/products` isteklerinde bu middleware kullanılmakta, böylece kullanıcı giriş yapmamışsa isteğe yanıt olarak 401 Yetkisiz hatası dönülmektedir [53]. JWT, kullanıcı giriş yaparken (örn. `/api/auth/login` rotasında) oluşturulur ve istemci tarafına gönderilir; istemci bu token'ı genellikle bir çerezde veya Authorization başlığında saklar. Sonraki her API çağrısında token sunucuya iletilir ve `authMiddleware` tarafından kontrol edilir. Bu mekanizma, veri akışının güvenliğini sağlar.

Bir API isteği doğrulandıktan sonra ilgili **controller** fonksiyonu çalışır. Controller katmanı, iş mantığının uygulandığı yerdir. Örneğin, istemci tarafından yeni bir ürün ekleme isteği geldiyse `productController.createProduct` fonksiyonu devreye girer. Bu fonksiyon, gelen isteğin gövdesinden (body) yeni ürün verilerini alır, gerekirse doğrulamalar yapar (örneğin aynı SKU kodu ile

ürün varsa hata döner) ve ardından veritabanına kaydetmek için Prisma ORM'ini kullanır. Prisma sayesinde bir veritabanı ekleme işlemi, tek satırlık bir çağrıyla gerçekleştirilebilir: Örneğin `prisma.product.create({ data: {...} })` şeklinde bir kod yeni ürünü veritabanına ekler [54]. Aşağıda, ürün oluşturma akışına dair koddan küçük bir örnek verilmiştir:

```
// Yeni ürün oluşturma - POST /api/products
productRouter.post(
  "/",
  requireAuth,                      // Kimlik doğrulama middleware'i, JWT kontrolü yapar
  productImageUpload.single("image"), // Ürün resmini işleyecek Multer middleware'i
  createProduct                     // İstek verisini veritabanına kaydeden controller fonksiy
);
```

Örnek route tanımı (`productRoute.js`) [55]. Yeni ürün oluşturma isteği, kimlik doğrulama ve dosya yükleme ara yazılımlarından geçtikten sonra `createProduct` fonksiyonuna yönlendirilir.

Controller, Prisma ile veritabanı işlemine dair sonucu aldıktan sonra (örn. yeni eklenen ürün satırı) bunu uygun bir yanıt olarak istemciye döner. Genellikle yanıtlar JSON formatındadır. İstemci tarafında Next.js, gelen yanıtı alır ve kendi durumunu günceller. Örneğin yeni eklenen ürünün bilgilerini alarak ürün listesini günceller ya da bir işlem başarılı olduğunda kullanıcıya bildirim gösterir.

Veri akışının bir diğer boyutu da **gerçek zamanlı güncellemeler**dir. Bu projede esas itibariyle istek-cevap modeli kullanılmış olup, anlık bildirimler veya web soketleri ile push mekanizmaları bulunmamaktadır. Ancak bazı kullanıcı deneyimi geliştirmeleri için istemci tarafında **polling** (belirli aralıklarla durumu çekme) veya filterleme sırasında **debounce** tekniği (kullanıcı yazmayı bitirene kadar bekleme) gibi yöntemler kullanılıyor olabilir. Örneğin, siparişlerin durumu belli aralıklarla güncellenmek istenirse, istemci tarafında periyodik bir istek tasarlanabilir ya da kullanıcı bir sayfayı yeniden yüklediğinde sunucudan güncel veriler çekilir.

Ayrıca, proje kapsamında veri akışının özel bir kısmı da **dosya içe/dışa aktarma** işlemleridir. Kullanıcı, arayüzden örneğin "Ürünleri Dışa Aktar" dediğinde, istemci tarafı ilgili backend API'ına (örn. `/api/files/export?type=products&format=excel`) bir istek gönderir. Sunucu tarafında dosya oluşturma controller'ı (örn. `fileExportController.exportProductsToExcel`) çalışır; bu fonksiyon veritabanından tüm ürünleri çeker ve istenen formata göre bir dosya (Excel, CSV, JSON, XML) oluşturarak yanıt olarak iletir [56] [57]. Bu yanıt, HTTP indirme olarak kullanıcıya sunulur. Benzer şekilde veri içe aktarma (import) işlemlerinde istemci, kullanıcı tarafından yüklenen dosyayı backend'e gönderir ve backend bu dosyayı okuyup veritabanına işler.

Özetle, istemci ve sunucu arasındaki etkileşim net, düzenli ve güvenli bir şekilde kurulmuştur. **Frontend**, kullanıcıyla etkileşimi yönetip doğru API çağrılarını yaparken; **Backend** bu çağrıları işler, veritabanıyla etkileşime geçer ve sonuçları geri döndürür. Bu mimari ayrım, ölçekleme ve bakım kolaylığı sağlar: Örneğin, veritabanı değiştirilmek istenirse sadece Prisma şeması ve ilgili controller işlemleri güncellenecek, istemci bu değişiklikten etkilenmeyecektir (API arayüzü sabit kaldığı sürece). Veri akışının her adımında loglama ve hata yakalama mekanizmaları da bulunmaktadır; böylece olası hatalar konsolda veya loglarda yakalanıp düzeltilebilir. Sistemin bütününde, verinin girişinden çıkışına kadar tutarlı bir akış ve veri bütünlüğünü koruyan kontrol noktaları (doğrulamalar, kimlik denetimi vs.) mevcuttur.

