Mikroservis Tabanlı Modern E-Ticaret Platformu

İleri Yazılım Mimarisi Dönem Projesi

Harun Oktay — *Proje Hazırlayan* — ***20011080***

Dr. Göksel BİRİCİK — *Proje Danışmanı*

Repo Linki: https://github.com/HarunOktay/Microservice-Based-E-Commerce

Video Linki: https://youtu.be/1fyKC_hmX54

January 23, 2025

Abstract

Bu projenin amacı yazılım mimarisi hakkında çözümü anlama, tasarlama ve gerçekleştirme pratiklerini kazanmaktır. Bu amacı gerçekleştirmek için basit bir mikroservis tabanlı e-ticaret platformu oluşturulmuştur. Sistem, kullanıcı yönetimi, ürün kataloğu ve sipariş işleme gibi temel e-ticaret fonksiyonlarını bağımsız mikroservisler olarak docker ortamında çalıştırmaktadır. Go programlama dili kullanılarak geliştirilen backend servisler ve Python'un Streamlit kütüphanesi ile oluşturulan modern kullanıcı arayüzü birbirlerinden bağımsız servisler olarak tasarlanmıştır. Tüm servisler Docker kullanılarak entegre edilmiştir. Bu servisler kendi aralarında HTTP requestleri üzerinden iletişim kurmaktadır. Ayrıca nginx reverse proxy kullanılarak servisler arası iletişim optimize edilmiş ve yük dengelemesi sağlanmıştır.

İstanbul - 2024

Contents

List of Figures

Teşekkür

Bu çalışmada tüm Yıldız Teknik Üniversitesi Bilgisayar Mühendisliği öğretim görevlilerine ve özel olarak sevgili hocam Dr. Göksel BİRİCİK'e teşekkürlerimi sunarım. Kendisinin verdiği BBG dersi sayesinde bilgisayar mühendisliğinin temellerini oluşturdum. Bu sağlam temelin ardından sistem analizi ve tasarımı dersinde pratik tasarım ilkeleri ve gerçek dünya deneyimleriyle bilgilerimi pekiştirme fırsatı buldum. Son olarak yüksek lisans derslerinden İleri Yazılım Mimarisi dersinde yazılımın ve süreçlerin temellerini daha detaylı öğrenmemi ve bu proje kapsamında kendimi geliştirmemi sağladığı için de kendisine ayrıca teşekkür ederim.

1 Literatür Taraması

Mikroservis mimarisi, modern yazılım geliştirme yaklaşımlarında önemli bir paradigma değişimini temsil etmektedir. Son on yılda, geleneksel monolitik mimarilerden mikroservis tabanlı mimarilere geçiş, yazılım endüstrisinde dikkat çekici bir trend haline gelmiştir Dragoni et al. (2017). Bu çalışmada, mikroservis mimarisinin gelişimi, temel prensipleri ve güncel uygulamaları sistematik bir şekilde incelenmiştir. Mikroservis mimarisinin temelleri, 2011 yılında yazılım mimarları tarafından ortaya atılmış ve zamanla evrimleşerek günümüzdeki formunu almıştır. Dragoni ve arkadaşlarının Dragoni et al. (2017) yaptığı çalışmada, mikroservis mimarisinin monolitik yapılardan farklılaşan yönleri detaylı bir şekilde ele alınmıştır. Araştırmacılar, mikroservislerin bağımsız dağıtılabilirlik ve ölçeklenebilirlik özelliklerinin, modern yazılım sistemlerinin karmaşık gereksinimlerini karşılamada önemli avantajlar sağladığını vurgulamışlardır.

Zhang ve ekibinin Zhang et al. (2019) gerçekleştirdiği kapsamlı performans analizi çalışmasında, mikroservis tabanlı sistemlerin ölçeklenebilirlik ve hata toleransı açısından üstün özellikleri ortaya konmuştur. Araştırmacılar, özellikle yüksek trafikli sistemlerde mikroservis mimarisinin sağladığı izolasyon ve bağımsız ölçeklendirme yeteneklerinin kritik öneme sahip olduğunu göstermişlerdir.

