T.C.

SİVAS CUMHURİYET ÜNİVERSİTESİ

MÜHENDİSLİK FAKÜLTESİ

BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ

DOCKER KONTEYNIRLAR VE MICROSERVIS UYGULAMASI: JAVA SPRING BOOT ILE

Barış Can AKDAĞ
BATUHAN ÇAM

LİSANS BİTİRME PROJESİ

07-2019
SİVAS

TEZ BİLDİRİMİ

Bu tezdeki bütün bilgilerin etik davranış ve akademik kurallar çerçevesinde elde edildiğini ve tez yazım kurallarına uygun olarak hazırlanan bu çalışmada bana ait olmayan her türlü ifade ve bilginin kaynağına eksiksiz atıf yapıldığını bildiririm.

İmza

Öğrencinin Adı SOYADI
Barış Can AKDAĞ

Tarih: 03.07.2019

İmza

Öğrencinin Adı SOYADI
Batuhan ÇAM

Tarih: 03.07.2019

ÖZET

DOCKER KONTEYNIRLAR VE MIKROSERVIS UYGULAMASI: JAVA SPRING BOOT ILE

Barış Can AKDAĞ
Batuhan ÇAM

Danışman: Dr. Öğr. Üyesi Halil ARSLAN

Mikroservis sadece bir işi yapan, bir fonksiyonaliteyi gerçekleştiren çok küçük kod parçacıklarıdır. Geliştirme süreçleri, bağımlılıkları, boyutları olabildiğince küçük olan atomik servislerdir.

Günümüzün popüler teknolojilerinden olan Java Spring Boot ile uygulama geliştirilmiştir. Mikroservislerin yönetimi, bakımı ve genişletilmesi için DevOps araçları kullanılmıştır.

Konu kapsamı gereğince Docker Konteynırlar, Mikroservis tasarımları ve DevOps araçları kullanılmıştır. Örnek bir uygulama yapılmıştır. Uygulama, Docker ve DevOps araçları detaylı bir şekilde açıklanmıştır.

.

Anahtar Kelimeler: Mikroservis, DevOps, Docker Konteynır

İÇİNDEKİLER

KISALTMALAR

Kısaltmalar

DevOps	:Developers and Operations
Docker	:Containerization Platform

1. GİRİŞ

Öncelikle konumuz Docker Konteynırlar üzerinde çalışan Java Spring Boot ile geliştirilmiş Mikroservis Uygulamasıdır. Proje kapsamında Spring Boot(Java) kullanarak Mikroservis Mimarisinde bir uygulamada geliştirilmiştir.

Konunun detaylarına ilerleyen bölümlerde değinilecektir. Bölümlere geçmeden önce Mikroservis Mimarisini tasarlarken ve Mikroservis Mimarisi ile ilişkili konuları ve kullanılan teknolojiler Bölüm 1.1'de gösterilmiştir. Kullanılan araçlar Bölüm 1.2'de gösterilmiştir.

Örnek bir uygulama geliştirilerek Docker Konteynır ve DevOps Araçları proje tamamlanmıştır.

1.1. Teknolojiler

- Spring Boot – Application Framework
- Zuul – API Gateway(Load Balancer)
- Consul - Service Registration And Discovery
- Docker – Containerization Platform
- RabbitMQ – Asenkron Mikroservice Communication
- Logstash – Log Collector
- Elasticsearch – Log Indexer
- Kibana – Data Vizualization
- Angular – Front-End Framework
- Bootstrap – UI Template

1.2. Araçlar

- Java- Programing Language
- Maven-Build Tool
- Git – Version Control System
- Docker – Deployment Tools

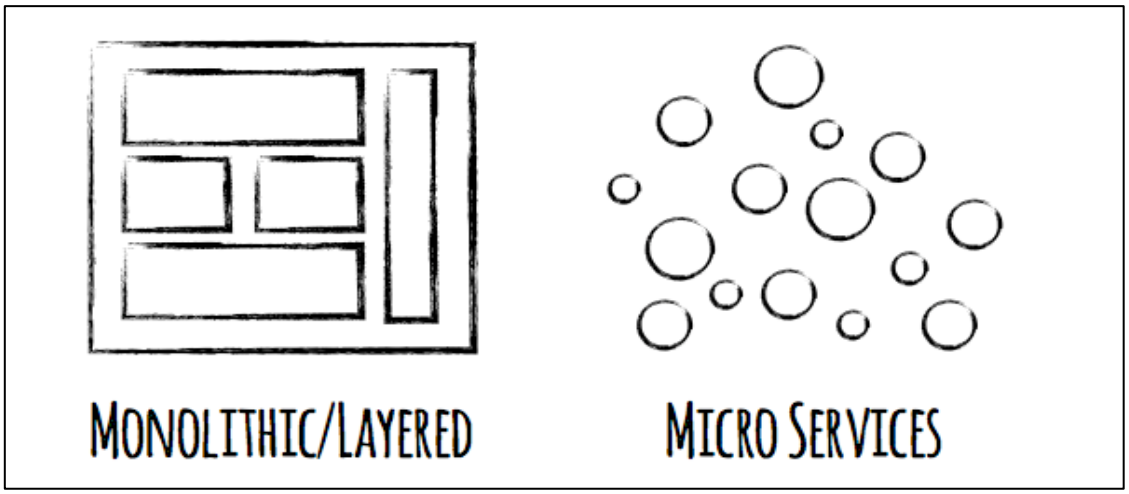
2. MİKROSERVİS MİMARİSİ

Mikroservis en kısa tabiriyle küçük, otonom ve bir arada çalışan servislerdir. Yazılım projesine yeni fonksiyonellikler eklendikçe kodlar büyümektedir. Bir zaman sonra, projeye hakim olmak, eklentiler yapmak ve karşımıza çıkan sıkıntıları çözmek zor bir hal almaya başlamaktadır. Normalde, monolitik bir proje içerisinde bu gibi problemlerle mücadele edebilmek için kodu olabildiğince soyut ve modüller oluşturulmalıdır.

Monolithic Architecture(Monotolik Mimari) ve Mikroservis Mimarisinin farklarının bilinmesi gerekmektedir.

2.1. Monolotik vs Mikroservis Mimarisi

Monolithic Architecture yazılımın Self-Contained (kendi kendine yeten) olarak tasarlanması anlamına gelmektedir. Bir standart doğrultusunda "tek bir parça" olarak oluşmasıdır. Bu mimarideki Component'ler Loosely Coupled (gevşek bağlanmış) olmasından ziyade, Interdependent (birbirine bağlı) olarak tasarlanmaktadır.



Şekil 2.1. Monolotik ve Mikroservis

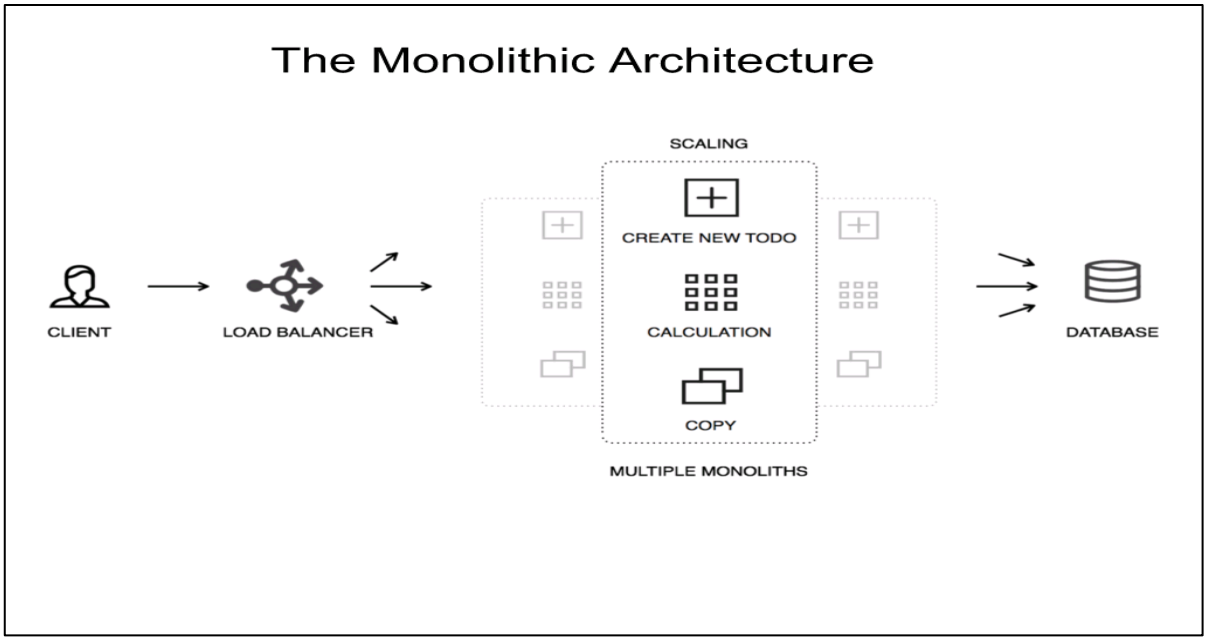
Günümüze baktığımızda kurumsal projeler, Servis Tabanlı Mimari (Service Oriented Architecture - SOA) ile geliştirilmeye başlanmıştır ve büyük ölçüde yerini zaten almış durumdadır. Geleneksel SOA mimarisinde geliştirilen tüm Component'lerin de, tek bir çatı altında olduğunu da görünmektedir. Yakın geçmişten bu yana SOA ile birlikte Manageability (Yönetilebilirlik), Maintenance (Bakım) ve Interoperability (Birlikte Çalışabilirlik) gibi kavramlar göz önüne alınmıştır.

Günümüzdeki şirketlerin IT yaklaşımlarına baktığımızda ise genelde IT For Business kapsamında olduğundan dolayı her zaman pazarlama odaklı gitmektedirler. Bu doğrultuda sürekli artan bir entegrasyon ihtiyaçları doğmaktadır. Bu bitmeyen ihtiyaçlar doğrultusunda ise Monolithic Architecture ile tasarlanmış olan SOA'lar gitgide istemsizce büyümektedir.

Bu noktaya kadar her şey "büyüme" haricinde normal görünebilir fakat problem nerede/ne zaman başlamaktadır? İşte bu soruya geçmeden önce Monolithic Architecture'ın dezavantajlarını ele alınmaktadır.

2.1.1. Monolithic Architecture'nin Getirdiği Bazı Dezavantajlar

Şekil 2.2.'deki resme bakıldığında bölünmez, Self-Contained olarak tasarlanmış Monolithic bir yapı görünmektedir. Scaling için bir Load Balancer arkasına koyulmuştur fakat bu durumda da Scale edilmek istenin Component'in aksine, tüm Monolithic yapının kopyasını farklı ortamlarda saklamak durumunda kalınmıştır.



Şekil 2.2. Monolotik Mimari

Diğer dezavantajlarını maddelemek gerekirse:

- Tüm Component'lerin aynı Framework, aynı programlama dili ile geliştirilmesinin gerekmesidir.

- Bir Component üzerinde olan değişiklik için, tüm Monolithic yapının tekrar Deploy edilmesi ve Restart edilmesi durumunda kalmasıdır.
- Versiyon yönetiminin gitgide zorlaşmasıdır.
- Birbirlerine olan bağımlılıklarından dolayı, bir Component için yapılan değişimden diğer Component'in etkilenebilmesidir.
- Continuous Delivery'nin uygulanmasının zorlaşmasıdır.

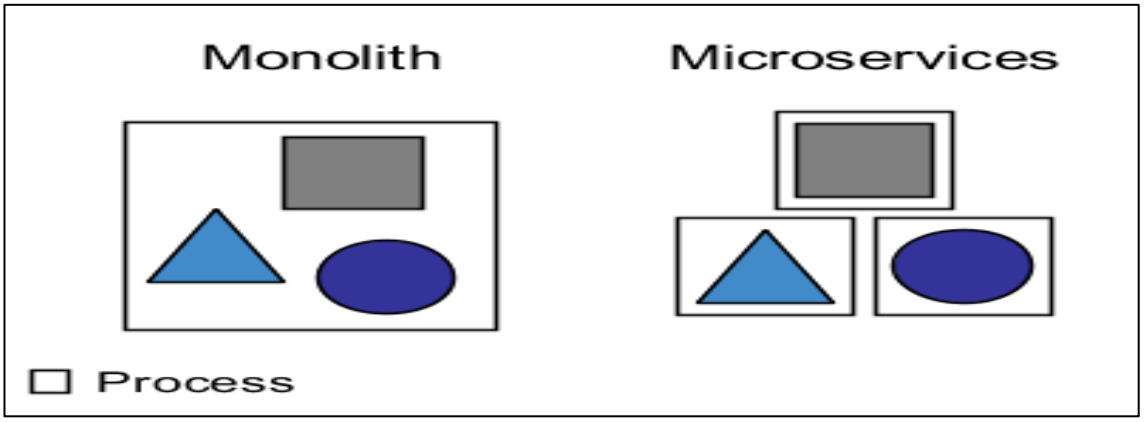
Bu dezavantajların bazıları Monolithic Mimarinin büyümesi ile gelmese de en Major problemlerden bir tanesi Monolithic Mimari üzerindeki Component'lerden herhangi birinde olan değişikliğin Deployment'ı yapıldığında, bu durumdan diğerlerinin de etkilenebiliyor/etkilenebilecek olmasıdır.

Örnek üzerinde düşünülmektedir. Belediye otobüslerine hangi durakta olduklarını ve her durak içinde ilgili otobüsün gelmesine tahmini kalan sürenin gösterimini yapan ekranların servisini geliştirilmektedir. Bizden otobüs içerisindeki ekranda gösterilmesi gereken bazı yeni özellikler istenmektedir. İlgili geliştirmeyi ilgili ekip yaptı ve Deployment'ı gerçekleştirilmiştir. Diğer servis olan duraklardaki otobüs sürelerini gösteren fonksiyonun geliştirilmesinde yarım kalan var ise veya otobüs içerisindeki ekranlar için geliştirilen serviste bir hata oluşursa ve diğer servise olan bağımlılığından dolayı, her iki serviste kullanılamaz hale gelirse? Bu ve bunun gibi farklı varyasyon ve senaryolar göz önüne alındığında, Monolithic mimaride geliştirilen servislerde yazılım ekiplerinin birbirleri ile iletişim becerilerinin yüksek olması gerektiği, farklı özellikler geldikçe Code Base'in daha da karmaşıklaşacağı ve Micro Deployment'lar yapılamayacağı görülmektedir. Buradaki tek problemimiz Scale etmek ve Micro Deployment'ları sağlayabilmek değildir.