6. Öne çıkan özellikler ve kullanıcı senaryoları

Bu bölümde, uygulamanın öne çıkan özellikleri gerçekçi bir senaryo eşliğinde ele alınacaktır. Bir işletme yöneticisinin sistemi nasıl kullanabileceğini adım adım düşünelim:

Senaryo 1 – Envanterin Oluşturulması: İlk kez sisteme giriş yapan bir yönetici, öncelikle **ürün kataloğunu oluşturmak** isteyecektir. Yönetici, web arayüzündeki ilgili formu kullanarak yeni bir ürün ekler. Örneğin "ABC Laptop" adında bir ürün ekleme ekranına girilir; ürünün kategorisi "Elektronik" olarak seçilir, markası ve tedarikçisi tanımlanır, stok kodu (SKU) girilir, satış fiyatı ve maliyet fiyatı belirtilir. Ayrıca ürünün açıklaması ve barkod numarası gibi bilgiler doldurulur. İstenirse ürün görseli de bu aşamada yüklenebilir. "Kaydet" dendiğinde, sistem bu yeni ürünü veritabanına işler ve ürün listesine ekler. Liste görünümünde her ürün bir kart veya satır olarak gösterilir; örneğimizde ABC Laptop artık ürünler listesinde görünecektir. Ürün kartında, ürünün bağlı olduğu kategoriye göre bir ikon otomatik belirmiştir (örneğin Elektronik kategorisi için bir cihaz ikonu). Yine aynı ekranda ürünün mevcut stok durumu da görüntülenir (başlangıçta stok 0 olabilir veya yöneticinin belirttiği bir başlangıç stoğu varsa o sayı görünür). Sistem, **ürün ekleme, düzenleme, silme ve listeleme** işlemlerini kolaylaştırdığı için yönetici çok hızlı bir şekilde tüm ürünlerini sisteme girebilir [2] . Ardından yönetici, **müşteri bilgilerini eklemek** üzere müşteri yönetimi modülüne geçer. Örneğin bir müşterinin adı, soyadı, e-posta adresi ve telefon numarası girilerek yeni bir müşteri profili oluşturulur. Adres bilgileri ve notlar gibi detaylar da eklenebilir. Kayıt işlemi sonrası müşteri listesinde bu yeni müşteri görünür hale gelir. Telefon numarası girişi yapılırken sistemin otomatik formatlama özelliği devrededir – yönetici "05321234567" yazsa bile sistem bunu "0 (532) 123 4567" formatına çevirebilir ve ülke kodu alanı sayesinde uluslararası formatta da kayıt tutabilir [5] . Bu sayede müşteri verileri tutarlı biçimde saklanır.

Senaryo 2 – Sipariş Oluşturma ve Takibi: Ürünler ve müşteriler sisteme girildikten sonra, bir satış gerçekleştiğinde **yeni bir sipariş** oluşturulabilir. Yönetici, arayüzdeki "Yeni Sipariş" ekranına giderek bir sipariş kaydı başlatır. Önce sipariş için ilgili müşteri seçilir (örneğin az önce eklenen müşteri), ardından siparişe eklenmek üzere ürünler seçilir. ABC Laptop ürününden 2 adet ekleyip, ayrıca listeden başka bir ürün daha (ör. bir mouse) ekleyebilir. Sistem, seçilen her ürünü bir **sipariş kalemi** olarak gösterecek; miktar ve birim fiyat bilgilerini alacaktır. Tüm kalemler eklendiğinde toplam tutarı hesaplayıp görüntüler. Yönetici ödeme türünü "Kredi Kartı" olarak işaretleyip siparişi oluşturur. Sipariş oluşturulduktan sonra, sipariş detay sayfası açılır. Başlangıçta sipariş durumu **Beklemede (PENDING)** olarak işaretlenir. Bu sayfada siparişin içeriği, müşteri bilgileri, seçilen ödeme yöntemi gibi bilgiler özetlenir. Sipariş hazırlanıp kargoya verilmeye hazır olduğunda, yönetici sipariş durumunu **Kargoya Verildi (SHIPPED)** olarak günceller. Bunu yapmak için sipariş detay sayfasında bir durum değiştirici arayüz kullanılır (örneğin bir açılır menü ya da adım adım ilerleyen bir timeline arayüzü). Durum güncellendiği anda, altta yer alan **zaman çizelgesi (timeline)** bu değişikliği yansıtır [6] . Timeline, siparişin yaşam döngüsündeki adımları grafiksel olarak gösteren bir bileşendir; örneğimizde "Sipariş Oluşturuldu -> Kargoya Verildi" adımları tarih/saat ile birlikte işaretlenmiş olacaktır. Bu arada yönetici, kargo firmasına göre bir **takip numarası** almış ise bunu da sipariş kayıtlarına ekler (sistem, takip numarası ve kargo firması bilgisini sipariş tablosunda saklayabiliyor). Kargo süreci başladıktan sonra uygulama, entegre harita özelliği sayesinde siparişin teslimat aşamalarını **harita üzerinde görselleştirir**. Sipariş detayında bir "Teslimat Haritası" bileşeni belirir ve burada Türkiye haritası üzerinde (OpenStreetMap tabanlı) bir yol çizgisi ve konum işaretçileri gözükür. Örneğin sistem varsayılan olarak İstanbul merkezini teslimat adresi kabul edip bir rota simülasyonu oluşturur: Kargonun çıktığı depo, ara transfer merkezleri ve müşterinin adresi şeklinde bir poligon çizilir. Kargonun o anki durumu **Kargoya Verildi** ise haritada kargonun güncel konumu olarak ara noktalardan biri mavi bir işaretçi ile gösterilir. Kullanıcı bu işaretçinin üzerine tıkladığında, küçük bir açılır pencerede kargo firmasının adı ve "Son konum güncellemesi: 12:30" gibi bir bilgi görüntülenir [58] . Sipariş durumu **Teslim Edildi (DELIVERED)** olarak güncellendiğinde ise harita üzerindeki rota çizgisi kesiksiz hale gelir ve bitiş noktası (müşterinin adresi) kırmızı bir işaretçi ile

