**T.C.**

**SOSYAL GÜVENLİK KURUMU BAŞKANLIĞI**

**MİKROSERVİS MİMARİSİNİN İNCELENMESİ: BİR SOSYAL GÜVENLİK KURUMU YAZILIM UYGULAMASI ÖRNEĞİ**

**Sosyal Güvenlik Uzmanlığı Tezi**

**Hazırlayan**

**Hallaç Mansur AKBAŞ**

**Tez Danışmanı**

**Murat BİNAY**

**Sosyal Güvenlik Uzmanı**

**Ağustos 2024**

**Ankara**

Hallaç Mansur AKBAŞ tarafından hazırlanan bu çalışmanın Sosyal Güvenlik Uzmanlığı Tezi olarak uygun olduğunu onaylarım. …./…./2024

Tez Danışmanının Adı SOYADI: Murat BİNAY

İmzası:

Bu çalışma, Kurulumuz tarafından oy birliği / oy çokluğu ile Sosyal Güvenlik Uzmanlığı Tezi olarak kabul edilmiştir.

|  |  |  |
| --- | --- | --- |
|  | **Adı Soyadı ve Unvanı** | **İmzası** |
| Başkan | : ………………………………………………… | …....………... |
| Üye | : …………………………………………………. | …....………... |
| Üye | : ………………………………………………… | …....………... |
| Üye | : …………………………………………………. | …....………... |
| Üye | : ………………………………………………… | …....………... |

Tarih: …./…./………

# 

# TEŞEKKÜR

Bu çalışma sırasında tecrübe ve bilgisi ile beni yönlendiren tez danışmanım Emre MUTLU’ya, ihtiyaç duyduğum konularda desteklerini esirgemeyen çalışma arkadaşlarım Hasan İbrahim KOÇAK’a ve Ramazan SARIALTIN’a, ayrıca her koşulda yanımda olan anneme, babama ve kardeşlerime teşekkürü borç bilirim.

**Ankara 2024 Hallaç Mansur AKBAŞ**

# BEYAN

Bu tezin tasarımı, hazırlanması, yürütülmesi, araştırmalarının yapılması ve bulgularının analizlerinde akademik kurallara ve bilimsel etik davranış ilkelerine uyduğumu, ayrıca, bu kural ve ilkelerin gereği olarak, çalışmada bana ait olmayan tüm veri, düşünce ve sonuçları andığımı ve kaynağını gösterdiğimi beyan ederim.

…./…./……..

Adı ve SOYADI: Hallaç Mansur AKBAŞ

İmzası:

# ÖZET

**Sosyal Güvenlik Uzmanlığı Tezi**

**MİKROSERVİS MİMARİSİNİN İNCELENMESİ:**

**BİR SOSYAL GÜVENLİK KURUMU YAZILIM UYGULAMASI ÖRNEĞİ**

**Hallaç Mansur AKBAŞ**

İnce istemci, uzak birimlerdeki kullanıcı ihtiyaçlarını zamanla izler ve uzak birime merkezdeki kaynaklardan gerekli ve yeterli miktardakini ayırarak etkin çözümlerle ve daha az kaynak kullanımıyla önemli bir iyileştirme sağlar.

İnce istemci sanallaştırma teknolojisi kullanır. Sanallaştırma; fiziksel olarak merkezde olan büyük bir havuzu (yazılım sayesinde) ihtiyaç olduğu kadar mantıksal birimlere bölerek kaynakları daha verimli hale getirmektir. Böylece eskiden her birimde ayrı ayrı bulunan fiziksel kaynaktaki kullanılmayan kapasite daha etkin bir biçimde kullanılacak ve merkezi yönetimle sistem performansı kat kat artacak ve yönetim maliyetinin de düşük olmasını sağlayacaktır.

SGK, sunucu sanallaştırma altyapısı ile bütün hizmetlerini kamuya sunmaktadır. Sunucu sanallaştırma, merkezdeki etkinliği sağlamakla birlikte mevcut taşra altyapılarında bir sanallaştırma altyapısı olmadığından buralardaki performanslara doğrudan bir etki yapamaz. İşte tam da burada ince istemci mimarisi ve alt yapısının gerekliliği ortaya çıkmaktadır. Bu çalışmada ilk olarak mevcut sanallaştırma altyapısı incelenmiş, topolojisi çıkartılmış ve bileşenleri anlatılmıştır. Ardından ince istemci mimarisinin SGK taşra birimleri kullanıcı masa üstü ortamına uygulanabilirliği test edilmiştir ve olumlu sonuçlar alınmıştır. Ayrıca Bilişim Ağı, Enerji Verimliliği, Yönetim Politikası ve Felaket Kurtarma konularında kurum bilişim teknolojileri altyapısında uygulanabilecek çözümler belirlenmiştir. Tezin amacı İnce İstemci tekonolojileri kullanımı halinde Kurum’da haberleşme yönetim ve yedekleme açısından ortaya çıkan faydaların değerlendirilmesi ve ince istemci ile ilgili bilinç oluşmasıdır.

**Anahtar Kelimeler:** Mikroservis, Yazılım Mimarisi, Monolitik Mimari, Yazılım Tasarımı, Ölçeklenebilirlik, Konteynerizasyon, DevOps

# ABSTRACT

**Social Security Expertise Thesis**

**EXAMINATION OF MICROSERVICE ARCHITECTURE: A SAMPLE OF A SOCIAL SECURITY INSTITUTION SOFTWARE APPLICATION**

**Hallaç Mansur AKBAŞ**

Thin client, the remote unit monitor user needs over time and with the necessary and adequate amount of resources in the next one effective solution by separating the central resources and remote unit it provides a significant improvement with less resource use.

Thin client uses virtualization technology. Virtualization, with a large pool in the center physically (thanks to software) to logically divide up our resources that we need to become more efficient units. Thus enabling each unit formerly unused capacity on the physical resources available separately to be used more effectively and will increase central management with system performance and management many times that of the low.

SGK, offers all the services to the public with the server virtualization infrastructure. Server virtualization; while providing a direct impact on activity in the center performance can’t be a direct impact on province performance as there is not a virtualization infrastructure around the existing provincial infrastructure. It is arises here is exactly the thin client architecture and requirements of the infrastructure. In this study first of all, examined the existing virtualization infrastructure, topology designated and components described. Then SGK provincial units of the thin client architecture have been tested the applicability of the user desktop environment and positive results were. In addition, IT network, energy efficiency, management policies and disaster recovery solutions identified are applicable to corporate information technology infrastructure. The purpose of this thesis is if the use of thin client technologies at the institution, evaluation of the benefits in terms of communication management and backup and aimed to create awareness about thin client.

**Key Words:** Microservice, Software Architecture, Monolithic Architecture, Software Design, Scalability, Containerization, DevOps

# YÖNTEM

Bu çalışmada ulusal ve yabancı kaynaklardan faydalanılarak hazırlanmıştır. Sanallaştırmanın ve sanallaştırma türlerinin detaylı araştırılmasında yurtiçi ve yurtdışındaki makalelerden, analizlerden, dergi ve kitaplardan, kurumların yapmış oldukları çalışmalardan yararlanılmıştır. Ayrıca konu ile ilgili ulusal ve yabancı internet sitelerinin de ağırlıklı olarak kullanımı düşünülmektedir. Tezin yazımının üç aşamadan meydana gelmesi düşünülmektedir.

Birinci bölümde Sosyal Güvenlik Kurumu bilgi işlem ağı ve altyapısı, ince istemcinin temeli olan sanallaştırma kavramanın tarihçesinden, kurum veri merkezlerinden ve hali hazırda yürütüyor olunan projelerden bahsedilecektir. Ayrıca diğer sanallaştırma yöntemleri olan; sunucu sanallaştırma, masaüstü sanallaştırma, uygulama sanallaştırma, veri depolama sanallaştırma, ağ sanallaştırma, veri ve veritabanı sanallaştırma, bellek sanallaştırma gibi sanallaştırma türleri detaylı bir şekilde incelenecektir.

İkinci bölümde ise ince istemcinin ne olduğu, mimarisi, uygulama alanları ve ince istemci ile sanallaştırma modeli uygulanırken, hangi hususlara dikkat edilmesi gerektiği açıklanmıştır. Ayrıca ince istemcinin geleneksel masaüstü bilgisayarlardan mantıksal farkı ve ince istemciye geçiş süreci de yine bu bölümde incelenecektir.

Son bölümde ise ince istemci teknolojisi kullanılarak taşralarda sanallaştırmaya geçiş sonucu elde edilebilecek faydalar ve geçiş süreci anlatılmıştır. Yapılması düşünülen yeni sanallaştırma teknolojisinin Sosyal Güvenlik Kurumu’na olası getirileri riskleri ve götürüleri üzerinde durulmuştur. Ayrıca ince istemci teknolojisi kullanılarak oluşturulan sanallaştırmanın ne gibi güvenlik getirileri olduğu ve enerji verimliliği açısından (otuz bin kullanıcılı üç yüz ayrı ve uzak lokasyonda bulunan bir kurumda) ne gibi getirileri olduğu da bu bölümde anlatılmıştır. Ayrıca pilot bölge seçilen bir taşra biriminde birkaç kullanıcının ince istemci kullanarak günlük işlerini yapması olanağını oluşturup, bu durumun sonuçlarını kurum geneline göre detaylı analiz edilmesi düşünülmekte ve ince istemci teknolojisinin kullanıldığı sanal bir platform kurup bu platform üzerinden günlük işler yapıp buna göre bir inceleme yapılması düşünülmektedir.

# İÇİNDEKİLER

[TEŞEKKÜR i](#_Toc168402755)

[BEYAN ii](#_Toc168402756)

[ÖZET iii](#_Toc168402757)

[ABSTRACT iv](#_Toc168402758)

[YÖNTEM v](#_Toc168402759)

[İÇİNDEKİLER vi](#_Toc168402760)

[ŞEKİLLER LİSTESİ vii](#_Toc168402761)

[TABLOLAR LİSTESİ ix](#_Toc168402762)

[KISALTMALAR x](#_Toc168402763)

[GİRİŞ 1](#_Toc168402764)

[BİRİNCİ BÖLÜM 2](#_Toc168402765)

[YAZILIM MİMARİSİ 2](#_Toc168402766)

[1. TANIM 2](#_Toc168402767)

[2. YAZILIM MİMARİSİNİN YARARLARI 2](#_Toc168402768)

[3. YAZILIM MİMARİSİ ÇEŞİTLERİ 3](#_Toc168402769)

[3.1. Monolitik Mimari 3](#_Toc168402770)

[3.2. Servis Odaklı Mimari 3](#_Toc168402771)

[3.3. Mikroservis Mimarisi 5](#_Toc168402772)

[İKİNCİ BÖLÜM 6](#_Toc168402773)

[MİKROSERVİS MİMARİSİ 6](#_Toc168402774)

[1. MİKROSERVİS MİMARİSİ TARİHÇESİ 6](#_Toc168402775)

[2. MİKROSERVİS MİMARİSİ TANIMI VE ÖZELLİKLERİ 7](#_Toc168402776)

[2.1. Mikroservis Mimarisinin Avantajları ve Dezavantajları 8](#_Toc168402777)

[2.2. Mikroservis Mimarisinin Monolitik Mimari ile Karşılaştırılması 9](#_Toc168402778)

[2.3. Mikroservis Mimarisinin SOA ile Karşılaştırılması 9](#_Toc168402779)

[3. MİKROSERVİS MİMARİSİYLE İLİŞKİLİ KAVRAMLAR 10](#_Toc168402780)

[3.1. Alan Odaklı Tasarım (DDD) 10](#_Toc168402781)

[3.2. Bulut Bilişim (Cloud Computing) 11](#_Toc168402782)

[3.3. Docker 13](#_Toc168402783)

[3.3.1. Docker’ın Temel Bileşenleri 13](#_Toc168402784)

[3.3.2. Sanal Makine ve Docker Karşılaştırması 15](#_Toc168402785)

[3.3.2.1. Mimari 15](#_Toc168402786)

[3.3.2.2. Kaynak Yönetimi 16](#_Toc168402787)

[3.3.2.3. Performans 16](#_Toc168402788)

[3.3.2.4. Güvenlik 16](#_Toc168402789)

[3.4. Kubernetes 16](#_Toc168402790)

[3.4.1. Kubernetes’in Temel Bileşenleri 18](#_Toc168402791)

[3.4.1.1.Master Node 19](#_Toc168402792)

[3.4.1.2.Worker Node 20](#_Toc168402793)

[ÜÇÜNCÜ BÖLÜM 21](#_Toc168402794)

[METODOLOJİ 21](#_Toc168402795)

[1. NOTLAR 21](#_Toc168402796)

[1.1. Not 21](#_Toc168402797)

[1.2. Bbbbbbbbb bbbbbb 21](#_Toc168402798)

[1.2.1. Ccccccc cccccc cccccccc ccccccccccccccccccc 21](#_Toc168402799)

[1.2.2. Ddddddddddd 21](#_Toc168402800)

[1.2.2.1. Ddddddddddd 21](#_Toc168402801)

[ÜÇÜNCÜ BÖLÜM 22](#_Toc168402802)

[SONUÇLAR VE DEĞERLENDİRME 22](#_Toc168402803)

[1. KURUM BİLGİ İŞLEM MİMARİSİ 22](#_Toc168402804)

[1.1. Aaaaaa aaaaaaaaa aaaaaaaaa 22](#_Toc168402805)

[1.2. Bbbbbbbbb bbbbbb 22](#_Toc168402806)

