



**T.C.
SELÇUK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ**

**MONOLİT YAPIDAKİ BANKACILIK
YAZILIM SİSTEMLERİNİN MİKRO SERVİS
MİMARİSİ ESASLI BİR AÇIK BANKACILIK
PLATFORMUNA DÖNÜŞÜMÜ**

Fikri AYDEMİR

DOKTORA TEZİ

Bilişim Teknolojileri Mühendisliği Anabilim Dalı

**Ocak 2024
KONYA
Her Hakkı Saklıdır**

TEZ KABUL VE ONAYI

Fikri AYDEMİR tarafından hazırlanan “**Monolit Yapıdaki Bankacılık Yazılım Sistemlerinin Mikro Servis Mimarisi Esaslı Bir Açık Bankacılık Platformuna Dönüşümü**” adlı tez çalışması 04/01/2024 tarihinde aşağıdaki jüri tarafından oy birliği ile Selçuk Üniversitesi Fen Bilimleri Enstitüsü Bilişim Teknolojileri Mühendisliği Anabilim Dalı’nda DOKTORA TEZİ olarak kabul edilmiştir.

Jüri Üyeleri

İmza

Başkan

Prof. Dr. Harun UĞUZ

.....

Danışman

Prof. Dr. Fatih BAŞÇİFTÇİ

.....

Üye

Prof. Dr. Halife KODAZ

.....

Üye

Doç. Dr. Tahir SAĞ

.....

Üye

Dr. Öğr. Üyesi Züleyha YILMAZ ACAR

.....

Yukarıdaki sonucu onaylarım.

Prof. Dr. Ömer Faruk YÜKSEL
FBE Müdürü

Bu tez çalışması **5210012** nolu **TÜBİTAK 1505** Üniversite-Sanayi iş birliği projesi çerçevesinde gerçekleştirilmiştir.

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.

DECLARATION PAGE

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

İmza

Fikri AYDEMİR

Tarih: 04.01.2024

ÖZET
DOKTORA TEZİ

**MONOLİT YAPIDAKİ BANKACILIK YAZILIM SİSTEMLERİNİN MİKRO
SERVİS MİMARİSİ ESASLI BİR AÇIK BANKACILIK PLATFORMUNA
DÖNÜŞÜMÜ**

Fikri AYDEMİR

**Selçuk Üniversitesi Fen Bilimleri Enstitüsü
Bilişim Teknolojileri Mühendisliği Anabilim Dalı**

Danışman: Prof. Dr. Fatih BAŞÇİFTÇİ

2024, 261 Sayfa

Jüri
Prof. Dr. Fatih BAŞÇİFTÇİ
Prof. Dr. Harun UĞUZ
Prof. Dr. Halife KODAZ
Doç. Dr. Tahir SAĞ
Dr. Öğretim Üyesi Züleyha YILMAZ ACAR

Literatürde Centralized Online Real-Time Environment (CORE) banking systems olarak geçen ve Türkçe’de çekirdek bankacılık sistemleri ya da temel bankacılık sistemleri olarak ifade edilebilen sistemler, bankaların sunduğu tüm servislerin uygulama mantığını barındıran yazılım sistemleri olmaları bakımından finansal işlemlerin devamlılığı için büyük önem arz ederler. Bu sistemler tarafından sağlanan bankacılık servislerine örnek olarak mevduat yönetimi, para transferleri, kredilendirme ve raporlama servisleri verilebilir. Bu servislere erişim konusunda dünya genelinde Açık Bankacılık Standartları geliştirilmiştir. Bu standartlar, bankacılık servislerine erişimin birer uygulama ara yüzü (Application Programming Interface- API) olarak nasıl tasarlanması gerektiğini ortaya koyarken bu servislerin altyapısını oluşturan temel bankacılık sistemlerinin nasıl tasarlanması gerektiği konusunu kapsamazlar.

Günümüz temel bankacılık sistemlerinin çoğu (Örn.: Main-frame sistemleri) bulut bilişimin yaygınlaşmasından önceki dönemde geliştirilmiş olup sunum, uygulama ve veri erişim katmanları gibi temel yazılım katmanlarının hepsini içine alan yekpare bir mimari yapı ortaya koymaktadır. Monolit mimari olarak da adlandırılan bu yapıda yazılım bileşenleri, spesifik bir donanım altyapısının (Örn.: İşlemci, RAM) kullanımını gerektiren ortak bir kod ve veri tabanı üzerinde geliştirilmiştir. Bu durum bankalar gibi geniş çaplı yazılım altyapısına sahip kurumları bakımı oldukça maliyetli eski sistemlerin ve teknolojilerin pençesine düşürmekte ve dolayısıyla bankacılık sistemlerinin modern bulut-tabanlı teknolojilere (Örn.: Sanallaştırma, Kubernetes ve konteyner (container) teknolojileri) adaptasyonunu güçleştirmektedir.

Bulut tabanlı teknolojilerdeki gelişmeler doğrultusunda ortaya çıkmış olan Mikro Servis Mimarisi her biri temel düzeyde (atomik) işleve haiz olan ve birbirinden bağımsız ancak koordineli olarak çalışabilen mikro servislerin geniş çaplı bir dağıtık yazılım sistemini oluşturmak üzere bir araya geldiği bir dağıtık yazılım sistemi mimarisidir. Bu mimari doğru kullanıldığı takdirde monolit mimarinin dezavantajlarını ortadan kaldırıp bulut-tabanlı bir altyapıya dönüşümü kolaylaştıracak bir potansiyeledir.

Buradan hareketle bu doktora çalışması kapsamında ele alınan problem monolit mimari esaslı temel bankacılık yazılım sistemlerinin mikro servis mimarisi temelinde ve açık bankacılık standartlarına uyumlu bir yazılım platformuna nasıl dönüştürülebileceğidir. Bu problemin çözümü için temelinde mikro servis mimarisini esas alan bir bankacılık yazılım sisteminin tasarlanıp geliştirilmesi için gerekli standartlar, teknikler ve tasarım kalıpları araştırılmış ve bunları uygulayan bir mimari model ortaya konmuştur. Bu modelin gerek akademik gerekse endüstriyel kapsamdaki en belirgin katkısı yüksek yük anında bile ivmelenerek artan performanslı bir yapı ortaya koyması ve mikro servisler arası dağıtık SAGA işlemlerinin yönetimini aktör modeli temelinde performans darboğazına mahal bırakmadan akıllıca çözmesidir.

Anahtar Kelimeler: Çekirdek (Temel) Bankacılık Sistemi, Açık Bankacılık Standartları, API, Monolit Mimari, Mikro Servis Mimarisi, SAGA, Dağıtık SAGA İşlemlerinin Yönetimi, Aktör Modeli

ABSTRACT

Ph.D. THESIS

TRANSFORMING MONOLITHIC CORE BANKING SYSTEMS INTO A MICROSERVICES BASED OPEN BANKING PLATFORM

Fikri AYDEMİR

**The Graduate School of Natural and Applied Science of Selçuk University
The Degree of Doctor of Philosophy
In Information Systems Engineering**

Advisor: Prof. Dr. Fatih BAŞÇİFTÇİ

2024, 261 Pages

Jury

Prof. Dr. Fatih BAŞÇİFTÇİ

Prof. Dr. Harun UĞUZ

Prof. Dr. Halife KODAZ

Assoc. Prof. Dr. Tahir SAĞ

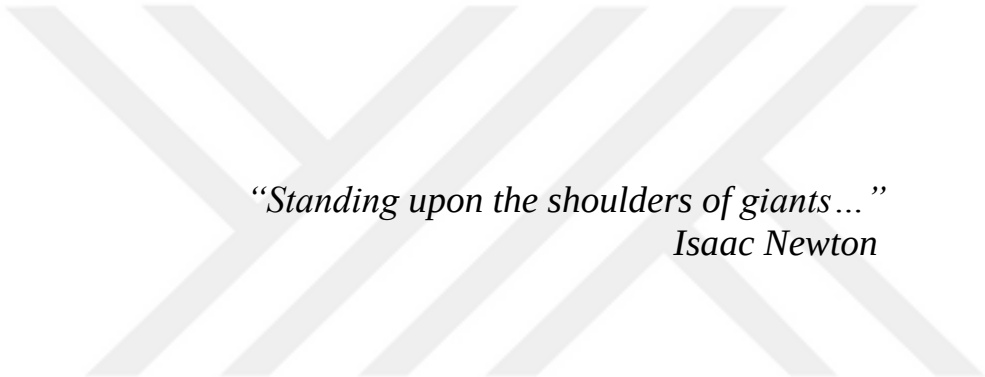
Assist. Prof. Dr. Züleyha YILMAZ ACAR

Centralized Online Real-Time Environment (CORE) banking systems lie in the heart of any bank today as they embrace all the services banks provide to execute their day-to-day business. These services, namely banking services, can include, transaction processing, payment processing, auditing, reporting, risk management, customer management, and e-banking. Standardization studies, namely Open Banking Standards, have been carried out around the World to enable access to these core banking services through well-defined Application Programming Interfaces (APIs). While these standards set high-level API standards for accessing core banking services; they do not set technical standards on how a core banking software infrastructure should be designed based on the modern cloud-native architectural trends, such as Micro Services Architecture (MSA).

Many of today's banking systems were built based on the monolith software architecture that was once a common practice in building distributed software systems before the advent of cloud computing-related technologies, such as virtualization and containerization. Monolith architecture encompasses all core banking services into a single, fully integrated piece of software where the codebase can become overwhelmingly complex and incomprehensible over time. As a consequence, it becomes rather challenging to make bug fixes and feature additions in such software and this can lead to serious maintenance costs for banks in adapting new technologies. MSA has emerged as a software engineering practice in line with the developments in virtualization and containerization technologies to overcome the problems that were encountered in monolith software architecture. In MSA, core banking services can be designed and operate independently (i.e., autonomously) to form a large-scale distributed core banking system that can benefit from the cloud computing technologies (e.g., Kubernetes, containerization) and lessen the complexity lying in the core banking systems design.

From this perspective, the specific problem that has been aimed to be solved in this doctoral study has been "How a core banking system with monolithic structure can be transformed into an MSA-based core banking system that is compliant with open banking standards?". For solving this problem, MSA-related techniques and best practices have been investigated and an MSA-based infrastructure model has been proposed by keeping performance efficiency in consideration. In the context of this proposed infrastructure model, a new technique has been developed particularly for managing distributed SAGA transactions among microservices by utilizing the Actor Model. This technique together with the proposed performance-efficient infrastructure constitutes our main academic contribution in our doctoral study.

Keywords: Core Banking Systems, Open Banking Standards, API, Monolith Architecture, Microservices Architecture, SAGA, Distributed SAGA Transactions Management, Actor Model



“Standing upon the shoulders of giants...”
Isaac Newton



*Yanımda olduđu her an değeri ni daha da derinden hissettiğim sevgili eşim
Derya'ya ve içimdeki sevgi pınarının kaynağı olan sevgili annem Firuzan
Hanımefendi'ye ithaf ediyorum.*

ÖNSÖZ

Bu doktora çalışmasında mikro servis mimarisi temelinde performans odaklı bir çekirdek bankacılık sistemi oluşturmak amacıyla bir yazılım mimarisi modeli ortaya konulmuştur. Bu model endüstriyel bir vaka çalışması kapsamında uygulanmış ve üzerinde yük testleri gerçekleştirilmiştir. Bu testlerin nasıl yapıldığı ve performans sonuçları bu tez dokümanında detaylarıyla paylaşılmıştır. Ayrıca mikro servisler arasındaki dağıtık işlemlerin (distributed transactions) yönetimini sağlayan SAGA tasarım deseninin aktör modeli ekseninde genişletildiği yeni bir algoritmik yöntem yine bu mimari model çerçevesinde geliştirilmiştir. **Aktör Esaslı SAGA** (Actorized-SAGA) olarak adlandırılabilen bu yöntem, komut-orkestrasyon ve kareografi olarak adlandırılan mevcut iki SAGA varyantında çözümü bulunmayan veri okuma-yazma senkronizasyonu (Örn.: dirty read) problemini bertaraf etmektedir. Bu yöntem ayrıca orkestrasyon SAGA varyantında merkezi koordinatör üzerinde oluşabilecek performans dar boğazını da ortadan kaldırmaktadır. Söz konusu yöntem prototiplenip doğruluğu Kuveyt Türk Katılım Bankası veri merkezinde sağlanan bir Kubernetes cluster'ında uygulamalı olarak teyit edilmiştir. Dolayısıyla bu doktora çalışmam ortaya koyduğu akademik katkısı ilaveten ticari bir bankada endüstriyel bir kullanım alanı bulması ve ürünleşmesi nedeniyle şahsım adına gurur verici bir endüstriyel doktora çalışması halini almıştır.

Bu vesileyle gerek eğitimim gerekse çalışmalarım sırasında sabırlı ve özverili yaklaşımlarıyla bana her anlamda destek olan kıymetli eşime, valideme, pederime ve ablalarım ayrı ayrı şükranlarımı sunmayı bir borç bilirim.

Ayrıca araştırmalarım ve çalışmalarım sırasında değerli akademik katkıları ve yönlendirmeleri nedeniyle danışman hocam Sayın Prof. Dr. Fatih BAŞÇİFTÇİ Beyefendi'lere şükranlarımı sunarım.

Son olarak doktora çalışmamda önerdiğim modeli Kuveyt Türk Katılım Bankası bünyesinde uygulamama ve nihai yapının ürünleşmesine vizyoner duruşlarıyla destek olan Architech Bilgi Sistemleri Genel Müdür Yardımcıları Sayın Hâdi RUBACI Beyefendi ile Sayın Zeynep Y.T. ALTAŞ Hanımefendi'ye, Analitik Bankacılık Müdürü Sayın Erhan ARIKAN Beyefendi'lere ve Kurumsal Mimari Müdürü Sayın Sinem Şen Hanımefendi'ye şükranlarımı sunarım.