“Teslimat Adresi” olarak vurgulanır [59] [58]. Bu harita entegrasyonu özelliği, OpenStreetMap altyapısını Leaflet kütüphanesi aracılığıyla kullanır ve gerçek adres-coordinat dönüşümü (geocoding) yerine simüle edilmiş verilerle çalışır – örneğin proje kodunda İstanbul içinde rastgele yakın koordinatlar üretilerek bir örnekleme yapılmıştır [60]. Bu nedenle harita, geliştirme/demo amaçlı gerçek zamanlı olmayan bir izleme sağlamaktadır. Ancak altyapı olarak bakıldığında, gerçek bir API ile entegre edildiğinde (Google Maps, Mapbox v.b.) siparişlerin gerçek konum takibini yapmak mümkün olacaktır. Timeline ve harita özellikleri birlikte, sipariş yönetimini görsel ve anlaşılır hale getirir; bir siparişin durumunun ne olduğu ve teslimata dair bilgiler tek bir ekranda kolayca takip edilebilir.

Senaryo 3 – Arama, Filtreleme ve Raporlama: Sistem kullanılmaya devam ettikçe, ürün ve müşteri sayıları artacaktır. Bu noktada **gelişmiş arama ve filtreleme** özelliği öne çıkar. Örneğin yönetici, ürün listesinde belirli bir kategoriye ait ürünleri görmek isteyebilir. Arayüzde kategori filtreleme menüsünden ilgili kategoriyi seçtiğinde, ürün listesi anında bu kategoriyle eşleşen ürünlerle güncellenir. Ayrıca fiyat aralığı filtresi de uygulayarak sadece belli bir fiyatın üzerindeki ürünleri listeleyebilir. Birden fazla filtre aynı anda etkinleştirilebilir; uygulama seçili filtreleri üst kısımda küçük etiketler (badge) olarak gösterir (ör. “Kategori: Elektronik, Fiyat: >1000 TL”) [9]. Kullanıcı dilerse tek tıkla bu filtreleri temizleyerek listeyi ilk haline döndürebilir. Benzer şekilde müşteri listesinde de arama ve filtreleme mevcuttur. Örneğin yüzlerce müşteri arasından “İstanbul” şehrinde olanları görmek için şehir filtresi uygulanabilir veya arama çubuğuna müşterinin adının bir parçası yazılarak anında arama yapılabilir [8]. Sistem, arama işlemlerinde yüksek performans için muhtemelen metin alanlarında indeksleme yapmıştır; bu sayede büyük veri setlerinde dahi arama gecikmesiz hissedilir. Bu **gerçek zamanlı arama** özelliği, kullanıcının listelerde kaybolmasını engeller ve istenen kayda hızlı erişim sağlar.

Uygulamanın bir diğer öne çıkan özelliği, **veri dışa aktarma ve raporlama** kabiliyetidir. Yönetici, belirli aralıklarla envanter raporları almak isteyebilir. Sistem, ürün verilerinin CSV, Excel, JSON veya XML formatlarında dışa aktarılmasına izin verir. Örneğin “Ürünleri Dışa Aktar (Excel)” seçeneğiyle tüm ürün listesini ve temel alanlarını içeren bir Excel dosyası anında indirilebilir [57]. Bu dosya, Excel üzerinde açılıp düzenlenebilir veya farklı paydaşlarla paylaşılabilir. Benzer şekilde müşteri listesi veya sipariş kayıtları da CSV gibi formatlarda çıkarılabilir. **Veri içe aktarma** tarafında ise, örneğin yeni bir ürün listesi toplu olarak sisteme yüklenecekse, sistem CSV/Excel import özelliği sunar: Yönetici uygun formatta hazırladığı bir dosyayı yükleyerek çok sayıda ürünü tek seferde ekleyebilir. Bu işlemler backend’deki özel servislerle (ExcelJS, csv-parse kullanılarak) gerçekleştirilir [56]. Bu özellik, özellikle başka sistemlerden veri taşınırken ya da başlangıçta mevcut envanter verilerini yüklerken büyük kolaylık sağlar.