Bu durum birbirlerinden bağımsız (Independently) olarak gerçekleşse idi fena olmaz mıydı? İşte bu noktada geleneksel SOA mimarisi yaklaşımı yerine yenilikçi SOA ile mikroservis yaklaşımı ortaya atılmaktadır. Mikroservis yaklaşımı için ise geleneksel SOA'nın getirdiği karmaşıklığı ve yönetimini kolaylaştıran bir kavramdır denmektedir.

2.2. Mikroservislere Genel Bakış

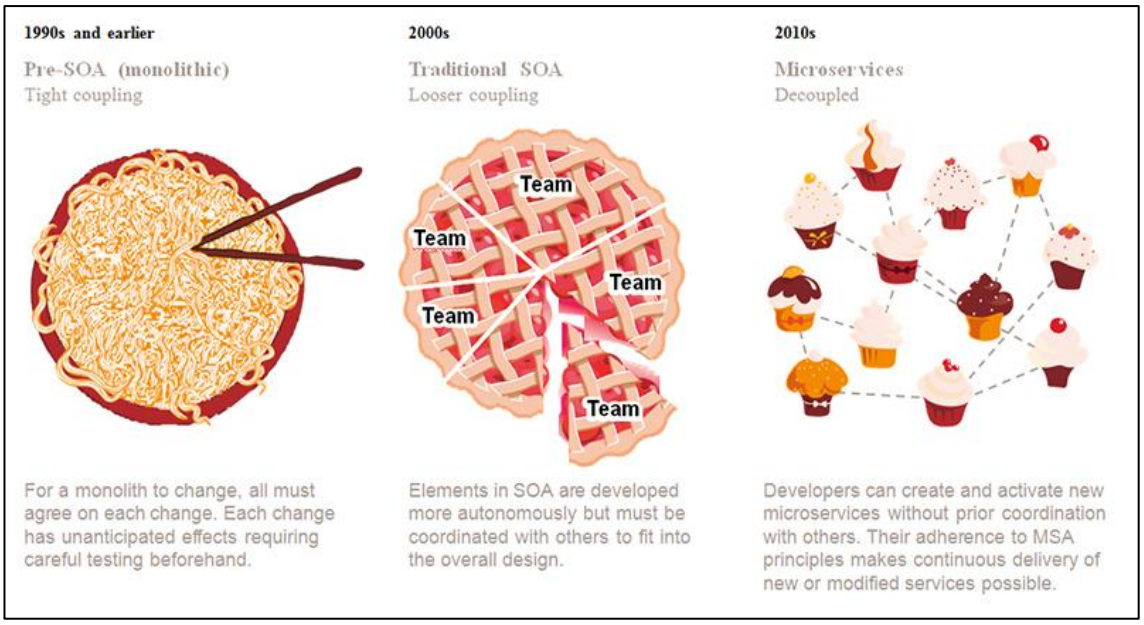
Resme (Şekil 2.3.) ilk baktığımızda Monolitik yapı gibi tüm sistemin Self-Contained olarak geliştirilmesi yerine, her bir Parçanın/Component'in kendi bünyesinde Self-Contained olarak modüler bir şekilde geliştirildiğini görülmektedir.



Şekil 2.3. Monolitik ve Mikroservis Karşılaştırlması

2.2.1. Mikroservis Mimarisinin Getirdiği Bazı Avantajlar

Mikroservis yapısı sürekli ve plansız bir şekilde büyüyen Monolithic yapıdaki servislerin, beraberinde getirdiği karmaşıklığı ve yönetim zorluklarını çözmeye odaklanmaktadır. SOA'ya alternatif bir model değildir. Geleneksel SOA yaklaşımı yerine yenilikçi SOA yaklaşımı ile beraber, biraz önce de bahsettiğimiz gibi karmaşıklığı ve yönetimi pratikleştirmeye çalışan bir kavramdır.



Şekil 2.4. Mikroservis Gelişimi

Mimaranin avantajları verilmiştir.

•Servisler farklı dillerde ve farklı Framework'lerde geliştirilebilir.

- Birbirlerinden bağımsız olarak her bir servis değişebilir, kolay Test ve Build yapılabilir.
- Continuous Delivery'e olanak sağlar ve hızlı Deployment'lar gerçekleştirilebilinir.
- Her bir servisi birbirinden bağımsız olarak Scale edebilme olanağı sağlar.
- Her bir servis birbirinden bağımsız olacağı için, Code Base'i sade ve Maintenance'ı kolay olacaktır.
- Versiyonlama kolay bir şekilde yapılabilecektir.

Bunlara ek olarak da geliştirilecek olan yeni özellikler ise kolay bir şekilde implemente edilebilir olacaktır. Bu avantajlardan yararlanan teknoloji firmaları ve nedenleri anlatılmıştır;

Uber, Netflix, Amazon, Ebay firmaları Mikroservis Mimarsinin kullanmaktadırlar. Bunlar gibi büyük firmaların sorunları, yükü kolay bir şekilde Scale edebilmek ve Deployment süreçlerini Continuous Delivery ile kolay bir şekilde ele alabilmektedir. Gerektiğinde saniyeler arasında, dakikalar arasında Deployment işlemlerini gerçekleştirmektedirler.

2.2.2. Mikroservis Mimarisinin Getirdiği Bazı Dezavantajlar

Birbirlerinden bağımsızlaşan farklı servisler aynı Business Objelerini kullanacaklarından dolayı kaçınılmaz bir kod tekrarı meydana gelecektir.

- Servisler farklı platform ve ortamlarda çalışabileceklerinden dolayı yönetim ve Monitoring maliyeti ortaya çıkacaktır.
- Birden çok Database ve Transaction'ların yönetimi zor olabilir.

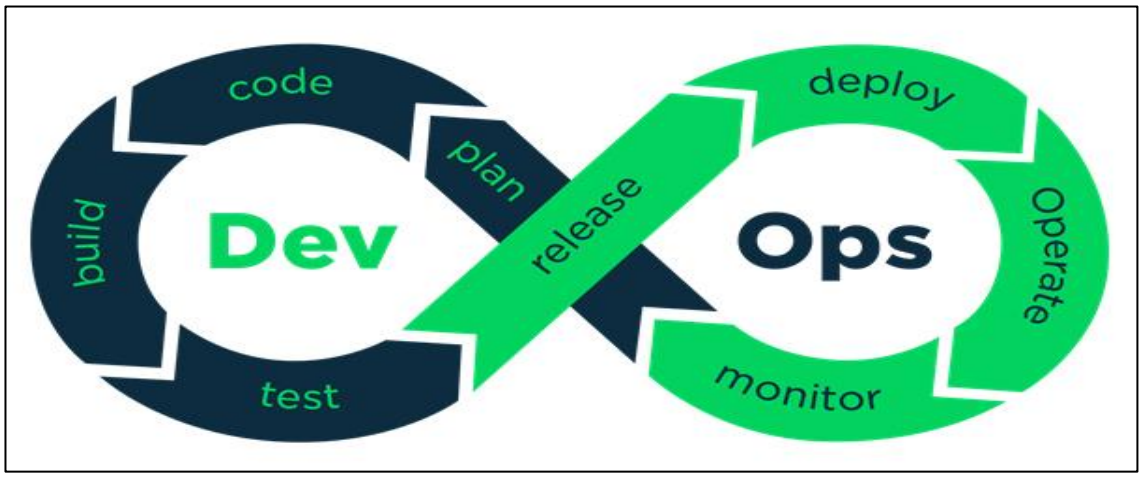
Bu maddelerin zaten birçoğu adreslenmiş durumda ve otomasyon araçları ile yönetilebilmektedir. Bunlara ek olarak da zaten ihtiyaçlar doğrultusunda mikroservis yaklaşımının getirileri göz önüne alındığında ise bu dezavantajlar görmezden de gelinebilmektedir. Yönetim kısmında ise DevOps kavramı ile kolay bir şekilde ele alınabilmektedir. Transaction yönetimi işlemlerinede DTC Distrubuted Transantion Manager) ile çözüm bulunabilmektedir.

3. MİKROSERVİS MİMARİSİ VE DEVOPS

Mikroservislerin avantajları ve Monolitich mimariden farklarından bahsettikten sonra mikroservis kavramı ile birlikte hayatımıza giren ve yukarıda bahsedilen dezavantajlarını absorbe etmek için kullanılan bazı kavramları ve teknolojileri de bilmeli ve bu konulara da değinilmiştir.

Mikroservis Mimarisi yukarıda da bahsedildiği üzere DevOps kavramıyla iç içedir (Management, Scale, vs).

Öncelikle ne olduğuna ve neden ortaya çıktığına değinilmiştir.



Şekil 3.1. DevOps Mimarisi

DevOps Bilgi Teknolojileri departmanı içerisinde bulunan iki temel birimi (Developers And Operations) Geliştiriciler (Yazılım Geliştiriciler, Yazılım Testçileri, vb.), Operasyon (Sistem Mimari ve Altyapı Ekipleri, Güvenlik ve Ağ ekipleri vb.) bir arada etkili bir iletişim içerisinde beraber çalışmalarıdır. DevOps'u aslında bir felsefe, yaklaşım veya bakış açısı olarak değerlendirebiliriz. Yazılım geliştiricilerin alışık olduğu Scrum, Agile, Kanban ve diğer yöntemledir.

3.1. DevOps Takımı

Oluşturulacak uygulamaya ait planları yapmaktır. Uygulamayı oluşturmaktır (Kodlamak). Uygulama Release ve Publish (Versiyonlama ve Yayınlama) yapmaktır. Uygulama iyileştirme (Update), Uygulama Test Süreçlerini gerçekleştirmek gibi sorumlulukları vardır.

Oluşturulan uygulamaların barındırılacağı ve kullanılacağı ortamı tasarlamaktır. Uygulamaların çalışması için gereken sistem bileşenleri ile iletişime geçebilmeleri için gerekli ağ ve güvenlik yapılandırmalarının yapılmasını sağlamaktır. Uygulamanın kaynak kullanımını belirlemektir. Uygulamanın gerekli izleme (Monitoring) araçları ile takibini sağlamaktır. Uygulamanın sistem kaynaklarını kullanım düzeyine göre kaynak arttırımını sağlamak (Scale Up ve Scale Down) gibi sorumlulukları vardır.

Yukarıda belirtilen takımların birbirleriyle yaptığı çalışmalar ve iletişim yoğunluğu sayesinde devreye alınması gereken yenilik ve düzeltmeler çok kısa süre içinde işleme alındıkları için verim ve başarı yüksek olmaktadır. CI&CD (Sürekli Entegrasyon ve Sürekli Dağıtım) mantığı DevOps kavramı ile beraber oluşmaktadır. Ortaya çıkarılan ürünlerin bir otomasyon çevresinde ilerlemesi Dağıtım (Deploy), Versiyonlama (Release) ve Test süreçleri olarak DevOps içerisindeki tüm personelin uygulamanın kodlanmasından çalıştırılmasına ve yaşam döngüsünden haberi olmasını sağlıyor.

Yazılım geliştirme sektöründe olan diğer meslek arkadaşlarının ekleyeceği birden fazla düşünce bakış açısı vardır. DevOps felsefesi tüm BT sektörü için daha güncel ve zamana uygun şekilde modernize olmuş hali ile hayatımız da ve sektörümüzde yer alıyor. Bu felsefe ve iş yapış türüne adapte olmak bize yeni olan tüm teknolojiler ile daha hızlı tanışmamızı ve adapte olmamızı sağlıyor. Tüm bu süreç topluluğuna ise DevOps deniyor.

Görüldüğü üzere DevOps anlayışı Mikroservis Mimarisini inşaa etme de ve devamlılığını sağlama da önemli bir anlayıştır. Bir Mikroservis Takımının (Dev ve Ops) beraber çalışması, Mikroservis Mimarisinin de bir anlayışı olan CD, CI gibi kavramların üzerine oluşmaktadır. Özellikle Mikroservis Mimarisinin en önemli dezavantajlarından olan Management ve Monitoring gibi sorunları gidermekte DevOps anlayışı önemli bir rol oynamaktadır.

Mikroservis Mimarisinde hızlı Deployment'lar hızlı geliştirmeler ve birbirinden bağımsızlık gibi konularda da uygulama anlamında DevOps araçlarına önemli ölçü de ihtiyaç duyulmaktadır. Bu yüzden Mikroservis Mimarisinde bir uygulama geliştirilecekse DevOps kavramına aşina olunması gerekmektedir.