Fikri AYDEMİR
Konya- Ocak 2024

İÇİNDEKİLER

ÖZET	iv
ÖNSÖZ.....	viii
İÇİNDEKİLER	ix
ŞEKİLLER LİSTESİ.....	xii
TABLolar LİSTESİ.....	xvi
SİMGELER VE KISALTMALAR	xvii
1. GİRİŞ	1
1.1. FinTek	5
1.1.1. FinTek hizmetleri ve sektörel kullanım alanları	6
1.1.2. FinTek ekosistemi.....	8
1.1.3. Dünya ve Türkiye’deki FinTek Yatırımları.....	11
1.2. Ödeme Hizmetleri Yönergesi (Payment Services Directive)	19
1.2.1. Ödeme hizmetleri yönergesi oluşum süreci.....	19
1.2.2. Ödeme hizmetleri yönergesi amaçları	20
1.2.3. Ödeme hizmetleri yönergesi hizmet sağlayıcıları.....	21
1.3. Açık Bankacılık	23
1.4. Application Programming Interface (API-Uygulama Programlama Ara Yüzü) .	26
1.4.1. API tanımı.....	27
1.4.2. API’nin tarihçesi	28
1.4.3. REST ve GraphQL.....	29
1.4.4. GraphQL teknolojisinin endüstriyel kullanım örnekleri.....	31
1.5. Açık Bankacılık API Standartları	35
1.5.1. Açık Bankacılık Standartlarının Kullanım Alanları	36
1.5.2. Veri alanları	41
1.5.3. Onay modelleri	42
1.5.4. Mesaj biçimleri	42
1.5.5. Kimlik doğrulama	42
1.5.6. Kullanıcı deneyimi.....	43
1.5.7. Hesap bilgilerine erişim rıza (Consent) uç noktaları (API’leri)	43
1.5.8. Hesaplar (Accounts) uç noktaları (API’leri).....	44
1.5.9. Ödeme başlatma (Payment Initiation) uç noktaları	47
1.6. Monolit Mimari.....	49
1.6.1. Tanım	49
1.6.2. Monolitlerin avantajları ve dezavantajları	51
1.6.3. Örnek monolit uygulamalar	53
1.7. Mikro Servis Mimarisi.....	56
1.7.1. Tanım	56
1.7.2. Mikro Servislerin karakteristik özellikleri	57
1.8. Mikro Servis Tasarım Kalıpları	62
1.8.1. API Geçidi (API Gateway) tasarım kalıbı	63
1.8.2. Devre kesici (Circuit Breaker) tasarım deseni	64

1.8.3. CQRS (Command Query Responsibility Segregation) tasarım deseni	65
1.8.4. SAGA tasarım deseni.....	67
1.9. Event-Bus.....	72
1.9.1. RabbitMQ	74
1.9.2. Apache Kafka	75
1.10. Önbellekleme (Caching)	78
1.11. ORM (Object Relational Mapping)	81
1.11.1. Dapper Mikro ORM.....	82
1.11.2. Entity Framework	83
1.12. Tez Çalışmasının Amacı.....	84
1.13. Tez Dokümanı Organizasyonu	87
2. KAYNAK ARAŞTIRMASI.....	89
2.1. FinTek Konulu Çalışmalar.....	89
2.2. Açık Bankacılık ve API Teknolojileri Konulu Çalışmalar	90
2.3. Mikro Servis ve Monolit Mimari Konulu Çalışmalar	93
2.4. Mikro Servis Mimarisi Temelli Sistemlerde Dağıtık SAGA İşlemlerinin (Transactionlarının) Koordinasyonu Konulu Çalışmalar	96
3. MATERYAL VE METOT	98
3.1. Problemin Tanımı	98
3.2. Hipotez.....	99
3.3. Metot.....	99
4. BANKACILIK İŞLEV SINIFLARI.....	103
4.1. Hesap ve Mevduat İşlevleri Sınıfı	104
4.2. Para Transfer İşlevleri Sınıfı.....	107
4.3. Kredi/Finansman İşlevleri Sınıfı.....	109
4.4. Kredi Kartı İşlevleri Sınıfı	111
4.5. Ödeme İşlevleri Sınıfı.....	112
4.6. Genel Bankacılık İşlevleri Sınıfı.....	114
4.7. Bölüm Özeti.....	115
5. TASARIM VE GELİŞTİRME ÇALIŞMALARI.....	116
5.1. Monolit Mimari Nedir?.....	117
5.2. Monolit BOA Mimarisine Genel Bakış	119
5.2.1. Bir BOA monolitinin anatomisi.....	119
5.2.2. Dağıtık BOA monolitleri	120
5.2.3. Dağıtık BOA monolitlerini yaygınlaştırma stratejileri	123
5.3. Mikro Servis Mimarisi Nedir?	125
5.4. Açık Bankacılık Mimarisi – Open Banking Architecture (OBA)	127
5.4.1. Açık bankacılığın ortaya çıkış süreci, genel felsefesi ve tanımı.....	127
5.4.2. OBA’da istemci kanallarının sınıflandırması	130
5.4.3. OBA’nın temel altyapı bileşenleri	131
5.4.4. OBA bileşenlerinin yaygınlaştırma (deployment) stratejisi	134
5.4.5. OBA’ya kuş bakışı.....	136
5.4.6. Diğer OBA bileşenleri	137
5.4.7. OBA’da request-response log (jurnal) altyapısı	142

5.4.8. OBA mimarisinin bütününe genel bakış.....	144
5.4.9. Kaynak kod yapılanması.....	147
5.5. Proof of Concept (PoC) Çalışmaları.....	152
5.5.1. Mobil şube canlandırılmalı PoC çalışması.....	153
5.5.2. Mikro servisler arası asenkron (publish/subscribe) iletişimi tesis eden PoC çalışması.....	156
5.6. Yük Testi Çalışmaları	158
5.6.1. Monolit BOA altyapısını barındıran test cluster'ının genel yapısı	160
5.6.2. Mikro servis mimarisi esaslı OBA altyapısını barındıran test cluster'ının genel yapısı	162
5.6.3. OBA üzerinde icra edilen yük testi konfigürasyonu.....	169
5.6.4. Yük testi sonuçları	170
6. DAĞITIK SAGA İŞLEMLERİNİN (TRANSACTIONS) AKTÖR MODELİ ESASLI YÖNETİMİ.....	176
6.1. Buluşun Teknik İçeriği	176
6.2. Buluşun Amacı	182
6.3. Buluşun Detaylı Açıklaması	184
6.5. Bölüm Özeti.....	205
7. SONUÇLAR	206
7.1. Tez Çalışmasının Genel Özeti	206
7.2. Yük Testi Değerlendirmesi.....	209
7.3. Aktör Modeli Esaslı Dağıtık İşlem (Transaction) Yönetiminin Sahada Uygulanışı	211
7.4. Mikro Servis Mimarisinin Endüstride Genel Kullanımı Hakkındaki Değerlendirme	216
KAYNAKLAR	219
EK-1: OBA Mimarisine Genel Bakış	233
EK-2: Docker-compose.yml	234
Docker-compose.override.yml	235
EK-3: Nakit Çekme SAGA Akışı Tanımı:	241
EK-4: Dağıtık SAGA Akışlarının Yönetiminin Sahada Uygulanışı	243

ŞEKİLLER LİSTESİ

Şekil 1.1. Monolit Mimarinin Temsili Görünümü	1
Şekil 1.2. Otonom ve Koordineli Olarak Çalışan Mikro Servislerin Temsili Görünümü	2
Şekil 1.3. Mikro Servis Mimarisinin Monolit Bir Sistem Üzerine Uygulanışı	3
Şekil 1.4. Mikro Servis Mimarisi Kullanılarak Tesis Edilmiş Bir Açık Bankacılık Platformu	4
Şekil 1.5. 2015 – 2019 Yılları Arasında FinTek Hizmetlerinin Müşteri Kullanım Seviyeleri (Anonim, 2019 (2)).....	7
Şekil 1.6. FinTek Hizmetlerinin Ülkeler Tarafından Kullanılma Oranları (Anonim, 2019 (2))	7
Şekil 1.7. FinTek Ekosistemi (Lee,2016).....	8
Şekil 1.8. Hesap Sağlayıcı FinTek’ler (Anonim, 2019 (3)).....	10
Şekil 1.9. Türkiye’de Para Transferi ve Ödeme Hizmetlerini Sunan FinTek’ler (Anonim, 2019 (3)).....	11
Şekil 1.10. FinTek Endüstrisi Hasılatının Yıllar Üzerinde Değişimi (Howarth, 2023) .	11
Şekil 1.11. FinTek Endüstrisi Hasılatının Yıllar Üzerinde Değişimi (Howarth, 2023) .	12
Şekil 1.12. Dünyada FinTek Yatırım Tutarının 2020 ile 2022 Arasında Değişimi (Shewale, 2024)	13
Şekil 1.13. Türkiye’de FinTek Yatırım Tutarlarının 2017 ile 2022 Yılları Arasında Değişimi (T.C. Cumhurbaşkanlığı Finans Ofisi, 2023).....	13
Şekil 1.14. Türkiye’de FinTek Sayılarının Şubat 2023 İtibariyle Durumu (T.C. Cumhurbaşkanlığı Finans Ofisi, 2023).....	14
Şekil 1.15. İngiltere’de FinTek’lere Yapılan Yatırım Tutarı ve Adedi Gelişim Grafiği (Anonim, 2021 (13)).....	15
Şekil 1.16. İngiltere’de FinTek’lere Yapılan Yatırım Tutarı ve Adedinin Çeyrek Dönemlerdeki Gelişim Grafiği (Anonim, 2021 (13)).....	15
Şekil 1.17. Kıta Bazında FinTek Yatırımlarının Dağılımı (Anonim, 2021 (13)).....	16
Şekil 1.18. 2021 Yılında FinTek Yatırımlarının Farklı Kıtalarda Dağılımı (Howarth, 2023).....	17
Şekil 1.19. 2021 Yılında FinTek Yatırımlarının Dünya Genelinde Sektör Bazlı Dağılımı (Howarth, 2023).....	17
Şekil 1.20. FinTek Yatırımlarında İlk 10 Ülke (Anonim, 2019 (3))	18
Şekil 1.21. 2021 Yılı İlk Yarısında FinTek’lere Yapılan Yatırımların En Yüksek Olduğu 5 Ülke (Anonim, 2021 (13))	19
Şekil 1.22. PSD2 Zaman Tüneli	20
Şekil 1.23. Ödeme Başlatma Hizmet Sağlayıcısı Modeli.....	22
Şekil 1.24. Hesap Bilgileri Hizmet Sağlayıcısı Modeli.....	23
Şekil 1.25. Geleneksel Bankacılık ve Açık Bankacılık (Anonim, 2019 (3)).....	24
Şekil 1.26. API Sayıları ve Geliştirici Deneyimine Göre Banka Grafiği (Anonim, 2023 (1))	25
Şekil 1.27. Yıllara Göre Bankaların API Sayılarındaki Değişim Tablosu (Anonim, 2023 (2))	26
Şekil 1.28. A Programının B Programınca Sunulan İşlemlere B Programı API’sini Çağırarak Erişmesi.....	27
Şekil 1.29. HTTP Protokolü Esaslı Web API Çağırımı	28

Şekil 1.30. Hesap Ek No'larını (Suffix) Getiren REST API Çağırımı	29
Şekil 1.31. Belirli bir Hesabın Belirli bir Ek No'suna Ait Hesap Bilgilerini Getiren REST API Çağırımı	30
Şekil 1.32. Hesap Hareketlerini Getiren REST API Çağırımı	30
Şekil 1.33. Hesap Hareketlerini Getiren GraphQL Tabanlı API Çağırımı	31
Şekil 1.34. GraphQL ve REST Teknolojilerinde API Çağırım Karmaşıklığının Temsili Resmi	32
Şekil 1.35. Ödeme Başlatma Hizmeti Genel Yapısı	38
Şekil 1.36. Hesap Bilgisi Servisi Genel Yapısı	40
Şekil 1.37. Üç Katmanlı Monolit Mimari	50
Şekil 1.38. Dağıtık Monolitlerden Oluşan bir Yazılım Sisteminin Kütüphane (DLL) ve Veri Tabanı Bakımından Bağımlılıkları	51
Şekil 1.39. SoundCloud Monolit Mimarisi	53
Şekil 1.40. BOA Monolit Mimarisi	54
Şekil 1.41. Uber Monolit Mimarisi	55
Şekil 1.42. Mikro Servis Mimarisi Benzetimi	57
Şekil 1.43. Monolit Mimaride Organizasyon Yapısının Dağılımı	58
Şekil 1.44. Mikro Servis Mimarisi Organizasyonel Dağılımı	59
Şekil 1.45. Mikro Servisler Arası Asenkron İletişim	60
Şekil 1.46. Monolit Mimariden Mikro Servis Mimarisine Geçişte Veri Tabanı Ayırımı	62
Şekil 1.47. API Geçidi Üzerinden Mikro Servis API'lerinin Çağırımı	64
Şekil 1.48. Devre Kesici Bileşen Kapalı Durumdayken	65
Şekil 1.49. Devre Kesici Bileşen Açık Durumdayken	65
Şekil 1.50. CRUD Aksiyonlarının Tek bir Veri kaynağı Üzerinde Uygulanışı	66
Şekil 1.51. CQRS Tasarım Deseninde Yazma ve Okuma Veri Kaynaklarının Fiziksel ve Mantıksal Ayırıştırılması	66
Şekil 1.52. Bir E-Ticaret Sistemi Kapsamındaki Mikro Servislerin Orkestrasyon Esaslı SAGA Deseni Çerçevesinde Etkileşim Senaryosu (Kanber, 2021)	68
Şekil 1.53. Orkestrasyon Esaslı SAGA Deseni Çerçevesinde Roll-back (Geri alma) Senaryosu İşleyişi (Kanber, 2021)	69
Şekil 1.54. Bir E-Ticaret Sistemi Kapsamındaki Mikro Servislerin Koreografi Esaslı SAGA Deseni Çerçevesinde Etkileşim Senaryosu (Kanber, 2021)	71
Şekil 1.55. Koreografi Esaslı SAGA Deseni Çerçevesinde Roll-back (Geri alma) Senaryosu İşleyişi (Kanber, 2021)	72
Şekil 1.56. Event-Bus Genel İşleyiş Yapısı (Torre ve ark., 2021)	73
Şekil 1.57. RabbitMQ Genel İşleyiş Yapısı (Roy, 2017)	74
Şekil 1.58. Event-Bus Genel İşleyiş Yapısı	76
Şekil 1.59. Uygulama ve Veri Tabanı Arasındaki ORM Bağlantısı (Edpresso Team, 2021)	81
Şekil 1.60. ORM Araçları Performans Analiz Sonuçları	83
Şekil 1.61. Entity Framework Temel Yaklaşımları (Alagöz, 2018)	84
Şekil 1.62. İktisadi Platform ve Öğeleri	85
Şekil 4.1. Hesap Bilgisi Edinme İşlevi Etkileşimleri	105
Şekil 4.2. Hesap Hareketleri Sorgulama Etkileşimleri	106
Şekil 4.3. Hesap Bakiyesi Sorgulama Etkileşimleri	106