Son olarak, uygulamada yöneticilerin karar almasına yardımcı olacak **dashboard (gösterge paneli)** ve analiz özellikleri bulunmaktadır. Anasayfada veya ayrı bir raporlama bölümünde, sistemdeki kritik metrikler ve istatistikler grafikler halinde sunulabilir. Örneğin, toplam satışlar, aylık sipariş sayıları, en çok satan ürün kategorileri gibi bilgiler **Recharts** kütüphanesi ile çizilen grafikler olarak gösterilebilir. Ayrıca müşteri kitlesine ilişkin bir **segmentasyon dağılımı** grafiksel olarak sunulabilir: Sistemdeki müşteri kayıtları hangi segmentte (örn. %10 VIP, %20 Yeni müşteri, %50 Regular, vs.) buna dair bir pasta grafik veya çubuk grafik gösterimi yapılabilir. Projede müşteri segmentasyonu için enum değerleri tanımlanmış olup (REGULAR, VIP, PREMIUM, NEW, vb. segmentler [39]), örnek bir **SegmentationDashboard** bileşeni de bulunmaktadır. Bu gösterge panelinde, müşterilerin segmentlere göre dağılımı ve belki her segment için toplam sipariş sayısı gibi bilgiler yer alır. Yine benzer şekilde ürünler için stok durumu özetleri, kritik stok seviyesinin altındaki ürün sayısı veya toplam müşteri sayısı gibi KPI (Key Performance Indicator) verileri de gösterge panelinde vurgulanabilir. Bu tür bir kuşbakışı görünüm, yöneticinin sistemdeki durumu hızlıca değerlendirmesini sağlar. Örneğin, grafikten VIP müşterilerin sayısının arttığı görülebilir veya son ayki satışların önceki aya kıyasla düştüğü anlaşılabilir. Bu bilgiler ışığında işletme stratejileri geliştirilebilir.

Özetlemek gerekirse, **Envanter & E-Ticaret Yönetim Sistemi** pratik kullanıcı senaryolarını destekleyen zengin özelliklere sahiptir. Ürün ve müşteri yönetiminden sipariş takibine, arama/filtrelemeden raporlamaya kadar uçtan uca bir çözüm sunar. Kullanıcı arayüzü modern ve etkileşimlidir; görsel bileşenler (ikonlar, harita, timeline, grafikler) ile deneyim zenginleştirilmiştir. Bu senaryolar, sistemin gerçek bir işletme ortamında nasıl değer katacağını ortaya koymaktadır.

7. Güçlü ve zayıf yönler

Projenin güçlü yönleri ve avantajları şu şekilde özetlenebilir:

- **Kapsamlı Özellik Seti:** Uygulama, envanter yönetiminden müşteri ve sipariş yönetimine kadar geniş bir yelpazede işlev sunuyor. Bir işletmenin ihtiyaç duyacağı pek çok modül (ürün, stok, müşteri, sipariş, raporlama vs.) tek bir sistemde bütünleşik halde mevcut. Özellikle ürün yönetimi konusunda kategori, varyant, özellik, etiket gibi detayların düşünülmüş olması ve sipariş tarafında teslimat takibi gibi ileri düzey özelliklerin bile eklenmiş olması, projenin ne denli kapsamlı olduğunu gösteriyor. Bu durum, kullanıcı için "tek adres" çözüm olma avantajı sağlıyor – ayrı ayrı araçlar yerine tek bir platform üzerinden tüm iş süreçlerini yürütebilir.

- **Modern Teknoloji Yığını:** Proje güncel ve yaygın olarak kabul görmüş teknolojilerle inşa edilmiş. Next.js ve React gibi frontend araçları, TypeScript desteğiyle birlikte, hem geliştirme verimini hem de uygulamanın performansını artıran unsurlar. Backend'de Node.js/Express ve Prisma kullanılması da benzer şekilde geliştirme sürecini hızlandırırken güçlü bir performans temeli sunuyor. Prisma ORM, veritabanı işlemlerinde hem tip güvenliği hem de kolaylık sağladığı için, bu tercih uzun vadede kod bakımı açısından avantajlı. Ayrıca Tailwind CSS + Shadcn UI gibi kombinasyonlar, tutarlı bir tasarım sistemi oluşturmayı kolaylaştırıyor. Bu modern stack, projeyi geleceğe dönük kılıyor ve yeni geliştiricilerin koda alışmasını da kolaylaştırıyor (çünkü kullanılan teknolojiler hakkında bol dokümantasyon ve topluluk desteği mevcut).

- **İyi Organize Edilmiş Kod ve Mimari:** Kodun frontend/backend olarak ayrılması ve iç yapıda modüler bir klasör organizasyonuna sahip olması önemli bir artı. Bu sayede takım içinde iş bölümü yapmak veya belirli bir alana odaklanmak mümkün. Örneğin front-end geliştiricisi, backend koduna minimum düzeyde girerek kendi işine odaklanabilir. Controller-route yapısının netliği, her özelliğin ilgili dosyalarda bulunması (örn. ürün işlemleri `productController` ve `productRoute` içinde) kod bakımını kolaylaştırır. Ayrıca kod içinde bolca yorum satırı ve JSDoc açıklamaları bulunması, anlaşılırlığı artırıyor. Örneğin rota tanımlarında her endpoint'in üstünde ne yaptığı açıklanmış veya controller fonksiyonlarında iş adımları Türkçe yorumlarla belirtilmiş durumda [61] [62]. Bu, projeyi yeni devralan bir geliştiricinin bile hızlıca konsepti kavramasına yardımcı olur. Güvenlik açısından da middleware kullanımı net bir şekilde projeye entegre edilmiş, bu da iyi bir mimari pratik (her istekte kimlik doğrulama kontrolü gibi) olduğunu gösterir.