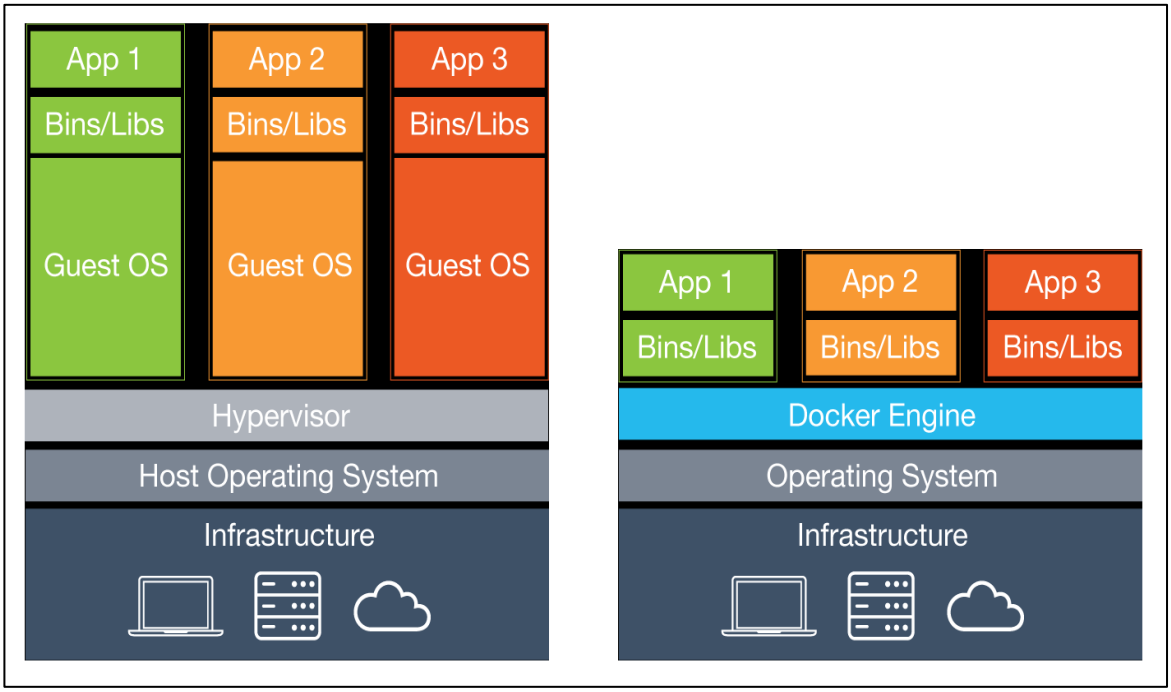
3.2. Docker

Docker, yazılım geliştiriciler ve sistemciler için geliştirilen açık kaynaklı bir sanallaştırma platformudur. Docker ile Linux, Windows ve MacOSX üzerinde Linux ve Windows Sanal Containerler (Makineler) çalıştırılmaktadır. Bu platform sayesinde web

sistemlerinin kurulumunu, testini ve dağıtımını kolaylıkla gerçekleştirilmektedir. En önemli özelliği belki de "Benim bilgisayarımda çalışıyordu, sunucuda neden çalışmıyor? Sorununu ortadan kaldırıyor olmasıdır.

Docker'ın sanallaştırma yapısı, bilinen sanal makinelerden (VirtualBox, Vmware vb) farklı olarak bir Hypervisor katmanına sahip değildir. Bunun yerine Docker Engine üzerinden, konak işletim sistemine erişmekte ve sistem araçlarını paylaşımlı kullanmaktadır. Böylece klasik VM'lere göre daha az sistem kaynağı tüketmektedir.

Docker, LXC sanallaştırma mekanizması üzerine kurulu. Bir Docker Imajı, Container denilen birimlerde çalıştırılıyor. Her bir Container bir süreç (process) kullanmaktadır. Bir makinada gücüne bağlı olarak binlerce Docker Container'ı birden çalıştırabilmektedir. Container Imajları ortak olan sistem dosyalarını paylaşmaktadır. Dolayısıyla disk alanından tasarruf etmektedir. Şekil 3.2.'de görüldüğü gibi uygulama Container'ları ortak bin (exe) ve kütüphaneleri kullanıyorlar. Ancak klasik sanal makine sistemlerinde her bir uygulama için ayrı işletim sistemi ve kütüphane dosyaları ayrılmak zorundadır.



Şekil 3.2. VM Vs Docker

Docker, yazılımların kurulu son hallerinin imajını alıp tekrar kullanılabilir olmasını sağlamaktadır. Bu imajları bir kere oluşturup diğer sunuculara gönderebilirsiniz ya da her sunucuda farklı imajlar oluşturulabilmektedir. Dockerfile adı verilen talimat

dosyalarına bakarak her sunucu aynı imajı yeniden inşa eder. Bu sayede manuel bir müdahale gerekmemektedir.

Bir diğer özellik ise Dockerfile ve imajların geliştirilebilir olmasıdır. Talimatlara birkaç adım daha eklemek isterseniz en baştan komutları vermek yerine kaldığı son yerden devam etmekte ve bu da zaman kazandırmaktadır.

3.2.1. Docker'ın VM'e Göre Avantajları

VM'ler her bir çalışan örneği için full bir işletim sistemine sahiptir. Docker ise hem full işletim sistemi yerine boyut olarak küçültülmüş imajları kullanır hem de konak işletim sistemi kütüphanelerini paylaşımlı olarak kullanmaktadır. Fakat bu durum, Docker'i sistem kaynak tüketim dostu yaparken, izolasyon seviyesini ise düşürmektedir.

3.2.2. Docker Compose

Docker Compose, kompleks uygulamaların tanımlanmasını ve çalıştırılmasını sağlayan bir Docker aracıdır. Docker Compose ile birlikte birden fazla Container tanımını tek bir dosyada yapılabilmektedir, tek bir komut ile uygulamanızın ihtiyaç duyduğu tüm gereksinimleri ayağa kaldırarak uygulama çalıştırılmaktadır.

Docker Compose ile birden fazla Container çalıştırılabilmektedir, bu Container'lardan bazılarının birbirine bağımlı kalmasını istenmektedir. Örneğin bir WordPress ayağa kaldırmak istenmektedir. Bu durumda bir MySQL ve WordPress Image tanımı yapılmaktadır, WordPress'i veritabanına (MySQL'e) bağımlı hale getirilmektedir (depent). Bu sayede ilk olarak veritabanı ayağa kalkar, sonra da uygulamanız (WordPress) çalıştırılır.

Temel kullanım alanının yanında Docker Compose'u şu farklı amaçlar için de kullanılmaktadır.

- Development Environments: İşe yeni başlayan biri için Compose ile çok kısa sürede geliştirme yapabileceği bir ortam hazırlanmaktadır.
- Automated Testing Environments: CI Pipeline için kullanılabilir. Test otomasyonları için çalışacak uygulama ve Script'ler çalıştırılabilir.
- Single Host Deployments: Tek bir host üzerinde herşeyin çalışması sağlanabilir.

Docker Compose'da diğer Docker CLI (Command Language Interface) komutlarını rahatça kullanılmaktadır. Docker sadece Mikroservis Uygulamaları için kullanılması zorunlu olan bir teknoloji değildir.

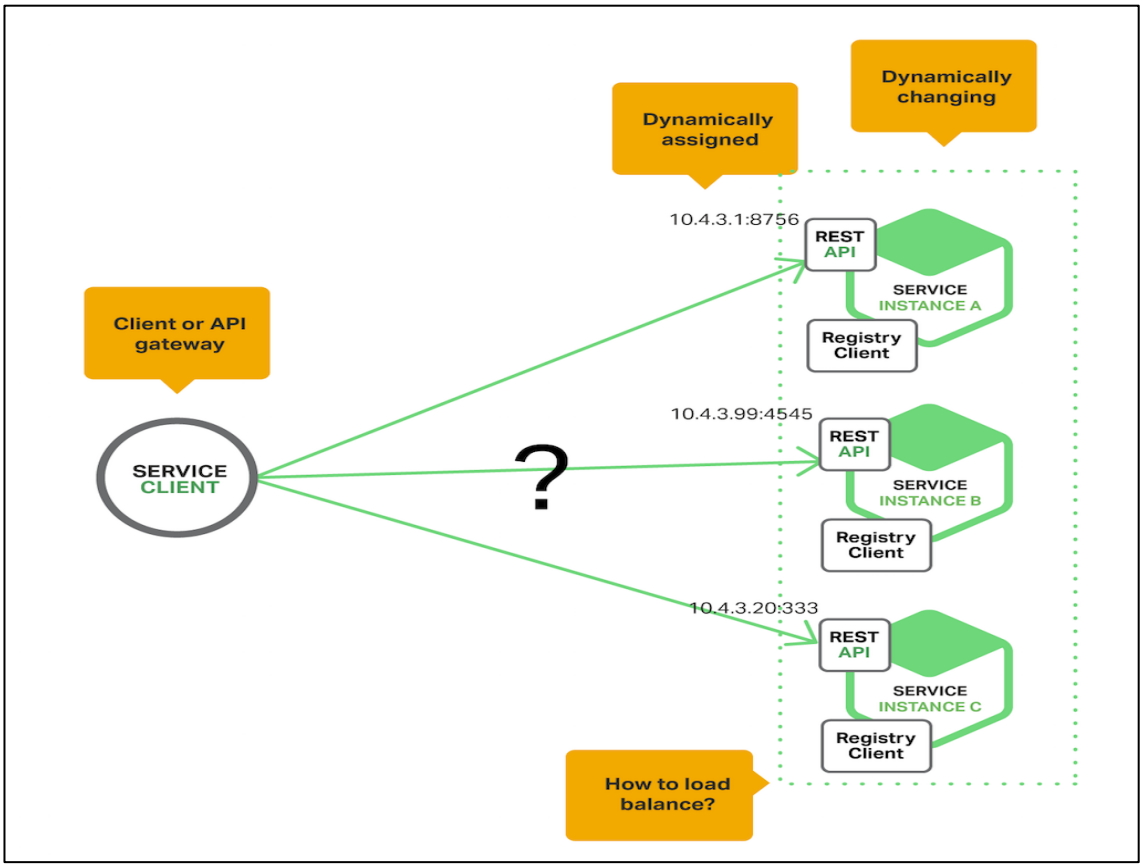
Bu uygulamada ve Microservice uygulamalarında Docker'ı kullanma amacımız yukarıdaki kısımda belirtilen performans ve kullanım kolaylığı(image'ler vs) gibi özellikleridir.

Ayrıca Microservice uygulamalarını da Docker ile ayağa kaldırmak zorunda da değildir. Sunucusunuz Mimarilerde de Mikroservislerini deploy etmektedir(AWS,Azure Cloud vs). Mimariye ve kullanılan teknolojilere odaklandığımızdan ve Sunucusuz Mimari konusunun da başlı başına işlenmesi gereken ayrı bir konu olduğundan bu kısma değinilmemiştir ve her şeyi kendimiz ayarlayıp görme açısından Docker kullanılmıştır.

3.3. Service Registration And Discovery

Mikroservisler ile alakalı diğer konularda bahsedildiği gibi, bizlere kattığı artıların yanında bazı zorlukları da beraberlerinde getirdiklerinden bahsedilmiştir. Bu zorluklardan belki de en önemlileri ise, Management ve Monitoring konularıdır.

Monolithic yapılara baktığımızda ise her şey tek bir çatı altındadır. Yazılımcılar için proje git gide büyümeye başladığında pek de hoş olmasa da, operasyonel anlamda sistemciler için oldukça rahattır. Mikroservis yapılarına baktığımızda ise, işler operasyonel anlamda değişmektedir. Çünkü n tane farklı sorumlulukları yerine getiren Distributed Mikroservisler ortaya çıkmaya başlıyor ve işte bu noktada Mikroservislerin beraberlerinde getirdiği Management ve Monitoring gibi ortak zorluklar ile "Service Discovery" kavramı ortaya çıkmaktadır.



Şekil 3.3. Client Request

Şekil 3.3.'e bakıldığında, Auto Scale olan ve Dynamic olarak değişen Instance'lara sahip bir yapı görülmektedir. Burada dikkat edersek IP adresleri de Dynamic olarak Assign edilmektedir. Bu gibi Dynamic Case'ler karşısında, Client hangi IP adresine istek atacağını bilemeyecektir. Service Discovery ise bu gibi durumları nasıl otomatik olarak Handle edebilmeye odaklanmaktadır.

Service Discovery temel olarak üç kavram üzerinde durmaktadır:

- Discovery: Servislerin Dynamic bir ortamda Cluster içerisindeki diğer servisler ile iletişim kurabilmeleri için, birbirlerinin IP ve port bilgilerini bulmaya ihtiyaçları vardır. Discovery ise bunu sağlamaktadır.
- Health Check: Health Check işlemi ile sadece Up olan servislerin sistemde kalmaları, Down olanların ise Dynamic bir şekilde sistem dışı kalmaları sağlanmaktadır.

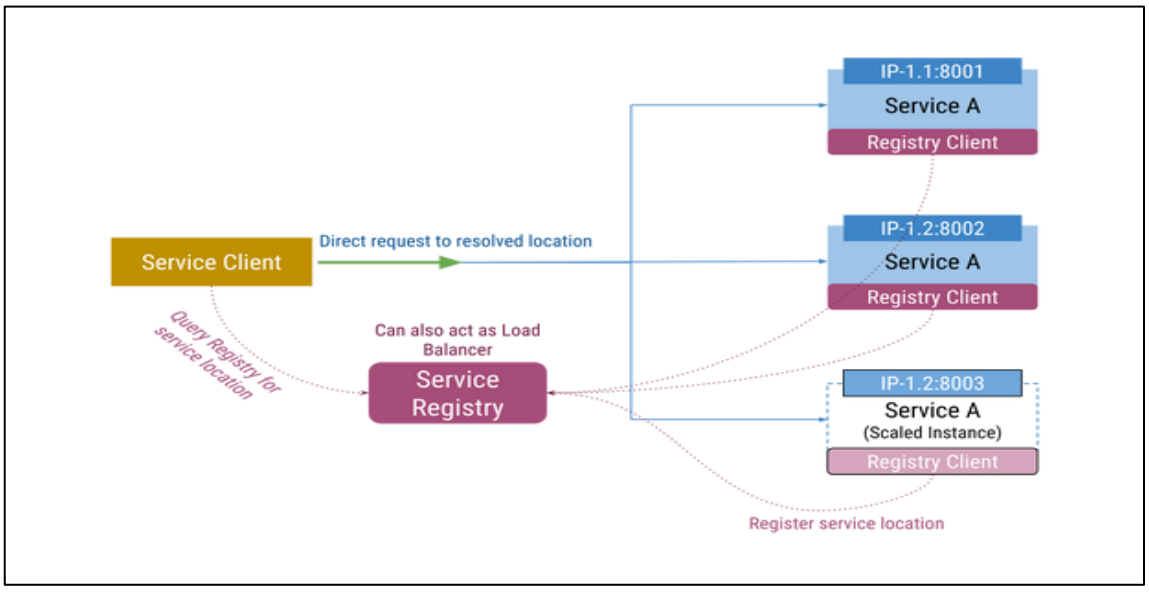
- Load Balancing: Hepimizin de bildiği gibi bir hizmete gelmiş olan Request'in, bu hizmeti sağlayan diğer Instance'lara da Dynamic olarak dağıtılmasını sağlamaktır.