[1.2.1. Ccccccc cccccc cccccccc ccccccccccccccccccc 22](#_Toc168402807)

[1.2.2. Ddddddddddd 22](#_Toc168402808)

[1.2.2.1. Ddddddddddd 22](#_Toc168402809)

[DÖRDÜNCÜ BÖLÜM 23](#_Toc168402810)

[UYGULAMA ÖRNEĞİ 23](#_Toc168402811)

[1. KURUM BİLGİ İŞLEM MİMARİSİ 23](#_Toc168402812)

[1.1. Aaaaaa aaaaaaaaa aaaaaaaaa 23](#_Toc168402813)

[1.2. Bbbbbbbbb bbbbbb 23](#_Toc168402814)

[1.2.1. Ccccccc cccccc cccccccc ccccccccccccccccccc 23](#_Toc168402815)

[1.2.2. Ddddddddddd 23](#_Toc168402816)

[1.2.2.1. Ddddddddddd 23](#_Toc168402817)

[SONUÇ ve ÖNERİLER 24](#_Toc168402818)

[KAYNAKÇA 25](#_Toc168402819)

# ŞEKİLLER LİSTESİ

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

**Şekil 2: Mikroservis Mimarisinin Gelişimini Etkileyen Dönüm Noktaları.........................6**

**Şekil 3: Bulut Bilişim Hizmet Modelleri...............................................................................11**

**Şekil 4: Docker Bileşenleri....................................................................................................14**

**Şekil 5: Docker ve Sanal Makine Mimarileri.......................................................................15**

**Şekil 6: Kubernetes Bileşenleri.............................................................................................18**

**Şekil 7: Strangler Yaklaşımı.................................................................................................29**

**Şekil 8:** **Talep Bileşeninin Alanları.......................................................................................38**

**Şekil 9: Sistem İşleyişi............................................................................................................40**

**Şekil 10: Yazılım Mimarilerinin Evrimi................................................................................5**

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

**Şekil 1: Yazılım Mimarilerinin Evrimi..................................................................................5**

# TABLOLAR LİSTESİ

# KISALTMALAR

4/1-(b) 5510 sayılı Kanunun 4 üncü maddesinin 1 inci fıkrasının (b) bendi

SOA Service-Oriented Architecture (Servis-Odaklı Mimari)

REST Representational State Transfer (Temsili Durum Transferi)

SGK Sosyal Güvenlik Kurumu

HTTP Hypertext Transfer Protocol (Hiper Metin Transfer Protokolü)

API Application Programming Interface (Uygulama Programlama Arayüzü)

DevOps Development and Operations (Geliştirme ve Operasyonlar)

DDD Domain Driven Design (Alan Odaklı Tasarım)

IT Information Technologies (Bilgi Teknolojileri)

IaaS Infrastructure as a Service (Hizmet olarak Altyapı)

PaaS Platform as a Service (Hizmet olarak Platform)

SaaS Software as a Service (Hizmet olarak Yazılım)

CD Continuous Delivery (Sürekli Teslimat)

CI Continuous Integration (Sürekli Entegrasyon)

CLI Command Language Interface (Komut Satırı Arayüzü)

CNCF Cloud Native Computing Foundation

SSL Secure Sockets Layer (Güvenli Soket Katmanı)

ESB Enterprise Service Bus (Kurumsal Hizmet Veri Yolu)

DAO Data Access Object (Veri Erişim Nesnesi)

JSP JavaServer Pages (JavaServer Sayfaları)

HTML Hypertext Markup Language (Hiper Metin İşaretleme Dili)

XML Extensible Markup Language (Genişletilebilir İşaretleme Dili)

IBM International Business Machines

ORM Object Relational Mapping (Nesne ile İlişkisel Eşleme)

SQL Structured Query Language (Yapılandırılmış Sorgulama Dili)

# GİRİŞ

Günümüzün hızla gelişen ve değişen teknoloji ortamında yazılım geliştirme süreç ve yöntemleri ihtiyaçlara cevap verebilmek için sürekli bir dönüşüm içindedir. Hız, performans ve depolama kapasiteleri artan bilgisayar sistemleri daha karmaşık ve güçlü yazılımların geliştirilmesine olanak sağlamıştır. Bunun sonucunda üretilen büyük ölçekli yazılımları geliştirmek ve yönetmek için yeni mimari yaklaşımlar ortaya çıkmıştır.

Klasik yazılım geliştirme mimarisinde tek bir kod tabanı üzerinde geliştirme yapılır ve geliştirilen yazılım bir bütün halinde kullanıma alınır. Özellikle büyük ölçekli yazılımların geliştirilmesi ve yönetilmesi konusunda bu yaklaşım bazı sınırlamaları ve zorlukları beraberinde getirir. Bu mimarinin getirdiği zorlukları aşmak ve daha esnek, ölçeklenebilir ve dağıtılabilir yazılım çözümleri sunabilmek için farklı bir mimari model olan mikroservis mimarisi ortaya çıkmıştır.

Mikroservis mimarisinin kullanımı Netflix, Amazon, Spotify gibi dünyadaki önemli kurumsal şirketlerin bu mimariyi kullanmada öncülük etmesiyle yazılım dünyasında yaygınlaşmıştır (Cebeci & Korçak, 2020). Bu mimari büyük monolitik uygulamaların yerine küçük, bağımsız ve birbiri ile entegre çalışabilen servislerden oluşan bir uygulama yapısı önermektedir. Her bir mikroservis, belirli bir işlevi gerçekleştirir ve bu servisler bir araya gelerek büyük uygulamaları oluşturur.

SGK bünyesinde hizmet veren yazılımlar, milyonlarca insanın hayatını doğrudan etkileyen kritik öneme sahip yazılımlardır. Bu sistemlerin doğru bir şekilde tasarlanması ve geliştirilmesi hem kurumun işleyişi hem de hizmet verilen vatandaşlar için büyük önem taşımaktadır. Gerek büyük uygulama yazılımlarının karmaşıklığının üstesinden gelmek gerekse yazılım ekiplerinin rahat ve uyumlu bir şekilde çalışabilmesini sağlamak için sunulan bir çözüm olan mikroservis mimarisinin kurum yapısına uygunluğu ve kullanılabilirliği değerlendirilecektir. Avantajları ve dezavantajları göz önüne alınarak ne kadar optimize bir çözüm olduğu hem teknik açıdan hem kültürel açıdan ele alınacaktır.

# BİRİNCİ BÖLÜM

# YAZILIM MİMARİSİ

## TANIM

Yazılım mimarisi, bir yazılım sisteminin oluşturulması için gerekli olan yapıların ve bu yapıları oluşturmak için izlenen disiplinin bütünüdür. Yazılım mimarisi, yazılım bileşenlerini, bu bileşenler arasındaki ilişkileri ve hem bileşenlerin hem de ilişkilerinin özelliklerini içerir (Bass, et al., 2012, p. 3).

Yazılımlar belirli bir işi yerine getirmek için yapılır. Yazılım planının soyut bir hedef halinden bir işleme çözüm getiren somut bir hale gelmesi kompleks bir süreçtir. Yazılım mimarisi, süreçleri yönetebilmek ve iş hedeflerini gerçekleştirebilmek için teorik bir çerçeve sağlar (Bass, et al., 2012, p. 2). Bu çerçeve, iş gereksinimlerini, kullanıcı ihtiyaçlarını ve teknik kısıtlamaları dikkate alarak somut ürünün meydana gelmesinde kritik bir rol oynar.

## YAZILIM MİMARİSİNİN YARARLARI

Yazılım mimarisinin sağladıkları yararlar aşağıdaki gibidir:

* Yazılım sisteminin temel yapısını ve bileşenlerinin birbiriyle nasıl etkileşime girdiğini tanımlar. Tüm paydaşlar için daha anlaşılabilir hale getirir (Bass, et al., 2012, p. 2).
* Süreç yönetimi ve maliyet tahmini yapılmasını sağlar (Perry & Wolf, 1992).
* Paydaşlar arasındaki iletişimi güçlendirir (Bass, et al., 2012, p. 21).
* Yapılan yazılım mimarisinin parçaları farklı sistemlerde yeniden kullanılabilir. Bu geliştirme süresini ve maliyetini kısaltmaya yardımcı olur (Bass, et al., 2012, p. 21).
* Yazılım gelecekteki muhtemel değişikliklere uyum sağlayabilecek bir yapıda geliştirilmesini sağlar.
* Yazılımın esnekliğini ve ölçeklenebilirliğini artırır. Böylece yazılımın büyütülebilmesine ve değiştirilebilmesine olanak sağlar.

## YAZILIM MİMARİSİ ÇEŞİTLERİ

Yazılım mimarisinin evrimi, değişen teknolojik gereksinimler ve iş zorluklarına cevap olarak gelişmiştir. Yazılım mimarisi literatüründe adı geçen üç mimari stili incelenecektir.

### Monolitik Mimari

Monolitik mimari, yazılım geliştirme dünyasında uzun süredir kullanılan geleneksel bir mimari yaklaşımdır. Bu yaklaşım, yazılımın tüm fonksiyonlarının tek bir birim olarak geliştirilmesini ve dağıtılmasını ifade eder (Chen, et al., 2017, pp. 466-475). Bu yaklaşımda kullanıcı arayüzü, iş mantığı, veri erişim katmanı ve uygulama entegrasyonu işlevlerinin tümü tek parça halindedir. Tek bir kod tabanı üzerine kurulu bu sistemler, başlangıçta geliştirilmesi ve anlaşılması kolay sistemlerdir. Çünkü tüm bileşenler tek bir programlama dilinde ve çalışma zamanı ortamında geliştirilir.

Zamanla yazılımın ihtiyaçları karşılaması için yapılan değişiklikler ve eklemelerle yazılımın boyutu ve kod karmaşıklığı artar. Bu durum yazılımın ölçeklendirme ve sürekli entegrasyon gibi ihtiyaçları karşılayamamasına neden olur. Kodda yapılan herhangi bir değişiklik bütün yapının yeniden konuşlandırılmasını(deploy) gerektirir. Ayrıca, yazılımın herhangi bir yerindeki hata tüm yazılımın çalışmamasına neden olabilir. Bu durum hata izolasyonunun ve sistem esnekliğinin olmaması demektir.

Bu mimari yaklaşımın özellikle daha küçük yazılımlar için veya bir yazılımın boyutu ve kapsamı nispeten sabit ve yönetilebilir olduğunda çeşitli avantajları da vardır. Tüm bileşenler aynı uygulamada olduğundan uygulamanın farklı bölümleri arasındaki iletişim hızlıdır. Hata ayıklama ve test etme işlemleri basittir. Harici hizmetlere herhangi bir bağımlılık olmaksızın yönetilecek tek bir uygulama olduğundan konuşlandırma(deploy) kolaydır.

### Servis Odaklı Mimari

Servis-odaklı mimari (SOA), sistem bileşenlerinin bağımsız hizmetler olduğu dağıtılmış sistemler geliştirmenin bir yoludur (Sommerville, 2010). Bileşenler birbirleriyle ağ üzerinden bir iletişim protokolü ile haberleşir. SOA’da temel amaç işlevleri farklı servislere ayırmaktır.

SOA kullanımının yazılım sistemlerine esnek, verimli ve güvenli bir şekilde çalışabilmesi için getirdiği bazı avantajlar vardır. Bunlar:

* SOA’da her servis spesifik bir işlevi yerine getirmek üzere oluşturulur. Bu servisler gerektiğinde yeniden kullanılabilir. Bu da iş ve zamandan tasarruf edilmesini sağlar.
* Servisler işlevlerini yerine getirirken iç uygulamanın mantığını yani nasıl yaptıklarını kullanıcıdan gizlerler. Kullanıcıya basit bir arayüz sunarak karmaşıklığı ortadan kaldırır.
* Servisler birbirinden bağımsız olarak geliştirilebilir ve konuşlandırılabilir.
* Değişen iş gereksinimlerini karşılamak için gelişmiş esneklik sağlar. Yeni servisler eklenebilir, var olan servisler güncellenebilir ve iş süreçleri yeniden yapılandırılabilir.
* Farklı sistemlerin birlikte çalışabilmesi ve veri paylaşması kolaydır.
* Modüler bir yapıya sahiptir ve sistem bileşenleri izole bir şekilde çalıştığından güncellenmesi durumunda birbirinden etkilenmez.
* Ölçeklenebilirliği sayesinde artan kullanıcı sayısı ve/veya iş yükü durumunda sistem performansını korumak için sistem büyütülebilir.

Bunun yanı sıra SOA bazı dezavantajları da beraberinde getirir. Bunlar:

* Ağ performansı ve servisler arası iletişim gecikmeleri sistemin çalışma performansını yavaşlatabilir.
* Sistem birçok servis içermesi durumunda bu servisler arasındaki iletişimi yönetmek ve izlemek zor olabilir. Bu karmaşıklık hataları tespit etmeyi ve çözmeyi zorlaştırır (Jayasooriya, 2024).
* Birçok farklı servis bağlantı noktası içerdiğinden güvenlik riskleri doğar. Servislerin arasındaki iletişimin güvenliğini sağlamak ek güvenlik önlemleri gerektirir.

### Mikroservis Mimarisi

Tezin ana konusu olduğu için bir sonraki ana başlık altında incelenmiştir.