Mikroservis güvenliği konusunda Yu ve arkadaşlarının Yu et al. (2020) yaptığı sistematik inceleme, bu mimarinin güvenlik açısından hem avantajlarını hem de zorluklarını ortaya koymuştur. Servisler arası iletişimin güvenliği, kimlik doğrulama ve yetkilendirme mekanizmalarının tasarımı gibi konular detaylı olarak ele alınmıştır. Montesi ve Weber'in Montesi et al. (2016) çalışması ise, servisler arası güvenli iletişim protokollerinin tasarımı ve implementasyonu konusunda önemli katkılar sağlamıştır.

Container teknolojilerinin mikroservis mimarisiyle entegrasyonu konusunda Burns ve ekibinin Burns et al. (2016) yaptığı araştırma, özellikle Kubernetes gibi container orchestration sistemlerinin mikroservis deploymentlarındaki rolünü ayrıntılı bir şekilde incelemiştir. Araştırmacılar, container teknolojilerinin mikroservis mimarisinin temel prensiplerini desteklemedeki önemini vurgulamışlardır.

Li ve arkadaşlarının Li et al. (2021) service mesh teknolojileri üzerine yaptığı çalışma, mikroservis iletişiminde yeni bir paradigma değişimini işaret etmektedir. Service mesh'in, servisler arası iletişimin güvenilirliği, izlenebilirliği ve yönetilebilirliği üzerindeki olumlu etkileri detaylı bir şekilde analiz edilmiştir.

Veri tutarlılığı konusunda Garcia-Molina ve Salem'in Garcia-Molina et al. (2018) araştırması, dağıtık sistemlerde veri yönetiminin karmaşıklığını ve çözüm yaklaşımlarını ele almıştır. Özellikle mikroservis mimarisinde her servisin kendi veritabanına sahip olması prensibinin getirdiği zorluklar ve çözüm stratejileri detaylı bir şekilde incelenmiştir.

2 Yazılım Mimarisi

Geliştirdiğimiz e-ticaret sistemi, mikroservis mimarisi üzerine kurulu olup birbirleriyle HTTP protokolü üzerinden haberleşen dört ana servisten oluşmaktadır. Frontend servisimiz Streamlit framework'ü kullanılarak geliştirilmiş olup, kullanıcı arayüzünü sağlamaktadır. Frontend servisimizin kodlarına baktığımızda (app.py), diğer servislerle doğrudan HTTP istekleri üzerinden haberleştiğini görmekteyiz. Örneğin, yeni bir kullanıcı oluşturulduğunda frontend "http://user-service:8001/api/users" endpoint'ine POST isteği göndermektedir.

Sistemimizin omurgasını oluşturan üç mikroservis (user-service, product-service ve order-service) Go programlama dili ve Gin web framework'ü kullanılarak geliştirilmiştir. Her servis kendi port'unda çalışmakta ve kendi verisini yönetmektedir. Örneğin user-service 8001 portunda, product-service 8002 portunda ve order-service 8003 portunda hizmet vermektedir. Bu servislerin her biri basit bir in-memory veritabanı olarak map yapısını kullanmaktadır.