Şekil 4.4. a) Hesap Etkileşimleri ile ilgili Domain Nesnelerini Gösteren UML Class Diyagramı b) Bakiye Etkileşimleri ile ilgili Domain Nesnelerini Gösteren UML Class Diyagramı	107
Şekil 4.5. Para Transferi Sırasında Meydana Gelebilecek Etkileşimler	108
Şekil 4.6. Para Transferi Domain Nesnelerini Temsil eden Class Diyagramı	109
Şekil 4.7. Fon Talebi ile İlişkili Aktör Etkileşimleri	110
Şekil 4.8. Kredi/Finansman Domain Nesnelerini Temsil eden Class Diyagramı	110
Şekil 4.9. Kart Detay Bilgisi ile İlişkili Aktör Etkileşimleri	111
Şekil 4.10. Para Transferi Domain Nesnelerini Temsil eden Class Diyagramı	112
Şekil 4.11. Ödeme Başlatma ile İlişkili Aktör Etkileşimleri	113
Şekil 4.12. Planlı Ödeme Bilgi İsteği İşlevi ile İlişkili Aktör Etkileşimleri	113
Şekil 4.13. Bankacılık Ürün Bilgeleri Listesini Getirmek için Meydana Gelen Etkileşimler	114
Şekil 5.1. Monolit Mimarinin Temsili Görünümü	117
Şekil 5.2. Yekpare bir Monoliti oluşturan yazılım katmanları	118
Şekil 5.3. Dağıtık Monolitlerden Oluşan bir Yazılım Sisteminin Kütüphane (DLL) ve Veri Tabanı Bakımından Bağımlılıkları	119
Şekil 5.4. Bir BOA Monolitinin Yekpare Yapısının Temsili Görünümü	120
Şekil 5.5. Tek bir BOA Monolitinin Genel BOA Sistemindeki Tek bir Yap-Boz Parçasına Benzetimi	121
Şekil 5.6. BOA Monolitlerinin Dağıtık bir Sistem Olarak Kenetlenmiş Temsili Görünümü	122
Şekil 5.7. BOA Monolitlerinin Dedike Network Bölümlerinde Koşumu	123
Şekil 5.8. Bir Host'ta Birden Çok Uygulama Barındıran Yaygınlaştırma Stratejisi (Richardson, 2016)	124
Şekil 5.9. Bir Sanal Makinede Bir Uygulama Barındıran Yaygınlaştırma Stratejisi (Richardson, 2016)	125
Şekil 5.10. Monolit Yaklaşımından Mikro Servis Esaslı Yaklaşımına Dönüşüm	126
Şekil 5.11. Bağımsız Ancak Koordineli Çalışan Bisiklet Parçalarının Mikro Servislere Benzetimi	126
Şekil 5.12. Geleneksel Bankacılık Ekosistemi	127
Şekil 5.13. Açık Bankacılık Ekosistemi	128
Şekil 5.14. Mobil uygulamaların Twitter servislerine yine Twitter tarafından sağlanan API'ler üzerinden erişimi	129
Şekil 5.15. Client (İstemci) Uygulamalarının API'ler Vasıtasıyla Açık Bankacılık Mikro Servislerine Erişimi	129
Şekil 5.16. OBA'da İstemcilerin Teknoloji ve Kanal Bazlı Mantıksal Sınıflandırması	130
Şekil 5.17. OBA'da Temel Altyapı Bileşenlerinin bir API Çağırımını (API Request) İşleyişi	132
Şekil 5.18. OBA Mikro Servislerinin Konteyner (Container) Teknolojisi Tabanlı Yaygınlaştırma Stratejisi	135
Şekil 5.19. OBA Mimarisine Kuş Bakışı	136
Şekil 5.20. Bir BFF'in OPA üzerinden API Yetki Kontrolü (Authorization) Yapması	138
Şekil 5.21. Consul ve Vault'un Mikro Servis Konfigürasyonunu besleyişi	140

Şekil 5.22. OBA’da Uygulama Log Kayıtlarının FluentBit Teknolojisini Kullanarak İşlenmesi ve Kayıt Altına Alınması.....	142
Şekil 5.23. OBA’da Jurnal Kayıtlarının Toplanıp İşlenmesi ve Kayıt Altına Alınması	143
Şekil 5.24. OBA Mimarisinin Bütününe Genel Bakış (Daha yüksek çözünürlüklü bir imaj için EK-1’e bakılması rica olunur.)	145
Şekil 5.25. OBA (Çatı Proje) İçeriği	148
Şekil 5.26. OBA.Core Solution’ı ve Barındırdığı Class-Library Projeleri (.NET 7) ...	150
Şekil 5.27. Mobil Şube Canlandırılmalı PoC	153
Şekil 5.28. Asenkron İletişimi Örnekleyen PoC.....	156
Şekil 5.29. BOA Test Cluster’ının Genel Yapılanması.....	160
Şekil 5.30. OBA Altyapısının Test Edildiği Test Cluster’ının Genel Yapılanması	163
Şekil 5.31. Monolit BOA Altyapısının Ortalama Verim Sonuçları	172
Şekil 5.32. Monolit BOA Altyapısının Ortalama Cevap Süreleri	172
Şekil 5.33. Mikro Servis Mimarisi Esaslı OBA Altyapısının Ortalama Verim Sonuçları	174
Şekil 5.34. Mikro Servis Mimarisi Esaslı OBA Altyapısının Ortalama Cevap süresi .	175
Şekil 6.1. Komut-Orkestrasyon Esaslı SAGA Tasarım Deseni.....	177
Şekil 6.2. Koreografi Esaslı SAGA Tasarım Deseni.....	177
Şekil 6.3. Nakit Çekme SAGA Akışı	179
Şekil 6.4. Döviz Alış SAGA Akışı	179
Şekil 6.5. Aktörler Arası Mesajlaşma Yapısı	185
Şekil 6.6. SAGA Akışlarının Aktörler üzerinden Çalıştırılması (M harfi Mikro Servisi temsil eder)	186
Şekil 6.7. Bir aktörün SAGA Akışı İşletimi Sırasında Kullandığı State-Machine Diyagramı	193
Şekil 6.8. Buluş Kapsamındaki Algoritmik Yöntemin Akış Diyagramı	195
Şekil 6.9. Nakit Çekme SAGA Akışının Aktör Marifetiyle İşletilmesi	198
Şekil 6.10. Döviz Alış SAGA Akışı Talebinin Aktörün Mesaj Kuyruğuna Atılması..	202
Şekil 6.11. Döviz Alış SAGA Akışı Talebinin Aktör Tarafından İşletilmesi	204
Şekil 7.1. Mikro Servis Temelli OBA Altyapısının Sahada Uygulanış Fazları	207
Şekil 7.2. Hesaba Para Transferi SAGA akışı	208
Şekil 7.3. Mikro Servis Temelli OBA ile Monolit BOA’nın Ortalama Verim Karşılaştırması	210
Şekil 7.4. Mikro Servis Temelli OBA ile Monolit BOA’nın Ortalama Cevap Sürelerinin Karşılaştırması	211
Şekil 7.5. Buluşun Sahada Uygulanışını Yansıtan Sistem Mimarisi (Daha büyük bir görünüm için EK-4’e bakınız)	214
Şekil 7.6. Mikro Servis Mimarisi ile Monolit Mimarinin Karşılaştırması	217

TABLÖLER LİSTESİ

Tablo 1.1. Fintek Hizmetlerinin Ülke ve Sektör Özelinde Kullanım Oranları (Howarth, 2023)	8
Tablo 1.2. API Standartlarında Kullanım Alanları	36
Tablo 1.3. Open Banking UK Rıza Uç Nokta Standartları (Gyara, 2019).....	44
Tablo 1.4. Berlin Group Rıza Uç Nokta Standartları (Anonim, 2023 (3))	44
Tablo 1.5. Open Banking UK Hesap Bilgileri Uç Nokta Standartları (Michael, 2017 (1))	45
Tablo 1.6. Berlin Group Hesap Bilgileri Uç Nokta Standartları (Anonim, 2023 (3)) ...	46
Tablo 1.7. Open Banking UK Ödeme Başlatma Uç Nokta Standartları (Michael, 2017 (2))	47
Tablo 1.8. Berlin Group Ödeme Başlatma Uç Nokta Standartları (Anonim, 2023 (3))	48
Tablo 1.9. Monolit Mimari Avantajları (Richardson, 2019; Bucchiarone, 2018)	52
Tablo 1.10. Monolit Mimari Dezavantajları (Richardson, 2019; Bucchiarone, 2018)..	52
Tablo 4.1. Bankacılık İşlev Sınıfları (a).....	103
Tablo 4.2. Bankacılık İşlev Sınıfları (b).....	104
Tablo 5.1. Bir BOA Monolitinin Koştuğu bir Sanal Makinanın Haiz Olduğu Konfigürasyon	161
Tablo 5.2. BOA'ya Uygulanan 6 Ayrı Yük Testi Koşumlarının Kapsamı	161
Tablo 5.3. Tek bir APISIX POD'unun Kapasite Konfigürasyonu.....	164
Tablo 5.4. Tek bir Keycloak POD'unun Kapasite Konfigürasyonu	165
Tablo 5.5. Tek bir OPA POD'unun Kapasite Konfigürasyonu	166
Tablo 5.6. Tek bir Sample-BFF POD'unun Kapasite Konfigürasyonu	167
Tablo 5.7. Kubernetes Test Cluster'ında Konumlandırılan OBA Bileşenlerinin Herbirinde Gerçekleşen Toplam Kaynak Kullanımları	168
Tablo 5.8. OBA ve BOA'nın Toplam Kaynak Kullanımı Bakımından Karşılaştırması	168
Tablo 5.9. OBA için Uygulanan Yük Testi Koşumları.....	169
Tablo 5.10. Monolit BOA Altyapısına Tatbik Edilen Yük Testi Sonuçları.....	171
Tablo 5.11. Açık Bankacılık Mimarisi (OBA) için Yük Testi Sonuçları	173

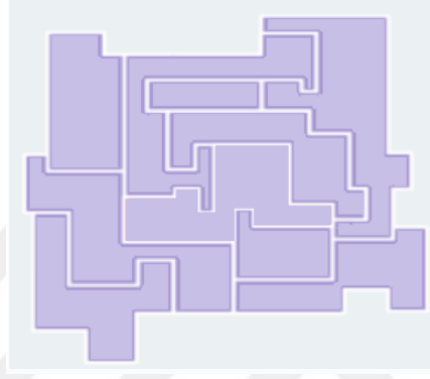
SİMGELER VE KISALTMALAR

ABP	Açık Bankacılık Platformu
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface (Uygulama Programlama Ara yüzü)
APP	Application
ATM	Automated Teller Machine
BFF	Backend for Frontend
BDDK	Bankacılık Düzenleme ve Denetleme Kurumu
BOA	Business Oriented Architecture
CNCF	Cloud Native Computing Foundation
CORE	Centralized Online Real-Time Environment
CPU	Central Processing Unit
DAPR	Distributed Application Runtime
DLL	Dynamically Linked Library
DMZ	Demilitarized Zone
EBF	European Banking Federation
FinTek	Finansal Teknoloji Firması
GKD	Güçlü Kimlik Doğrulama
gRPC	Google Remote Procedure Call
HBHS	Hesap Bilgisi Hizmeti Sağlayıcısı (Hesap Hizmeti Bilgisini Sunan Tüzel Kişi)
HHS	Hesap Hizmet Sağlayıcısı (Nezinde ödeme hesabı bulunan ödeme hizmeti sağlayıcısı)
I/O	Input/Output (Girdi/Çıktı)
İB	İşlem Bilgisi
İDK	İşlem Doğrulama Kodu
JSON	Java Script Object Notation
MA	Monolithic Architecture (Monolit Mimari)
MQTT	MQ Telemetry Transport
MSA	Micro Services Architecture (Mikro Servis Mimarisi)
MVC	Model-View-Controller
NLB	Network Load Balancer
OBA	Open Banking Architecture

ÖBHS	Ödeme Emri Başlatma Hizmet Sağlayıcısı (Ödeme emri başlatma hizmetini sunan tüzel kişi)
ÖDK	Ödeme Kuruluşu
OAuth	Open Authentication
OPA	Open Policy Agent
OS	Operating System (İşletim Sistemi)
ÖH	Ödeme Hesabı
ÖHK	Ödeme Hizmeti Kullanıcısı (Müşteri)
ÖHS	Ödeme Hizmet Sağlayıcısı
ÖHVPS	Ödeme Hizmetlerinde Kullanılan Veri Paylaşım Servisleri
PoC	Proof of Concept
PÖS	Perakende Ödeme Sistemi
PSD2	Payment Services Directive 2 (Ödeme Hizmetleri Önergesi 2. Sürümü)
REST	Representational State Transfer
RPC	Remote Procedure Call
SAGA	Software Automation, Generation, and Administration
SM	Sanal Makine
SMS	Short Message Service
SDK	Software Development Kit
STOMP	Simple (or Streaming) Text Orientated Messaging Protocol
TPP	Third Party Provider (Üçüncü Parti Hizmet Sağlayıcı – Yani FinTek Firması)
TCMB	Türkiye Cumhuriyet Merkez Bankası
UG	Uygulama Geliştirme
UUID	Universally Unique Identifier
VU	Virtual User
VM	Virtual Machine
YÖS	Yetkili Ödeme Hizmeti Sağlayıcısı (Ödeme emri başlatma hizmeti ve/veya hesap bilgisi hizmeti sunan tüzel kişiliklerin genel adı)
WAF	Windows Application Firewall

1. GİRİŞ

Yazılım mühendisliğinde yaygın olarak karşılaşılan tasarım problemlerine çözüm getirmek amacıyla ortaya konulan mimari yaklaşımlar mevcuttur. Bu yaklaşımlardan en gelenekseli olan *Monolit Yazılım Mimarisinde (Monolithical Architecture-MA)* bir yazılım sistemini oluşturan tüm karmaşık bileşenler Şekil 1.1’de temsili olarak gösterilen yap-boz benzeri bir yapıda yekpare bir yazılım sistemini oluşturmak üzere kenetlenerek bir araya getirilirler.