Service Discovery'i uygulayabilmek için "Client-Side Service Discovery" ve "Server-Side Service Discovery" olmak üzere iki farklı Pattern bulunmaktadır.

3.3.1. Client-Side Service Discovery Pattern

Bu yaklaşımda Servis Instance'ları, kendi Network Location'larını Service Registery üzerine kayıt eder. Service Registery ise Service Discovery'nin bir parçasıdır. Bu sayede buradaki servislere, hangi IP ve port üzerinden erişilebileceği bilgisi Service Registery üzerinde bulunmaktadır. Client ise herhangi bir Request'i göndermeden önce Service Registery'e gelerek, Request göndermek istediği servisin lokasyon bilgilerini elde eder ve o bilgiler doğrultusunda Request işlemini gerçekleştirir. Kompleks olarak gözükebilir ama genel anlamda bakılacak olursa kolay bir işlemdir.

Bir diğer yandan bu pattern Load Balancer hatalarından sistemi korur, Balancing işlemini ise Client'a bırakır. Client, Registery üzerinden istediği servisin IP ve port bilgilerini alır ve birden çok IP adresine sahip ise kendi belirleyeceği bir IP adresine Request işlemini gerçekleştirir. Bu noktada dezavantajına baktığımızda ise artık Client'lardır, Service Registery ile konuşması gerektiğini bilmek zorundadırlar ve Balancing işlemlerini kendileri Handle etmelidirler.



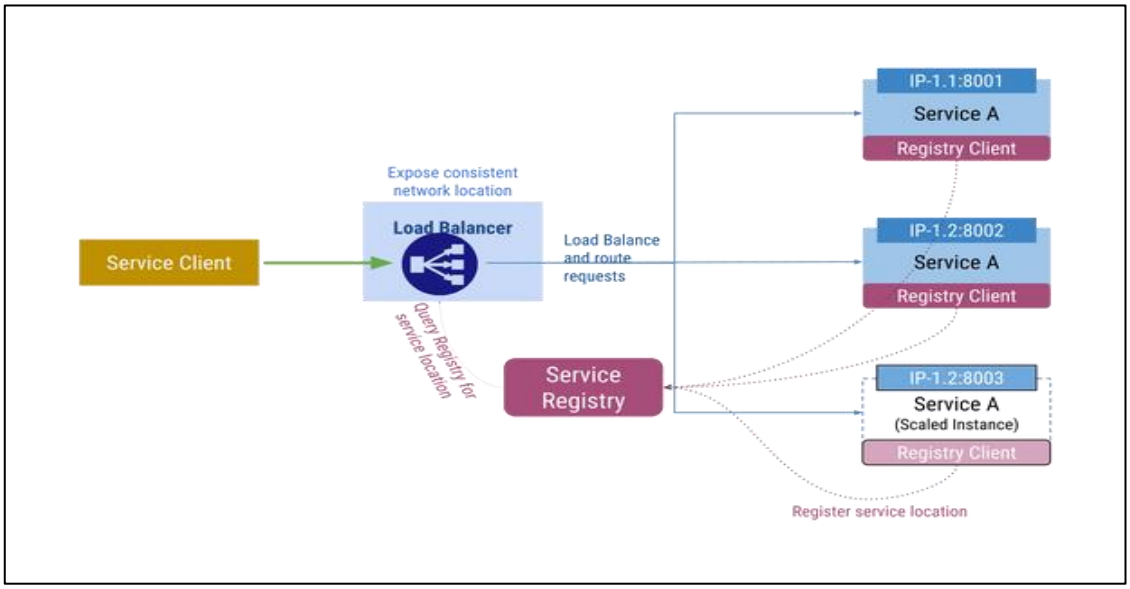
Şekil 3.4. Client-Side Service Discovery Pattern

3.3.2. Server-Side Service Discovery Pattern

Bu yaklaşımda ise ilk nokta; Client'ın artık ilk olarak Service Registery ile konuşmak yerine direkt olarak Load Balancer üzerine istek atmasıdır. Load Balancer ise Service Registery üzerinden ilgili servisin lokasyon bilgilerini alarak, Route işlemini kendisi gerçekleştirmektedir.

Pattern'ın avantajı ise, Client ile Service Registery'nin Decoupled bir şekilde olmaları ve Client'ın hangi servisin hangi Node'da olduğundan bir haberi olmamasıdır. Dezavantajı ise buradaki Load Balancer'ın Single Point Of Failure durumda olmasıdır.

Service Discovery ve Register kavramını ve Pattern'lerini neden kullanılması gerektiğini geniş bir şekilde açıkladıktan sonra, şimdi bu Pattern'leri bizim için sağlayan bir araç olan Consul incelenmiştir. Uygulamada Consul kullanılmıştır. Consul ile ilgili diğer detaylar Uygulama Anlatım kısmında anlatılacaktır.



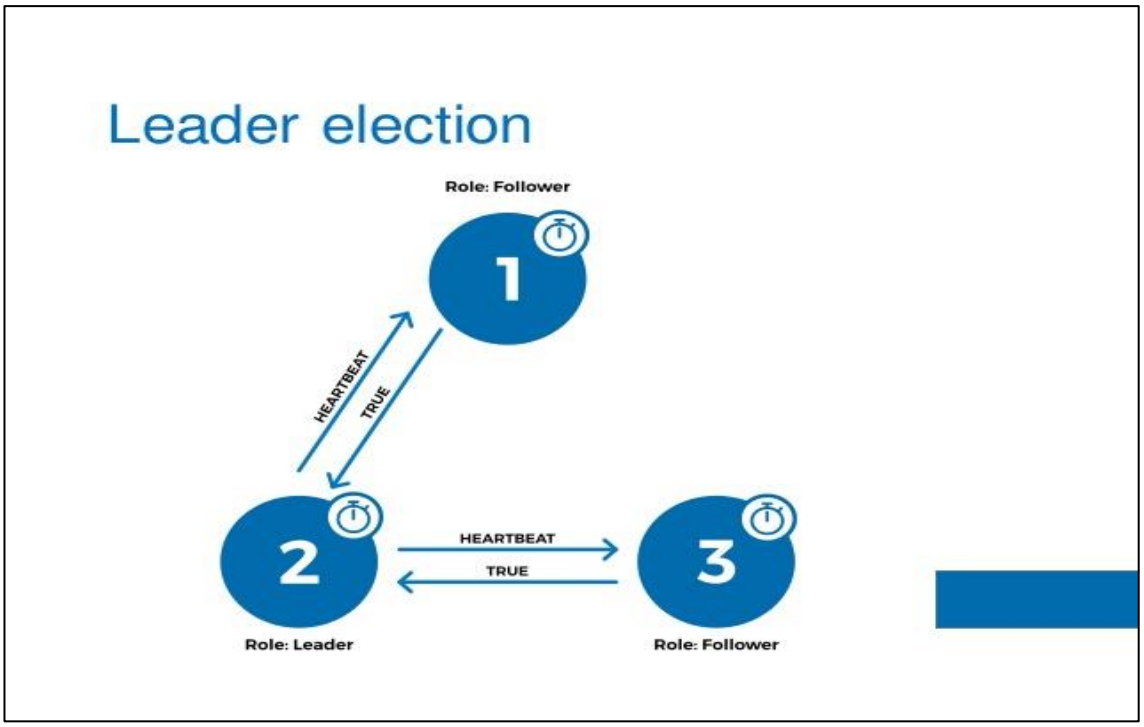
Şekil 3.3.2. Server-Side Service Discovery Pattern

3.4. Consul

Consul, kapsamlı bir service discovery aracıdır. Öncelikle Consul'un mimarisine biraz değinecek olursak tutarlılık için Server Node'larında Raft consensus'u kullanmaktadır.

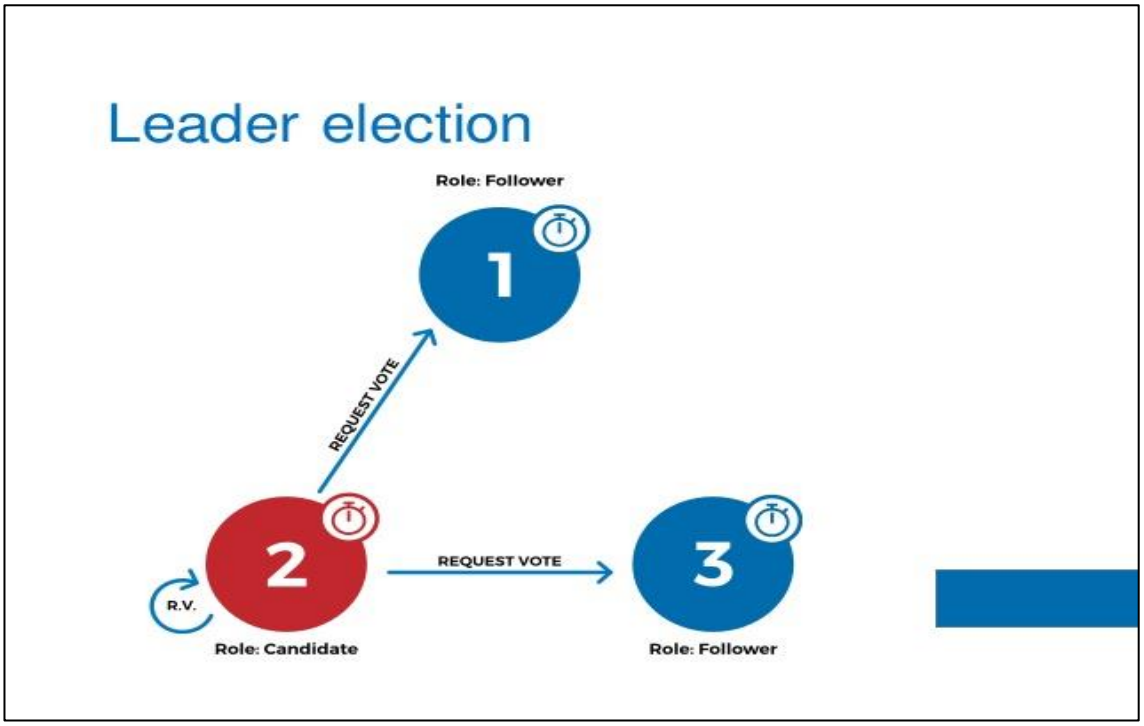
Raft Consensus, Paxos temelli bir Consensus algoritmasıdır. Distributed Computing'de, Consensus'u sağlayabilmek için kullanılan bir protokol olarak tanımlanmıştır. Raft' a geri dönecek olursak, Paxos'a göre daha basit ve anlaşılabilir bir algoritma olarak tasarlanmıştır. Consul ise Raft'ı, node'lar arasındaki tutarlılık durumunu veya Leader Election'ı sağlayabilmek için kullanmaktadır.

Örnek vermek gerekirse: 3 adet Server Node'u olduğunu düşünelim. Bu Node'lar aralarındaki Consistency'i sağlayabilmek için Leader Node, aşağıdaki gibi diğer Node'lara bir Heartbeat göndermektedir.



Şekil 3.4. Leader Election

Heartbeat timeout'a uğrar ise, x Node'undan birtanesi yeni bir Election(oylama) başlatacaktır. Election ise Node'lar arasından hangisinin yeni Leader olacağına karar verilebilmesi için yapılmaktadır. Bu işlemin gerçekleşebilmesi için ise Election'ı başlatan Node'dır, aşağıdaki gibi diğer Node'lardan leader olabilmek için oy istemektedir.



Şekil 3.4.1. Leader Election Vote

Diğer Node'lardan gerekli oyu alabilirse, Leader olarak seçilmektedir. Bu oy isteme işlemine ise "Quorum" denmektedir ve bu işlem için (n/2)+1 kadar üye gerekmektedir. Consul'ün diğer bazı architectural detaylarına baktığımızda ise:

- Etkileşim için bir REST Endpoint'i sunmaktadır.
- Dynamic Load Balancing işlemini gerçekleştirebilmektedir.
- Multiple Datacenter desteği vardır.
- In-built olarak kapsamlı bir service Health Checking sağlamaktadır.
- Service Database'i için, Distributed Key-Value Store'a sahiptir.

Bunlara ek olarak Consul, Highly Fault Tolerant'a sahiptir. Tüm Consul Service Cluster'ı Down olduğunda dahi, bu durum Service Discovery işlemini durdurmayacaktır. Bu işlemi ise Consul, Serf ile sağlamaktadır. Serf, tamamen bir Gossip Protokol'ü olup bir nevi Node Orchestration Tool'udur. Serf Nembershipment'ı yönetmek, Failure Detection ve Event Broadcasting yapabilme işlemlerini sağlayabilmektedir. Ayrıca server Node'ları için ise Clustering sağlamaktadır. Consul'u tanımlamaya çalışırken birçok farklı konuya değinilmiştir. Toparlamak gerekirse eğer, gördüğümüz gibi birçok bileşeni mevcuttur.