**Şekil 1**: Yazılım Mimarilerinin Evrimi



**Kaynak**: (Ilyukha, 2024)

# İKİNCİ BÖLÜM

# MİKROSERVİS MİMARİSİ

## MİKROSERVİS MİMARİSİ TARİHÇESİ

Mikroservis terimi yazılım dünyasına çeşitli aşamalardan geçerek gelmiştir. Dr. Peter Rodgers, 2005 yılında “Micro-Web-Services” terimini ilk kullanan kişi olarak kabul edilmektedir. “Mikroservis” terimi ise 2014 yılında Martin Fowler ve ThoughtWorks’teki meslektaşlarının bu kavramı sağlam ve ölçeklenebilir sistemler oluşturmanın bir yolu olarak tartışmasıyla önem kazanmıştır (Sanchez, 2024).

**Şekil 2**: Mikroservis Mimarisinin Gelişimini Etkileyen Dönüm Noktaları



**Kaynak:** (Sanchez, 2024)

Mikroservis mimarisi fikrinin temelleri 2000’li yılların başlarına kadar gider. Monolitik mimarilerin uygulamalar büyüdükçe karmaşıklığının artması uygulamaların birbirinden ayrı olarak geliştirilip bir protokol üzerinden haberleşmesi fikrinin yani SOA fikrinin doğmasına yol açmıştır.

2010’larda Amazon ve Netflix gibi dünya çapındaki şirketlerin çevikliği ve ölçeklenebilirliği artırmak için büyük monolitik uygulamalarını daha küçük ve bağımsız olarak dağıtılabilen servislere ayırması mikroservis mimarisinin popülaritesini artırdı.

2013’te Docker ve 2014’te Kubernetes teknolojilerinin çıkması mikroservis mimarisinin kullanımına kolaylıklar getirdi. Docker her bir mikroservisin hafif konteynerler olarak paketlenmesine olanak tanırken Kubernetes daha kolay bir biçimde mikroservislerin dağıtılmasını ve yönetilmesini sağladı. Bu araçların geliştirilmesi ve kullanımının yaygınlaşması mikroservis mimarisinin yazılım dünyasında yerini daha da sağlamlaştırmasına neden oldu.

Sonraki yıllardan günümüze kadar gelen süreçte mikroservis mimarisinin kullanımı artmaya ve büyük ölçekli web uygulamaları için standart haline gelmeye devam etmektedir. DevOps uygulamalarının ve sürekli teslimatın gelişimi mikroservis mimarisinin yazılım dünyasındaki yerini desteklemektedir.

## MİKROSERVİS MİMARİSİ TANIMI VE ÖZELLİKLERİ

Mikroservis mimarisi, her biri kendi sürecinde çalışan ve hafif mekanizmalarla haberleşme sağlayan küçük servislerden oluşan tek bir uygulamayı geliştirmeye yönelik bir yaklaşımdır (Fowler & Lewis, 2014). Başka bir şekilde tanımlanırsa, tek bir uygulamayı küçük, özerk servislerden oluşan bir koleksiyona ayıran merkezi olmayan bir yazılım mimarisidir.

Mimariyi oluşturan her bir servis tek sorumluluk prensibine uyarak kendi için tanımlanmış olan göreve odaklanır. Servisler kendi işine odaklanan parçalara bölündüğü için küçüktür. Birbirleriyle haberleşirken HTTP gibi teknolojiden bağımsız protokoller kullanırlar.

Mikroservis mimarisinin kendine özgü nitelikleri aşağıdaki şekilde sıralanabilir:

* Servisler sadece tek bir iş yapmak üzerine tasarlandığı için küçüktür.
* Her bir servis büyük ölçüde özerktir. Başka bir ifade ile servisler birbirlerine gevşek bağlıdır ve birbirlerinin süreçlerine müdahale edemezler. İzole bir şekilde çalışırlar.
* Mikroservisler ihtiyaca göre birbirlerinden bağımsız bir şekilde ölçeklendirilebilir.
* Teknoloji bağımsızdır. Her bir servis farklı programlama dilini, farklı bir çerçeveyi(framework) ve veritabanını kullanabilir.
* Mikroservis mimarisi karmaşıktır ve yönetilmesi zor bir yapıya sahiptir.

### Mikroservis Mimarisinin Avantajları ve Dezavantajları

Mikroservis mimarisi monolitik mimaride ortaya çıkan zorlukların üstesinden gelmek için oluşturulan bir çözümdür. Artı yönleri aşağıda sıralanmıştır:

* Mikroservis mimarisi, her servis için yerine getireceği göreve uygun programlama dilini ve araçlarını seçme esnekliği sağlar ve yenilikçiliği teşvik eder.
* Mikroservislerden herhangi biri arızalanırsa diğer servisler çalışmaya devam ederek sistemin genel hata toleransını artırır.
* Uygulamanın tamamını ölçeklendirmek yerine ihtiyaca göre sadece belirli bir servisi ölçeklendirme olanağı sunar.
* Mikroservis mimarisi ile tek bir serviste değişiklik yapılabilir ve sitemin geri kalanından bağımsız olarak dağıtılabilir. Hızlı dağıtıma olanak sağlar (Newman, 2015).
* Herhangi bir kod tabanında çalışan kişi sayısını en aza indirmek ekip büyüklüğü ve üretkenlik gibi önemli parametreler için uygun seviyeyi yakalamayı kolaylaştırır (Newman, 2015).

Bunların yanında mikroservis mimarisi kendine özgü eksi yönleri de içinde barındırır. Bunların aşağıdaki gibidir:

* Birden fazla servisi yönetmek ve düzenlemek karmaşıktır. Ek araç kullanımı gerektirir.
* Mikroservisler ağ üzerinden iletişim kurar bu sistemin çalışmasında yavaşlamaya neden olabilir (Fowler & Lewis, 2014).
* Veri tutarlılığını sağlamak zordur. Potansiyel veri bütünlüğü sorunlarına yol açar (Kızılpınar, 2021).
* Mikroservisler doğası gereği dağıtık yapıda oldukları için hata ayıklama ve test işlemleri zordur.
* Mikroservisler daha fazla potansiyel giriş noktasına sahip olduğu için saldırıya daha açıktır.

### Mikroservis Mimarisinin Monolitik Mimari ile Karşılaştırılması

Mikroservis mimarisi geleneksel yazılım geliştirme yaklaşımı olan monolitik mimarinin zayıflıklarını gidermek için oluşturulmuştur. İki yaklaşımın farkları karşılaştırması şu şekildedir:

* Mikroservis mimarisinde ölçeklendirme daha kolaydır. Uygulamanın sadece belli bir bölümü ölçeklendirilebilir. Monolitik mimaride ise uygulama bir bütün halinde ölçeklendirilmek zorundadır. Bu da fazladan maliyet getirir (Ayrancıoğlu, 2019).
* Mikroservis mimarisinde yapılan bir değişiklik sadece belli bir servisi etkilerken monolitik mimaride tüm uygulama yeniden konuşlandırma yapılması gerekliliği doğar. Güncelleme monolitik mimaride daha zordur.
* Mikroservis mimarisinde yeni bir teknolojiye geçmek ve kullanmak kolayken monolitik mimaride tüm uygulamanın geçirilmesi gerektiğinden daha zordur.
* Mikroservis mimarisinde hata ayıklama ve test etme işlemleri ek araçlar gerektirir ve zahmetlidir. Fakat monolitik mimaride daha basittir.
* Mikroservis mimarisinde servisler arası haberleşme ağ üzerinden yapılır ve bu ek yük getirir. Monolitik mimaride böyle bir sorun yoktur.

### Mikroservis Mimarisinin SOA ile Karşılaştırılması

SOA ve mikroservis mimarileri, yazılım geliştirme sürecine çeviklik ve esneklik sağlamayı amaçlayan mimari desenlerdir ancak yaklaşımları bakımından belli farklılıkları vardır. Bunlar aşağıdaki gibidir:

* Mikroservisler, tipik olarak daha büyük SOA servislerine göre daha spesifik olacak şekilde tasarlanmıştır. Yani tek bir işleve veya birbiriyle yakından ilişkili küçük bir işlevsellik kümesine odaklanır.
* Mikroservisler sıklıkla kendi veritabanına sahiptir, SOA servisleri veritabanlarını paylaşır.
* Mikroservisler haberleşme için REST veya mesajlaşma kuyrukları gibi daha hafif protokoller kullanırken SOA, ESB aracılığıyla haberleşir.
* Mikroservisler bağımsız bir şekilde konuşlandırılabilirken SOA servisleri genellikle eşzamanlı bir şekilde konuşlandırılır.
* Mikroservisler dağıtık yapısı nedeniyle doğası gereği ölçeklenebilirler.

Projenin boyutu ve karmaşıklığı, beklenen ölçeklenebilirlik kapasitesi, organizasyon becerileri ve mevcut altyapısı göz önünde bulundurarak mimari ele alınmalıdır.

## MİKROSERVİS MİMARİSİYLE İLİŞKİLİ KAVRAMLAR

Mikroservis mimarisi diğer yazılım metotları ve teknolojileri ile birlikte çözüm getirir. Alt başlıklarda mikroservis mimarisinin tasarımında ve çalıştırılmasında ilişkili olduğu kavramlar ve teknolojilerden bahsedilecektir.

### Alan Odaklı Tasarım (DDD)

DDD, etki alanının uzmanlarından gelen girdilere göre bir etki alanıyla eşleşecek yazılımın modellenmesine odaklanan önemli bir yazılım tasarımı yaklaşımıdır (Anon., 2024). DDD’de ilk amaç yazılım paydaşlarının ortak bir iletişim dili oluşturmasıdır. İkinci amacı ise ölçeklenebilir ve anlaşılabilir bir yazılım mimarisi oluşturmaktır.

İlk kez Eric Evans’ın kitabında sözü geçen terim yazılım sisteminin temel alanının anlaşılmasını ve modellenmesini vurgular. Oluşturulacak yazılıma ilişkin ortak bir anlayış oluşturmak için yazılıma katkıda bulunacak paydaşların arasındaki iş birliğini teşvik eder. DDD etki alanına odaklanarak geliştiricilerin gerçek dünyadaki iş gereksinimleriyle yakından uyumlu yazılım çözümleri oluşturmasına olanak tanır.

DDD, herkesin etki alanı kavramlarını anlamasını sağlamak için yazılım geliştiricileri ve iş dünyası arasında ortak bir kelime dağarcığı olan her yerde bulunan dil(Ubiquitous Language) oluşturulmasını savunur. Etki alanını kendi içinde ilişkili parçalara böler.

DDD, sınırları belirlemek ve sistemi bölmek için uygun bir çözüm sağlar (Pachghare, 2016). DDD, etki alanlarını net bir şekilde arayüzlerle birbirinden ayırdığı için yazılımların dağıtık bir şekilde geliştirilmesine olanak tanır. Mikroservis mimarisi dağıtık bir mimari olduğu için DDD ile tasarlanan yazılımlar her bir etki alanı bir mikroservis olacak şekilde ayrı şekilde konuşlandırılmaya elverişlidir.

### DevOps

### Bulut Bilişim (Cloud Computing)

Bulut bilişim, minimum yönetim çabası veya hizmet sağlayıcı etkileşimi ile hızlı bir şekilde sağlanabilen ve serbest bırakılabilen yapılandırılabilir bilgi işlem kaynaklarının paylaşılan bir havuzuna her yerde, uygun, isteğe bağlı ağ erişimi sağlamaya yönelik bir modeldir (Mell & Grance, 2011). Bulut bilişim ile birlikte bilgi teknolojileri kaynaklarına erişim, kullanım ve yönetim biçimleri değişmiştir. Çeşitli hizmetlerin internet üzerinden sunulmasını sağlamaktadır. Böylece bu hizmetten yararlananlar fiziksel sunuculara ve diğer altyapılara sahip olma ve bunların bakımını yapma masraflarından ve karmaşıklığından kurtulurlar.

Bulut bilişim her biri farklı ihtiyaçlara ve senaryolara hitap edecek şekilde üç hizmet modeli ortaya çıkmıştır: Bunlar; Hizmet Olarak Altyapı (IaaS), Hizmet Olarak Platform (PaaS) ve Hizmet Olarak Yazılım (SaaS). Her model farklı bir soyutlama seviyesini temsil eder ve çeşitli bilgi işlem senaryolarında farklı kullanıcı ihtiyaçlarını karşılar.

**Şekil 3**: Bulut Bilişim Hizmet Modelleri



**Kaynak:** (Red Hat, 2022)

IaaS, bulut hizmetinin en temel seviyesidir. Kullanıcılara sunucu, ağ, sanallaştırma ve depolama gibi temel bilgi işlem altyapısına erişim sunarak kurumların pahalı donanımlara yatırım yapmak yerine kaynakları talep üzerine ve ihtiyaç duyulduğunda satın almasına olanak tanır. Kurumların talebe göre kaynakları hızlı bir şekilde ölçeklendirmelerine veya azaltmalarına olanak tanır. IaaS aynı zamanda fiziksel sunucuları ve diğer veri merkezi bileşenlerini yönetmenin karmaşıklığından ve maliyetinden kaçınmak isteyen kurumlar için de çok önemli bir teknolojidir.

PaaS kurumların süreçle ilişkili altyapıyı oluşturma ve sürdürme karmaşıklığı olmadan uygulamaları geliştirmelerine, çalıştırmalarına ve yönetmelerine olanak tanıyan bir platform sağlayan bulut hizmeti modelidir. PaaS, işletim sistemleri, geliştirme araçları, veritabanı yönetim sistemleri ve daha fazlasını içerir ve web uygulaması yaşam döngüsünün tamamını (oluşturma, test etme, dağıtma, yönetme ve güncelleme) destekler. PaaS, temel altyapıyı yönetmek zorunda kalmadan uygulama geliştirmenin yaratıcı yönüne odaklanmak isteyen geliştiriciler için özellikle faydalıdır.