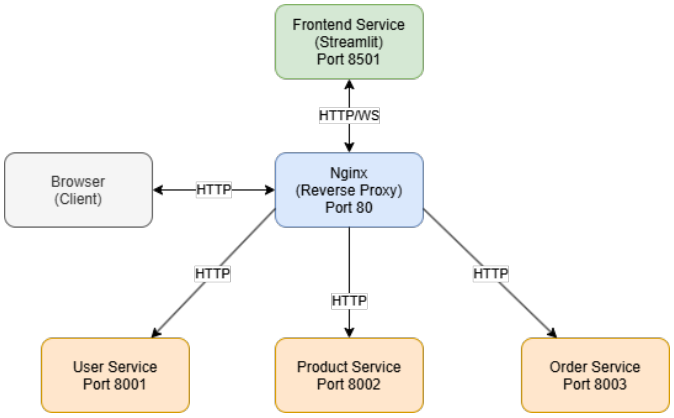

Figure 1: Yazılım Mimarisi

Nginx, sistemimizde reverse proxy görevi görmektedir. nginx.conf dosyasına baktığımızda, gelen isteklerin URL'lerine göre ilgili servislere yönlendirildiğini görebiliriz. Örneğin "/api/users" ile başlayan tüm istekler user-service'e, "/api/products" ile başlayan istekler product-service'e yönlendirilmektedir. Ancak burada ilginç bir durum söz konusudur: Frontend servisimiz Nginx'i bypass ederek doğrudan mikroservislere istek yapmaktadır. Bu durum aslında bir tasarım eksikliği olarak değerlendirilebilir.

Docker Compose yapılandırmamız (docker-compose.yml) tüm servisleri container'lar içinde çalıştırmakta ve birbirleriyle iletişim kurabilmelerini sağlamaktadır. Nginx container'ı 80 portunu dışarıya açarak, dış dünyadan gelen istekleri karşılamaktadır. Diğer servisler de kendi portlarında çalışmakta, ancak bu portlar sadece Docker network'ü içinde erişilebilir durumdadır.

Sistemin en önemli özelliklerinden biri, her servisin bağımsız olarak çalışabilmesi ve ölçeklendirilebilmesidir. Örneğin, product-service'in kodlarına baktığımızda (product-service/main.go), servisin kendi başına çalışabilen, basit bir REST API sunduğunu görebiliriz. Benzer şekilde order-service (order-service/main.go) de siparişleri bağımsız olarak yönetmektedir.

Frontend tarafında Streamlit'in sağladığı dinamik UI özellikleri kullanılarak, kullanıcı dostu bir arayüz oluşturulmuştur. Frontend, servislerin durumunu sürekli kontrol ederek, hangi servislerin aktif olduğunu göstermekte ve hata durumlarını kullanıcıya bildirmektedir. Bu yapı, sistemin sağlığının izlenmesi açısından önemli bir özellik sunmaktadır.

3 Nginx ile Reverse Proxy Yönetimi

Modern mikroservis mimarilerinde trafik yönetiminin merkezinde yer alan Nginx, bu e-ticaret projesinde API yönlendirmelerini koordine eden kritik bir bileşen olarak konumlandırılmıştır. `nginx/nginx.conf` dosyası üzerinde tanımlanan kurallar, Docker ortamında çalışan dört temel servisin (`frontend`, `user-service`, `product-service`, `order-service`) uyum içinde çalışmasını sağlar. Sistemin temel prensibi, kullanıcı tarafından görülen tek giriş noktası olarak Nginx'i kullanırken arka uç servislerini tamamen izole bir ağ içinde tutmaktır.

Projenin yapılandırma dosyası incelendiğinde, her bir API endpoint'inin nasıl özelleştirilmiş yönlendirme kurallarına sahip olduğu görülür:

```
location /api/users {
    proxy_pass http://user-service:8001;}
location /api/products {
    proxy_pass http://product-service:8002;}
location /api/orders {
    proxy_pass http://order-service:8003;}
location / {
    proxy_pass http://frontend:8501;}
```

Docker Compose dosyasında tanımlanan ağ yapısı, Nginx'in servis keşfi için hayati önem taşır. `expose` komutuyla belirtilen portlar sadece iç ağda erişilebilirken, `ports` ile Nginx'in 80 numaralı portu dış dünyaya açılır. Bu sayede kullanıcılar mikroservislerin doğrudan port numaralarına erişemez ve tüm trafik Nginx'in güvenlik filtrelerinden geçmek zorunda kalır.