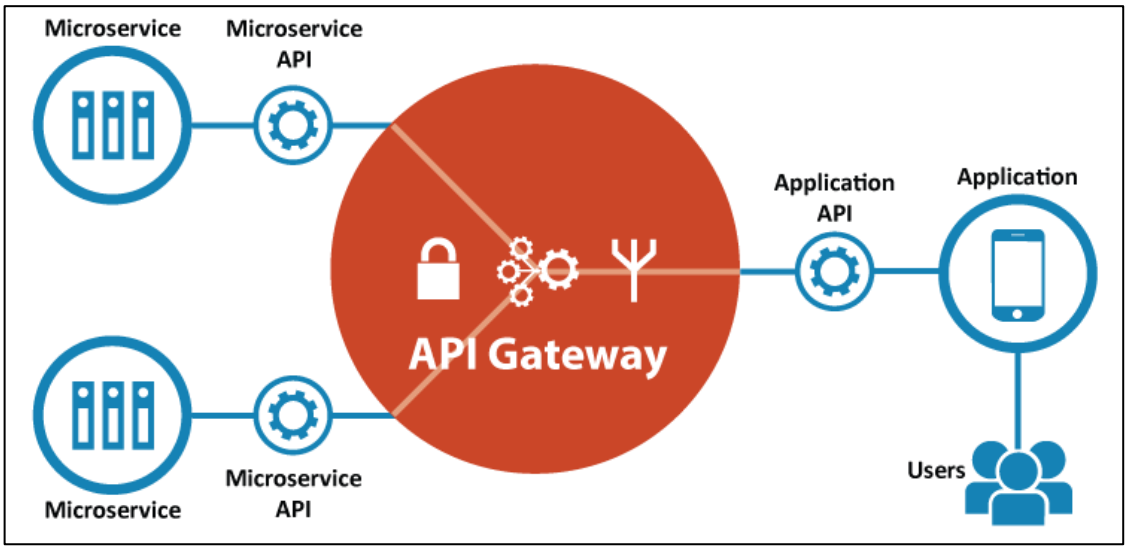
Bir bütün olarak baktığımızda ise altyapımızdaki servisleri Discovery edilmesi ve Configuring işlemlerini yapılması için geliştirilmiş bir Tool'dur. Consul'un mimarisi ile ilgili son olarak da Health Checking konusuna da değinmek gerekirse, Health Checking işlemini ise Client Agent'ları aracılığı ile yapabilmektedir.

3.5. API Gateway

Projede kullanılan Consul aracının marifetlerini anlatılmıştır. Service Discovery kısımında bahsedilen Server-Side Service Discovery Pattern konusunu biraz daha açmak gerekmektedir.

Proejede Server-Side Service Discovery Pattern'i kullanılmıştır. Client'ın Load Balancer ile iletişim de olduğu ve Load Balancer'ın client'dan gelen isteği Consul'a ilettiği Pattern'dir.

Load Balancer'ın diğer bir adı da API Gateway'dir. API Gateway(Load Balancer) konusuna değinilmiştir.



Şekil 3.5. API Gateway

Kullanıcı uygulamadan bir Request attığında arkada neler dönüyor bunu bilmemektedir. Api-Gateway bu Request'e cevap verebilmek için içerde belki de onlarca mikroservise gidip gelebilmektedir. Burada datanın nasıl Fetch ve Aggregate edileceği konusunda API-Gateway Pattern'i neye ihtiyacımız olduğunu çözmek için vardır.

Api-Gateway temel davranışları;

- Router: Mikroservisler arasında haberleşmeyi sağlar. Bir servisten diğerine gelen istekleri iletir.(Service Discovery)

- Data Aggregator: Mikroservisler arasında bilgi toplayarak ve bunları zengin bir Response halinde bağlı olduğu Api Consumer'a iletir. Bu durumda Backend For Frontend (BFF) gibi davranmıştır.
- Protocol Abstraction Layer: Api-gateway'e Rest Api veya GraphQL üzerinden gelen isteklerle yani protocol ve teknoloji farketmeksizin içerdeki mikroservislerin iletişimini sağlar.
- Centralized Error Management: Bir servise ulaşılamadığı zaman veya servis aşırı yavaş cevap vermeye başladığı zaman Api-Gateway ölümcül hataların yayılmaması için Cache den Default Response'lar sağlamaya başlar. Sistemi daha güvenilir ve esnek hale getirmek için erişelemeyen servis yeniden ayağa kalkana kadar kapatılır.

API Gateway kavramı Client ile Mikroservisleri birebir ilişkiye sokmamak için kullanılmıştır (Yüzlerce mikroservis birbirleriyle haberleşebilir vs). API Gateway'e gelen isteğin hangi servise gideceğine Consul karar vermektedir. Service Discovery bize yardımcı olmamaktadır. Service Discovery kavramı bize sadece API Gateway'den gelen isteğin arka tarafta hangi Service portunda olduğunu ve cevap verecek servisi bulunmasını sağlamaktadır.

API Gateway ne de Service Discovery aslında içeride diğer servislere gidip tek tek istek atmamaktadır. Bu servislerin cevap verme süresinide uzatacağından doğru bir yaklaşım olmamaktadır.

Teori de API Gateway Client'den isteği alır ve geriye birleştirilmiş bir kaç servisten alınan dataları birleştirip dönmüş şekilde bir cevap vermektedir.

Pratik de diğer servislerin verilerine ulaşmak ve onları birleştirip API Gateway'e geri dönmek için belli kodlar yazılmalı ve servisler arasında iletişim sağlayacak bir araç kullanılmalıdır.

Farklı Port'larda ki mikroservisler birbirleriyle asenkron şekilde haberleştirilmeli ve dataları birleşecek olan servislerde birleştirilip cevap vermelidir.

Bize bu konu da yardımcı olacak en güzel araç RabbitMQ'dür.

3.6. RabbitMQ

RabbitMQ bir mesaj kuyruğu sistemidir. Benzerleri Apache Kafka, Msmq, Microsoft Azure Service Bus, Kestrel, ActiveMQ olarak sıralanmaktadır. Amacı herhangi bir kaynaktan alınan bir mesajın, bir başka kaynağa sırası geldiği anda iletilmesidir. Mantık olarak Redis Pub/Sub'a benzemektedir. Ama burada yapılacak işler bir sıraya

alınmaktadır. Yani iletimin yapılacağı kaynak ayağa kalkana kadar tüm işlemler bir Quee'de sıralanmaktadır. Fakat aynı durum Redis Pub'Sub için geçerli değildir. RabbitMQ çoklu işletim sistemine destek vermesi ve açık kaynak kodlu olması da en büyük tercih sebeplerinden birisidir.

Bazı işlemlerin anlık yapılmasına ihtiyaç yoktur. Örnek vermek istenir ise sisteme yeni bir haber girildiğinde, ya da var olan bir haberin güncellenmesi anında Cache'in düşürülmesi, bir başka örnek de Upload edilen "Gif" dosyalarının Scale işleminin yapılmasıdır. Zaman ayarlı mesaj ve otomatik mailler de yine RabbitMQ'ya güzel bir örnek olmaktadır. Sıraya alınan bu işlemlerin asenkron bir şekilde yapılması, hem çalışan uygulamanın boş yere bekletilmemesinden hem de sunucu üzerindeki işlem maliyetinin minimuma indirilmesinden dolayı RabbitMQ iyi bir tercih sebebi olmaktadır. Ayrıca Scalable olmasından dolayı da değişen trafikli yapılarda ayrıca tercih edilmektedir.

RabbitMQ'nün ne olduğunu ve neden kullanılması gerektiğini anlatılmıştır. Projede RabbitMQ'yü mikroservisleri asenkron olarak birbirleri ile haberleştirmek için kullanılmıştır. Detaylar Uygulama Anlatımı kısmında anlatılacaktır.

RabbitMQ ve mikroservisler arası iletişimi de anladıktan sonra bir diğer konu olan loglama ve log toplama, arama konuları da değinilmiştir.

Mikroservislerin veya diğer mimarilerdeki projelerin bir sorunu değil bir ihtiyacıdır. Bir sistemin tüm işlem bilgilerinin sürekli olarak toplanması ve bir şekilde gösterilmesi ihtiyacı vardır.

3.7. Elasticsearch, Logstash, Kibana ve Beast(Elastic Stack)

BT sistemlerinden çeşitli sebeplerle log toplama ve arama ihtiyacı doğmaktadır. Örneğin bir alışveriş sitesinde kullanıcı davranışlarını loglamak (müşteriler hangi tarayıcılarla, hangi ülkelerden geliyorlar, hangi ürünleri geziyorlar vb.) bu amaçlardan birisi olabileceği gibi, Windows+Linux hibrit bir ortamdaki sistemlerden (kullanıcı yada sunucu) yada IPS/IDS/WAF/NAC/Firewall/Proxy/DHCP/DNS vb. gibi sistemlerden de güvenlik yada regülasyon amaçlı log toplamak istenmektedir.

Şekil 3.7. Elastic,Logstash ve Kibana

Elastic Stack, açık kaynak kodlu log toplama, arama ve analiz bileşenlerinin tamamına verilen genel addır. Kabaca aşağıdaki bileşenlerden oluşur.

Elasticsearch;

- Kibana
- Logstash
- Beats ailesi
- Winlogbeat
- Filebeat
- Packetbeat

Bu bileşenler açıklanmıştır.

3.7.1. Elasticsearch

Elasticsearch esasında bir arama ve veri indeksleme motorudur. Geriplanında, Apache Lucene projesini kullanılmıştır. Verileri Logtash yada doğrudan Beats Agent'ları üzerinden alarak onları aranabilir halde indexlemektedir. REST ve JSON teknolojilerini kullanarak HTTP protokolü üzerinden veri üzerinde arama, ekleme, silme vb. operasyonları yapmanızı sağlar. En büyük avantajlarından birisi esnekliği ve bu sebeple sağlanan kolay ölçeklenebilme avantajıdır.

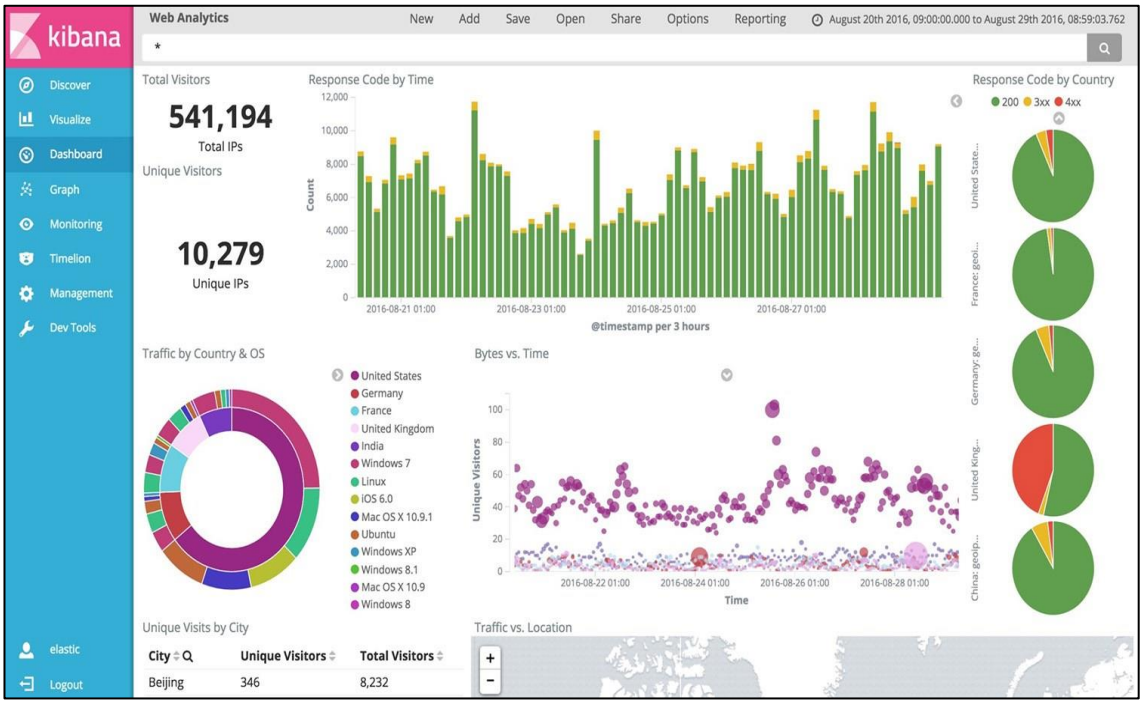
3.7.2. Logstash

Log toplayan, ihtiyaca göre onları işleyen ve Elasticsearch'e bu logları indekslenmek üzere gönderen sistemdir. Tipik bir konfigürasyonu şöyledir; input { stdin { } } output { stdout { } }

Burada input alanında logların Logstash bileşenine nasıl geldiğini, output'da ise işlendikten sonra nereye gönderildiği ifade edilmektedir. Bu örnekte örneğin konsoldan gelen logu yine konsola basılmaktadır. Bu input alanı 514 Syslog, bir Beats Ajanı (Örneğin Windows bir sistemden log alan ve Logstash'a gönderen Winlogbeat) da olabilir.

3.7.3. Kibana

Elasticsearch Lucene tabanlı bir arama ve log indeksleme moturudur. REST ve JSON teknolojileri kullanarak Curl, Python vb. onlarca farklı imkanla bu veritabanından veri okuyup/aratıp/yazıbilmektedir. Bunu bir web arayüzüyle görsel olarak yapmak istediğimizde ise arkada bu işlemleri bizim yerimize yapan Kibana karşımıza çıkıyor Kibana için Elastichsearch , Logstash ‘in arka planda yaptıklarından sonra bir Client tarafı ile gösterilmesidir.



Şekil 3.7.1. Kibana

3.7.4. Beats

Beats ürün ailesinin üyelerini kabaca sistemlerden Logstash'a log gönderen ajanlar olarak düşünülmektedir.

Bazı örnekleri şu şekildedir;

Winlogbeat: Windows Eventlog okuyup bunu indensklenmek üzere gönderen Beats ailesi üründür.

Packetbeat: Kurduğunuz sisteme dair network verisi getiren bileşendir. Bu sayede örneğin kritik bir sunucunuzdaki network anormallikleri izlenmektedir.

Filebeat: Bir dosyayı izleyerek bu dosya içeriğindeki değişiklikleri gönderen Beats ailesi ürünüdür. Örneğin /var/log/httpd.log dosyanızı bu ajana gösterek bu dosyayı loglamaktadır.