- **Zengin Kullanıcı Deneyimi:** Proje, kullanıcı deneyimini zenginleştiren birçok özellikle birlikte geliyor. Örneğin harita üzerinde kargo takibi, sipariş durum timeline'ı, gerçek zamanlı arama, animasyonlu geçişler (muhtemelen Framer Motion ile) ve bildirimler, uygulamanın profesyonel bir hissiyat vermesini sağlar. Bu özellikler sadece "hoş" olmakla kalmayıp fonksiyonelliğe de katkıda bulunuyor (harita entegrasyonu, sipariş durumunu anlamayı kolaylaştırır gibi). Arayüzde kullanılan ikonlar, responsive tasarım (Tailwind ile mobil uyumlu olarak inşa edilebilir) ve Radix tabanlı etkileşimli bileşenler de kullanıcıların sistemi benimsemesini kolaylaştırır. Ayrıca dil desteği altyapısının bulunması (Next.js Intl kullanımının entegre edilmesi) potansiyel olarak uygulamayı farklı dillere çevirme imkanı sunar, bu da uluslararası kullanım için bir artıdır.

- **Esneklik ve Ölçeklenebilirlik:** Veritabanı şemasının detaylı ve ilişkisel olarak doğru kurgulanmış olması, uygulamanın ileride genişlemesine olanak tanıyor. Örneğin şu an belki tüm kullanıcılar yönetici seviyesinde, ancak sistem mimarisi rolleri eklemeyi zorlaştırmayacak şekilde planlanmış (User modeli zaten var, ilişkiler userId üzerinden). Yine aynı şekilde ürünler için ek özellikler veya sipariş sürecine iade, değişim gibi ek modüller eklenmek istenirse altyapı bunu kaldırabilecek durumda. Tek bir şirket yerine çoklu şirket desteği eklenmek istense (multi-tenant yapılar), belki Store kavramı bunun için şimdiden eklenmiş bile. Bu yönüyle proje, bir **MVP** (Minimum Viable Product) olmanın ötesine geçip, daha ileri düzey kullanım senaryolarını da göz önünde bulunduruyor izlenimi veriyor.

Buna karşın, projenin bazı zayıf yönleri veya iyileştirilebilecek noktaları da mevcut:

- **Karmaşıklık ve Öğrenme Eğrisi:** Proje bir hayli geniş kapsamlı olduğundan, yeni bir geliştirici veya son kullanıcı için ilk bakışta karmaşık gelebilir. Özellikle veritabanı modeli çok sayıda tablo içeriyor (ürünle ilgili belki 10'dan fazla tablo var). Eğer tüm bu özellikler son üründe kullanılmayacaksa, bazılarının varlığı gereksiz karmaşıklık yaratabilir. Örneğin "Component" ve "StorePage" gibi mağaza oluşturucu özellikleri, envanter yönetimi bağlamında ikincil öneme sahip. Kodda bu bölümlerin varlığı genel kod tabanını büyütmüş ve anlaşılmasını biraz zorlaştırmış olabilir. Eğer bu özellik tam entegre değilse, developer açısından zihinsel yük yaratabilir (hangi kodun gerçekten kullanıldığı, hangisinin opsiyonel olduğu belirsizleşebilir). Benzer şekilde, Prisma ile modellenmiş ama arayüzde henüz karşılığı tam olmayan bazı alanlar (örn. müşteri `lifetimeValue` veya `rfmData` gibi) mevcut; bunlar geleceğe dönük düşünülmüş fakat şu an devreye alınmamış olabilir. Bu gibi durumlar, kullanıcılara sunulmadığı sürece kodda tutulmasa da olur veya daha iyi dokümante edilmelidir.

- **Performans ve Optimizasyon Meseleleri:** Next.js ve Express seçimleri genel olarak yüksek performans sunsa da, uygulamanın bazı bölümleri yük altında performans sorunları yaşayabilir. Örneğin, ürün listesinde yüzlerce kayıt varsa ve her kayıt için birden fazla ilişki çekiliyorsa (kategori, stok, vs.), sunucu tarafında sorgu optimizasyonu önem kazanır. Prisma burada yardımcı olsa da, geliştirici sorguları yazarken dikkatli olmalı. Filtreleme ve arama özelliği için eğer sunucu tarafında özel bir indeksleme stratejisi yoksa, çok büyük tablolar üzerinde arama biraz yavaşlayabilir. Yine harita entegrasyonu gibi kütüphaneler, istemci tarafında ekstra yük getirebilir (Leaflet CSS/JS boyutu vs.). Bu gibi potansiyel performans noktalarında henüz önlemler alınıp alınmadığı belirsizdir. Projenin geliştirildiği teknoloji stack'i ölçeklenebilir, ancak gerçek hayatta büyük veri hacmiyle test edilip optimize etmek gerekebilir.