Frontend servisimiz (Streamlit uygulaması) artık backend servislerine doğrudan erişmek yerine, tüm isteklerini Nginx üzerinden yapmaktadır:

```
# Kullan c  olu turma
response = requests.post("http://nginx/api/users",
                         json={"name": name, "email": email})
#   rn   listesi alma
products = requests.get("http://nginx/api/products").json()
# Sipari  olu turma
response = requests.post("http://nginx/api/orders",
                         json={"username": username, ...})
```

Performans optimizasyonu bağlamında, Nginx'in buffer yönetimi ve zaman aşımı parametreleri projenin özel ihtiyaçlarına göre ayarlanmıştır. Sistem durumu kontrolleri için timeout değerleri 2 saniye olarak belirlenmiştir:

```
response = requests.get("http://nginx/api/users", timeout=2)
response = requests.get("http://nginx/api/products", timeout=2)
response = requests.get("http://nginx/api/orders", timeout=2)
```

Gerçek kullanım senaryolarında karşılaşılan tipik sorunların debug edilmesi için Nginx log yapılandırması özelleştirilmiştir. Dağıtık yapının avantajları, sistem yükü arttığında `user-service` gibi bileşenlerin kolayca ölçeklendirilebilmesiyle kendini gösterir. Yeni container'lar eklendiğinde Nginx'in otomatik servis keşfi sayesinde yapılandırma dosyalarında değişiklik yapmaya gerek kalmaz.

4 Docker Compose

Docker Compose, birden fazla Docker konteynerini tanımlamak ve yönetmek için kullanılan bir araçtır. `docker-compose.yml` dosyası, uygulamanın mikroservis mimarisini oluşturan bileşenleri ve bu bileşenlerin nasıl yapılandırılacağını belirler. Bu dosya, tüm servislerin birbirleriyle etkileşimini ve bağımlılıklarını tanımlayarak, uygulamanın tutarlı bir şekilde çalışmasını sağlar.

4.1 Servisler ve İşlevleri

`docker-compose.yml` dosyasında, uygulamanın her bir mikroservisi ayrı bir servis olarak tanımlanmıştır. Bu servisler, uygulamanın farklı işlevlerini yerine getirir ve birbirleriyle etkileşimde bulunur. Nginx servisi, uygulamanın ön yüzü olarak görev yapar ve gelen HTTP isteklerini yönlendirir. Diğer mikroservisler (frontend, user-service, product-service, order-service) arasında köprü görevi görür. Ayrıca, WebSocket desteği için gerekli ayarları içerir. Frontend servisi, kullanıcı arayüzünü sağlayan Streamlit uygulamasını çalıştırır ve 8501 portunu kullanarak Nginx üzerinden erişilir. User-service, kullanıcı yönetimi ile ilgili işlevleri yerine getirir ve 8001 portunu kullanır. Product-service, ürün yönetimi ile ilgili işlemleri yönetir ve 8002 portunu kullanır. Order-service ise sipariş yönetimi ile ilgili işlevleri yerine getirir ve 8003 portunu kullanır.

4.2 Bağımlılıklar ve Başlatma Sırası

Her bir servis, diğer servislerin çalışmasına bağlı olarak yapılandırılmıştır. Örneğin, Nginx servisi, frontend ve diğer mikroservislere bağımlıdır. Bu bağımlılıklar, `depends_on` direktifi ile tanımlanmıştır ve uygulamanın düzgün bir şekilde çalışmasını sağlamak için kritik öneme sahiptir. Docker Compose, bu bağımlılıkları dikkate alarak servisleri doğru sırayla başlatır ve her bir servisin diğerlerine erişebilmesini garanti altına alır.