SaaS, bulut bilişimin en yaygın olarak bilinen şeklidir. SaaS, yazılım uygulamalarını abonelik temelinde İnternet üzerinden sunar. SaaS sağlayıcıları altyapıyı, platformları ve hatta verileri yönetir; bu da kullanıcıların uygulamaları tek tek bilgisayarlara yüklemelerine veya çalıştırmalarına gerek olmadığı anlamına gelir. Bu sadece yazılım edinme masraflarını azaltmakla kalmaz, aynı zamanda bakım ve desteği de basitleştirir. Yaygın örnekleri arasında e-posta, müşteri ilişkileri yönetimi sistemleri ve iş birliğine dayalı araçlar yer almaktadır. Bu model, hızlı kurulum ve dağıtım, maliyet etkinliği ve ölçeklenebilirlik açısından avantajlıdır ve İnternet bağlantısı ve tarayıcısı olan herhangi bir cihazdan erişilebilen çözümler sunar.

Bu hizmet modelleri, altyapı tedarikinden uygulama geliştirme ve yazılım sunumuna kadar farklı ihtiyaçlara hitap eden bulut bilişim hizmetleri yelpazesini temsil etmektedir. Bu modeller, bulut bilişimin azaltılmış sermaye giderleri, daha düşük operasyonel maliyetler, gelişmiş ölçeklenebilirlik, gelişmiş erişilebilirlik ve daha iyi yönetilebilirlik gibi faydalarını kapsamaktadır. Bu modelleri anlamak, operasyonlarını ve stratejik yeteneklerini geliştirmek için bulut teknolojilerinden yararlanmak isteyen kuruluşlar için çok önemlidir.

Bulut ortamları, ölçeklenebilirlikleri, esneklikleri ve dağıtılmış yapıları ile mikroservis mimarisi için ideal bir altyapı sağlar. Yükü dengelemek ve her bir mikroservisin optimum performansını sağlamak için çok önemli olan dinamik kaynak tahsisini ve yönetimini desteklerler. Mikroservis mimarisi ve bulut bilişim teknolojisi birbirini tamamlayan bir yapıya sahiptir.

### Docker

Yazılım konteynerizasyonunda devrim yaratan açık kaynaklı bir platform olan Docker, konteynerleri kullanarak uygulamaların oluşturulmasını, dağıtılmasını ve çalıştırılmasını basitleştirir. Konteynerler, bir geliştiricinin bir uygulamayı kütüphaneler ve diğer bağımlılıklar gibi ihtiyaç duyduğu tüm parçalarla birlikte paketlemesine ve hepsini tek bir paket olarak göndermesine olanak tanır. Docker bunu yaparak öngörülebilirliği ve verimliliği artırır. Uygulamaların ayrılmış ve kontrollü bir ortamda çalışmasını sağlar.

Docker hem geliştirme hem de dağıtım iş akışlarını verimli, ölçeklenebilir ve güvenli bir şekilde yapmak için hafif konteynerleştirme teknolojisini kullanır. Teknoloji, Docker konteynerleri içinde çalışan uygulamaların birbirinden ayrı ve izole olmasını sağlamak için birden fazla güvenlik katmanını destekler, bu da kötü niyetli saldırıların veya sistem arızalarının bitişik konteynerleri etkileme riskini en aza indirir.

Sonuç olarak Docker, farklı ortamlarda çok çeşitli uygulamaları yönetmek için ölçeklenebilir, güvenli ve verimli bir platform sağlayarak yazılım geliştirme ve dağıtım alanında önemli bir ilerlemeyi temsil etmektedir. Docker, CI/CD iş akışları için ideal bir çözümdür; çünkü geliştirme, test ve üretim ortamları arasında tutarlılık sağlar. Ayrıca uygulamaları dağıtık olarak yönetmeye elverişli olduğundan mikroservis mimarisiyle uyumludur. Bununla birlikte, Docker'ın getirdiği yenilikler ve esneklikler, büyük ölçekli sistemlerde kaynak kullanımını optimize etmek ve yönetmek için de önemli fırsatlar sunmaktadır.

### Docker’ın Temel Bileşenleri

Docker'ın mimarisi, dağıtılmış uygulamaları çalıştırmak için hafif, taşınabilir ve verimli bir araç sağlamak üzere birlikte çalışan birkaç temel bileşenden oluşur. Docker'ın temel bileşenleri aşağıda incelenecektir ve bu bileşenlerin Docker ekosistemi içindeki işlevleri ve rolleri açıklanacaktır.

**Docker Engine**: Docker'ın işlevselliğinin merkezinde, Docker konteynerlerinin oluşturulmasını ve güvenliğini sağlayan hafif bir çalışma zamanı ve paketleme aracı olan Docker Engine yer almaktadır. Bu teknoloji istemci-sunucu tabanlıdır. Üç ana bileşeni vardır: Konteynerleri yöneten dockerd, dockerd ile iletişim sağlayan REST API’lar ve komut satırı.

**Docker Image**: Konteyner oluşturmak için kullanılan salt okunur bir şablondur (docker.docs, 2024). Kaynak kodlarını, kütüphaneleri ve bağımlılıkları içerir. Docker Hub veya özel kayıt defterleri(registry) üzerinden indirilebilir. Oluşturulan Docker Image’lar Docker Hub sitesine yüklenebilir.

**Docker Container**: Konteyner, bir Docker Image’ının çalıştırılabilir bir örneğidir (docker.docs, 2024). Docker API veya CLI ile konteynerleri başlatmak, durdurmak ve yönetmek mümkündür. Bir Docker Image çalıştırıldıktan sonra belleğe alınır.

**Docker Registry**: Docker Image’larının depolandığı yerdir. Docker Hub herkesin kullanabileceği bir Docker Registry’dir.

**Docker Client**: Kullanıcı ile etkileşimi sağlayan bileşendir. Bir CLI ile bunu gerçekleştirir.

**Şekil 4**: Docker Bileşenleri



**Kaynak:**(Basumallick, 2022)

### Sanal Makine ve Docker Karşılaştırması

Konteynerizasyon teknolojisi Docker ile sanal makine teknolojileri hem taşınabilirlik hem de izolasyon sağlayan sanallaştırma teknolojileridir. Her ikisi de donanım kaynaklarının daha verimli bir biçimde kullanılmasını sağlarlar. Bununla birlikte birbirlerine karşı avantajlara ve dezavantajlara sahiptir.

**Şekil 5**: Docker ve Sanal Makine Mimarileri



**Kaynak:** (Cloud Academy Team, 2023)

### 3.3.2.1. Mimari

Sanal Makineler, tek bir fiziksel makinede birden fazla ve farklı işletim sisteminin çalışmasına izin vererek tam bir donanım sistemini taklit eder. Bir hipervizör tarafından desteklenen bu teknoloji, donanımı soyutlar ve her bir sanal makineye bir dizi özel kaynak ayırarak tam bir sanal işletim sistemi oluşturulmasını sağlar. Sanal makineler, çeşitli bilgi işlem ortamlarında yüksek derecede izolasyon ve uyumluluk sağlama konusunda mükemmeldir, bu da onları tam işletim sistemi kontrolü gerektiren uygulamalar için uygun hale getirir.

Buna karşılık Docker, konteynerleştirme teknolojisini kullanarak uygulama dağıtımında verimliliği ve hızı en üst düzeye çıkarmak için tasarlanmıştır. Sanal makinelerin aksine, Docker konteynerleri tam bir işletim sistemi oluşturmaz; bunun yerine, tek bir paylaşılan işletim sistemi çekirdeği üzerinde çalışan uygulamayı ve bağımlılıklarını içerir. Bu mimari fark, Docker'ın önyükleme süresini ve kaynak ek yükünü önemli ölçüde azaltan, aynı fiziksel sunucuda birden fazla uygulamayı veya mikroservisi yönetmek için yüksek ölçeklenebilirlik ve performansı destekleyen hafif, çevik konteynerleştirme yaklaşımının temelini oluşturur.

### 3.3.2.2. Kaynak Yönetimi

Sanal makinelerde kaynak tahsisi nispeten statiktir, her sanal makine sabit miktarda kaynak ayırır, bu da yetersiz kullanım veya kaynak kıtlığına yol açabilir. Docker ise daha dinamik bir kaynak tahsis modeli benimseyerek konteynerlerin kaynakları talep üzerine kullanmasına olanak tanır. Bu esneklik, değişken iş yüklerine sahip uygulamalar için çok önemlidir, genel sistem verimliliğini artırır ve israfı azaltır.

### 3.3.2.3. Performans

Sanal makineler tam bir işletim sistemi çalıştırdığı için ek yük ve kaynak taleplerine neden olur. Docker ise üzerinde çalıştığı sistem kaynaklarını paylaşarak kullanır. Bu nedenler sanal makineler daha yavaştır. Docker konteynerleri verimlilik için ana bilgisayar çekirdeğinden yararlanarak daha hızlı başlatma süreleri ve daha düşük kaynak tüketimi sağlar.

### 3.3.2.4. Güvenlik

Sanal makineler, uygulamaları donanım düzeyinde izole ederek sağlam bir güvenlik çerçevesi sunar ve bu da onları sistemler arası güvenlik ihlallerine karşı daha güçlü hale getirir. Her sanal makine kendi güvenlik protokolleriyle bağımsız olarak çalışır, bu da onları yüksek güvenlik veya tam izolasyon gerektiren uygulamalar için uygun hale getirir.

Docker'ın güvenliği, konteynerlerin ana işletim sistemi çekirdeğini paylaşması nedeniyle bir şekilde tehlikeye girmektedir. Bir konteynerde bir güvenlik açığından yararlanılırsa, bu diğerlerini de etkileme potansiyeline sahiptir ve güvenliğin çok önemli olduğu ortamlarda önemli bir risk oluşturur. Buna rağmen, Docker bu tür riskleri azaltmak için gelişmiş güvenlik kontrollerini entegre etmiştir, ancak sanal makinelere kıyasla daha az izole kalmaktadır.

### Kubernetes

Yazılım geliştirme alanında, verimli uygulama dağıtımı ve yönetimine duyulan ihtiyaç, konteyner orkestrasyon araçlarının ortaya çıkmasına neden olmuştur. Açık kaynaklı bir sistem olan Kubernetes bu teknolojilerin en önemlilerindendir (Phelan, 2021). Google tarafından tasarlanan ve sonrasında CNCF altında geliştirilmesi sürdürülen Kubernetes, konteynerizasyon araçları tarafından oluşturulan konteynerleri yönetir.

Kubernetes daha öncesinde Google tarafından geliştirilen ve kullanılan, bir cluster yönetim sistemi olan Borg yazılımından esinlenerek geliştirilmiştir (kubernetes, 2015). Kubernetes, Borg’un bazı sınırlamalarının üstesinden gelmeyi ve yeteneklerini daha geniş bir kullanıcı tabanına yaymayı amaçlayan Borg’un açık kaynaklı bir sürümü olarak tasarlanmıştır. Geliştirilmesinde Go programlama dili kullanılmıştır.

Kubernetes’in kullanımı birçok avantajı beraberinde getirir. Kubernetes'teki otomatik ölçeklendirme, değişen iş yüklerini ele almada sistemin verimliliğini ve uyumunu artıran temel bir özelliktir. Bu özellik, Kubernetes'in bir uygulamanın çalışan örneklerinin sayısını mevcut talebe göre otomatik olarak ayarlamasına olanak tanır, böylece optimum kaynak kullanımı sağlar ve manuel müdahale olmadan performansı korur. Otomatik ölçeklendirme yalnızca kaynak verimliliğine katkıda bulunmakla kalmaz, aynı zamanda statik cluster kurulumlarında aşırı provizyon ihtiyacını azaltarak maliyet yönetimine de katkıda bulunur. Kubernetes, karmaşık uygulamaların ihtiyaçlarına gerçek zamanlı olarak uyum sağlayabilen esnek bir altyapıya sahiptir.

Kubernetes herhangi bir nedenle çalışmayan konteynerleri otomatik olarak değiştirir veya yeniden başlatır. Kendi kendini iyileştirme özelliği ve trafiğin yalnızca çalışan konteynerlere yönlendirilmesini sağlayan servis keşfi ve yük dengeleme yetenekleri vardır. Kubernetes çoklu hata toleransı seviyelerini destekler. Bileşenler arızalandığında bile uygulamaların sorunsuz bir şekilde çalışmaya devam etmesini sağlar. Bu, Kubernetes'i kritik uygulamalar için güvenilir bir platform haline getirir.

Kubernetes geri alma özellikleri sayesinde yeni bir dağıtımın sorunlara yol açması durumunda bir uygulamanın önceki sürümlerine kolayca geri dönülmesini sağlar. Bu, yeni özelliklerin ve güncellemelerin durumu bozmadan eklenebileceği ve test edilebileceği istikrarlı bir dağıtım sağlar.

Kubernetes, çeşitli araçlar aracılığıyla sistemin metriklerini izleyebilen ve işleyişi hakkında içgörüler sağlayabilen ayrıntılı izleme yeteneğine sahiptir. Bu veriler performans ayarlama, kapasite planlama ve anomali tespiti için hayati önem taşır ve yöneticilerin ve geliştiricilerin yüksek hizmet kalitesi ve kullanılabilirlik düzeylerini korumalarını sağlar.