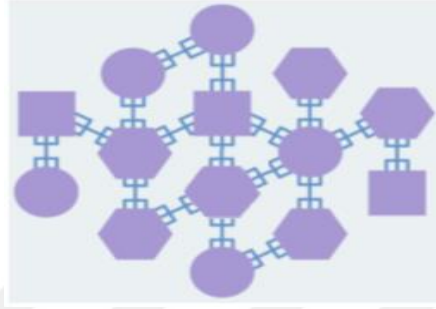


Şekil 1.1. Monolit Mimarinin Temsili Görünümü

Monolit mimaride oluşturulan bir yazılım sistemi tek tip bir geliştirme teknolojisini (yani programlama dilini) barındıran tek bir kod tabanı üzerinde yine tek bir veri tabanını kullanacak şekilde yekpare bir yapı olarak geliştirilir. Bu yapı üzerinde bir güncelleştirme gerektiğinde veya üzerindeki iş yükü artışı nedeniyle donanım kapasitesinin artırılması (yani ölçeklenmesi) gerektiğinde yapının bütünüyle kapatılıp açılması gerekebilmekte ve dolayısıyla müşteri memnuniyetsizliğine neden olabilecek uzun sistem kesintileri yaşanabilmektedir. Diğer yandan yapının tek tip bir teknoloji ile geliştirilmiş olması yeni teknolojilere adaptasyonu güçleştirmektedir. Bu bakımdan monolit mimari günümüz şartlarındaki büyük ölçekli yazılım sistemlerinin tasarımında tercih edilebilecek bir nitelikte değildir zira yekpare monolit yapı büyüdükçe sistemin genel karmaşıklığını yönetmek zorlaşmaktadır (Richardson, 2019).

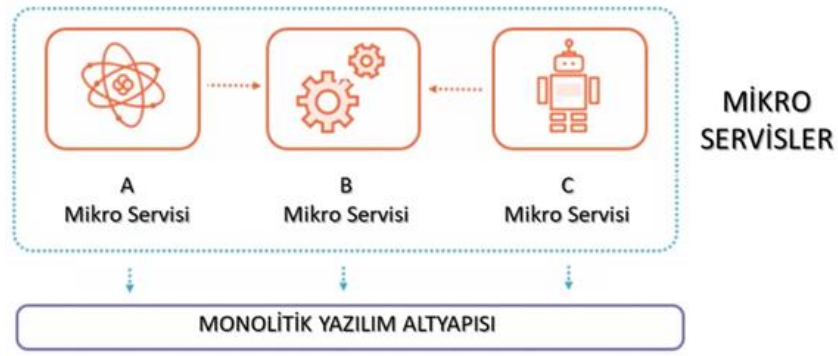
Mikro Servis Mimarisi (Micro Services Architecture-MSA) ortaya koyduğu çözümler bakımından monolit mimarinin dezavantajlarını ortadan kaldırabilecek daha modern bir yazılım mimarisi ortaya koyar (Fowler ve Lewis, 2014). Bu mimaride yazılım sistemini oluşturan bileşenler birbirinden bağımsız olarak geliştirilip, test edilip

yaygınlaştırılabilen otonom bileşenler olarak tasarlanmak suretiyle yazılım sisteminin bütünündeki karmaşıklığın yönetimi kolaylaştırılır. Bu yapıdaki bileşenler her ne kadar işlevlerini birbirlerinden bağımsız (otonom) olarak icra ediyor olsalar da birbirleri ile koordineli bir ahenk içerisinde çalışırlar (Richardson, 2019; Fowler ve Lewis, 2014). Bu bakımdan monolit mimaride birbirine sıkı sıkıya kenetlenmiş yap-boz benzeri yapının yerini otonom ancak koordineli olarak çalışan bileşenlerin (yani mikro servislerin) ahengi alır. Bu durum Şekil 1.2’de temsili olarak gösterilmektedir.



Şekil 1.2. Otonom ve Koordineli Olarak Çalışan Mikro Servislerin Temsili Görünümü

Sanallaştırma teknolojilerinde ve bulut hesaplamadaki gelişmeler doğrultusunda ortaya çıkmış olan MSA, monolit mimarinin dezavantajlarını ortadan kaldırmasının yanı sıra bulut bilişim platformları üzerinde çalışabilecek uygulama geliştirmeye de imkân tanıyan bir dağıtık yazılım mimarisi ortaya koymaktadır. MSA’da, her biri temel düzeyde (atomik) işleve haiz olan otonom mikro servisler geniş çaplı dağıtık bir yazılım sistemini oluşturmak üzere bir araya gelerek monolit yazılım altyapısının üzerine aşağıda Şekil 1.3’te temsil edildiği gibi kurulabilirler. Bu servisler birbirinden bağımsız ve otonom yapıları nedeniyle şelale metodolojisi yerine daha modern *Agile* (çevik) yazılım geliştirme metodolojileri kullanılarak parça parça geliştirilip yaygınlaştırılabilir ve hem yatayda hem de düşeyde ölçeklenebilirler (Richardson, 2019). Bu bakımdan mikro servisler, hâlihazırdaki monolit altyapı sistemlerini bütünüyle sil-baştan yeniden geliştirmek yerine ilave bir katman gibi giydirilmek suretiyle değişiklik maliyetlerini minimize edecek akılcı bir çözüm olarak ön plana çıkmaktadır.



Şekil 1.3. Mikro Servis Mimarisinin Monolit Bir Sistem Üzerine Uygulanışı

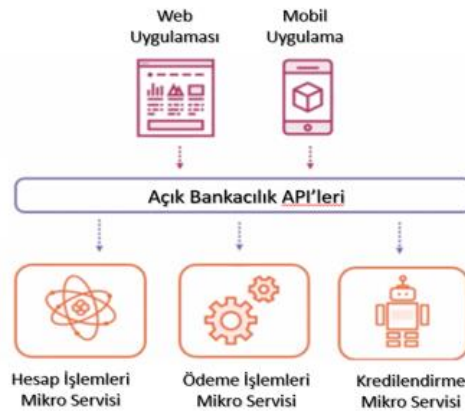
Temelleri 1970'lere dayanan ve literatürde “*Centralized Online Real-Time Environment (CORE) Banking Systems*” olarak ifade edilen sistemler, bankaların sunduğu tüm servislerin uygulama mantığını barındıran yazılım sistemleri olması bakımından finansal işlemlerin devamlılığının sağlanması adına büyük önem arz ederler. Türkçe’de *Çekirdek Bankacılık Sistemleri* veya *Temel Bankacılık Sistemleri* olarak adlandırılabilen bu sistemler tarafından sağlanan servislere örnek olarak mevduat yönetimi, para transferleri, kredilendirme, transaction (işlem) yönetimi ve raporlama servisleri verilebilir (Haralayya, 2021; Gartner, 2011). Bu tez dokümanının geri kalanında bankacılık sistemi denildiğinde işaret edilen kavram esasen *Temel Bankacılık Sistemi (CORE Banking System)* olarak anlaşılmalıdır.

Günümüz bankacılık sistemlerinin pek çoğu bulut bilişimin yaygınlaşmasından önceki dönemde monolit mimari esaslı olarak tasarlanıp geliştirilmiştir (Megargel ve ark., 2020). Genel itibarıyla *Şelale* (waterfall) yazılım geliştirme yöntemine göre geliştirilmiş olan bu monolit bankacılık sistemlerinin (Örnek: Main-frame bankacılık sistemleri) günümüzün bulut-tabanlı (cloud-native) yazılım geliştirme yaklaşımında kullanılan “Kubernetes” ve “Container” gibi donanım bağımsız sanallaştırma teknolojileri üzerinde çalışacak şekilde uyarlanması oldukça maliyetli ve güç bir iştir (Fritzsche ve ark., 2019; Chen ve ark., 2017; Fowler ve Lewis, 2014).

Diğer yandan monolit mimari olsun mikro servis mimarisi olsun bankacılık yazılım sistemlerinin Trendyol, hepsiburada gibi dijital platformlarca erişilmesi günümüzde kaçınılmaz bir gereklilik haline gelmiştir zira bu platformlar bankalara ciddi nakit akışına neden olmaktadır. Günümüzde bu erişim pek çok banka tarafından web servisleri üzerinden sağlanmakta olup bunun bazı dezavantajları mevcuttur. Bu dezavantajların en önemlisi web servislerine erişimin yalnızca kullanıcı adı ve şifre esası üzerine kurgulanmış tek faktörlü bir kimlik doğrulama yöntemi kullanılarak yapılıyor

olmasıdır. Kullanıcı adı ve şifre bilgileri Finansal Teknoloji Firmalarına (FinTek'lere) her ne kadar güvenli bir şekilde iletiliyor olsa da deşifre olma riski olması bakımından bankalar için güvenlik zafiyeti oluşturmaktadır. Diğer yandan web servisleri üzerinden sunulan bankacılık hizmetleri belirli bir standart dâhilinde olmayabilmekte ve buna bağlı olarak bankaların her FinTek için farklı birer web servisi geliştirmesi gerekmektedir. Bu da bankalara ciddi bir zaman ve iş gücü kaybı olarak yansımaktadır (Bucchiarone, 2018).

Bu dezavantajları ortadan kaldırmak amacıyla dünya genelinde *Açık Bankacılık (Open Banking)* adı altında standartlaştırma çalışmaları gerçekleştirilmektedir (Gozman ve ark., 2018). Bu çalışmalardan belki de en önemlisi ve bilineni Avrupa birliği tarafından geliştirilip kabul edilen ve *Ödeme Hizmetleri Şartnamesi2 (Payment Services Directive2–PSD2)* olarak adlandırılan şartnamedir (Anonim, 2015). Bu şartname kapsamında FinTek uygulamalarının bankaların yazılım altyapılarına erişimi web servislerinin aksine güvenliği tesis edilmiş ve standartlaştırılmış *Uygulama Programlama Ara Yüzleri (Application Programming Interfaces-APIs)* aracılığı ile sağlanmaktadır. Bu bağlamda FinTek'ler tarafından geliştirilen uygulamalar işlevlerini yerine getirirken bankaların web servislerini kullanmak yerine bankalar tarafından açık bankacılık standartları kapsamında sunulan daha güvenli API'leri çağıracaklardır (Premchand ve Choudhry, 2018). Bu bağlamda açık bankacılık ölçütlerinde oluşturulacak API'ler üzerinden erişilen ve mikro servis mimarisi esasına göre tasarlanmış bir bankacılık yazılım altyapısı aşağıda Şekil 1.4'te görüldüğü gibi temsil edilebilir. Bu Şekilden anlaşılabacağı üzere açık bankacılık API'leri FinTek'lerce geliştirilen mobil/web uygulamaları ile mikro servisler arasında köprü görevi görmektedir.



Şekil 1.4. Mikro Servis Mimarisi Kullanılarak Tesis Edilmiş Bir Açık Bankacılık Platformu

PSD2 gibi açık bankacılık standartları, bankacılık sistemlerince sağlanan servislere erişimin standartlaşmış API'ler vasıtasıyla nasıl yapılabileceği konusunda kurallar ortaya koyarken bu API'lere altyapı sağlayan temel bankacılık sistemlerinin (Core Banking Systems) altyapı mimarisi konusunda bir standart ortaya koymamaktadır. Bu bakımdan doktora çalışmamız esasen bu kapsamındaki eksikliği giderici bir niteliktedir. Buradan hareketle doktora çalışmamız kapsamında çözüm bulmaya çalıştığımız **problem** şu şekilde ifade edilebilir:

“Bir bankanın sahip olduğu monolit yapıdaki temel bankacılık sistemi, Açık Bankacılık Standartlarına uyumlu bir bankacılık platformuna Mikro Servis Mimarisi temelinde nasıl dönüştürülebilir?”