- **Test ve Kalite Güvencesi Eksikliği:** Kod incelendiğinde, bir test altyapısının planlandığı ancak muhtemelen henüz tam olarak yazılmadığı görülüyor. README'de test çalıştırma komutu olsa da repo içinde unit/integration test dosyalarına rastlanmıyor. Bu, ileride refaktör veya ek özellik geliştirme sırasında geriye dönük hatalar yapma riskini artırır. Otomatik testlerin olmaması, manuel test yükünü de artıracaktır. Aynı şekilde, Continuous Integration (CI) ve kod kalitesi araçlarına dair bir iz yoksa (ör. GitHub Actions, code coverage takibi), projenin büyümesiyle birlikte hataları erken yakalamak zorlaşabilir. Bu, teknik borç olarak değerlendirilebilir ve gelecekte ele alınması gerekebilir.

- **Dokümantasyon ve Kullanım Kılavuzu:** Projenin README dosyası kurulum ve özellik listesi açısından oldukça iyi bir başlangıç sunuyor. Ancak son kullanıcı perspektifinden tam bir kullanım kılavuzu veya ekran görüntüleriyle desteklenmiş bir doküman mevcut değil. Örneğin, bir işletme yöneticisi uygulamayı kurduktan sonra ilk ne yapacağını (ürün mü eklemeli, yoksa önce kategorileri mi tanımlamalı?) deneme yanılma ile bulmak durumunda kalabilir. Ekranların nasıl kullanılacağına dair bir Wiki veya kullanıcı rehberi olmaması, uygulamanın adaptasyonunu

yavaşlatabilir. Özellikle geniş kapsamlı sistemlerde, her modülün kullanımını anlatan kılavuzlar veya en azından arayüzde ipucu (tooltip) mekanizmaları faydalı olur. Bu noktada dokümantasyon bir eksiklik olarak not edilebilir.

- **Eksik Özellikler veya Entegrasyonlar:** Bazı özelliklerin prototip seviyede kaldığı göze çarpıyor. Örneğin harita takibi simülasyon ile çalışıyor; gerçek bir kargo API'sine bağlanıp canlı veri alamıyor. Benzer şekilde, ödeme modülü tanımlı olsa da, gerçek bir ödeme entegrasyonu (ör. Stripe, İyzico vs.) yok (Payment modeli manuel kayıtlar için kullanılabilir durumda, ancak gerçek zamanlı ödeme akışı yok). Bu gibi eksiklikler, proje kullanımda iken elle veri girişi gerektirebilir (ör. ödemenin tamamlandığını elle işaretlemek gibi). Mağaza oluşturucu kısmının tamamlanıp tamamlanmadığı da belirsiz; eğer tamamlanmadıysa, arayüzde belki gizlenmiştir ama kodda yer alıyor. Bu da proje bütünlüğü açısından bir zayıflık sayılabilir, çünkü bitmemiş özelliklerle dolu bir ürün izlenimi verebilir. Güçlü yön çok, ancak her şeyi aynı anda yapmaya çalışmak yerine belli başlı işlevleri %100 mükemmel yapmak daha iyi olabilirdi. Bu denge bazı yerlerde gözetilmemiş olabilir.

- **Deployment (Canlıya Alma) Konuları:** Projenin kurulum adımları local geliştirme için çok net. Ancak bunu bir üretim ortamına koymak için ek adımlar gerekebilir. Örneğin ölçeklendirme için front-end ve back-end'in ayrı sunucularda barındırılması, bir ters proxy ayarlanması (Next.js front-end'i için), veritabanının bulut servisinde tutulması gibi konular dokümante edilmemiş. Docker desteği yok gibi görünüyor; bu, devreye almayı teknik bilgi gerektirir hale getiriyor. Bir Dockerfile veya docker-compose olsaydı, belki hem Postgres hem uygulama kolayca ayağa kalkabilirdi. Bu eksiklik, teknik yeterliliği sınırlı ekipler için zorluk çıkarabilir. Ayrıca yedekleme, loglama, monitör etme gibi işletim konularına dair bir bilgi yok, bu da ileride ele alınması gereken noktalar.

Özetle, projenin zayıf yönleri büyük ölçüde **tamamlanmamış özellikler ve optimizasyon eksikleri** etrafında toplanıyor. Ancak bunların çoğu giderilebilir veya projenin doğası gereği sonraki aşamalarda üstüne koyulabilecek şeyler. Mevcut haliyle proje, sağlam bir temel ve geniş bir özellik seti sunuyor; zayıf yönler ise bu temelin cilalanması gereken noktaları işaret ediyor. Geliştirici ekibin bu noktalara dikkat etmesi ve iyileştirmeler yapması durumunda proje çok daha profesyonel bir seviyeye ulaşacaktır.

8. Geliştirme veya kullanım için öneriler

Projenin mevcut durumunu göz önüne alarak, hem geliştirme ekibi için ileriye dönük hem de son kullanıcı açısından kullanımda dikkat edilebilecek bazı öneriler şu şekildedir:

- **Test Altyapısının Tamamlanması:** İleride yapılacak refaktörler veya yeni özellik eklemeleri için, proje mutlaka kapsamlı bir test suite'ine sahip olmalıdır. Bu nedenle, halihazırda planlanmış görünen (README'de `npm test` komutunun varlığı ile anlaşılıyor) test altyapısını tamamlamak önemlidir [26]. Her bir kritik fonksiyon için birim testleri (unit test) yazılması, özellikle controller seviyesinde farklı senaryoların (başarılı ekleme, validasyon hatası, yetkisiz erişim vb.) test edilmesi önerilir. Ayrıca, uçtan uca (end-to-end) testler ile temel kullanıcı senaryolarının (ör. ürün ekleyip sipariş oluşturma) otomatikleştirilmesi güvenilirliği artıracaktır. Bu sayede ileride yapılacak değişikliklerde mevcut fonksiyonların bozulmadığından emin olunabilir.