Kubernetes yetenekleri ile mikroservis mimarisini destekler. Bir mikroservis ekosisteminde çevik geliştirme ve dağıtım döngüleri için gerekli olan CI/CD uygulamalarını kolaylaştırır. Böylece geliştiricilerin ve kuruluşların bulut teknolojilerin tüm potansiyelinden yararlanmasına da olanak tanır.

### Kubernetes’in Temel Bileşenleri

Kubernetes yapısı master ve worker node olmak üzere iki ana bileşenden oluşur. Bu bileşenlerde kendi içinde bölümlere ayrılmaktadır. Kubernetes’i oluşturan bütün parçalar bu bölüm başlığı altında incelenecektir.

**Şekil 6**: Kubernetes Bileşenleri



**Kaynak:**(Pantic, 2022)

Kubernetes mimarisini oluşturan bileşenleri anlamak için bilinmesi gereken kavramlar vardır. Bunlar aşağıda açıklanmıştır.

**Cluster:** Konteyner uygulamaları çalıştıran node kümesidir. Bir cluster en az bir master node ve birden fazla worker node’dan oluşur.

**Pod:** Bir veyabirden fazla konteyneri içeren en küçük Kubernetes birimidir. Pod'lar, depolama ve ağ gibi kaynakları paylaşan bir veya daha fazla birbiriyle yakından ilişkili konteyneri çalıştırmak için tasarlanmıştır. Her bir Pod'a cluster içinde benzersiz IP adresi tahsis edilir, bu da diğer Pod'larla ve harici servislerle iletişim kurmasını sağlar. Pod’lar geçici Kubernetes varlıklarıdır.

**Service:** Mantıksal bir Pod kümesini ve bunlara erişmek için kuralları tanımlayan bir soyutlamadır. Bu, yük dengeleme gibi ayarları içerebilir. Servisler, uygulamaların trafik almasını sağlar.

**Volume:** Veri tutan Pod'lara bağlı bir depolama birimidir. Bir Pod'un geçici yerel depolamasının aksine, bir Volume kalıcıdır ve Pod yeniden başlatıldığında hayatta kalır. Yapılandırılması Kubernetes API'si içinde yönetilir.

**Namespace:** Birden fazla ekibe veya projeye yayılmış çok sayıda kullanıcının bulunduğu ortamlarda kullanılmak üzere tasarlanmıştır. Cluster kaynaklarını birden fazla kullanıcı arasında bölmenin bir yoludur.

**Deployment:** Pod'lara ve bunların altında yatan ReplicaSet'lere güncellemeler sağlamak için tasarlanmıştır. Bir Deployment, Pod'lar ve ReplicaSet'ler için hangi kapsayıcı imajlarının kullanılacağı, çalıştırılacak replika sayısı ve güncelleme ve geri alma işlemlerinin nasıl yapılacağı gibi güncellemeleri tanımlamanıza olanak tanır.

**Ingress:** Bir cluster servislere, tipik olarak HTTP'ye harici erişimi yöneten bir API nesnesidir. Ingress yük dengeleme, SSL sonlandırma ve ad tabanlı sanal barındırma sağlayabilir. İstekleri, istek ana bilgisayarına veya yoluna dayalı olarak hizmetlere yönlendirmenin ve bu kuralları tek bir kaynakta merkezileştirmenin bir yoludur.

### 3.4.1.1.Master Node

API Server, Etcd, Controller ve Schedular bileşenlerinden oluşur.

**API Server:** Cluster’a yapılan tüm REST isteklerini alan merkezi yönetim bileşenidir. Kullanıcıların, yönetim araçlarının ve diğer bileşenlerin cluster ile etkileşime girmesine olanak tanır.

**Etcd:** Kaynakların yapılandırılmasını, hizmetlerin keşfedilmesini ve cluster gibi dağıtılmış sistemlerin koordinasyonunu kolaylaştıran açık kaynaklı, dağıtılmış bir anahtar/değer deposudur (Armo, 2024).

**Controller**: Birden fazla controller’dan meydana gelen master node bileşenidir. Controller’lar, görevi API server aracılığıyla cluster’ın mevcut durumunu izlemek ve mevcut durumu istenen duruma getirmeye çalışan değişiklikler yapmak olan bileşenlerdir.

**Schedular:** Kaynak kullanılabilirliğine ve diğer zamanlama kısıtlamalarına dayalı olarak Pod’lar biçimindeki işleri çalışan düğümlere atamaktan sorumludur. Mevcut iş yükünü ve kaynak gereksinimlerini göz önünde bulundurarak bir Pod için en uygun düğümü seçer.

### 3.4.1.2.Worker Node

Konteynerlerin çalıştığı yerdir. Kubelet, Kube-Proxy, Container Runtime bileşenlerinden oluşur.

**Kubelet:** Her bir node’da çalışan bir ajandır. Konteynerlerin öngörüldüğü şekilde bir Pod içinde çalışmasını sağlamaktan sorumludur.

**Kube-Proxy:** Kubernetes'in Pod’lar ve servisler arasında ağ iletişimi sağlayan önemli bir bileşenidir (Adamson, 2023).

**Container Runtime:** Konteynerleri çalıştırmakla görevlidir. Konteynerlerin çalışması için gerekli altyapıyı sağlar.

# ÜÇÜNCÜ BÖLÜM

# SOSYAL GÜVENLİK KURUMU İÇİN STRATEJİLER

Gelişen dijital dönüşüm ortamında, bütün kamu kurumları gibi SGK de teknolojik adaptasyon gerektiren zorlukların üstesinden gelmek durumundadır. Teknolojik adaptasyonun sağlanabilmesi için sadece teknik fizibilite değil aynı zamanda kurumun personel kaynağının bu geçiş için stratejik uyumu da değerlendirilmelidir. Bu bölümde SGK’de mikroservis mimarisinin benimsenmesinin stratejik sonuçları araştırılmakta ve bu değişimin gelişmiş hizmet sunumu, operasyonel çeviklik vb. daha geniş hedeflerle nasıl uyumlu olduğu değerlendirilmektedir. Ayrıca mikroservis mimarisinin benimsenmesinin kurumun güvenli ve verimli sosyal güvenlik hizmeti sunma misyonuna olumlu katkıda bulunmasını sağlamak için değişim yönteminin önemi üstünde durulmaktadır.

## STRATEJİK UYGUNLUK VE FİZİBİLİTE

### Kurumsal Hedeflerle Uyumun Değerlendirilmesi

SGK için önemli bir kurumsal hedef, hizmetlerin tüm vatandaşlara erişilebilirliğini artırmaktır. Günümüzde erişilebilirliğin sağlanmasının önemli bir yolu da yazılım hizmetlerinin erişilebilirliğiyle olmaktadır. Direkt vatandaş tarafından kullanılan veya vatandaşa hizmet veren SGK personelinin kullandığı yazılım servisleri bu erişilebilirliğin bir parçasıdır. Mikroservis mimarisi, tüm sistemi etkilemeden güncellenebilen, iyileştirilebilen ölçeklendirilebilen ve bağımsız olarak dağıtılabilir hizmetlerin geliştirilmesini sağlayarak bu konuda önemli bir rol oynayabilir. Bu da yazılım servislerinin daha hızlı güncellenmesi ve hizmetlere daha kısa zamanda erişim sağlayabilir.

SGK’nin yeni mevzuat veya politika değişikliklerine yanıt olarak yazılımlarını hızlı bir şekilde güncellemesi gerekir. Mikroservis mimarisi, yazılım sistemini bölümlendirerek sistemin belirli bölümlerinin bağımsız olarak güncellenmesine olanak tanır ve mevzuat değişikliklerinin yapılmasıyla ilgili zaman ve maliyeti azaltır. Yazılım sisteminin tümünün güncellemeden bölüm bölüm güncellenebilmesi sık değişen bir mevzuatı olan bir kurum için önemlidir.

Mikroservis mimarisinin erişilebilirliği artırma ve yenilikçilik gibi stratejik hedefler ile uyumlu olması, SGK için faydalı bir teknolojik strateji olma potansiyelini ortaya koymaktadır.

### Teknik Fizibilite Çalışması

Bu çalışmanın amacı, SGK’de mikroservis mimarisine geçilmesi için yapılması gereken işlemlerin neler olduğunu, bunun getireceği avantajlar ve dezavantajların neler olabileceğini, hangi problemlere çözüm olabileceğini açıklamaktır.

Tezin başında düşünülen ve yapılması planlanan mikroservis mimarisi ve monolitik mimari ile aynı uygulamanın geliştirilip karşılaştırılması bu tez için mümkün olmamaktadır. Bunun nedenlerinden biri, mikroservis mimarisinin büyük çaplı uygulamalar için avantaj sağlayacağıdır. Büyük bir uygulamanın uygun bir şekilde bölünmesi zaman, büyük bilgi birikimi ve tecrübe gerektirir. Verilen sürede bunların analiz edilip ölçüm yapılması mümkün olmamaktadır. Diğer bir neden ise ölçüm yapılmasının zorluğudur. Mikroservis mimarisinin avantajlarını görmek için önce monolitik mimari üzerinde gerekli ölçümler ve gözlemler yapılmalı daha sonra sistemi mikroservisler ile dizayn edip aynı işlemler tekrar yapılmalıdır. Bu yüzden bu çalışma SGK’de mikroservis mimarisinin uygulanabilmesi için neler yapılması gerektiğine odaklanmaktadır.

### 1.2.1. Mevcut Durum

SGK’nin mevcut altyapısında halihazırda 6425 adet uygulama hizmet vermektedir (Sosyal Güvenlik Kurumu, 2024). Bu uygulamaları çalıştırmak için gerekli olan donanım, işlemci gücü, ağ kapasitesi bulunmaktadır. Dolayısıyla mikroservis mimarisi ile oluşturulmuş bir uygulamanın da çalışabilmesi için gerekli donanım, işlemci gücü ve ağ hızı mevcuttur. Ayrıca yazılım uygulamalarını konteyner haline getirebilmek için Docker, konteynerlerin orkestrasyonu için Kubernetes, kaynak kod ve proje yönetimi için GitLab teknolojileri kullanılmaktadır. GitLab, CI/CD yaklaşımını desteklemektedir.

SGK, diğer devlet kurumları gibi verilerini ve yazılımlarını kendi sunucularında barındırmaktadır. Günümüzde şirketler için bir seçenek olarak değerlendirilebilecek bulut bilişim hizmeti satın alarak verilerini ve/veya yazılım uygulamalarını bulutta işlemek güvenlik ve veri gizliliği gibi endişeler sebebiyle yasal olarak mümkün olmamaktadır.

Mikroservis mimarisinin SGK için düzgün bir şekilde değerlendirilebilmesi ve sisteme neler katabileceğinin gösterilebilmesi için mevcut bir sistem olan Emektar4B uygulaması üzerinden ele almanın daha iyi bir yaklaşım olacağı düşünülmektedir. Emektar4B, 4/1-(b) kapsamındaki kişilerin aylık tahsis işlemlerinin yapıldığı web tabanlı bir uygulamadır. Emektar4B, 2011 yılında yayımlanmıştır ve bu zamana kadar geliştirilmesi sürdürülmektedir.

Emektar4B uygulamasının 2011 yılından bugüne kadar kod karmaşıklığı sürekli artmış ve hala artmaktadır. Farklı geliştiriciler tarafından yapılan değişiklikler, uygulama kodlarının anlaşılmasını ve uygulamanın bakımını zorlaştırmıştır. Ek olarak uygulamanın eski bir teknoloji ile geliştirilmesi doküman ve örnek eksikliğine yol açmaktadır. Bu da yeni gelen ve bu teknolojiye aşina olmayan geliştirici için büyük bir zorluk ortaya çıkarmaktadır. Ayrıca entegre edilmesi düşünülen yeni teknolojiler uyumluluk sorunları nedeniyle kullanılamamaktadır.

Emektar4B uygulaması monolitik yapıda geliştirildiğinden ve buna bağlı olarak zamanla yazılım bileşenleri arasındaki bağımlılık artması nedeniyle tek bir yerinde yapılan düzenleme veya ekleme bütün uygulamanın yeniden derlenmesini gerektirir. Bu da uygulamanın yeniden dağıtımı yapılırken kesintilere neden olmaktadır. Ayrıca kodlar arasındaki bağımlılık artmıştır. Bunun sonucu olarak kodun bir yerinde yapılan değişiklik kodun birçok yerinde değişiklik yapılmasını gerektirmektedir.

Mikroservis mimarisinin getirdiği avantajlardan faydalanabilmek için DevOps yaklaşımının benimsenmesi bir zorunluluktur. Mikroservisler dağıtık ve karmaşık yapısı nedeniyle yönetilmesi daha fazla çaba gerektiren ve işlerin operasyonel tarafına ağırlık getiren bir mimaridir. SGK’de mevcut durumda DevOps altyapısını işletebilecek yeterli personel bulunmamaktadır.

### 1.2.2. İhtiyaçlar ve Analiz

Kurumda mikroservis mimarisinin başarılı ve verimli bir şekilde uygulanabilmesi için çeşitli gerekliliklerin sağlanması gerekmektedir. Bunlar; personelin eğitilmesi, DevOps yaklaşımının benimsenmesi ve araçlarının kullanılması, mikroservisleri doğru bir şekilde bölebilmek için DDD gibi metotların bilinmesi, dağıtık veritabanı yönetiminin yapılabilmesi, mikroservis mimarisine uygun test stratejilerinin benimsenmesi, mikroservisleri izleme araçlarının kullanılması olarak sıralanabilir.