Bu problemin çözümü için temelinde mikro servis mimarisini esas alan bir yazılım sisteminin ihtiyaçları ortaya konulmuş, bu ihtiyaçları karşılamak için gerekli teknikler ve tasarım kalıpları araştırılıp uygulanabilirliği tespit edilmiş ve bu doğrultuda bir mimari model geliştirilmiştir. Geliştirilen bu modelin doğruluğu Kuveyt Türk Katılım Bankası bünyesinde bir endüstriyel bir vaka çalışması kapsamında uygulamalı olarak ortaya konmuştur.

Bu bölümün geri kalanında bu problemin çözümü için doktora çalışmamız kapsamında araştırdığımız konular açıklanmıştır. Bu konular sırasıyla aşağıdaki gibidir:

- a. FinTekler
- b. Ödeme Hizmetleri Yönergesi
- c. Açık Bankacılık
- d. API Teknolojileri (REST ve GraphQL)
- e. Açık Bankacılık API Standartları (UK ve Berlin Grubu Standartları)
- f. Mikro Servis Mimarisi ve Monolit Mimari
- g. Ön bellekleme (Caching)
- h. Object Relational Mapping (ORM) Teknolojileri

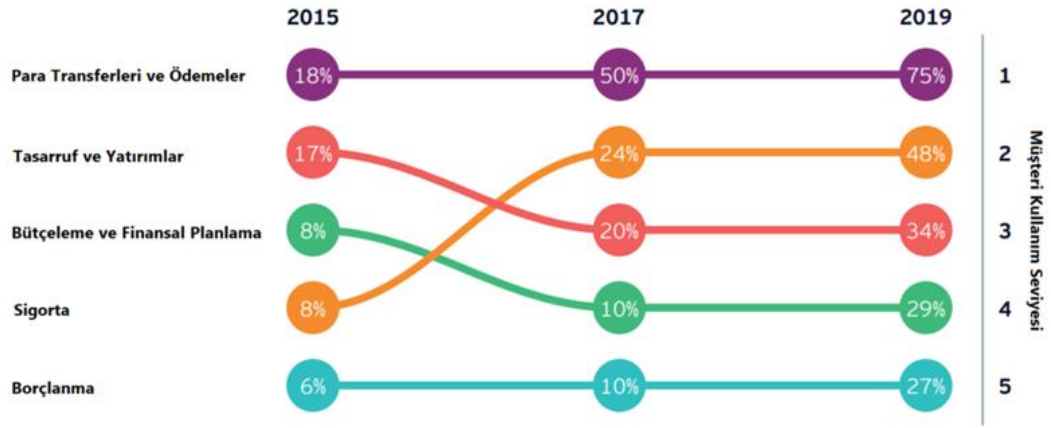
1.1. FinTek

Günümüzde teknolojinin her alanda hızla gelişmesiyle birlikte birçok sektörde olduğu gibi finans sektöründe de dijitalleşme hız kazanmış ve daha kaliteli hizmet verebilmek amacıyla birçok teknolojik atılım yapılması gerekmiştir (Devrim, 2007). Bu amaç doğrultusunda ortaya çıkmış olan FinTek terimi finansal ve teknoloji kelimelerinin birleştirilmesiyle ortaya çıkmış olup sözlük anlamı itibarıyla bankacılık hizmetlerini ve

finansal hizmetleri desteklemek veya etkinleştirmek amacıyla kullanılan bilgisayar programları ve teknolojileri bütünü olarak ifade edilebilir (Oxford Dictionary, 2021). FinTek terimi bu sözlük anlamına ek olarak ikinci bir anlam da yüklenmektedir. Bu ikinci anlam özellikle bilişim teknolojilerini kullanarak finansal faaliyetleri iyileştirmeyi amaçlayan yeni bir finans sektörünü ve dolayısıyla bu sektörde faaliyet gösteren firmaları ifade etmek amacıyla kullanılmaktadır (Schueffel, 2016). Bu ikinci anlam özellikle Microsoft firmasının kurucularından Bill Gates'in 1994'te ortaya koyduğu "Bankacılık gereklidir, bankalar değil" sloganı çerçevesinde ortaya çıkmış ve yaygınlaşmıştır. Bu bağlamda FinTek'ler bankalara alternatif olmakla beraber aynı zamanda bankaların kendini geliştirmesine ve müşterilerine yeni fırsatlar sunmasına imkân sağlayan finansal teknoloji firmaları olarak ifade edilebilir. Bu tezin geri kalanında FinTek ifadesiyle bu manaya işaret edilmektedir.

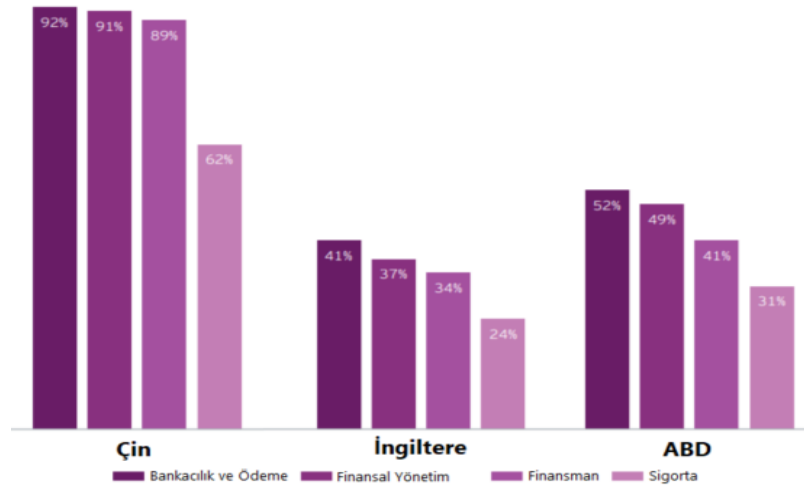
1.1.1. FinTek hizmetleri ve sektörel kullanım alanları

Günümüzde FinTek'lerin aktif olarak hizmet sağladığı sektör alanlarına örnek olarak dijital ödemeler, para transferleri, tasarruf ve yatırım sektörü ile sigortacılık sektörü verilebilir. EY adlı şirketin yaptığı bir araştırmaya göre bu sektörler içerisinde FinTek'lerin en çok hizmet sunduğu sektör para transferi ve ödeme hizmetleri olup 2015 yılı içerisinde sektör payı %18 oranındayken; 2019 yılında %75 seviyesine ulaşmıştır. Para transferi ve ödeme hizmetlerini, %48'lik bir oran ile sigortacılık hizmetleri takip etmektedir. Sigortacılık hizmetlerinin sektör payı 2015 yılı içerisinde %8 iken 2019 yılında %48 seviyelerine ulaşmıştır. Sigortacılık hizmetlerini sırasıyla tasarruf ve yatırımlar, bütçeleme ve finansal planlama ve borçlanma hizmetleri takip etmektedir (EY, 2019). Bu sektör alanlarının 2015-2019 yılları arasında müşteri kullanım oranlarına göre sıralanmış hali ise Şekil 1.5'te görülmektedir.



Şekil 1.5. 2015 – 2019 Yılları Arasında FinTek Hizmetlerinin Müşteri Kullanım Seviyeleri (Anonim, 2019 (2))

EY şirketinin 2019 yılında yaptığı bir başka araştırmada FinTek'lerin yukarıda bahsi geçen sektörler kapsamında hangi ülkede en yoğun iş yaptıkları incelenmiştir. Buna göre bu sektörlerin en yoğun iş yapılan ülkenin %61'lik oranla Çin olduğu belirtilmektedir. Çin'i takip eden diğer ülkeler sırasıyla Amerika Birleşik Devletleri (%23) ve İngiltere'dir (%18). Aynı araştırma kapsamında ortaya konulan bir diğer bilgi de Çin'in en çok kullandığı hizmet sektörü %92'lik oranla bankacılık ve ödemelerdir (Anonim, 2019 (2)). Şekil 1.6'da FinTek hizmetlerinin Çin, Amerika Birleşik Devletleri ve İngiltere tarafından kullanım yoğunluk oranları görülmektedir.



Şekil 1.6. FinTek Hizmetlerinin Ülkeler Tarafından Kullanılma Oranları (Anonim, 2019 (2))

2023 yılı özelinde FinTek hizmetlerinin ülke bazlı kullanım alanları incelendiğinde ise Çin'in yine en önde olduğu ve onu sırasıyla A.B.D., Meksika, Güney Amerika ve İngiltere'nin takip ettiği görülmektedir. Bu bakımdan İngiltere'nin

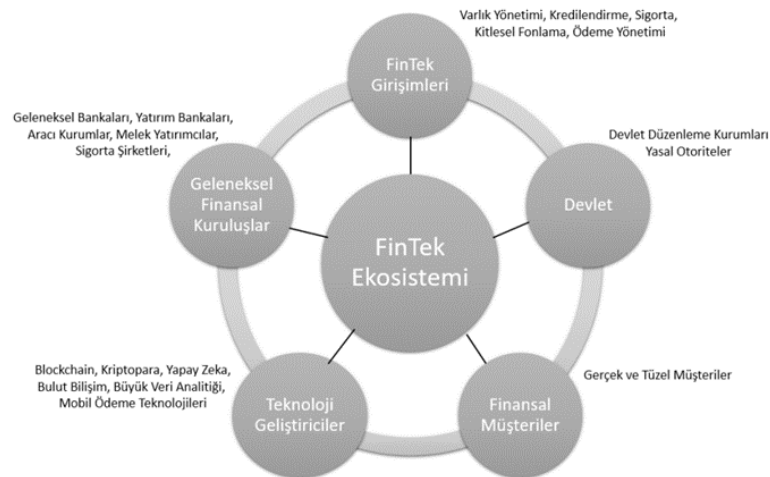
sıralamada genel olarak geride kalmaya başladığı anlaşılmaktadır (Howarth, 2023). Bu durum aşağıda Tablo 1.1’de ülke ve sektör özelinde özetlenmektedir.

Tablo 1.1. Fintek Hizmetlerinin Ülke ve Sektör Özelinde Kullanım Oranları (Howarth, 2023)

Ülke	Para Transferleri ve Ödemeler	Finansal Planlama	Yatırımlar	Sigorta
Çin	%92	%91	%89	%62
A.B.D.	%52	%49	%41	%31
Meksika	%49	%36	%31	%23
Güney Afrika	%47	%43	%34	%26
İngiltere	%41	%37	%34	%24

1.1.2. FinTek ekosistemi

FinTek firmalarının büyümesinde ve çoğalmasında stabil bir FinTek ekosisteminin varlığı önem arz etmektedir. FinTek’ler ve etrafındaki etkileşimde bulundukları FinTek ekosistemi Şekil 1.7’de görüldüğü gibi 5 temel unsurdan oluşmaktadır (Lee, 2016).



Şekil 1.7. FinTek Ekosistemi (Lee,2016)

Şekil 1.7’de görülen FinTek ekosistemi içerisindeki en temel unsur olan FinTek girişimleri finansal teknoloji alanında ürün ve hizmet geliştiren firmaları temsil eder. Kısaca FinTek olarak da ifade edilen bu firmalar kendi içerilerinde varlık yönetimi,

kitlesel fonlama, sigortacılık, ödeme sistemleri farklı çalışma alanlarında faaliyet gösterebilmektedir.

FinTek ekosistemi içerisindeki ikinci temel unsur olan teknoloji geliştiriciler, FinTek'ler tarafından sunulan finansal teknoloji odaklı yazılım uygulamalarını (aplikasyonları) geliştiren yazılım geliştiricilerini ifade eder. Bu geliştiricilerin çalışma alanlarına yapay zekâ, blok zinciri (blockchain), sanal gerçeklik, kripto-para ve mobil ödeme örnek olarak verilebilir.

FinTek ekosistemi içerisindeki üçüncü derecede önemli unsur ise *Bankacılık Denetleme ve Düzenleme Kurulları* ve *Bağımsız Denetleme Şirketleri* ile bu kurul ve şirketlerin uyması gereken hükümleri ortaya koyan *Devlet Kuruluşlarını* ifade eder. FinTek ekosistemindeki dördüncü temel unsur ise finansal hizmetleri kullanan *finansal müşteriler* olup tüzel ve gerçek olarak iki gruba ayrılırlar.

FinTek ekosisteminin son unsuru ise finans hizmeti veren *finansal kuruluşlardır*. Bu kuruluşlara bankalar, sigorta şirketleri ve hisse senedi aracı kurumları örnek olarak verilebilir. Bu 5 unsur ülke içerisinde yenilikçiliğe katkıda bulunarak bankacılık ve finans sektöründe iş birliği ve rekabeti kolaylaştırmakta ve ekonomiyi canlandırmaktadır (Lee, 2016).

Fintek ekosisteminin en temel unsuru olan FinTek girişimleri bankalar ile çeşitli hizmetler geliştirmek amacıyla yoğun iş birliği yapmaktadır. Bu iş birliği çalışmaları kapsamında bankalar FinTek'lere özellikle hesap işlemleri, ödeme işlemleri, döviz alım/satım işlemleri, kıymetli maden alım/satım işlemleri hizmetleri sağlayan API'ler sağlamaktadır (Anonim, 2019 (3)). Bu bölümün geri kalanında FinTek girişimlerinin kullanımına sunulan bankacılık API'lerine örnekler verilmektedir.