3.8. Bölüm Özeti

Bu kısımda bir Mikroservis Mimarisinin tam veririmlilik ile çalışabilmesi ve dezavantajlarının absorbe edilmesi için neler yapılması gerektiği, DevOps felsefesinin ve DevOps araçlarının bize Mikroservis Mimarisini inşaa ederken ne gibi katkıları olduğunu proje de kullanılan DevOps süreçleri ve DevOps araçları üzerinden açıklanmıştır. Sırada ki Bölüm’de ise bu araçların nasıl kullanıldığı Mikroservis Mimarisinin nasıl inşaa edildiğini projemiz üzerinden anlatılacaktır. Bu bölümde bazı kavramlar anlatıldığından dolayı Uygulama kısmında sadece uygulamanın nasıl çalıştığı anlatılacaktır, DevOps araçlarının nasıl sisteme dahil edildiği ve nasıl çıktılar alındığı gibi konulara değinilecek ve bu bölümde anlatılan konulara teorik olarak yer verilmeyecektir.

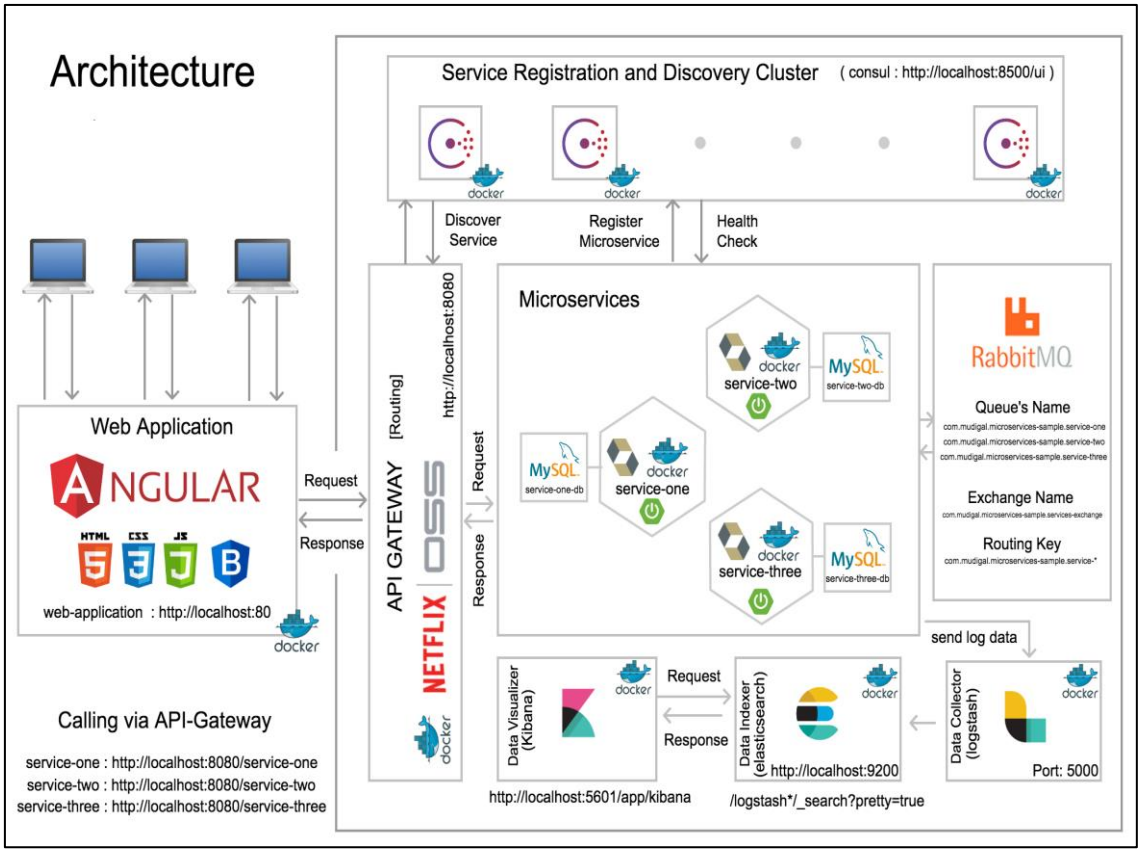
4. UYGULAMA ANLATIMI

4.1. Projenin Genel Mimarasi

Proje örnek uygulamada 3 adet örnek servisimiz bulunmamaktadır. Bunlar ürün, stok ve fiyat servisleridir. Bu üç servisin üç adet veritabanı vardır. Bunlar ürün db, stok db ve fiyat db'dir.

Servisler Deploy olurken ürün servisi rastgele bir ürün adı üretir. Stok servisi rastgele stok adedi üretir.

Fiyat servisi rastgale bir fiyat üretir ve bunları kendi Database'lerine kayıt ederler. Her servis rastgale ürettiği servisi RabbitMQ kuyruğuna atarlar ve servisler kuyruğu sürekli dinlenmektedir. Burada servisler birleriyle Asyc olarak haberleşmektedir. Yani ürün oluştu stok oluştu ve fiyat oluştu bu bilgiler tüm servislerin veritabanında tutulur. Detaylı şekilde aşağıda açıklanacaktır.



Şekil 4.1. Uygulamanın Mikroservis Mimarisi

4.2 Projenin Docker-Compose.yml Çalışma Akışı

Image: Container'a hangi Image ile çalışacağı bildirilir.

Services: Düğümde Docker Compose ile yönetmek istediğimiz servisleri teker teker sıralarız. Projede bulunan tüm modüller servis olarak ayağa kalkar.

Container_name: Servis isimlerimizi veririz.Vermez ise otomatik olarak kendisi oluşturur.

Hostname: Servisin ana adını tutar.

Build: İlgili servisin hangi DockerFile ‘ a göre Build edileceğini bildirir.

Ports: Sol tarafındaki port Docker Image’ın kendi portu sağ taraftaki port Image’ın dışarıya açılan portu Eğer 8080: 444 1600-1700 şeklinde olsaydı 1600-1700 portundan gelen istekler Container’a Forward edilecektir.

Command: İlgili servis için kullanılan Image’ın sağladığı komuttan farklı bir komut kullanmak istersek Command’ a yazılır.

Expose: Portu dışarıya açmadan servisler arasındaki iletişimi sağlamak için yazılır.

Enviroment: Container için özel çevre değişkenleri tanımlanır.

Networks: Servisin dahil olacağı networku belirlenir.

Links: Başka servisteki Container’a bağlı olduğu bildirilir.

Depends_on: Servisin bağımlı olduğu olduğu servisler bildirilir. Docker-compose up komutu çalıştırıldığında Depends_on ‘a yazılan servisler sırası ile ayağa kaldırılır.

4.3. Projenin Başlatılması

İlk olarak uygulamayı nasıl Deploy edeceğimizi gösterilmektedir.

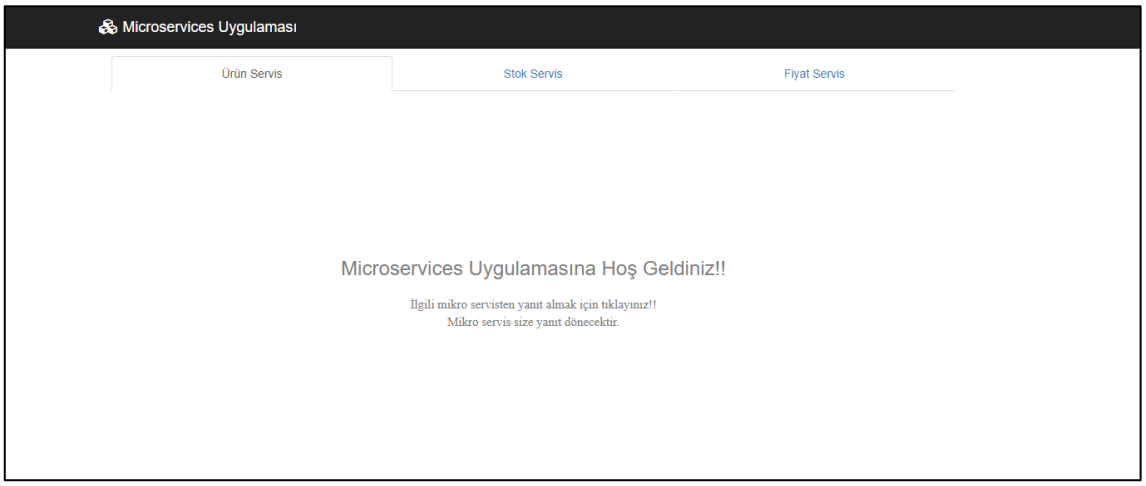
Yapılması gereken ./deploy.sh dosyasının çalıştırılması gerekmektedir.

./deploy.sh dosyasının içeriği Şekil 4.2’de mevcuttur.

```
if [ $# == 0 ]
then
     echo "Kullan:

Input Parameters :

     1) Profile adını giriniz 'docker'"
     exit 1
fi


cd ../../..

# Tüm maven paketleri build alınır.
mvn clean package

# docker-compose'un bulunduğu klasöre gidiyoruz.

cd build/docker/

# Profile aldığımız
# Parametreyi atıyoruz
export profile=$1

# Deploy
docker-compose up --build -d
```

Şekil 4.2. deploy.sh Dosyasının İçeriği

Proje Deploy olduğunda http://localhost adresinden uygulamaya erişim sağlanmaktadır.



Şekil 4.3. Uygulama Anasayfa

Uygulama Şekil 4.3.'te görüldüğü gibi açılmaktadır.

4.4. Arayüz Adımları

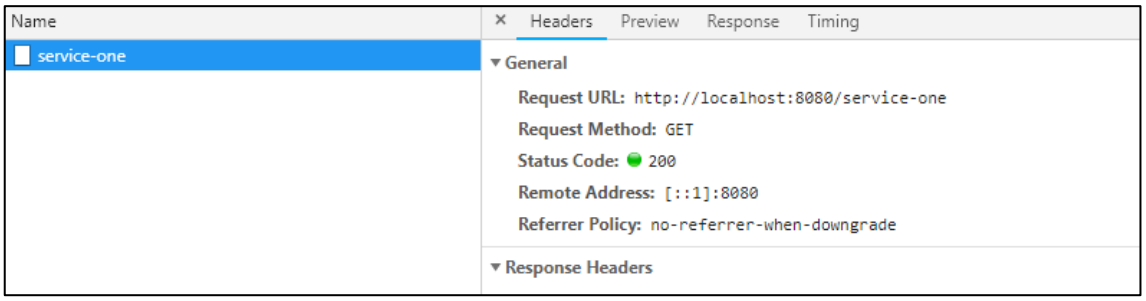
Uygulama Şekil 4.3.'de görüldüğü gibi açılacaktır.

Çağırmak istediğimiz servisi tablardan seçerek çağırılmaktadır.

Örnek olarak ürün servisini çağırılmaktadır. İsteğin hangi aşamalardan geçtiğini sırası ile anlatılmıştır.

İlk olarak Network'den gelen paketi incelenmektedir.

Görüldüğü üzere(Şekil 4.4) istek http://localhost:8080/service-one ‘a atılmaktadır



Şekil 4.4. Ürün Servisi Cevap

Response Şekil 4.5’de gösterilmiştir.

Şekil 4.5. Ürün Servisi Response

Burada önemli olan nokta istediğimiz direk olarak service-one (ürün) servisine itmemiştir. İstek ilk olarak Api Gateway’a gitmektedir.

Api Gateway’da gelen istediğin nasıl service-one (ürün) servisine gittiğine bakılmıştır.

4.5. Api Gateway Ayarları

Api Gateway Docker üzerinde bir servis olarak ayağa kalkmaktadır. Şekil 4.6’da yapılandırma ayarları gösterilmektedir.

```yaml
services:
  # ---------------------------
  # API Gateway
  # ---------------------------
  api-gateway:
    container_name: "api-gateway"
    hostname: "api-gateway"
    build: ../../api-gateway/target
    ports:
      - "8080:8080"
    expose:
      - "8080"
    links:
      - consul
      - logstash
    environment:
      - SPRING_PROFILES_ACTIVE=${profile}
    networks:
      - backend
```

Şekil 4.6. Api Gateway Ayarları

Api Gateway servis olarak başlatır. Gelen istediğin nereye yönlendirileceğini ve Cors ayarları yapılır.

Cors ayarları Şekil 4.7'dedir.

```java
/**
 *
 * @author Barış Can Akdağ
 *
 */

@EnableZuulProxy
@SpringBootApplication

public class ApiGatewayApplication {

    public static void main(String[] args) { SpringApplication.run(ApiGatewayApplication.class, args); }

    @Bean
    public CorsFilter corsFilter() {
        final UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
        final CorsConfiguration config = new CorsConfiguration();
        config.setAllowCredentials(true);
        config.addAllowedOrigin("*");
        config.addAllowedHeader("*");
        config.addAllowedMethod("GET");
        source.registerCorsConfiguration( path: "/**", config);
        return new CorsFilter(source);
    }
}
```