4.3 Port Ayarları ve Ağ Yapılandırması

Her bir servis, belirli bir port üzerinden erişilebilir şekilde yapılandırılmıştır. Bu port ayarları, Nginx'in diğer mikroservislere yönlendirme yapabilmesi için gereklidir. Örneğin, frontend servisi 8501 portunu kullanırken, user-service 8001 portunu kullanır. Tüm bu portlar, Docker ağı içinde erişilebilir durumdadır ve dış dünyaya sadece Nginx'in 80 numaralı portu açıktır. Bu yapı, güvenliği artırırken aynı zamanda servislerin birbirleriyle sorunsuz şekilde iletişim kurmasını sağlar.

Sonuç olarak `docker-compose.yml` dosyası, mikroservis mimarisini oluşturan bileşenlerin yapılandırmasını ve etkileşimini tanımlar. Bu dosya sayesinde, uygulamanın tüm servisleri kolayca başlatılabilir ve yönetilebilir. Docker Compose, geliştirme ve dağıtım süreçlerini basitleştirerek, uygulamanın daha verimli bir şekilde çalışmasını sağlar. Ayrıca, servislerin bağımsız olarak ölçeklendirilmesine ve güncellenmesine olanak tanır. Bu yapı, modern mikroservis mimarilerinin temel gereksinimlerini karşılayarak, uygulamanın esnek ve sürdürülebilir olmasını sağlar.

5 Front-End Service

Front-end servisi, kullanıcıların e-ticaret platformu ile etkileşimde bulunduğu arayüzü sağlayan temel bileşendir. Bu servis, Streamlit kütüphanesi kullanılarak geliştirilmiştir ve kullanıcıların ürünleri görüntülemesine, yeni kullanıcılar eklemesine ve sipariş oluşturmasına olanak tanır. Tüm bu işlevler, kullanıcı dostu bir arayüz üzerinden gerçekleştirilirken, arka planda Nginx üzerinden diğer mikroservislerle iletişim kurulur.

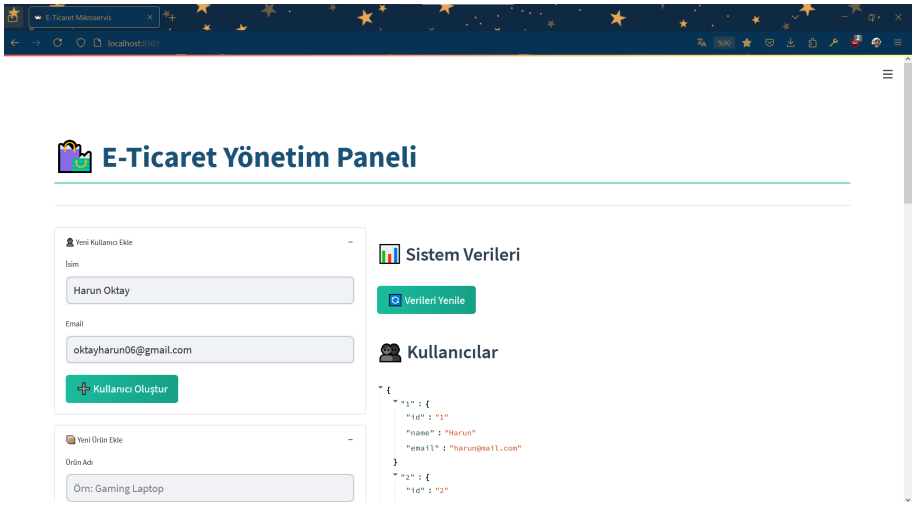

Figure 2: Ana Sayfa

5.1 app.py: Uygulamanın Ana Kod Dosyası

`app.py` dosyası, Streamlit uygulamasının ana kodunu içerir ve kullanıcı arayüzünü oluşturur. Bu dosya, kullanıcıdan alınan verileri işlemek için gerekli HTTP isteklerini yönetir. Örneğin, yeni bir kullanıcı eklemek için `requests.post()` metodu kullanılarak Nginx üzerinden `user-service` servisine istek gönderilir. Benzer şekilde, ürün listesi almak veya sipariş oluşturmak için ilgili mikroservislere HTTP istekleri yapılır. Bu sayede, kullanıcı etkileşimleri anında sunucuya iletilir ve yanıtlar kullanıcı arayüzünde dinamik olarak güncellenir.