Kuveyt Türk Katılım Bankası, hesap işlemleri için FinTek'lere üç adet API sağlamaktadır. Bu üç API'den ilki olan *Hesap Listeleme API'si* FinTek müşterilerinin Kuveyt Türk bünyesindeki hesaplarının listesinin çekilmesine olanak tanımaktadır. İkinci API olan *Hesap Hareketleri API'si*, bir FinTek müşterisinin Kuveyt Türk bünyesindeki hesaplarından herhangi birinde belirli bir dönemde gerçekleştirdiği hesap işlemlerinin çekilmesine olanak tanımaktadır. Bu hareketler çekilirken belirli bir zaman aralığına (yani başlangıç ve bitiş tarihine) ve/veya işlem adedine göre filtrelenerek çekilebilmektedir. Son olarak üçüncü API olarak ifade edebileceğimiz *Dekont API'si* ise bir FinTek müşterisinin Kuveyt Türk bünyesindeki hesaplarından herhangi birinde belirli bir dönemde gerçekleştirdiği hesap işlemleri ile ilişkili dekontların çekilmesine olanak tanımaktadır (2021, Kuveyt Türk API Market). Bu bağlamda bu üç API FinTek'lere

hesap sağlayıcı hizmetler geliřtirmelerine olanak tanımaktadır. Türkiye’de bu kapsamda hesap saęlama hizmetleri sunan FinTek’lerden bazıları řekil 1.8’da grlmektedir.



řekil 1.8. Hesap Sağlayıcı FinTek’ler (Anonim, 2019 (3))

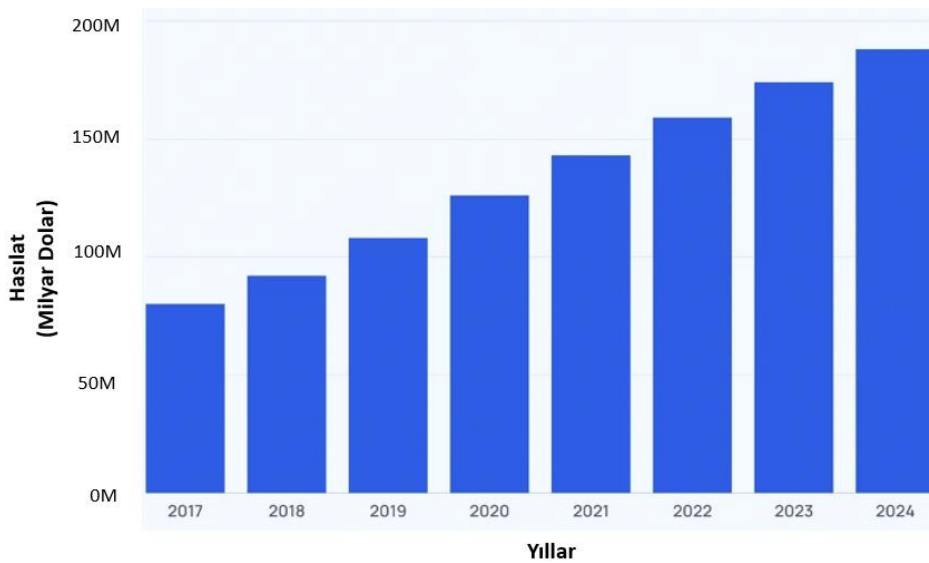
Kuveyt Türk Katılım Bankasından başka Albaraka Türk Katılım Bankası da benzer řekilde FinTek giriřimlerinin kullanımı iin API’ler saęlamaktadır. Bu API’lere rnek olarak para transferi iřlemleri verilebilir. FinTek’lerin mřterileri, Albaraka’nın saęlamıř olduęu bu API’ler zerinden para transferi iřlemlerini bankaya baęlı kalmadan, kullanmıř oldukları FinTek aplikasyonu zerinden gerekleřtirebilmektedirler. Albaraka, para transferi hizmetini IBAN’a, hesaba, kredi kartına, isme ve kolay adrese olacak řekilde sınıflandırmıřtır. Zira mřterilerin iřlemine gre para transferi gerekleřtirilirken kullanılan yntem farklılık gsterebilmektedir (Albaraka Türk API, 2021). Mřterilerin bu API’ler zerinden yapmıř olduęu para transferi iřlemi aynı banka ierisindeki hesaplar arasında gerekleřiyorsa havale, farklı bir bankaya gnderiliyor ise PS-EFT (Perakende deme Sistemi-Electronic Fund System) olarak adlandırılmaktadır. En yeni para transfer yntemi Fonların Anlık ve Srekli Transferi (kısacası FAST) olarak adlandırılmakta olup bu yntemde mřterilerin gn ierisinde her saatte bankalar arası para transferi gerekleřtirmesine olanak saęlanmaktadır. Türkiye’de deme ve para transferi hizmetleri zerine hizmet veren FinTek’ler řekil 1.9’da grlmektedir.



Şekil 1.9. Türkiye’de Para Transferi ve Ödeme Hizmetlerini Sunan FinTek’ler (Anonim, 2019 (3))

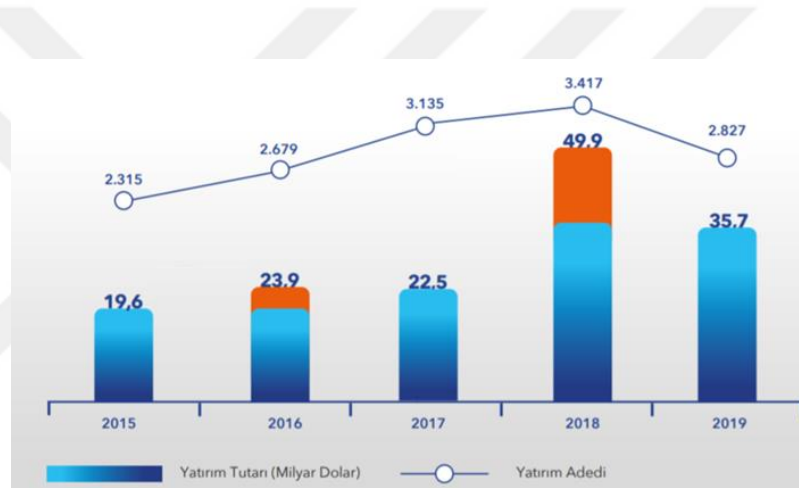
1.1.3. Dünya ve Türkiye’deki FinTek Yatırımları

2017 ile 2024 yılları arasında dünya genelindeki FinTek pazarındaki değişim incelendiğinde bu pazarda aktif olarak mevcudiyetini sürdüren firma sayısının yaklaşık 30.000’e ulaştığı ve pazardaki toplam hasılatın ise Şekil 1.10’da görüldüğü gibi 179 milyar amerikan doları (USD) seviyelerine ulaştığı görülmektedir. Bu rakamın 2028’e kadar 492 milyar dolar seviyelerine çıkması beklenmektedir (Howarth, 2023).



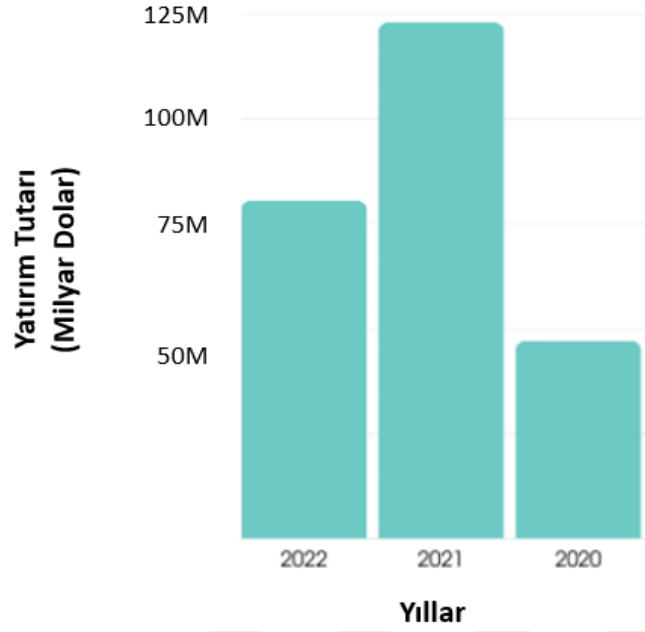
Şekil 1.10. FinTek Endüstrisi Hasılatının Yıllar Üzerinde Değişimi (Howarth, 2023)

Yatırım oranları göz önüne alındığında Innovate Finans tarafından yapılan bir çalışmaya göre FinTek'lere 2019 yılında 35,7 milyar dolar yatırım yapılmıştır. Bu tutar 2018 yılında 49,9 milyar dolar olarak gözlenmiştir ancak 2018 yılının ikinci çeyreğinde Ant Finansal' a yapılan 14 milyon dolarlık yatırım dışarıda tutulduğunda son iki yılda çok yakın değerlerin söz konusu olduğu gözlemlenmiştir. Yatırım şirketi kuran çok sayıda banka ve sigorta şirketi, oluşturdukları fonlar üzerinden ürün ve hizmetlerine katkı sağlayacaklarına inandıkları FinTek'lere yatırım yapma yoluna gitmektedirler. ABD gibi bankacılık sisteminin geride olduğu ülkelerde bankalar, FinTek'lere daha fazla yatırım yapmakta ve değişen tüketici ihtiyaçlarına, FinTek'lerin çözümleriyle yanıt vermeye çalışmaktadırlar. Şekil 1.11'de Innovate Finans şirketinin dünyada FinTek'lere yapılan yatırım tutarlarını ve adetlerini gösteren grafiğe yer verilmiştir (Anonim, 2019 (3)).



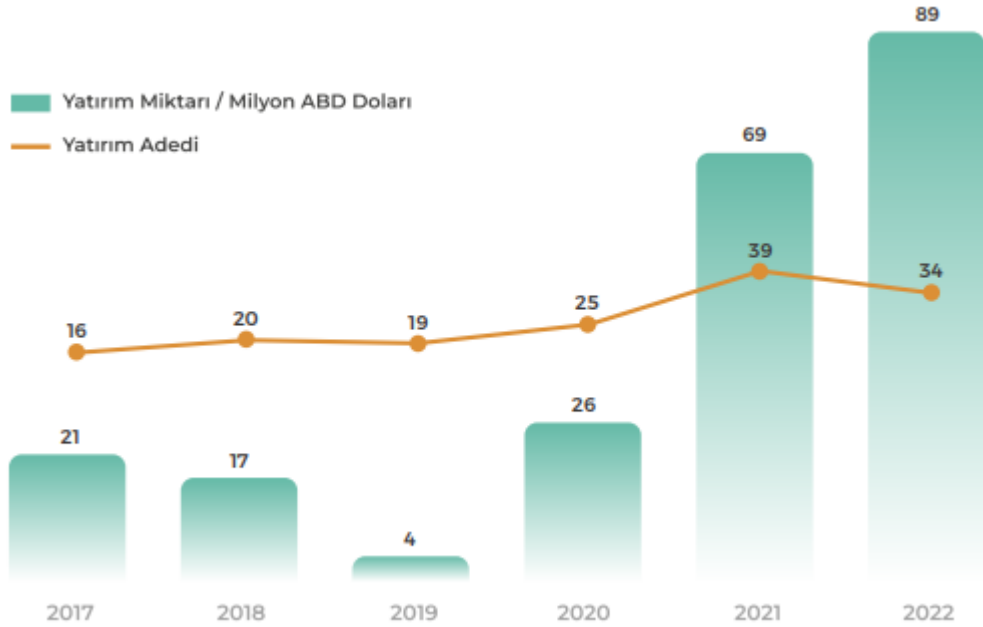
Şekil 1.11. FinTek Endüstrisi Hasılatının Yıllar Üzerinde Değişimi (Howarth, 2023)

Diğer yandan detayları Şekil 1.12'de görülen bir başka çalışmaya göre ise FinTek yatırım tutarları dünya genelinde 2021'e kadar artarak giderken 2022'de bir miktar azalmıştır (Shewale, 2024).



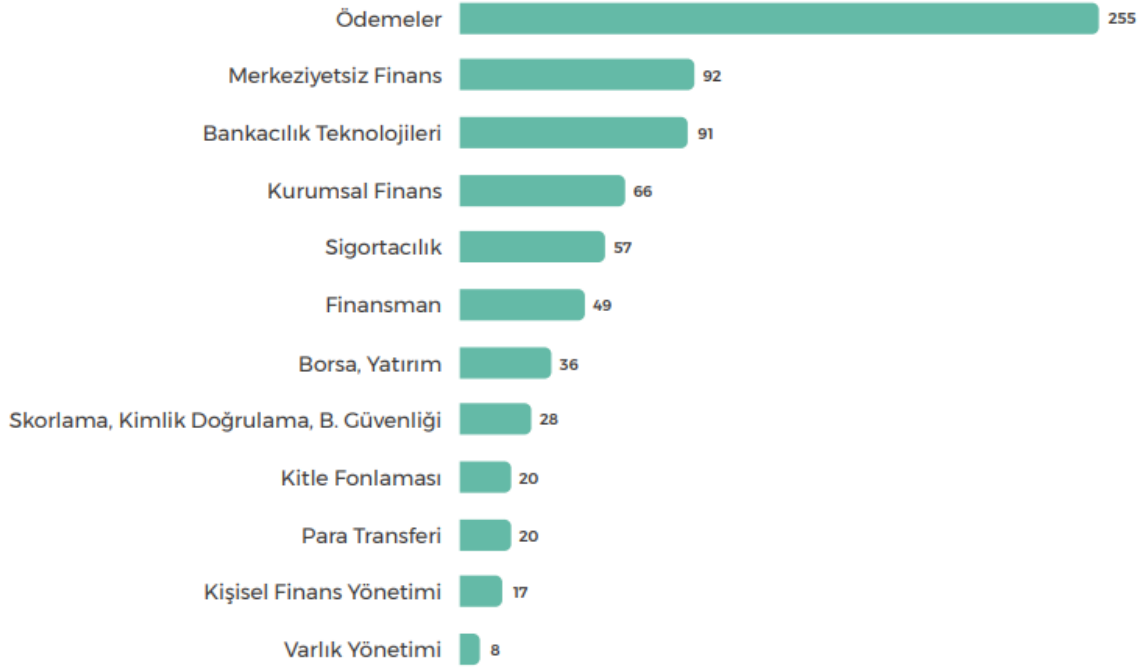
Şekil 1.12. Dünyada FinTek Yatırım Tutarının 2020 ile 2022 Arasında Değişimi (Shewale, 2024)

T.C. Cumhurbaşkanlığı Finans Ofisi'nin verilerine göre Türkiye'deki FinTek yatırımları 2022 yılında 89 milyon A.B.D. doları seviyesine ulaşmıştır. Bu durum Şekil 1.13'te görülmektedir (T.C. Cumhurbaşkanlığı Finans Ofisi, 2023).