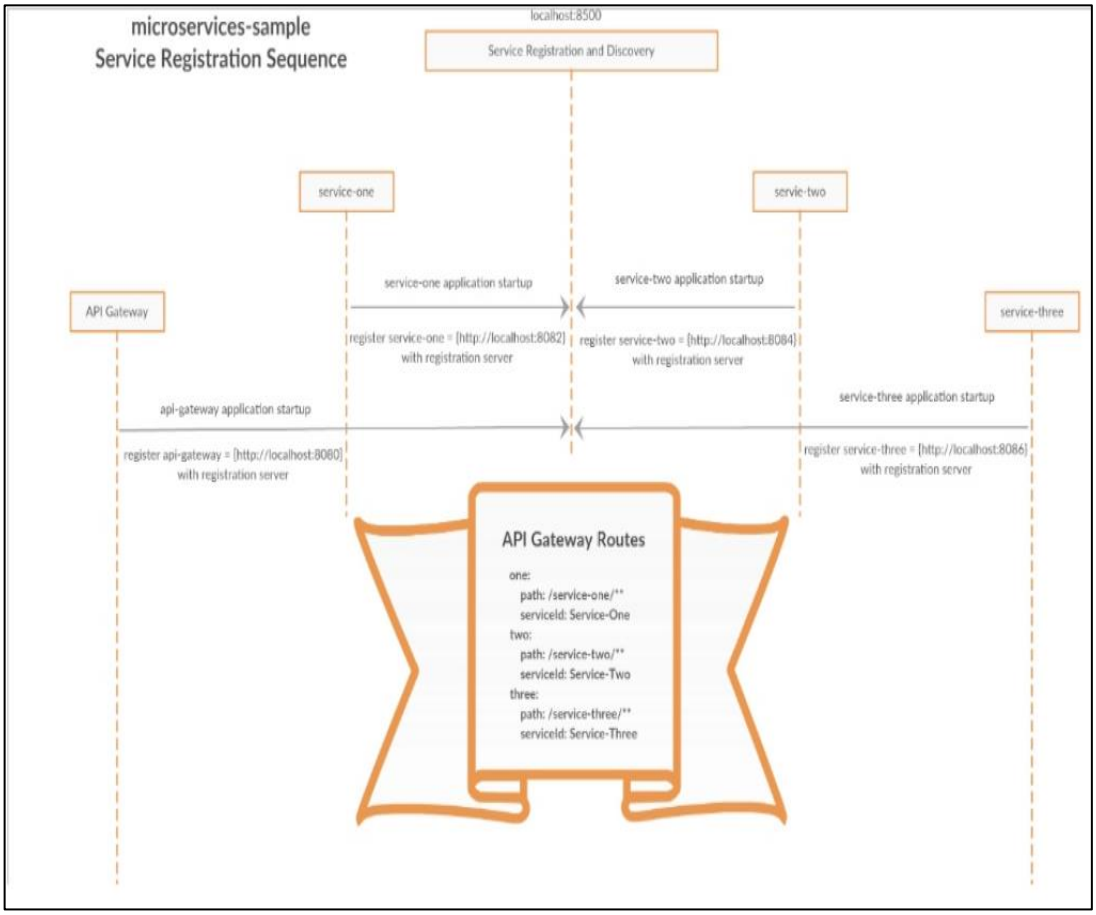
Şekil 4.7. Cors Ayarları

Yönlendirme ayarları Şekil 4.8.'dedir.

```yaml
server:
   contextPath: /
   port: 8080

zuul:
   ignoredServices: '*'
   routes:
      one:
         path: /service-one/**
         serviceId: Service-One
      two:
         path: /service-two/**
         serviceId: Service-Two
      three:
         path: /service-three/**
         serviceId: Service-Three
```

Şekil 4.8. Yönlendirme Ayarları

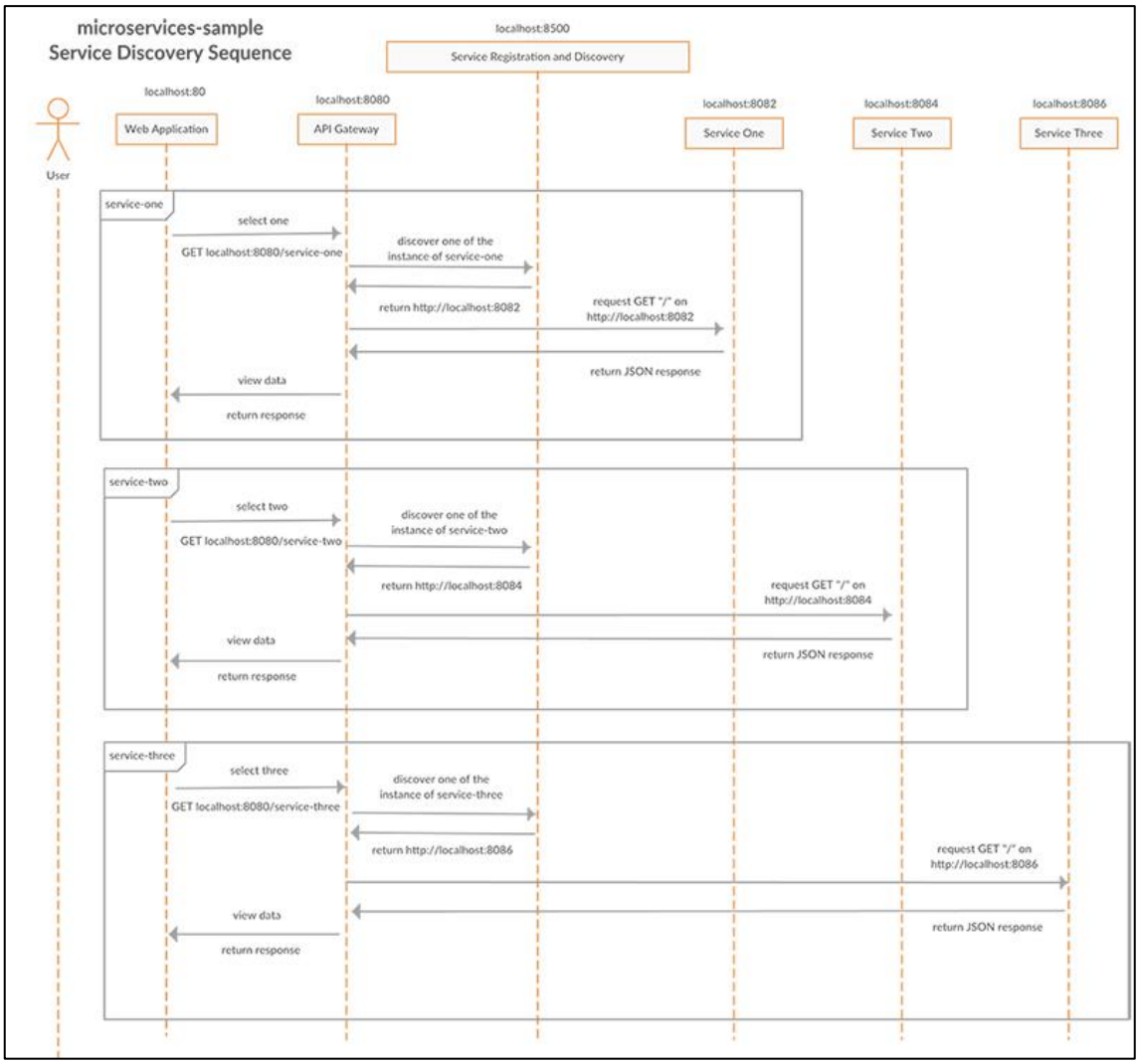
Şekil 4.8.'de görüldüğü gibi path : /service-one/** gelen isteklerin id'sini Service-One olarak işaretlenmiştir. Buradan sonra istek Consul'a düşmektedir. Consul tüm servislerin tutulduğu merkezi yapımızdır aşağıdaki resimde Consul ve Api Gateway bağlantısı gösterilmiştir.



Şekil 4.9. API Gateway

Gelen istediğin yönlendirmesini Api Gateway yapar ve Consul'a iletir. Consul id'si Service-One(ürün) olan servise istediği iletir. Şekil 4.5.4 Use-Case 'de detayları verilmiştir.

Şekil 4.10.'de görüldüğü gibi kullanıcı web uygulamasından service-one(ürün) servisine istek atar. Api Gateway isteği Consul'a iletir. Consul id'si Service-One olan servisi arar. Servisi bulduğu zaman gelen istediği Service-One(Ürün) servisine iletir. İstediği Service-One işler ve Api Gateway üzerinde geriye dönmektedir.



Şekil 4.10. Request Use Case-2

4.6 Consul Ayarları

Şekil 4.11.'de kodda Cluster için 3 adet Node'umuz olacağını belirtilmiştir.

data-dir: Parametresi ile, Agent'ın yaşam döngüsü boyunca State'inin tutulacağı klasörü gösterilmiştir.

Bootstrap-expect: Parametresine 3 değerini geçerek, Cluster'ın çalışabilmesi için en az 3 Agent'ın Server olarak çalışması gerektiği söylenmiştir. Yukarıda bahsedilen 2n+1 durumunu burada canlandırılmıştır. Parametrenin kullanılabilmesi için Server parametresi muhakkak eklenmelidir.

Server: Parametresi ile de Agent'ın Server Mode'da çalışması gerektiğini belirtilir.

Ui: Parametresi ile bu Agent'ın web arayüzü sunacağı belirtilir.

Rety-join ifadesi ile Agent'ın hangi Cluster'a dahil olacağı belirtilir.

Uygulamada 3 Node tek bir Cluster'a dahildir. Eğer uygulamada herhangi bir Node'da servis sıkıntı yaşar ise diğer Node'da çalışmaya devam edicektir.

Bu şekilde servisler Cluster Mimarisi ve Consul ile birlikte yönetmiş ve monitor edilmiştir.

```
# ----------------------------
# Registration and
# Discovery Cluster
# ----------------------------
consul:
  image: consul:0.7.2
  container_name: "consul"
  hostname: "consul"
  command: consul agent -server -client 0.0.0.0 -ui -bootstrap-expect=3 -data-dir=/consul/data -retry-join=consul2 -retry-join=consul3 -datacenter=blr
  ports:
    - "8500:8500"
    - "8600:8600"
  networks:
    - backend
consul2:
  image: consul:0.7.2
  container_name: "consul-2"
  hostname: "consul2"
  expose:
    - "8500"
    - "8600"
  command: consul agent -server -data-dir=/consul/data -retry-join=consul -retry-join=consul3 -datacenter=blr
  links:
    - consul
  networks:
    - backend
consul3:
  image: consul:0.7.2
  container_name: "consul-3"
  hostname: "consul3"
  expose:
    - "8500"
    - "8600"
  command: consul agent -server -data-dir=/consul/data -retry-join=consul -retry-join=consul2 -datacenter=blr
  links:
    - consul
    - consul2
  networks:
    - backend
```

Şekil 4.11. Consul Ayarları

Consul'u servisleri dinlemesi için de yapılandırılması gerekmektedir. İlk olarak Dockercompose.yml dosyasında Consul'u servisleri Depends_on veya link şeklinde verilmiştir (Şekil 4.11).

```
 7    api-gateway:
 8      container_name: "api-gateway"
 9      hostname: "api-gateway"
10      build: ../../api-gateway/target
11      ports:
12        - "8080:8080"
13      expose:
14        - "8080"
15      links:
16        - consul
17        - logstash
18      environment:
19        - SPRING_PROFILES_ACTIVE=${profile}
20      networks:
21        - backend
```

Şekil 4.12. API Gateway İçin Consul Yapılandırması

```
37  ---
38  # DOCKER CONFIGURATION
39  spring:
40    profiles: docker
41    cloud:
42      consul:
43        host: consul
44        port: 8500
45        discovery:
46          hostName: api-gateway
47          instanceId: ${spring.application.name}:${spring.application.instance_id:${random.value}}
48          healthCheckPath: ${management.contextPath}/health
49          healthCheckInterval: 15s
50
```

Şekil 4.13. API Gateway İçin Docker Yapılandırması

4.7. Logstash Uygulama Ayarları

Uygulamanın loglama tabanında üç tane teknoloji kullanılmaktadır. Bunlar Elasticsearch, Logstash ve Kibana'dır. Logstash üç servis için logları toplamaktadır ve bunları Elasticsearch'a atmaktadır. Şekil 4.14. ve Şekil 4.15.'de service-one (ürün) servisi için yapılandırması mevcuttur.

```
input {
  file {
    path =>  [ "/tmp/spring.log.json" ]
    codec =>   json {
      charset => "UTF-8"
    }
  }
}

output {
  elasticsearch { hosts => ["127.0.0.1:9200"] }
}
```

Şekil 4.14. Logstash Ayarları

```
<configuration debug="false">
    <include resource="org/springframework/boot/logging/logback/base.xml" />

    <!-- Method 2 -->
    <appender name="JSON" class="ch.qos.logback.core.FileAppender">
        <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
            <level>DEBUG</level>
        </filter>
        <encoder>
            <pattern>${FILE_LOG_PATTERN}</pattern>
        </encoder>
        <file>${LOG_FILE}.json</file>
        <encoder class="net.logstash.logback.encoder.LogstashEncoder">
            <includeCallerInfo>true</includeCallerInfo>
            <customFields>{"appname":"service-one","version":"1.0"}</customFields>
        </encoder>
    </appender>

    <!-- Method 3 -->
    <appender name="STASH" class="net.logstash.logback.appender.LogstashTcpSocketAppender">
      <destination>logstash:5000</destination>
      <!-- encoder is required -->
      <encoder class="net.logstash.logback.encoder.LogstashEncoder" />
    </appender>

    <root level="WARN">
        <appender-ref ref="CONSOLE" />
        <!-- Method 1-->
        <appender-ref ref="FILE" />
        <!-- Method 2 -->
        <appender-ref ref="JSON" />
    </root>
```

Şekil 4.15. Logstash Xml Ayarları

4.8. Elasticsearch Ayarları

Şekil 4.16.'de Elasticsearch Docker ayarları mevcuttur.

```
256  # ----------------------------------
257  # Indexing Server
258  # ----------------------------------
259  elasticsearch:
260    image: elasticsearch:5.2-alpine
261    container_name: elasticsearch
262    hostname: elasticsearch
263    environment:
264      - "cluster.name=elasticsearch"
265      - "ES_JAVA_OPTS=-Xms512m -Xmx2048m"
266      - "discovery.zen.ping.unicast.hosts=127.0.0.1"
267    ulimits:
268      memlock:
269        soft: -1
270        hard: -1
271      nofile:
272        soft: 65536
273        hard: 65536
274    ports:
275      - "9200:9200"
276      - "9300:9300"
277    expose:
278      - "9200"
279    networks:
280      - backend
281
```

Şekil 4.16. Elasticseach Docker Ayarları

4.9. Logstash Docker Ayarları

Şekil 4.17'de Docker üzerinde Logstash ayarları mevcuttur.

```
# ----------------------------
# Log Collection and Format
# ----------------------------
logstash:
  image: logstash:5.2-alpine
  container_name: log-stash
  hostname: logstash
  ports:
    - "5000:5000"
  expose:
    - "5000"
  volumes:
    - /Users/Shared/data/tools/logstash/:/tmp/
  command: >
    logstash --debug -e 'input { tcp { port => 5000 codec => json { charset => "UTF-8" } } } output { elasticsearch { hosts => "elasticsearch:9200" } }'
  depends_on:
    - elasticsearch
  networks:
    - backend
```