- **Gerçek Entegrasyonların Eklenmesi:** Uygulamanın bazı bölümleri şu an için simülasyon veya manuel işlem gerektiriyor. Örneğin, **harita/kargo takibi** özelliği gerçek bir kargo firması API'sine bağlanırsa çok daha fonksiyonel hale gelecektir. Google Maps Geocoding API veya OpenStreetMap Nominatim gibi bir servis kullanarak adres bilgisinin koordinata çevrilmesi

sağlanabilir; ayrıca büyük kargo şirketlerinin (PTT Kargo, DHL, UPS vs.) web servisleri entegre edilip gerçek takip verisi alınabilir. Bu sayede sistem, manuel konum üretmek yerine gerçek konum güncellemelerini gösterebilir. Proje kodunda bu ihtiyacın farkında olunduğu, yorum satırlarından anlaşılıyor [60]; ileri aşamada bu entegrasyonlar planlandığı gibi hayata geçirilmelidir. Benzer şekilde **ödeme sistemi** entegrasyonu (örn. Stripe veya iyzico ile), sipariş sürecinin kapanışını otomatikleştirir ve Payment tablosunu gerçek işlemlerle doldurur. Bu entegrasyonlar proje kapsamını büyütse de opsiyonel modüller olarak sunulabilir; örneğin bir yapılandırma ile harita takibi simülasyon modunda veya gerçek modda çalışacak şekilde ayarlanabilir.

- **Kullanıcı Arayüzü İyileştirmeleri ve Rehberlik:** Son kullanıcıların uygulamayı daha rahat öğrenip kullanabilmesi için arayüze bazı iyileştirmeler eklenebilir. Örneğin, ilk kullanımda yöneticiyi yönlendiren bir **onboarding sihirbazı** işe yarayabilir (önce kategori oluştur, sonra ürün ekle, sonra müşteri ekle gibi adımlar). Arayüzde kritik alanlara bilgi simgeleri eklenip üzerine gelindiğinde açıklama veren **tooltip**'ler konulabilir. Örneğin "Min Stok Seviyesi" alanının yanında küçük bir "i" simgesi ile bu alanın ne anlama geldiği anlatılabilir. Ayrıca, **çok dillilik** desteği halihazırda kütüphane olarak eklenmiş görünüyor (Next.js Intl) ancak uygulamada sadece Türkçe kullanılıyor gibi [63]. İleride uygulamayı İngilizce başta olmak üzere diğer dillere çevirmek için altyapı hazır; tüm metinlerin uluslararasılaştırma (i18n) sistemine alınması ve çeviri dosyalarının oluşturulması önerilir. Bu sayede uygulama farklı dilde konuşan kullanıcılara da sunulabilir, pazar genişler.

- **Performans Optimizasyonları:** Uygulama büyüdükçe performans konusu gündeme gelecektir. Bazı öneriler: Veritabanı sorgularında ihtiyaç duyulan alanların seçilmesi (Prisma ile select/include kullanarak), gereksiz büyük veri yüklerinin önlenmesi. Örneğin ürün listesini çekerken her ürünün tüm siparişlerini de getirmek gibi ağır işlemler yapmamaya dikkat edilmeli. Gerekli durumlarda veritabanına uygun **indeksler** eklenmeli (Prisma modeli içinde @@index ile tanımlanabilir). Özellikle arama yaparken kullanılan sütunlar için (ör. ürün adı, müşteri adı) indeksler performansı çok artırır. Frontend tarafında, Next.js'in sunduğu **otomatik kod bölme (code splitting)** ve önbellekleme özelliklerinden yararlanılabilir. Örneğin sayfalar arası geçişlerde verilerin önceden fetch edilmesi (Next'in getServerSideProps veya yeni App Router ile server component kullanımı) ile kullanıcıya daha akıcı bir deneyim sunulabilir. Tailwind CSS kullanımı sayesinde CSS boyutu optimize olacaktır (JIT derleme ile kullanılmayan CSS atılıyor), bu zaten iyi bir performans artısıdır. Harita gibi ağır komponentler sadece gerekli olduğunda yüklenmeli (dinamik import ile), bu da sayfa yükleme sürelerini azaltır.

- **Kod Büyüklüğünün Yönetimi ve Modülerlik:** Proje halihazırda oldukça modüler olsa da, daha da okunabilir kılmak için bazı iyileştirmeler yapılabilir. Örneğin backend'deki bazı devasa controller dosyaları daha küçük parçalara ayrılabilir veya **service layer** eklenebilir. Şu anda controller'lar doğrudan Prisma çağrıları yapıyor. Alternatif olarak, veritabanı işlemleri için ayrı **repository/service** sınıfları yazılıp controller'lar daha yalın tutulabilir (bu bir tercih meselesi, doğrudan Prisma kullanmak da kabul edilebilir). Mağaza oluşturucu veya dosya içe aktarma gibi opsiyonel özellikler ayrı modüller olarak paketlenebilir; böylece temel envanter yönetimi kodundan ayrılmış olurlar. Örneğin proje ileride monorepo'yu ayırmak isterse (birden fazla bağımsız deploy edilebilir servis gibi), mağaza oluşturucu belki ayrı bir servis olabilir. Bu tür bir modülerlik ileri seviye bir optimizasyon, an itibariyle kritik değil ancak planlama olarak akılda tutulabilir.