Uygulama geliştirmeye yönelik dağıtık ve parçalı yapıya sahip mikroservis mimarisi, operasyonel karmaşıklıkları ve ölçeklenebilirlik ihtiyaçlarını karşılamak için doğası gereği DevOps uygulamalarının entegrasyonunu gerektirir. Bu birliktelik, mikroservis ortamında çeşitli nedenlerden dolayı kritik öneme sahiptir.

Mikroservis mimarisindeki dağıtım karmaşıklığı, her servis bağımsız olarak geliştirildikçe, dağıtıldıkça ve ölçeklendirildikçe önemli ölçüde artmaktadır. Geleneksel IT operasyonları, DevOps'un temel unsurları olan otomasyon ve standartlaştırılmış süreçler olmadan, birden fazla servisteki sık dağıtımları yönetmekte zorluk çekmektedir. DevOps, derleme, test ve sürüm döngülerinin otomatikleştirildiği ve hızlandırıldığı CI/CD uygulamalarına olanak tanır. Bu yalnızca dağıtımların hızını artırmakla kalmaz, aynı zamanda manuel işlemlerle ilişkili hata riskini de azaltır.

Ölçeklenebilirlik mikroservislerin temel bir avantajıdır; ancak çeşitli ortamlarda çok sayıda hizmet örneğinin yönetilmesinde zorluklara neden olur. DevOps uygulamaları, Kubernetes gibi konteyner orkestrasyon teknolojilerini içerirler. Bunlar mikroservislerin dinamik ölçeklendirilmesini ve yönetimini kolaylaştırır. Konteyner orkestrasyonu, mikroservis tabanlı mimarilerin yönetimi için çok önemli olan konteynerli uygulamaların dağıtımını, ölçeklendirilmesini ve çalışmasını otomatikleştirir.

Dağıtılmış bir sistemde izleme ve loglama karmaşıktır ancak sistem sağlığını ve performansını korumak için hayati öneme sahiptir. DevOps, mikroservislerin bütünsel bir görünümünü sağlayan, etkileşimlerini ve performanslarını gerçek zamanlı olarak izleyen entegre izleme araçlarını içerir. Bu tür araçlar, hataların hızlı bir şekilde belirlenmesine ve çözülmesine yardımcı olur; bu, servislerin gevşek bir şekilde bağlı olduğu ve ağa bağımlı olduğu bir sistemde çok önemlidir. Kurumda halihazırda izleme ve hataları tespit etmeye yardımcı araçlar kullanılmamaktadır. Bu araçların temin edilerek kurumda kullanıma sunulması gerekmektedir. Ayrıca kurum personelinin de DevOps yaklaşımını benimsemesi gerekmektedir.

### 1.2.4. Riskler

Mikroservis mimarisine geçiş ölçeklenebilirlik, esneklik ve dağıtım hızı gibi çok sayıda avantaj sunarken aynı zamanda çeşitli riskleri ve zorlukları da beraberinde getirir. Bu riskleri anlamak, kurumlar için kritik öneme sahiptir.

Mikroservisler sistem karmaşıklığını artırır. Düzinelerce veya yüzlerce ayrı servisi yönetmek, tek bir monolitik uygulamayı yönetmekten daha zor olabilir. Her servisin kendi bağımlılıkları, kaynak gereksinimleri ve ölçeklendirme politikaları olabilir; bu da dağıtım, izleme ve yönetimi zorlaştırabilir.

Veri yönetimi, mimarinin dağıtık yapısı nedeniyle mikrosevislerde daha karmaşık hale gelir. Monolitik bir veritabanı sisteminin geleneksel ACID (Atomicity, Consistency, Isolation, Durability) özellikleri olmadan hizmetler arasında veri tutarlılığının sağlanması, olay kaynağı veya dağıtılmış işlemler gibi stratejilerin uygulanmasını gerektirir ve bu da karmaşıklığı artırabilir.

Mikroservisler sıklıkla bir ağ üzerinden iletişim kurar ve bu da gecikmeye neden olur. Bu durum, özellikle servisler kötü tasarlanmışsa veya ağ altyapısı yeterince sağlam değilse uygulama performansını düşürebilir. Ayrıca, ağ sorunları servisin kullanılamamasına veya servisten servise iletişimde arızalara yol açabilir.

Mikroservisler kaynak kullanımının artmasına da neden olabilirler. Her mikroservis ayrı çalışma zamanı ortamları veya veritabanları gerektirebilir. Bu monolitik bir mimariye kıyasla potansiyel olarak daha yüksek operasyonel maliyetlere yol açabilir.

Mikroservislerde güvenlik daha karmaşıktır. Her servis, güvenlik ihlalleri için potansiyel bir giriş noktasıdır. Erişim kontrollerini yönetmek ve bağlantıları şifrelemek de dahil olmak üzere tüm servislerde tutarlı bir şekilde güvenlik uygulaması daha zordur.

Test işlemleri monolitik uygulamalara kıyasla daha zordur. Mikroservislerin hem ayrı ayrı hem de diğer mikroservislerle birlikte test edilmesi gerekmektedir. Ayrıca uçtan uca testler için gerçek ortam koşullarını simüle etmek zor olabilir.

Sürekli dağıtım mikroservislerin ana avantajlarından biridir fakat düzgün bir şekilde yönetilmezse risk oluşturabilir. Dağıtım sıklığı, uygun CI/CD düzeni mevcut değilse istikrarsızlık sorunlarına yol açabilir.

Kurumdaki yazılım ekiplerinin birbiriyle koordinasyonu sağlaması gerekliliği mikroservis mimarisine uyumsuzluk riskini doğurabilir. Çünkü Mikroservis mimarisi kültürel bir değişimin sağlanmasını gerekli kılar.

## ORGANİZASYONEL ETKİLER

### Değişiklik Yönetimi

Mikroservis mimarisine geçiş, yalnızca teknik alanda değil, aynı zamanda kurumsal yapı ve kültürde de değişiklikleri içerir. Sorunsuz bir geçiş sağlayabilmek ve mikroservislerin tüm faydalarından yararlanabilmek için doğru bir yaklaşım önemlidir.

Mikroservisler yüksek derecede özerklik gerektirir. Her ekip kendi servislerinin tasarımı, geliştirilmesi ve dağıtımı ile ilgili kararlar alma yetkisine sahip olmalıdır. Monolitik geliştirme yaklaşımında analiz, tasarım, geliştirme (Front-End/Back-End) ve test ekipleri hepsi kendi alanıyla ilgili ekipte işlerini gerçekleştirirler. Mikroservis mimarisinde ise ekibin sorumlu olduğu uygulamanın bütün yazılım yaşam döngüsü aşamaları ekip tarafından yapılabilmelidir. Mevcut durumda kurumda kullanılan yaklaşım yazılım ekibinin bütün aşamalarda var olduğudur. Bunun olası bir mikroservis mimarisine geçişi kolaylaştıran bir etken olduğu görülmektedir. Ayrıca bu yaklaşım iş ihtiyaçlarının ve hedeflerin uyumlu olmasını sağlar.

Mikroservisler dağıtık uygulamalardır. Çalışanlara uygulamaların yapısı nedeniyle mikroservis mimarilerinde daha yaygın olan ağ sorunlarını anlamaları için gerekli eğitimlerin verilmesi gerekmektedir. Bunun yanında CI/CD ve konteynerizasyon teknolojilerindeki becerileri geliştirmek için de eğitim programları uygulanmalıdır.

Mikroservisler genellikle iş yetenekleri etrafında düzenlendiğinden DDD’yi anlamak ve uygulamak çok önemlidir, bu da iş ihtiyaçlarına göre servis sınırlarının nasıl belirleneceği konusunda eğitim gerektirir. Ayrıca, DevOps ilkeleri, araçları ve uygulamaları konusunda tüm IT personelinin kapsamlı bir şekilde eğitilmesini içeren bir DevOps eğitimine de ağırlık vermek gerekmektedir.

Ekipler daha merkezi olmayan hale geldiği için etkili iletişim kanalları oluşturulmalıdır. Ekipler arasında iş birliğini ve bilgi paylaşımını kolaylaştıran araçlar kullanılmalıdır.

### Paydaşlara Etkileri

Yazılım mimarisi değişiminin hem DevOps hem de geliştirici ekipleri üzerinde etkileri vardır. Mikroservis mimarisine geçiş, bu ekiplerin çalışma, iş birliği yapma ve yazılım teslim etme biçimlerini etkiler. Paydaşlar üzerinde etkileri anlamak, geçişi etkili bir şekilde yönetmek ve mikroservislerin potansiyel faydalarından yararlanmak için gereklidir.

Bir kurumda mikroservis mimarisine geçiş DevOps tarafında iş yükünün ve karmaşıklığının artması anlamına gelmektedir. Bunun nedeni bir dağıtım ile devreye alınabilecek bir uygulamanın parçalara ayrılarak birden fazla dağıtım şeklinde devreye alınmasıdır. DevOps ekibinin artan sorumluluğu karşılayacak kapasiteye ulaşması gerekmektedir. Bunu gerçekleştirmenin yolu ise DevOps ekibinin mikroservis mimarisi için geliştirilmiş yazılım araçlarını öğrenmesi ve uygulamasıdır.

Mikroservis mimarisi hem DevOps hem de geliştirici ekipleri için önemli zorluklar ve artan sorumluluklar da gerektirmektedir. Değişen dinamikleri ve gereksinimleri iyi anlamak büyük önem taşımaktadır. Geliştiriciler iş alanlarına daha ayrıntılı bir şekilde odaklanmalı ve servisler arası iletişimin sağlam ve güvenli olmasını sağlamalıdır. Sonuç olarak, mikroservis mimarisine başarılı bir şekilde adapte olmak yalnızca teknik değil, aynı zamanda daha fazla iş birliği ve sürekli öğrenmeye yönelik kültürel bir değişime de bağlıdır.

# DÖRDÜNCÜ BÖLÜM

# TEKNOLOJİK ADAPTASYON YOLLARI

Mikroservis mimarisine geçiş sorunsuz bir şekilde yapılabilmesi için belirli yöntemler uygulanmalı ve adımlar izlenmelidir. Bu bölümde zamanla kazanılmış tecrübelerden yararlanılarak elde edilmiş yaklaşımlar ve geçiş adımları ele alınmaktadır.

## GEÇİŞ STRATEJİLERİ

### Strangler ve Big Bang Yaklaşımları

Yazılım mimarisi dönüşümünde monolitik bir mimariden mikroservis mimarisine geçişte, iki yaygın geçiş stratejisi kullanılmaktadır: Big Bang yaklaşımı ve Strangler yaklaşımı. Her bir strateji farklı metodolojiler, avantajlar ve zorluklar sunmaktadır.

Big Bang geçiş yaklaşımı, tüm bileşenlerin monolitik mimariden mikroservis mimarisine tek seferde geçişini önerir. Bu strateji, tüm uygulamanın durdurulması, yeniden yapılandırılması ve mikroservis odaklı biçiminde tek bir ortak çabayla dağıtıma alınması demektir. Bu yaklaşımın ana avantajı, basit geçiş takvimi ve yeni bir mimari modele geçerek iki mimariyi paralel olarak çalıştırma aşamasını azaltmasında yatmaktadır. Ancak Big Bang yaklaşımı risklidir. Temel risk, geçiş sırasında sistemin çökmesi ve sistem genelinde hataların ortaya çıkması olasılığıdır. Bu yaklaşımda birden geçiş yapılması nedeniyle, yeni sistemde öngörülemeyen herhangi bir sorun önemli aksaklıklara yol açabilir. Bu yöntem ayrıca, sistemin tüm parçalarının tam olarak anlaşılması ve yeni mimaride işlev görecek şekilde titizlikle yeniden tasarlanması gerektiğinden, başlangıçta kapsamlı bir planlama gerektirir.

Strangler yaklaşımı ise monolitik mimarinin tek tek bileşenlerinin veya servislerinin metodik ve sıralı bir şekilde mikroservis mimarisine dönüştürüldüğü kademeli bir geçişi önerir. Bu yaklaşımda monolitik yapı ile mikroservisler birlikte çalışır ve yeni istekler mikroservis olarak geliştirilir. Yazılım uygulamasının belirli bölümlerinin izole edilmesine, dönüştürülmesine ve bağımsız olarak dağıtılmasına olanak tanıyarak kesinti en aza indirilir ve sürekli test ve konfigürasyon yapılması sağlanır.

Strangler yaklaşımı, belirli bir zamanda sistemin yalnızca bazı bölümleri değiştirildiğinden, yaygın sistem arızası riskinin azaltılması da dahil olmak üzere çeşitli avantajlar sunar. Ayrıca bu yöntem, daha esnek bir geliştirme ortamını destekler ve çevik geliştirme uygulamalarıyla daha uyumludur. Bununla birlikte Strangler yaklaşımı toplam geçiş süresini uzatır ve orijinal monolitik sistemin kalan parçalarının yeni mikroservislerle sürekli olarak etkileşime girmesi gerektiğinden karmaşıklığı geçici olarak artırır. Bu, sağlam bir ara entegrasyon mekanizması gerektirir ve geçiş döneminde iki mimari tarzın sürdürülmesinde zorluklara yol açabilir.