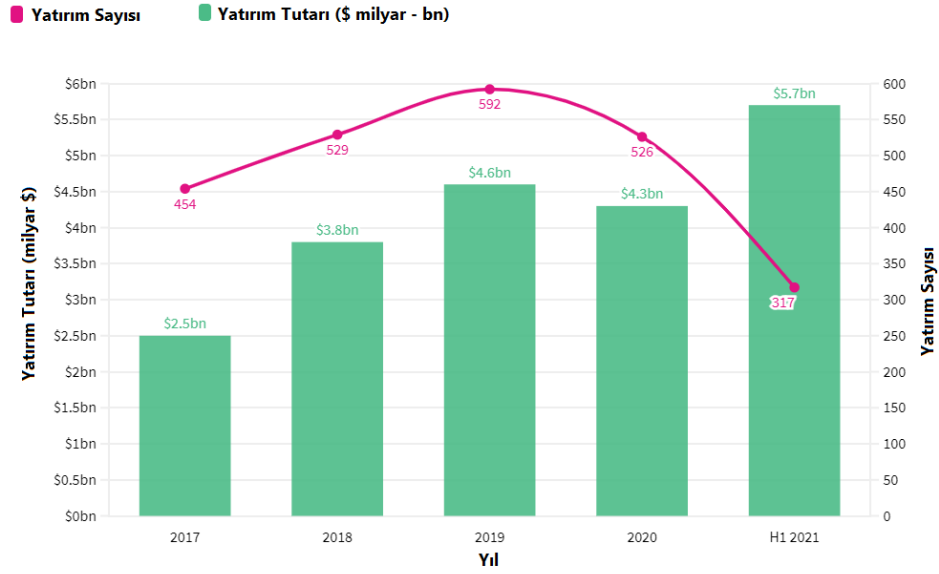
Şekil 1.13. Türkiye'de FinTek Yatırım Tutarlarının 2017 ile 2022 Yılları Arasında Değişimi (T.C. Cumhurbaşkanlığı Finans Ofisi, 2023)

Yine T.C. Cumhurbaşkanlığı Finans Ofisi'nin verilerine göre Şubat 2023 itibarıyla Türkiye'de 637'si aktif olmak üzere 739 FinTek kurulmuştur. Şekil 1.14'te görüldüğü gibi FinTek şirketleri dikeylerine göre değerlendirildiğinde ödemeler 255 kuruluş ile ilk sırada yer almaktadır. Bunu 92 kuruluşla merkeziyetsiz finans, 91 kuruluşla bankacılık teknolojileri ve 66 kuruluşla kurumsal finans dikeyleri takip etmektedir (T.C. Cumhurbaşkanlığı Finans Ofisi, 2023).

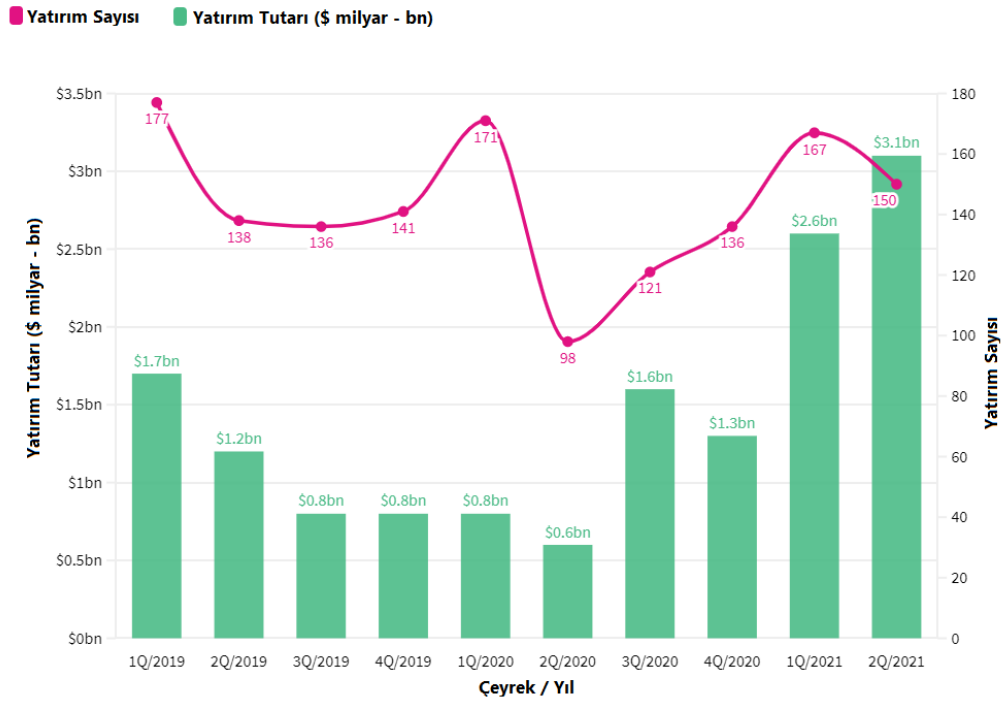


Şekil 1.14. Türkiye’de FinTek Sayılarının Şubat 2023 İtibariyle Durumu (T.C. Cumhurbaşkanlığı Finans Ofisi, 2023)

Yüzümüzü yeniden yurt dışına çeviricek olursak Innovate Finance’ın İngiltere özelinde yaptığı ve sonuçları Şekil 1.15’te görülen başka bir araştırmasına göre İngiltere FinTek’leri 2021 yılı ilk yarısında 317 yatırım almış ve toplamda 5,7 milyar dolarlık yatırım sermayesi elde edilmiştir (Innovate Finance). Bu bakımdan 2021 yılının ilk yarısında dahi FinTek yatırım miktarlarının oldukça yüksek seviyelere ulaştığı görülmektedir zira önceki yıllara göre yılın ilk yarısı için yatırım adedi düşük olmasına rağmen yatırım tutarının oldukça yüksek olduğu dikkat çekmektedir (Anonim, 2021 (13)).



Şekil 1.15. İngiltere’de FinTek’lere Yapılan Yatırım Tutarı ve Adedi Gelişim Grafiği (Anonim, 2021 (13))



Şekil 1.16. İngiltere’de FinTek’lere Yapılan Yatırım Tutarı ve Adedinin Çeyrek Dönemlerdeki Gelişim Grafiği (Anonim, 2021 (13))

Aynı araştırma kapsamında FinTek yatırımları çeyrek dönemler üzerinden incelendiğinde yukarıda Şekil 1.16’da görülen sonuçlar elde edilmiştir. Buna göre 2021 yılının ilk çeyreği ile ikinci çeyreğindeki yatırım tutarlarının önceki yıllara göre yüksek olduğu gözlemlenmektedir. Yatırım tutarının en yüksek olduğu dönem ise 3,1 milyar dolar ile 2021 yılının ikinci çeyreğindedir. Yatırım sayısına bakımında

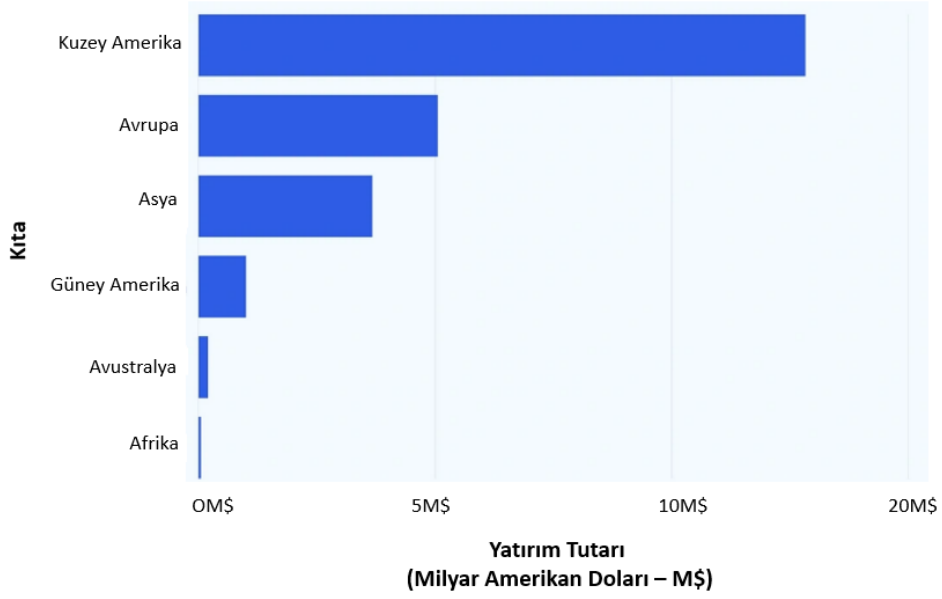
değerlendirildiğinde 2019 yılı ilk çeyreği ve 2020 ilk çeyreği yatırımcı sayısı en yüksek dönemler olmasına rağmen 2021 yılı ilk çeyreğindeki toplam yatırım tutarını geçememektedir. 2019 ilk çeyrek ve 2020 ilk çeyreğinde toplam 2,3 milyar dolar değerinde yatırım toplanırken 2021 yılının sadece ilk çeyreğinde bu tutar 2,6 milyar dolar olarak belirtilmiştir. Şekil 1.16’da 2019 ve 2021 yıllarında alınan yatırımların tutarı ve sayısı çeyrek dönemlere ayrılarak gösterilmiştir (Anonim, 2021 (13)).

FinTek yatırımlarının kıta özelinde değerlendirildiği Şekil 1.17’de 2019 yılına kadar Asya’nın lider konumda olduğu görülmektedir. Ant Finansal yatırımlarının gerçekleştiği bir araştırmada 2016 ve 2018 yıllarında yatırımların yarısından fazlasının Asya’ya yapıldığı belirtilmiştir. 2019 yılında ise bu tablo değişerek yatırımların %49’unu Kuzey Amerika toplamıştır. Kuzey Amerika’yı sırasıyla Avrupa, Asya ve diğer kıtalar takip etmiştir. 2018 ve 2019 yılları arasındaki Çin ve ABD arasındaki gerginliğin bu grafiği etkilediği belirtilmiştir (Anonim, 2019 (3)).



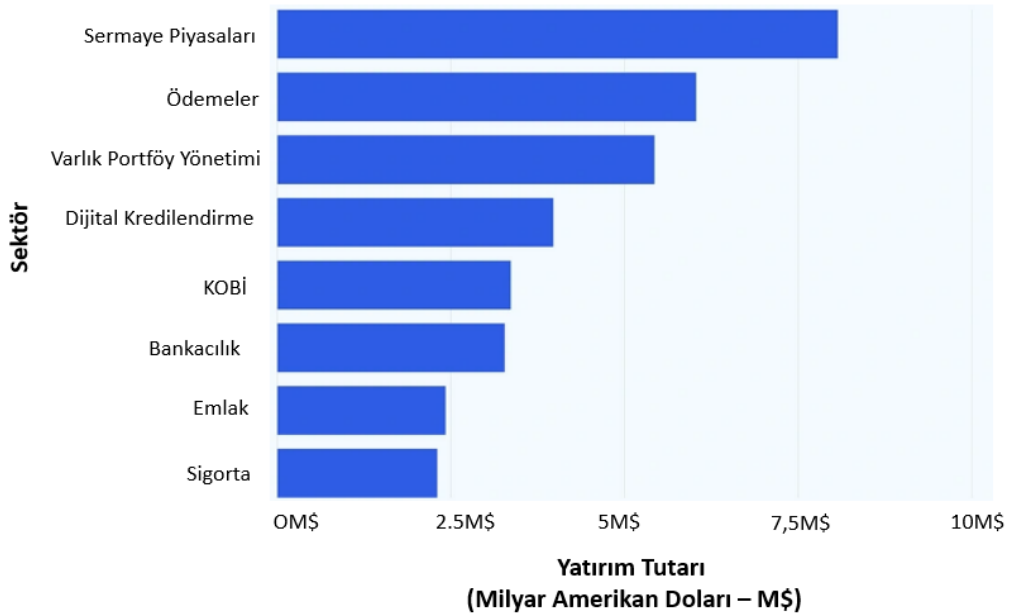
Şekil 1.17. Kıta Bazında FinTek Yatırımlarının Dağılımı (Anonim, 2021 (13))

Howarth tarafından yapılan incelemede 2021’in ilk çeyreğinde farklı kıtalar kapsamında alınan FinTek yatırım tutarları ise Şekil 1.18’de görüldüğü gibi olup yatırımların çoğunluğunun yine Kuzey Amerika kıtasında toplandığı görülmektedir (Howarth, 2023).



Şekil 1.18. 2021 Yılında FinTek Yatırımlarının Farklı Kıtalarda Dağılımı (Howarth, 2023)

Yine Howarth tarafından yapılan çalışmada 2021'in ilk çeyreğinde FinTek yatırımlarının dünya genelinde sektörel dağılımı incelenmiş olup Şekil 1.19'da görüldüğü gibi bir dağılım ortaya çıkmıştır (Howarth, 2023). Buna göre dünya genelinde FinTek yatırımının alındığı sektör sermaye piyasaları olup ödemeler onu ikinci sırada takip etmektedir. Buna karşın Şekil 1.4'te görüldüğü gibi Türkiye'de en öndeki sektör ödemeler olmuştur.