5.2 Dockerfile: Konteyner Yapılandırması

Front-end servisini bir Docker konteynerinde çalıştırmak için gerekli talimatlar `Dockerfile` içinde tanımlanmıştır. Bu dosya, Python 3.9-slim tabanlı bir imajdan başlar ve uygulamanın çalışma dizinini ayarlar. Gerekli Python paketleri, `requirements.txt` dosyasından otomatik olarak yüklenir. Streamlit uygulaması, 8501 numaralı portta çalışacak şekilde yapılandırılmıştır. Docker konteyneri başlatıldığında, Streamlit uygulaması otomatik olarak çalıştırılır ve kullanıcıların erişimine sunulur. Bu yapı, geliştirme ve dağıtım süreçlerini standartlaştırarak tutarlı bir çalışma ortamı sağlar.

5.3 requirements.txt: Bağımlılıkların Yönetimi

Uygulamanın çalışması için gerekli olan Python kütüphaneleri `requirements.txt` dosyasında listelenmiştir. Bu dosyada, Streamlit kütüphanesi kullanıcı arayüzünü oluşturmak için, `requests` kütüphanesi HTTP isteklerini yönetmek için, `altair` kütüphanesi veri görselleştirme için ve `protobuf` kütüphanesi Protobuf desteği için kullanılmaktadır. Bu kütüphaneler, uygulamanın işlevselliğini artırırken kullanıcı deneyimini de geliştirir. Bağımlılıkların bu şekilde merkezi olarak yönetilmesi, geliştirme sürecini kolaylaştırır ve uygulamanın farklı ortamlarda tutarlı şekilde çalışmasını sağlar.

5.4 Kullanıcı Etkileşimi ve İşlevsellik

Front-end servisi, kullanıcıların e-ticaret platformu ile etkileşimde bulunmasını sağlayarak, ürünleri yönetmelerine ve sipariş oluşturmalarına olanak tanır. Kullanıcılar, basit bir arayüz üzerinden yeni kullanıcılar ekleyebilir, ürünleri görüntüleyebilir ve sipariş oluşturabilir. Tüm bu işlemler, arka planda Nginx üzerinden ilgili mikroservislere iletilir ve yanıtlar anında kullanıcı arayüzüne yansıtılır. Bu yapı, kullanıcıların ihtiyaçlarını karşılamayı hedeflerken, aynı zamanda geliştiriciler için esnek ve ölçeklenebilir bir çözüm sunar.

6 Backend Services

Backend servisleri, e-ticaret platformunun temel işlevlerini yerine getiren mikroservislerdir. Bu servisler, kullanıcı yönetimi, ürün yönetimi ve sipariş yönetimi gibi işlevleri sağlar. Aşağıda bu servislerin ana bileşenleri ve işlevleri açıklanmaktadır.

6.1 User Service

`main.go`: User Service, kullanıcıların oluşturulması ve listelenmesi işlevlerini yerine getirir. User yapısı, kullanıcı modelini temsil eder ve global bir kullanıcı haritası (`users`) oluşturulur. Uygulama başlatıldığında, test verisi olarak iki kullanıcı (Harun ve Ahmet) eklenir. `/api/users` altında iki endpoint sunar: `POST /api/users` endpoint'i, yeni bir kullanıcı oluşturmak için kullanılır. İstemciden gelen JSON verisi işlenir ve kullanıcı haritasına eklenir. `GET /api/users` endpoint'i ise mevcut kullanıcıların listesini döndürür.

`Dockerfile`: User Service'in bir Docker konteynerinde çalıştırılması için gerekli talimatları içerir. Golang 1.21-alpine tabanlı bir imajdan başlar, gerekli bağımlılıkları yükler ve uygulamayı derler. 8001 portu üzerinden erişilebilir hale getirilir.