Özetle, Big Bang ve Strangler yaklaşımları arasındaki seçim uygulamanın özel ihtiyaçlarına, risk toleransına ve operasyonel yeteneklerine bağlıdır. Big Bang yaklaşımı hızlı ancak riskli bir geçiş sunarken, Strangler yaklaşımı daha kontrollü, ancak uzun süreli bir geçiş yolu sağlar ve tipik olarak sürekli teslimat ve modern çevik metodolojilerle daha iyi uyum sağlar.

**Şekil 7:** Strangler Yaklaşımı



**Kaynak:** *(Çilingir, 2023)*

### Pilot Projeler

Kurumda mikroservis mimarisi ile projeler geliştirmeye ve yönetmeye başlamanın en ideal şekilde yapılabilmesi için pilot projeler yapmak en uygun metotlardan biridir. Monolitik bir mimariden mikroservis mimarisine geçişin ilk aşaması için pilot projeler önerirken, birkaç stratejik kriterler göz önünde bulundurulmalıdır. Bu kriterler riskleri en aza indirmeyi, öğrenme fırsatlarını en üst düzeye çıkarmayı ve mikroservis mimarisinin faydalarını etkili bir şekilde göstermeyi amaçlamaktadır. Pilot projelerin seçilmesi ve tasarlanmasına ilişkin kritik hususlar aşağıda belirtilmiştir:

* Yazılım uygulamalarının olası arızalarının etkisini en aza indirmek için kritik olmayan uygulamalar seçmek gerekir. Bu uygulamalar geliştiricilere geçişin somut faydalarını gösterebilmelidir.
* Monolitik mimari içinde nispeten diğer bileşenlere göre daha az bağımlılığı olan bileşenler seçilmelidir. Böylece yazılım uygulamasının diğer kısımlarındaki değişiklik ihtiyacı azalır ve geçişin karmaşıklığı en aza iner.
* Performans sorunları olan bileşenlere öncelik verilmelidir. İlk olarak bu bileşenlerin dönüştürülmesi performans iyileştirmeleri sağlar.
* Başarı için net ölçütler sağlayabilen bileşenler tercih edilmelidir. Performans, dağıtım döngüleri ve kaynak kullanımındaki iyileştirmelerin kolayca ölçülebildiği ve gözlemlenebildiği hizmetleri seçmek avantajlıdır.
* Geliştirme ve operasyon ekiplerinin mevcut beceri ve uzmanlıklarının olduğu teknolojiler ve uygulamalar seçilmelidir.
* Daha az düzeltmeye ihtiyaç olan bileşenlerle başlamak avantajlıdır.
* Pilot proje sırasında ve sonrasında geri bildirim önemlidir. Bu, etkiyi değerlendirmek ve iyileştirmek için izleme araçlarını ve kullanıcı geri bildirim kanallarını içerir.

Kurum, pilot projeleri bu kriterlere göre dikkatlice seçerek mikroservis mimarisine geçişle ilgili riskleri etkili bir şekilde yönetebilir, faydaları erkenden gösterebilir ve sonraki geçişler için bir örnek oluşturabilir.

## GEÇİŞ ADIMLARI

Monolitik mimari ile oluşturulmuş bir yapıdan mikroservis mimarisine geçişte belli bir algoritmayı takip etmek gerekir. Bu geçişte farklı yöntemler ve teknolojiler kullanılabilse de algoritmanın ana hatları aynıdır.

### Monolitik Uygulamayı Analiz Etme

Monolitik bir mimariden mikroservis mimariye geçişin ilk aşaması, mevcut monolitik sistemin titiz bir şekilde incelenmesini ve ayrıştırılmasını gerektirir. Bu süreç, geçişte en etkili stratejileri belirlemek için gerekli olan sistemin bileşenlerini ve bağımlılıklarını tespit edilmesini de kapsar.

Monolitik mimarinin kapsamlı bir şekilde anlaşılması için eğer mevcut ise dokümantasyonun kapsamlı bir incelemesi yapılır. Bu inceleme, dokümantasyonun eksik veya eski olduğu, sistemin mevcut durumunu doğru bir şekilde yansıtmak için güncelleme veya tamamen yeniden yazma gerektiren alanların belirlenmesine yardımcı olur.

Mevcut performans verileri analiz edilerek sistemin verimliliğini ve yanıt verebilirliğini ölçülebilir. Bu kıyaslama, mikroservis mimarisine geçişin genel sistem performansı üzerindeki etkisini değerlendirmek için geçiş sonrası karşılaştırmalı bir analiz yapılmasına olanak tanıyan bir referans noktası olarak kullanılabilir.

Monolitik sistemin incelenmesiyle temel işlevler ve modüller belirlenir. Amaç, bu mikroservislerin bağımlılığını azaltmak, bakım kolaylığı, ölçeklenebilirlik ve bağımsız dağıtımı sağlayabilecek şekilde ayırmaktır.

Uygulamanın teknik borcunun bir değerlendirmesi yapılır. Bu, geçiş sürecini engelleyebilecek eski kodların, eski teknolojilerin ve ölçeklenemeyen bileşenlerin değerlendirilmesini sağlar. Teknik borcun ele alınması, mikroservis mimarisine geçişin verimsizlikleri veya potansiyel başarısızlık noktalarını sürdürmemesini sağlamak için çok önemlidir.

Özetle, monolitik sistemin analizi, mikroservis mimarisine geçişin temel bir adımını oluşturur. Analiz mikroservis mimarisine yönelik geçiş için zemin hazırlar. İş hedefleri ve teknik gereksinimlerle uyumu sağlar.

### Bileşenleri Tanımlama

Monolitik mimariden mikroservis mimarisine geçişin bu adımında ana odak noktası, mikroservis mimarisine dönüştürülebilecek bileşenleri tanımlamaktır. Monolitik mimarilerde kodlar zamanla karmaşıklaşır ve iç içe geçmeye başlar. Aynı işlevleri yerine getiren benzer fonksiyonlar oluşur. DDD ilkelerinin kullanılması bu süreçte düzgün bir ayrım yapılmasına olanak tanır. Monolitik sistem içindeki sınırlı bağlamlar (belirli iş sorumluluklarına ve iyi tanımlanmış arayüzlere sahip farklı işlevsel bölümler) tanımlanarak mikroservis mimarisi için mantıksal bölümler belirlenir. Bu sınırlı bağlamlar, benzersiz iş yeteneklerine göre tanımlanmış olup, iş alanlarının net bir şekilde ayrılmasını sağlar.

Mikroservislerin uygun sınırlarının belirlenmesi bu adımdaki en önemli zorluktur. Belirlenen her bir servis, tek bir işlevi yerine getirecek veya birbiriyle yakından ilişkili bir dizi işlevi yerine getirecek kadar küçük olmalı ve böylece yüksek uyumluluk sağlanmalıdır. Bu DDD ilkeleriyle yapılırsa geliştirme, test ve bakım kolaylığı sağlanmış olur.

Mikroservisler için uygun bileşenlerin belirlenmesinin yinelemeli bir şekilde yapılması daha yararlı olabilir. Bu yinelemeli yaklaşım geçiş süreci sırasında hata riskini azaltır.

Mikroservislere dönüştürülmek üzere monolitik sistemdeki bileşenlerin belirlenmesi, dikkatli planlama, alan uzmanlığı ve yinelemeli iyileştirme gerektiren kritik bir aşamadır.

### Mikroservisleri Tanımlama ve Geliştirme

Önceden tanımlanmış bileşenlerin mikroservis olarak geliştirilmesini içerir. Mikroservisler, sistemdeki diğer servislerden bağımsız olarak geliştirilmeye olanak tanıyan kendi teknoloji yığınları ve veritabanları ile özerk bir haldedir.

Mikroservislerin geliştirilmesi, ayrı dağıtım, ölçeklendirme ve testlere izin vermek için her mikroservise ayrı geliştirme ortamlarının kurulmasını gerektirir. Bu ayrım, geliştirme sürecinin çevikliğini artırarak ekiplerin sistemin diğer kısımlarını etkilemeden güncellemeleri ve iyileştirmeleri hızla uygulamasına olanak tanır. Geliştirme sırasında, hizmetlerin gevşek bağlı bir şekilde entegre edilmesi için gerekli olan servisler arası iletişim API'ler ile sağlanmalıdır. Bu sebeple uygulama için gerekli API’ler yazılmalıdır.

Her mikroservis, diğer mikroservislerle paylaşılmayan veritabanını kontrol eder ve böylece veri kapsüllemesini sağlar. Bu bağımsızlık, her bir servisin yönettiği verilerin bütünlüğünü ve güvenliğini korumak için önemlidir.

### Mikroservisler Arası İletişim

En uygun iletişim tarzını (senkron veya asenkron) seçmek için mikroservisler arasındaki veri alışverişi gereksinimlerini değerlendirilmelidir. HTTP veya RPC gibi senkron iletişimler gerçek zamanlı istemci-sunucu etkileşimleri için uygunken, mesaj kuyrukları veya olay akışları kullanan asenkron iletişimler ayrıştırılmış, olay odaklı mimarileri destekler. Seçilen iletişim tarzına bağlı olarak, hizmetlerin operasyonel ve performans ihtiyaçlarına en uygun protokoller seçilmelidir.

Eşzamanlı iletişim için ölçeklenebilir ve kullanımı kolay, durum bilgisi olmayan REST API'ler kullanılmalıdır. Bu API'lerin iyi belgelendirilmeli ve mevcut hizmet işlemlerini kesintiye uğratmadan değişiklikleri yapmak için sürümlendirilmelidir.

Yüksek trafik veya hizmet kesintisi sırasında bile güvenilir mesaj teslimatı sağlamak için mesaj kuyrukları kullanılabilir. Mesaj kuyrukları, başarılı bir şekilde işlenene kadar mesajları arabelleğe alır, böylece veri kaybını önler ve yük seviyelendirmeyi sağlar. Mesajların alındığını ve işlendiğini onaylamak için sistemler kurulabilir. Başlangıçta işlenemeyen mesajlar için yeniden deneme mekanizmaları uygulayarak geçici arızalar nedeniyle hiçbir mesajın kaybolmaması sağlanır.

Verimi ve gecikme süresini etkin bir şekilde yönetmek için iletişim kanallarının performansı sürekli olarak izlenmelidir. İletişim altyapısındaki darboğazları veya arızaları tespit etmek için izleme araçlarını kullanılmalıdır. İstemci isteklerini birden fazla mikroservise verimli bir şekilde dağıtarak kaynak kullanımını ve yanıt süresini optimize etmek için yük dengeleyicileri kullanılmalıdır.

### Bağımsız Dağıtma

Bu adım, mikroservislerin özerk olarak çalışabilecekleri ve sistemdeki diğer servislerden ayrı olarak yönetilebilecekleri bir sunucuda fiilen çalıştırılmasıdır. Dağıtım yapılmadan önce, her bir mikroservisin kendi veritabanına, tanımlanmış API'lere ve izole edilmiş çalışma zamanı ortamına sahip olacak şekilde tamamen kapsüllenmesi sağlanmalıdır. Bu kapsülleme, mikroservisin sistemin diğer bileşenleriyle olumsuz etkileşime girmeden bağımsız olarak çalışabilmesini garanti eder. Dağıtım süreci, geliştirme aşamasından üretime kadar tutarlı bir ortam sağlayan ve ortamlar arasındaki tutarsızlık olasılığını azaltan Docker gibi konteynerleştirme teknolojilerini içermelidir.

Servis keşfi, yük dengeleme ve otomatik ölçeklendirme dahil olmak üzere dağıtım karmaşıklıklarını yönetmek için Kubernetes gibi hizmet düzenleme araçları sürece dahil edilmelidir.

Geliştirme, test ve dağıtım süreçlerini otomatikleştiren sağlam CI/CD süreçleri oluşturulmalıdır. Bu otomasyon, entegrasyon ve otomatik testler yoluyla sorunların erkenden yakalanmasına yardımcı olur. Tüm testleri geçen herhangi bir değişikliğin otomatik olarak üretime dağıtılabilmesi sağlanmalı ve manuel müdahale en aza indirilmelidir.

### Veri Yönetimini Ayırma

Mikroservis mimarisinde bileşenleri ayrıştırma hem uygulama hem veritabanı bazında işlemler gerektiren bir adımdır. Monolitik mimaride tek bir veritabanı içinde bulunan veriler sistemin doğası gereği mikroservis mimarisinde ayrı olarak tutulmalıdır.

Mikroservis mimarisinde veri sahipliği ilkesi, her bir mikroservisin kendi verileri üzerinde özel kontrole sahip olmasını zorunlu kılar. Bu, hizmetlerin gevşek bir şekilde bağlanmasını sağlayarak özerkliği destekler ve sistem esnekliğini artırır. Bu bilgiler dikkate alındığında ideal bir yapı kurmak için veritabanı ayrıştırılmalıdır.

Mikroservisler arasında veri dolaşımını önlemek için her mikroservis kendi veritabanına sahip olmalıdır. Bu izolasyon, sistemin diğer bölümleri yoğunluk altındayken veya bakımdan geçerken bile mikroservisin verilerinin bütünlüğünü ve tutarlılığını korumaya yardımcı olur. Veritabanları her bir mikroservisin ihtiyaçları için özel olarak ayarlanmalıdır. Bu yaklaşım, veri modelinin karmaşıklığı artıran ve performansı düşüren gereksiz veri alanları olmadan mikroservisin işlevselliğini en iyi şekilde desteklemesini sağlar.

Uygulama mantığı ile veri erişim mantığı arasında net bir soyutlama katmanı sağlamak mikroservisleri veritabanına daha az bağımlı hale getirir. Her bir mikroservisin ihtiyaçlarına özel DAO geliştirerek ayrıştırılmış veri etkileşimleri sağlanmalıdır.