- **DevOps ve Dağıtım Önerileri:** Uygulamanın gerçek kullanımda sorunsuz çalışması için dağıtım süreçlerine yatırım yapılmalı. Mevcut durumda manuel kurulum adımları yeterli gelse de, Docker ortamı hazırlamak işleri kolaylaştıracaktır. Docker için hem backend hem frontend hem de

PostgreSQL içeren bir `docker-compose.yml` yazılması tavsiye edilir. Böylece geliştiriciler veya kullanıcılar tek komutla sistemi ayağa kaldırabilir. Sürekli entegrasyon (CI) süreçleri (ör. GitHub Actions ile) kurulup her commit'te testlerin çalıştırılması, otomatik build'lerin yapılması sağlanabilir. Versiyonlama ve yayınlamak konusunda da semantic versioning uygulanabilir; örneğin proje belirli aralıklarla 1.0, 1.1 gibi sürümler halinde paketlenebilir. Üretim ortamında Next.js ve Node sunucusunun arkasına bir **reverse proxy** (NGINX gibi) konularak, SSL terminasyonu ve yük dengeleme yapılması da önerilir. Ayrıca loglama için bir merkezi log sistemi (winston + Elasticsearch/Kibana gibi) veya hata takip için Sentry gibi araçlar entegre edilebilir. Bu DevOps iyileştirmeleri, uygulamanın profesyonel bir ortama taşınmasını kolaylaştıracaktır.

- **Kullanıcı Geri Bildirimleri ve Sürekli İyileştirme:** Son olarak, uygulamanın gerçek kullanıcılar tarafından kullanımından gelecek geri bildirimler çok değerli olacaktır. Örneğin kullanıcılar belirli bir özelliğin arayüzünün karmaşık olduğunu veya bir rapor türünün eksik olduğunu belirtebilir. Bu geri bildirimler düzenli olarak toplanıp yol haritasına eklenmeli. Yazılım, yaşayan bir organizma gibidir; kullanım oldukça ihtiyaçlar netleşir. Bu projede de belki bazı modüller çok sık kullanılmayacak (örneğin iletişim kayıtları az kullanılabilir) ama başka bir özellik talebi doğacaktır (ör. "müşteri borç takip özelliği isteriz" gibi). Bu nedenle, esnek mimarinin avantajı kullanılıp yeni istekler makul şekilde eklenebilir. Ekip içinde kod incelemeleri (code review) ve standartlara uyum devam ettirilmeli, böylece proje büyüse de kod kalitesi düşmemelidir.

Sonuç olarak, **Envanter & E-Ticaret Yönetim Sistemi** projesi halihazırda güçlü bir temel ve fonksiyonellik seti sunmaktadır. Yukarıdaki öneriler, bu temelin üzerine eklemeler yaparak projeyi daha da olgunlaştırmayı hedeflemektedir. Gerek teknik iyileştirmeler (testler, performans, entegrasyonlar) gerekse kullanıcı deneyimi geliştirmeleri ile proje, üretim ortamında başarılı bir şekilde hizmet verebilir. Ekip, geliştirme sırasında modern pratikleri ve kullanıcı odaklı yaklaşımı sürdürdükçe, ortaya çıkan ürünün değeri ve kullanılabilirliği artacaktır. Bu rapor kapsamında belirtilen geliştirme adımları, projenin hem bakımını kolaylaştıracak hem de kullanıcı memnuniyetini üst düzeye çıkaracaktır. Böylece proje, ismiyle müsemma şekilde işletmeler için vazgeçilmez bir **envanter ve e-ticaret yönetim** aracı haline gelebilir.

---

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 24 26 52 **github.com**
https://github.com/haitaa/inventory-tracker/blob/8af06858ed970f6fda013839d7c9f6beff6aaf92/README.md

19 20 21 22 23 **github.com**
https://github.com/haitaa/inventory-tracker/blob/8af06858ed970f6fda013839d7c9f6beff6aaf92/backend/package.json

25 63 **github.com**
https://github.com/haitaa/inventory-tracker/blob/8af06858ed970f6fda013839d7c9f6beff6aaf92/frontend/package.json

github.com
https://github.com/haitaa/inventory-tracker/blob/8af06858ed970f6fda013839d7c9f6beff6aaf92/backend/prisma/schema.prisma

github.com
https://github.com/haitaa/inventory-tracker/blob/8af06858ed970f6fda013839d7c9f6beff6aaf92/backend/routes/product-service/productRoute.js

github.com
https://github.com/haitaa/inventory-tracker/blob/8af06858ed970f6fda013839d7c9f6beff6aaf92/backend/controllers/product-service/productController.js

github.com
https://github.com/haitaa/inventory-tracker/blob/8af06858ed970f6fda013839d7c9f6beff6aaf92/backend/controllers/file-export-service/fileExportController.js

github.com
https://github.com/haitaa/inventory-tracker/blob/8af06858ed970f6fda013839d7c9f6beff6aaf92/frontend/components/orders/DeliveryMap.tsx