6.2 Product Service

`main.go`: Product Service, ürünlerin yönetimi ile ilgili işlevleri yerine getirir. Product yapısı, ürün modelini temsil eder ve global bir ürün haritası (`products`) oluşturulur. Uygulama başlatıldığında, test verisi olarak iki ürün (Laptop ve Phone) eklenir. `/api/products` altında iki endpoint sunar: `POST /api/products` endpoint'i, yeni bir ürün oluşturmak için kullanılır. İstemciden gelen JSON verisi işlenir ve ürün haritasına eklenir. `GET /api/products` endpoint'i ise mevcut ürünlerin listesini döndürür.

`Dockerfile`: Product Service'in bir Docker konteynerinde çalıştırılması için gerekli talimatları içerir. Golang 1.21-alpine tabanlı bir imajdan başlar, gerekli bağımlılıkları yükler ve uygulamayı derler. 8002 portu üzerinden erişilebilir hale getirilir.

6.3 Order Service

`main.go`: Order Service, siparişlerin yönetimi ile ilgili işlevleri yerine getirir. Order yapısı, sipariş modelini temsil eder ve global bir sipariş haritası (`orders`) oluşturulur. Uygulama başlatıldığında, test verisi olarak bir sipariş (Harun'un Laptop siparişi) eklenir. `/api/orders` altında iki endpoint sunar: `POST /api/orders` endpoint'i, yeni bir sipariş oluşturmak için kullanılır. İstemciden gelen JSON verisi işlenir ve sipariş haritasına eklenir. `GET /api/orders` endpoint'i ise mevcut siparişlerin listesini döndürür.

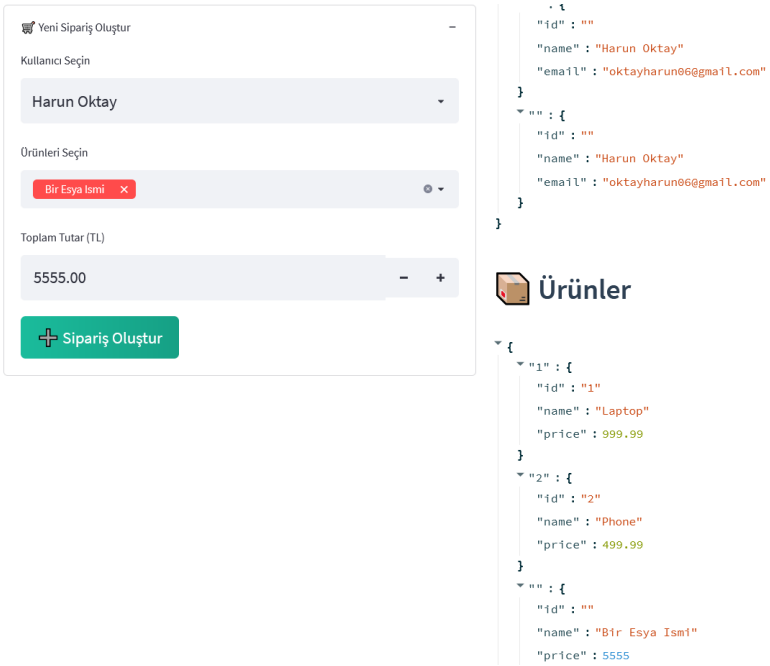

Figure 3: Siparis Kismi

`Dockerfile`: Order Service'in bir Docker konteynerinde çalıştırılması için gerekli talimatları içerir. Golang 1.21-alpine tabanlı bir imajdan başlar, gerekli bağımlılıkları yükler ve uygulamayı derler. 8003 portu üzerinden erişilebilir hale getirilir.