Şekil 4.17. Docker Üzerinde Logstash Ayarları

4.10. Kibana Ayarları

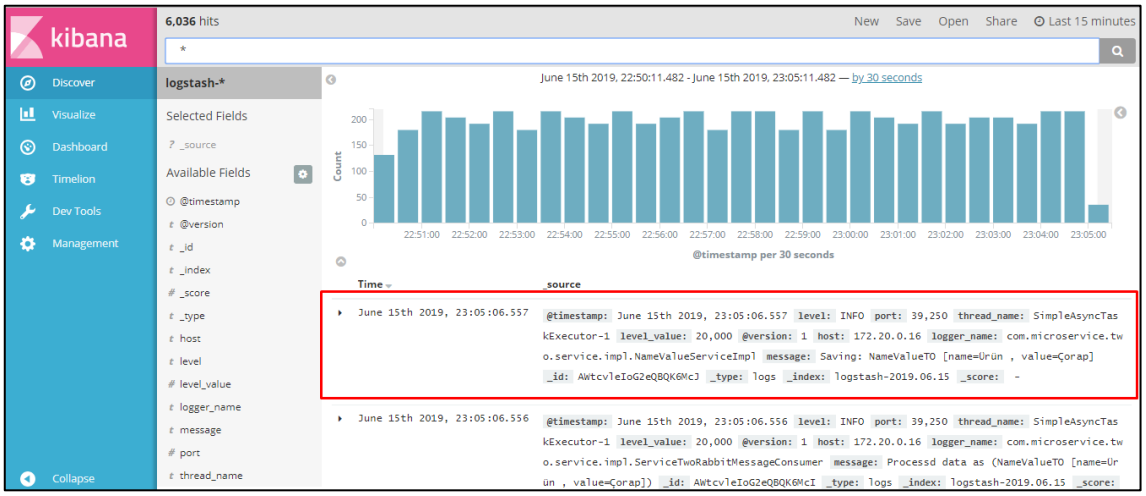
Şekil 4.18'da Docker üzerinde Kibana ayarları mevcuttur.

```
237
238   # ------------------------------------------------------------
239   # Log Vizualization
240   # ------------------------------------------------------------
241   kibana:
242     image: kibana:5.2
243     container_name: kibana
244     hostname: kibana
245     ports:
246       - "5601:5601"
247     expose:
248       - "5601"
249     environment:
250       - ELASTICSEARCH_URL=http://elasticsearch:9200
251     depends_on:
252       - elasticsearch
253     networks:
254       - backend
255
256   # ------------------------------------------------------------
257   # Indexing Server
```

Şekil 4.18. Docker Üzerinde Kibana Ayarları

http://localhost:5601/app/kibana

Projede tüm logları Kibana ile görselleştirebiliriz. Şekil 4.19.'de ekran görüntüsü mevcuttur.



Şekil 4.19. Kibana Ekran Görüntüsü

4.11. Weavescope Ayarları

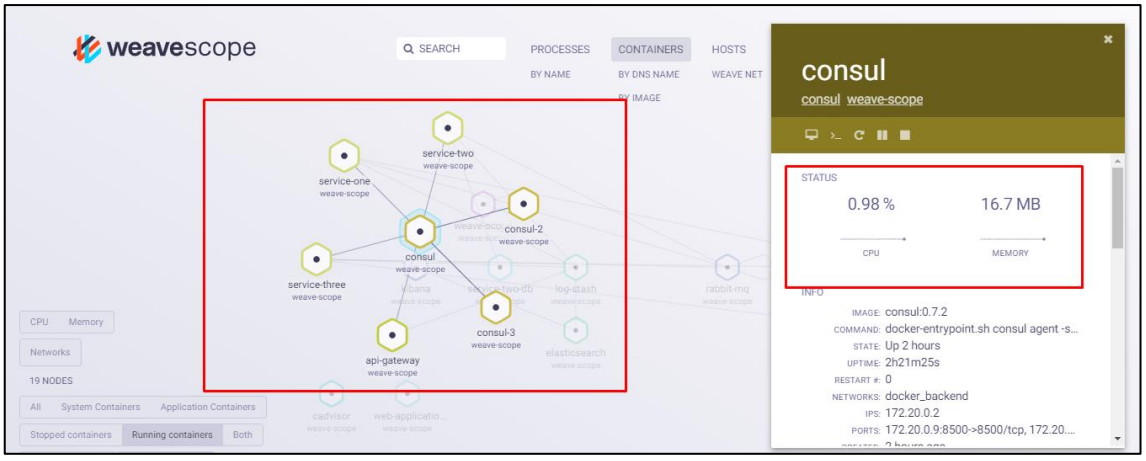
Şekil 4.20'de Docker üzerinde Weavescope ayarları mevcuttur.

```
215  # ------------------------------------------------------------
216  # Container Visualization
217  # ------------------------------------------------------------
218  scope:
219    image: weaveworks/scope:1.1.0
220    container_name: weave-scope
221    hostname: weave-scope
222    pid: "host"
223    privileged: true
224    ports:
225      - "4040:4040"
226    expose:
227      - "4040"
228    labels:
229      - "works.weave.role=system"
230    volumes:
231      - "/var/run/docker.sock:/var/run/docker.sock:rw"
232    command:
233      - "--probe.docker"
234      - "true"
235    networks:
236      - backend
237
238  # ------------------------------------------------------------
```

Şekil 4.20. Docker Üzerinde Weavescope Ayarları

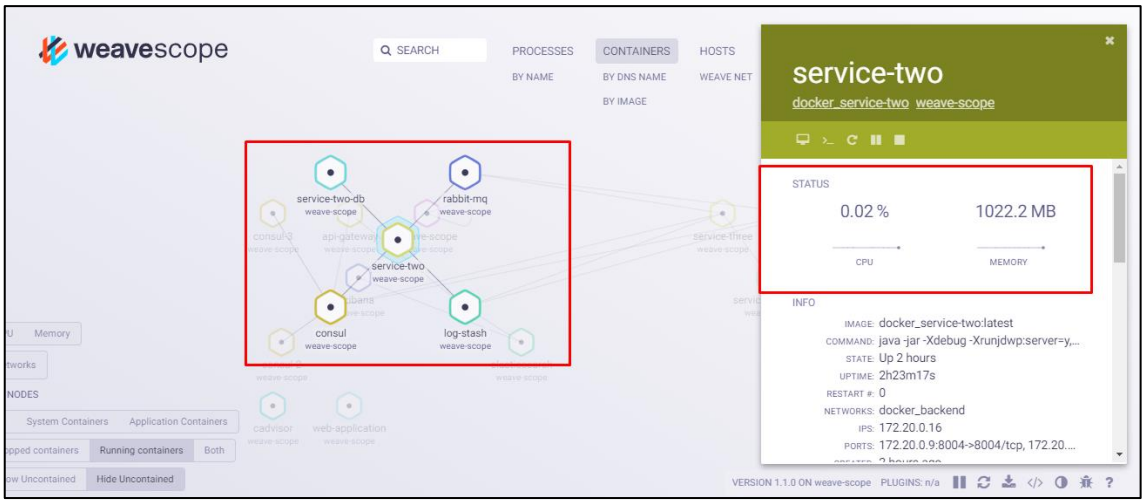
http://localhost:4040 erişim adresidir.

Projede görüntüsü Şekil 4.21.'dedir.



Şekil 4.21. Waevescope Ekran Görüntüsü

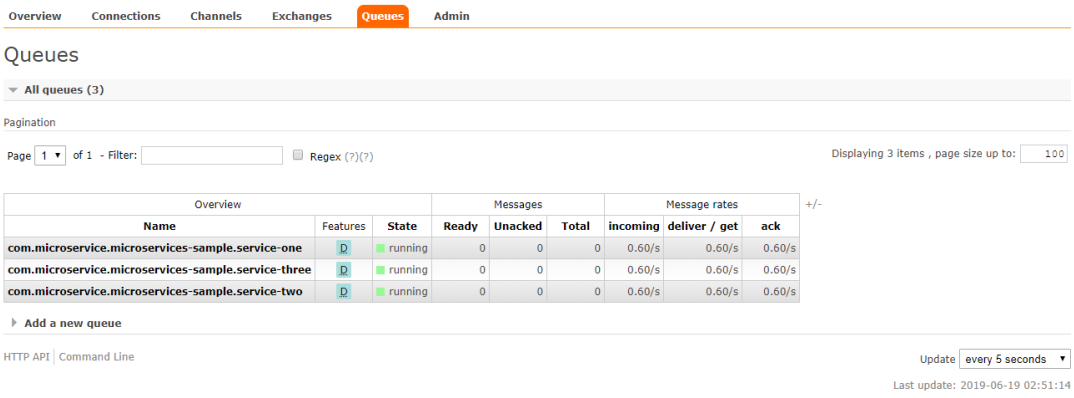
Ürün servisi hakkında bilgi almak istendiğinde Weavescope'dan bakılması yeterlidir (Şekil 4.22.).



Şekil 4.22. Weavescope Ürün Servisi

4.12. RabbitMQ Ayarları

Servisler arasındaki data transferi RabbitMQ ile yapılır.



Şekil 4.23. RabbitMQ Ekran Görüntüsü

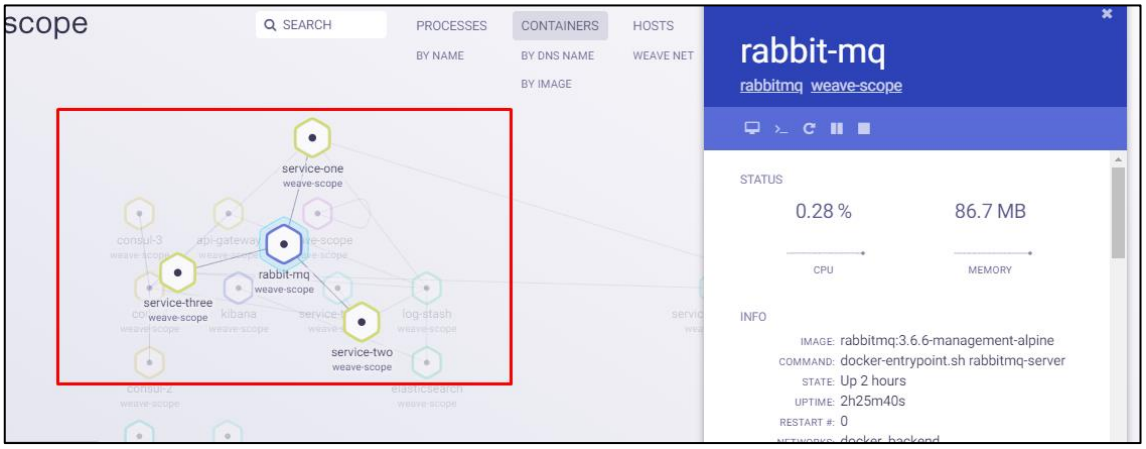
Şekil 4.24.'de Docker üzerinde RabbitMQ ayarları mevcuttur.

```
# ----------------------------
rabbit:
  image: rabbitmq:3.6.6-management-alpine
  container_name: rabbit-mq
  hostname: rabbitmq
  ports:
    - "5672:5672"
    - "15672:15672"
  expose:
    - "15672"
  environment:
    - CLUSTERED=true
    - RAM_NODE=true
    - CLUSTER_WITH=rabbit
    - RABBITMQ_DEFAULT_USER=mudigal
    - RABBITMQ_DEFAULT_PASS=mudigal
  networks:
    - backend
```

Şekil 4.24. Docker Üzerinde RabbitMQ

Servisler arasındaki iletişim RabbitMQ ile sağlanır.

İletişimi Weavescope'dan gösterilmiştir (Şekil 4.25.).



Şekil 4.25. Servisler Arasındaki İletişim

5. SONUÇLAR VE ÖNERİLER

5.1 Sonuçlar

Monolitik Mimarasinden Mikroservis Mimarisine geçildiğinde uygulama atomik parçalara bölünmektedir. Bu sayede parçaları yönetmek kolaydır. Uygulama esnek (flexible) bir şekilde çalışmaktadır. Atomik Servisin hangi programlama dilinde yazıldığı önemli değildir. Hepsi bütünün bir parçasıdır.

DevOps araçları ile bütün parçalar bir bütün haline gelmektedir. Servisler arasındaki iletişim, yönlendirme ve yük dengeleme işlemleri DevOps ile yönetilebilir bir duruma getirilir.

Kullanıcı sayısının artması ve teknolojinin gelişmesi ile birlikte monolitik yapılar artık ihtiyaçları karşılayamamaktadır. En büyük sorunun yönetim olduğu süreçte Mikroservis Mimarilerinin kullanımı artarak devam edicektir.

5.2 Öneriler

Mikroservis yapıları kompleksdir. Eğer geliştirdiğiniz sistem çok büyük ve çok fazla kullanıcıya hitap etmiyorsa Mikroservis Mimarisinde uygulamayı geliştirmek zaman ve maliyette artışa sebep olmaktadır.

Mikroservis Mimarasini yönetebilmek için DevOps araçları ile entegrasyon sağlanmalıdır.

Mikroservis Mimarisi tabanı doğru kurulduğunda bakım ve zaman azalmaktadır. Mikroservis Mimarisini kurmak kapsamlı bir süreç gerektirmektedir. Netflix Mikroservis Mimarisine 6 senelik bir çalışmak sonucunda geçmiştir.

Mikroservislerin DevOps ile iç içe olduğu unutulmamalıdır. DevOps araçlarının kullanılması ve entegrasyonu öğrenilmesidir.

6. KAYNAKLAR

[1] https://medium.com/architectural-patterns/microservice-nedir-73bdfddad197
[2] https://www.gokhan-gokalp.com/monolithic-ve-microservice-architecturea-genel-bir-bakis/
[3] https://gokhansengun.com/docker-nedir-nasil-calisir-nerede-kullanilir/
[4] https://medium.com/@selcukusta/consul-i%CC%87le-service-discovery-d%C3%BCnyas%C4%B1na-bak%C4%B1%C5%9F-60d81c06a45d
[5] https://github.com/vmudigal/microservices-sample
[6] https://www.docker.com/resources/what-container