Değişiklikleri sistematik bir şekilde yönetmek ve zaman içindeki veri değişikliklerini anlamak ve sorun gidermek için çok önemli olan veritabanı evriminin geçmişini korumak için veritabanı şemalarına sürüm kontrolü uygulanmalıdır. Değişiklikleri kontrollü ve geri döndürülebilir bir şekilde uygulamak için otomatik geçiş araçları kullanılmalıdır.

### Optimize Etme ve Ölçeklendirme

Performans darboğazlarını belirlemek ve çözmek için her bir mikroservis sürekli olarak izlenmelidir. Verimliliği ve yanıt sürelerini artırmak için veri erişim modelleri iyileştirilmeli ve servis mantığı geliştirilmelidir.

Kullanım istatistiklerine ve performans ölçümlerine göre kaynak tahsisleri ayarlanmalıdır. Ayrıca herhangi bir hizmetin sistem kaynaklarını tekeline almasını önlemek için kaynak kotaları ve sınırları uygulanmalıdır.

Gerçek zamanlı kullanıcı etkileşimleri, sistem günlükleri ve hata raporları dahil olmak üzere çeşitli kaynaklardan veri toplayan ve analiz eden sağlam geri bildirim mekanizmaları oluşturulmalıdır. Bu geri bildirim, hizmetlerin iyileştirilebileceği veya ayarlama yapılması gereken alanları belirlemek için önemlidir.

Mikroservis güncellemelerinde yinelemeli bir yaklaşım gereklidir. Burada en son içgörülere ve geri bildirimlere dayalı olarak düzenli olarak artımlı değişiklikler yapılır. Bu yaklaşım, kullanıcı ihtiyaçlarını ve sistem performansı hedeflerini daha iyi karşılamak için hizmetlerin sürekli gelişimini ve uyarlanmasını sağlar.

### Monolitik Parçaları Çıkarma

Monolitik bir mimariden mikroservis tabanlı bir mimariye geçerken monolitik uygulamanın mikroservislere başarıyla taşınan parçalarının sistematik olarak kaldırılması gerekir. Bir mikroservis tamamen çalışır hale geldiğinde ve belirlenen tüm görevleri beklendiği gibi yerine getirdiği doğrulandığında, monolitik uygulama içindeki ilgili bileşen kullanımdan kaldırılmak üzere ayrılabilir. Hangi bileşenlerin taşındığını ve hangilerinin kaldırılmayı beklediğini takip etmek için bu süreç boyunca detaylı belgeler tutulabilir.

Tamamen silmeden önce, monolitik bileşenleri tamamen kaldırmak yerine geçici olarak devre dışı bırakmak tavsiye edilir. Bu yaklaşım, sistemin kararlılığının izlenmesine ve mikroservisin devre dışı bırakılan parçanın işlevselliğini herhangi bir veri bütünlüğü kaybı olmadan yeterince karşıladığından emin olmasına olanak tanır. Bu gözlem süresi boyunca, mikroservis uygulamasındaki beklenmedik davranışları veya eksiklikleri tespit etmek için tüm sistem etkileşimleri titizlikle kaydedilmelidir. Önceden tanımlanmış bir gözlem penceresi boyunca herhangi bir sorun ortaya çıkmazsa, mikroservisin kapasitesine duyulan güven teyit edilmiş olur.

İşlemler en az kritik bileşenlerden başlayarak ve sistemin daha önemli parçalarına doğru ilerleyerek aşamalı olarak yapılmalıdır. Her çıkarma işleminin ardından, istenmeyen sonuçların kalan monolitik bileşenleri veya sistemin genel işlevselliğini etkilememesini sağlamak için bir dizi test yapılmalıdır. Her kaldırma işleminden sonra yapılan kapsamlı testler, sistemin sağlam ve güvenilir kalmasını sağlamak için regresyon, performans ve entegrasyon testlerini içerir.

# BEŞİNCİ BÖLÜM

# UYGULAMA ÖRNEĞİ

## UYGULAMANIN MEVCUT DURUMU

Emektar4B, nesne yönelimli bir programlama dili olan Java ile geliştirilmiş web tabanlı bir uygulamadır. Çerçeve olarak Struts 1 kullanılmaktadır. Struts 1 MVC(Model-View-Controller) mimarisini benimsemiş bir çerçevedir. Bu sayede kodlar ayrıştırılarak kavranması daha kolay ve karmaşıklığı daha az web uygulamaları oluşturmak için kullanılmaktadır.

Uygulamada web tabanlı kullanıcı arayüzü oluşturmak için JSP kullanılmaktadır. Bu teknoloji, geliştiricilerin HTML, XML veya diğer belge türlerine dayalı dinamik web sayfaları oluşturmasında kullanılır ve bu da kullanıcıların web tarayıcılarında içeriklerin gösterilmesini sağlar.

Uygulamada veritabanı olarak DB2 kullanılmaktadır. DB2, IBM tarafından geliştirilmiş bir ilişkisel veritabanı yönetim sistemi teknolojisidir. Verileri oluşturmak, yönetmek ve düzenli bir şekilde tutmak için kullanılır. Emektar4B’de kullanılan veriler bu teknoloji ile saklanmaktadır.

Uygulama içinden veritabanı ile ilgili işlemleri yapmak için bir ORM aracı olan hibernate teknolojisi kullanılmaktadır. Uygulama tarafında nesne olarak ele alınan veriler veritabanı tarafında ilişkisel olarak tutulmaktadır. Hibernate iki taraf arasındaki eşleştirme görevini yapar. Uygulama tarafında SQL kodu yazmadan standart bir şekilde veritabanı işlemleri yapılmasını sağlamaktadır.

WebSphere Application Server, IBM tarafından geliştirilmiş bir uygulama sunucusudur. Web tabanlı Emektar4B bu uygulama sunucusunda barındırılmaktadır. WebSphere Application Server güvenli ve ölçeklenebilir bir altyapı sağlamaktadır.

Kullanılan bu teknolojiler içinde Struts 1 eski bir teknolojidir ve modern çerçevelerin getirdiği yeteneklere sahip değildir. WebSphere Application Server maliyeti ve teknoloji bağımlılığı gibi dezavantajları barındırmaktadır.

Emektar4B uygulaması monolitik bir yapıdadır. Uygulamanın mevcut durumu düşünüldüğünde strangler yaklaşımının sorunsuz bir geçiş için daha uygun olduğu görülmektedir.

## GEÇİŞ SENARYOSU

Bu geçişte Strangler yaklaşımı benimsenmiştir. Emektar4B çok fazla bileşeni bünyesinde barındırdığı için birçok mikroservise ayrılabilir durumdadır. Bu çalışmada belli bir bileşen seçilip bir mikroservis olarak uygulamadan ayrı bir şekilde tasarlanıp çalıştırılacaktır. Geçiş senaryosu için seçilen bileşen diğer bileşenlerle minimum bağımlılık sergilemelidir. Böylece daha sorunsuz ve daha az maliyetli bir geçiş süreci sağlanabilir.

Emektar4B bileşenlerinin kapsamlı ve ayrıntılı bir analizi, TALEP bileşeninin diğer bileşenlere göre daha az bağımlılık gösterdiğini ortaya koymaktadır. TALEP bileşeninin hem uygulama hem de veritabanı katmanlarında sadece KISI bileşeniyle bağlantısı vardır. Bu özellik, onu özerk bir mikroservis olarak ayırma ve yeniden geliştirme için en uygun aday haline getirerek ayrıştırma ve entegrasyon süreçlerinin zorluğunu en aza indirmiştir.

**Şekil 8:** Talep Bileşeninin Alanları

|  |  |
| --- | --- |
| ALAN ADI | ALAN TİPİ |
| ID | NÜMERİK |
| KISI\_ID | NÜMERİK |
| TALEPTARIHI | TARİH |
| VARIDETARIHI | TARİH |
| VARIDESAYISI | NÜMERİK |
| TALEPTIPI | NÜMERİK |

**Kaynak:** Bu çalışma kapsamında hazırlanmıştır.

Java programlama dilinin kullanımının yeni mikroservisin geliştirilmesinde önemli avantajları olduğu görülmüştür. Java, kurum içinde en bilinen ve yaygın olarak kullanılan programlama dili olması nedeniyle bu çalışma için seçilmiştir. Geliştirme ekipleri arasında Java deneyiminin daha çok olması üretkenliği artırıp mikroservis mimarisine daha sorunsuz bir geçişi kolaylaştırır, zaman kazandırır ve hata potansiyelini en aza indirir.

Java çerçevelerinin mikroservis mimarisine olan desteği, seçimde bir başka nedendir. Java platformları, özellikle mikroservislerin geliştirilmesini, dağıtımını ve bakımını kolaylaştırmak için tasarlanmış Spring Boot gibi kapsamlı bir kütüphane ve çerçeve ekosistemi sunar. Bu araçlar, mikroservislerin verimli çalışması için kritik öneme sahip olan yapılandırma yönetimi ve REST API'leri için entegre destek gibi temel özellikler sağlar.

Mikroservislerin iletişimi için, doğasında bulunan basitlik ve çeşitli teknolojiler arasında kapsamlı desteğe sahip olmasından dolayı REST API tercih edilmiştir. Bu seçimde öncelikle REST teknolojisinin kullanım kolaylığı sağlayan ve geliştirme sürecini hızlandıran basit ve anlaşılır yapısı etkili olmuştur. REST; GET, POST, PUT ve DELETE gibi herkes tarafından anlaşılan ve uygulanması kolay olan standart HTTP yöntemlerinden yararlanır. Bu basitlik, mikroservisler arası iletişim karmaşıklığını azaltarak onu mikroservis mimarisi için ideal bir seçim haline getirmiştir.

REST API, Java programlama dili de dahil olmak üzere çok sayıda teknolojide geniş bir desteğe sahiptir. Bu, mevcut sistemler ve araçlarla sorunsuz entegrasyon sağlayarak geliştirici verimliliğini artırır ve entegrasyon sorunları potansiyelini azaltır. Sağlam ekosistemiyle Java, REST hizmetlerinin oluşturulmasını ve kullanımını basitleştiren kapsamlı kütüphaneler ve çerçeveler sunarak REST yapısının tercih edilen iletişim yöntemi olmasını sağlamıştır.

Mikroservis olarak oluşturulacak olan yeni uygulamaya ait TALEP bileşeninin tablosu da ayrılır ve veriler yeni tabloya taşınır. Oluşturulan tablo ile mikroservis arasında bağlantı sağlanır. Veritabanı yönetim sistemi olarak DB2 ve ORM olarak hibernate teknolojisi kullanılmıştır.

Eski uygulama, mevcut mikroservis ve yazılacak olan diğer mikroservisler birbirinden haberdar olması için Servis Keşfi mekanizması gerekir. Servis Keşfi mekanizması uygulaması sıfırdan yazılabilir veya daha önceden oluşturulmuş ve hazır halde gelen yapılar kullanılabilir. Netflix Eureka hazır gelen bir Servis Keşfi uygulamasıdır. Spring Cloud aracılığıyla elde edilip konfigüre edildikten sonra kullanıma alınmıştır. Netflix Eureka tüm istemcilerin listesini kaydeder ve yük dengeleme işlevi ile gelen istekleri servislere dağıtır. Yük dengeleme yapısı Kubernetes teknolojisi tarafından da yapılabilir. Sonraki aşamalarda da tıpkı bu aşamada olduğu gibi bütün mikroservislere gerekli bağımlılıklar indirilmeli ve konfigürasyonu yapılmalıdır.

Yeni mikroservis ve Servis Keşfi mekanizması çalışır duruma geldikten sonra eski monolitik Emektar4B uygulamasından mikroservise gelecek olan istekleri yönlendirmek için bir mekanizma gerekir. Bu API Gateway ile sağlanmıştır. API Gateway istekleri yönlendirme ve gerektiğinde filtreleme görevlerini gerçekleştiren bir ayrı bir uygulamadır.

**Şekil 9:** Sistem İşleyişi



**Kaynak:** Bu çalışma kapsamında hazırlanmıştır.

# ALTINCI BÖLÜM

# SONUÇ VE ÖNERİLER

X dağıtık sistemlerde veri bütünlüğünü sağlamak önemli. Bu yüzden 2PC ve sga yöntemi bunu sağlamaya yöneliktir. X dağıtık sistemlerde veri bütünlüğünü sağlamak önemli. Bu yüzden 2PC ve saga yöntemi bunu sağlamaya yöneliktir.

DevOps bölümünü yaz. 1.Bölümdeki mikroservis mimarisinde az da olsa bir şeyler yaz. Stratejik uygunluk ve fizibilite bölümündeki Tavsiye bölümünü zaten tezin sonunda vereceğim için oraya taşıyorum. Gerek olursa yeniden oraya alırım. Resimlerin yazılarını ayarlamayı unutma resim bir sayfada açıklaması diğer sayfada olmasın. Sayfa sayısı artması için SAGA dahil edilebilir. Mikroservisler arası iletişim yöntemleri yazılabilir. GEÇİŞ Senaryosu anlatırken Javayı söyledin ama Spring boot framework’ünü belirtmedin. Konuyu genişletmek için kullanabilirsin. İlk başta chatgpt ye kontrol ettir anlatımda bir yanlışlık var mı? Rest API

Konuyu genişletmek **Eureka Server,** Spring boot framework,

Neden pratik olarak sonuçları gösteremediğini iyi bir şekilde anlat.

# KAYNAKÇA

# EKLER