7 Sonuç

Bu proje, mikroservis mimarisini pratik olarak uygulayabilmek ve modern yazılım geliştirme süreçlerini deneyimlemek adına oldukça faydalı bir çalışma oldu. Nginx reverse proxy yapılandırması, Docker Compose ile konteyner yönetimi ve Golang tabanlı mikroservislerin entegrasyonu gibi birçok konuda doğrudan pratik yapma fırsatı buldum. Bu süreç, teorik bilgilerimi uygulamaya dökebilmem için mükemmel bir ortam sağladı.

Proje, mikroservis mimarisinin temel prensiplerini anlamak ve uygulamak için harika bir başlangıç noktası oldu. Her ne kadar basit bir yapıda olsa da, kullanıcı yönetimi, ürün yönetimi ve sipariş yönetimi gibi temel işlevleri içeren bu mimari, ileride daha karmaşık sistemler tasarlamak için sağlam bir temel oluşturdu. Özellikle Docker ve Nginx gibi araçların kullanımı, geliştirme ve dağıtım süreçlerini ne kadar kolaylaştırdığını gösterdi.

Bu proje, kendi gelişimim için büyük bir adım oldu. Mikroservis mimarisini doğrudan uygulayarak, servislerin bağımsız çalışması, ölçeklenebilirlik ve güvenlik gibi kavramları deneyimleme fırsatı buldum. Ayrıca, farklı teknolojilerin (Golang, Python, Docker, Nginx) bir arada nasıl kullanılabileceğini öğrenmek, gelecekteki projeler için büyük bir motivasyon kaynağı oldu.

İleriki çalışmalarda, bu projeyi daha da geliştirerek veritabanı bağlantıları eklemeyi, daha karmaşık servisler tasarlamayı ve güvenlik önlemlerini artırmayı planlıyorum. Özellikle veritabanı entegrasyonu ve dağıtık sistemlerde veri tutarlılığı gibi konular üzerinde çalışarak, bu projeyi daha gerçekçi bir ürün haline getirmeyi hedefliyorum. Bu süreç, hem teknik becerilerimi geliştirmemi sağlayacak hem de modern yazılım mimarilerini daha derinlemesine anlamama yardımcı olacak.

Sonuç olarak, bu proje sadece teknik bir deneyim değil, aynı zamanda kendi gelişim yolculuğumda önemli bir kilometre taşı oldu. Mikroservis mimarisini uygulamak, farklı teknolojileri bir araya getirmek ve bir sistemin tüm bileşenlerini entegre etmek, gelecekteki projeler için büyük bir özgüven kazandırdı. Bu deneyim, yazılım geliştirme süreçlerine daha bütünsel bir bakış açısıyla yaklaşmamı sağladı.

References

Burns, Brendan et al. (2016). "Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade". In: *Queue* 14.1, pp. 70–93.

Dragoni, Nicola et al. (2017). "Microservices: Yesterday, Today, and Tomorrow". In: *Present and Ulterior Software Engineering*, pp. 195–216.

Garcia-Molina, Hector and Kenneth Salem (2018). "Data Management Challenges in Microservice Architectures". In: *IEEE Data Engineering Bulletin* 41.1, pp. 3–14.

Li, Wei et al. (2021). "Service Mesh: Challenges, State of the Art, and Future Research Opportunities". In: *IEEE Software* 38.1, pp. 149–157.

Montesi, Fabrizio and Janine Weber (2016). "Circuit Breakers, Discovery, and API Gateways in Microservices". In: *Proceedings of the 9th International Conference on Service-Oriented Computing*, pp. 203–217.

Yu, Dongjin et al. (2020). "A Survey on Security Issues in Services Communication of Microservices-Enabled Fog Applications". In: *Concurrency and Computation: Practice and Experience* 32.4, e5436.

Zhang, Peng, Yutao Zhou, and Minzhi Li (2019). "Performance Analysis and Optimization in Microservice-based Systems". In: *IEEE Transactions on Services Computing* 12.4, pp. 468–483.