Şekil 1.19. 2021 Yılında FinTek Yatırımlarının Dünya Genelinde Sektör Bazlı Dağılımı (Howarth, 2023)

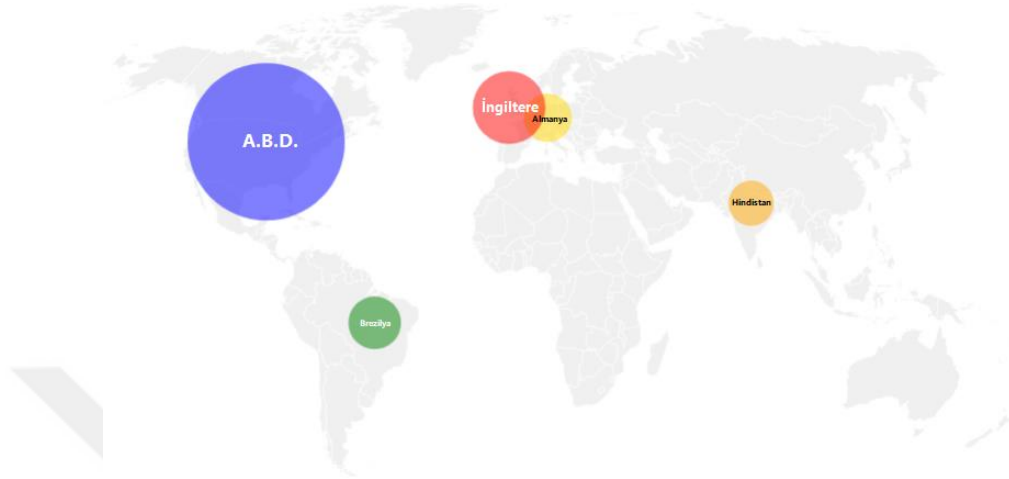
FinTek İstanbul'un 2019 yılında yapmış olduğu başka bir çalışmada FinTek yatırımlarının ülke bazında dağılımı incelenmiş ve Amerika Birleşik Devletleri (A.B.D.), İngiltere, Hindistan ve Çin'in Fintek yatırımlarında öne çıkan ülkeler olduğu görülmüştür. Buna göre yatırım tutarı olarak A.B.D 16,275 milyar dolar ile ilk sıradadır. A.B.D'yi sırasıyla 4,901 milyar dolar ile İngiltere, 3,852 milyar dolar ile Hindistan takip etmektedir. Yatırımlara tutar bakımından bakıldığında da ise A.B.D 1,095 adet ile ilk sırada yer almaktadır. A.B.D'yi sırasıyla 359 adet yatırım ile İngiltere, 166 yatırım ile Çin takip etmektedir. Şekil 1.20'de FinTek yatırımlarında başı çeken ilk 10 ülke görülmektedir (Anonim, 2019 (3)).



Şekil 1.20. FinTek Yatırımlarında İlk 10 Ülke (Anonim, 2019 (3))

Innovate Finans firmasının 2021 yılı ilk yarısı için yaptığı başka bir çalışmada, dünya genelinde FinTek'lere en çok yatırım yapılan ülke 960 adet yatırım ile A.B.D.'dir. A.B.D'deki FinTek'ler bu 960 adet yatırımdan toplamda 26,7 milyar dolar tutarında ödenek almıştır. A.B.D., bu yatırım tutarı ile 2021 yılı ilk yarısı için en çok destek tutarı alan ülke konumundadır. A.B.D.'den sonra en çok yatırımı 317 yatırım desteği ile İngiltere almıştır. İngiltere FinTek'leri toplamda 5,7 milyar dolar destek almışlardır. Bu iki büyük pazarı Brezilya, Almanya ve Hindistan takip etmektedir. Brezilya da 40 yatırım desteği ile 3 milyar dolar, Almanya da 56 yatırım desteği ile 2,5 milyar dolar, Hindistan da ise 132 yatırım desteği ile 2,2 milyar dolar destek FinTek'lere yapılmıştır. Hindistan, Almanya ve Brezilya'dan daha çok yatırım adedine sahip olsa da tutar bakımından bu iki

ülkeden az miktarda yatırım desteği toplamıştır. Şekil 1.21’de 2021 yılı ilk yarısında FinTek’lere yapılan yatırımların en yüksek olduğu 5 ülke görülmektedir (Anonim, 2021 (13)).



Şekil 1.21. 2021 Yılı İlk Yarısında FinTek’lere Yapılan Yatırımların En Yüksek Olduğu 5 Ülke (Anonim, 2021 (13))

1.2. Ödeme Hizmetleri Yönergesi (Payment Services Directive)

Ödeme Hizmetleri Yönergesi (Payment Service Directive 2-PSD2), İngiltere ve Avrupa Birliği’nde 13 Ocak 2018 itibariyle kurumlar ve üçüncü parti servis sağlayıcılar arasında geliştirilecek yeni uygulamaların yöntem ve işleyişini belirleyen devrim niteliğindeki yasal altyapı kurallarını içinde barındıran yasal bir şartnamedir. Bu bölümün geri kalanında bu şartnamenin oluşum süreci ve amaçları gibi konulara yer verilmektedir.

1.2.1. Ödeme hizmetleri yönergesi oluşum süreci

Ödeme sistemleri sektöründe yenilikçiliği, rekabeti ve güvenliği arttırmak isteyen Avrupa Birliği (AB) komisyonu, *Ödeme Hizmetleri Direktifi (Payment Service Directive-PSD)* adı altında bir şartname (yönerge) ortaya koymuştur. AB komisyonu bu şartnamenin ikinci sürümü olarak addedilebilecek olan ve bankacılık ve finans sektöründe devrim niteliği taşıyan *PSD2* şartnamesi taslağını 2013 yılı temmuz ayı içerisinde oluşturmuş ve 23 Aralık 2015 tarihinde Avrupa Birliği Resmî Gazetesi’nde yayınlayarak 12 Ocak 2016 tarihi itibariyle resmen yürürlüğe koymuştur. Bu çerçevede AB komisyonu, PSD2 şartnamesi kapsamındaki yönetmeliklerin yerel yasalara dönüşümü

için AB üyesi olan 28 ülke ile İzlanda, Norveç ve Lihtenştayn'a 13 Ocak 2018 tarihine kadar iki senelik bir geçiş süresi tanımıştır. Bu doğrultuda PSD2 yönetmeliği, tüm AB ülkelerinde 2018 yılı ocak ayı içerisinde yasalasmıştır. Buna ek olarak PSD2 kapsamında müşteri doğrulama ile ilgili teknik standartlar, 2018 yılı mart ayı içerisinde AB Resmî Gazetesi'nde yayınlanmıştır. Bu gelişmeler üzerine bankalar, Uygulama Programlama Arayüz'lerini (API) test amaçlı olarak *Üçüncü Parti Hizmet Sağlayıcıların (Third Party Provider-TPP)* kullanımına 2019 yılı mart ayı içerisinde açmışlardır. Burada *TPP* ile ifade edilmek istenen büyük veya küçük ölçekli bütün FinTek girişimleridir (yani FinTek firmalarıdır). PSD2'nin oluşum süreci boyunca gerçekleşen önemli gelişmeler Şekil 1.22'deki PSD2 zaman tüneline ifade edilmektedir.



Şekil 1.22. PSD2 Zaman Tüneli

1.2.2. Ödeme hizmetleri yönergesi amaçları

Elektronik ve mobil ödeme hizmetlerindeki çeşitliliğin artması ve yeni türde ödeme hizmetlerinin oluşması neticesinde PSD'nin ilk versiyonu güncelliğini yitirmiştir.

Bu nedenle, özellikle ödeme ve bankacılık alanlarında tüketicilerin mağdur duruma düşmesine neden olabilecek yasal belirsizlik ve güvenlik riskleri oluşmuştur. Diğer yandan ödeme ve hesap bilgilendirme hizmetleri gibi yardımcı hizmetlerin ortaya çıkması ve yaygınlaşması PSD2nin ilk versiyonu kapsamında önemli düzenlemelerin yapılmasını gerekli kılmıştır. Bu gerekliliği gidermek amacıyla PSD'nin ikinci versiyonu olan PSD2 hazırlanmış ve yürürlüğe konmuştur (Anonim, 2015; European Banking Federation, 2016). PSD2'nin amaçları şunlardır:

- Üçüncü parti servis sağlayıcılarının (Third Party Payment Service Provider-TPP) önündeki engelleri azaltarak inovatif uygulamaların gelişimini desteklemek.
- Yenilikçi mobil ödeme uygulamalarının, geliştirilmesini ve kullanılmasını teşvik etmek.
- Sektörde var olan elektronik ödeme uygulamalarına standart getirmek ve kurallarını geliştirmek.
- Üçüncü parti sağlayıcıların haklarını ve tabii oldukları zorunlulukları belirlemek.
- Kullanıma sunulacak yenilikçi uygulamalar kapsamında tüketicileri korumak.
- Uygulamaların düşük maliyetlerle hazırlanmasını teşvik etmek ve yenilikçi mobil ödeme uygulamalarının geliştirilmesini ve kullanılmasını teşvik etmek.

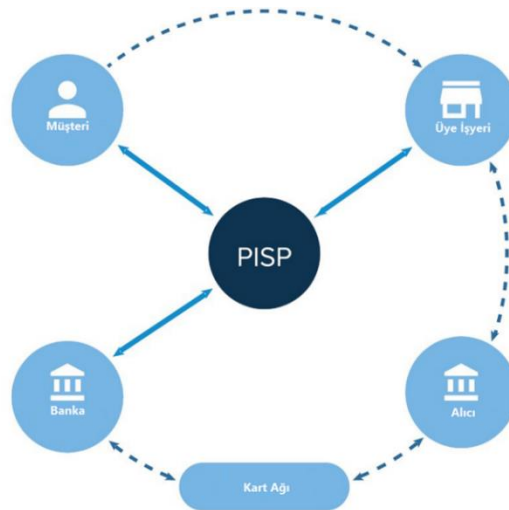
1.2.3. Ödeme hizmetleri yönergesi hizmet sağlayıcıları

PSD2 yönergesi kapsamında perakendeciler ve bankalar arasında bağlantı sağlanması amacıyla *Hesap Hizmeti Ödeme Hizmeti Sağlayıcıları (Account Servicing Payment Service Providers-ASPSP)* olarak da bilinen bankacılık kurumlarında hesabı bulunan bankacılık müşterilerinin hesaplarıyla ilgili bilgilerin *Üçüncü Taraf Ödeme Hizmet Sağlayıcılarının (Third Party Payment Service Providers-TPPs)* erişimine açılması gerekmektedir. Bankacılık kurumlarının açık erişim yetkisi vereceği üçüncü Taraf Hizmet Sağlayıcıları iki gruba ayrılabilir (Petrović, 2020):

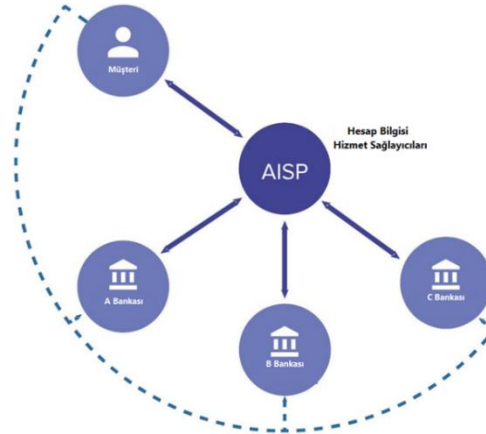
- Ödeme Başlatma Hizmet Sağlayıcıları (Payment Initiation Services Providers-PISPs) Grubu:** Müşterileri adına ödemeyi başlatıp satıcılara paranın ödenme güvencesini vererek müşterilerinin online (çevirim-içi) satın alma işlemlerini kolaylaştıran hizmetler sunan TPP'leri temsil eden gruptur.

Ödeme PISP'ler bankalar tarafından sağlanan uygulama ara yüzlerini (Application Programming Interfaces-APIs) kullanarak kullanıcı adına ödeme işlemini başlatır (müşteri bu izinlere erişim izni verdiğyse). Fonlar doğrudan kullanıcının banka hesabından satıcıya taşınır. Bu durum ödeme kartı şirketleri tarafından tahsil edilen ücretleri ortadan kaldırmaktadır. PISP minvalinde bankalar ile müşterileri arasındaki etkileşim Şekil 1.23'te görülmektedir. Bu Şekilden anlaşılacağı üzere PISP'ler müşterilerin üye iş yeri tarafından online (yani web tabanlı) yapacakları alım satım işlemlerini bankalarla entegre olarak kısaltmak suretiyle kart ağı üzerinden işlem yapmayı gerektiren geleneksel kart ağı bazlı alım satım işlemlerini bertaraf etmektedir.

- ii. **Hesap Bilgileri Hizmet Sağlayıcıları (Account Information Service Providers-AISPs) Grubu:** Müşterilerin bir veya birden fazla bankadaki hesap bilgilerini **konsolide** edip hesap ve bakiye bilgisi veren hizmetler sunan TPP'leri temsil eden gruptur. Bu gruptaki TPP'ler sağladıkları hizmetler üzerinden bankacılık müşterilerine finans durumlarını kolaylıkla yönetebilecekleri teknolojik bir platform sunmaktadırlar. AISP'ler ile bankalar ve müşteriler arasındaki etkileşim Şekil 1.24'te temsili olarak ifade edilmektedir. Bu Şekilden anlaşılacağı üzere AISP'ler müşterilere sırasıyla A, B ve C bankalarındaki hesap bilgilerini ve hareketlerini konsolide olarak imkânı sunmaktadır.



Şekil 1.23. Ödeme Başlatma Hizmet Sağlayıcısı Modeli



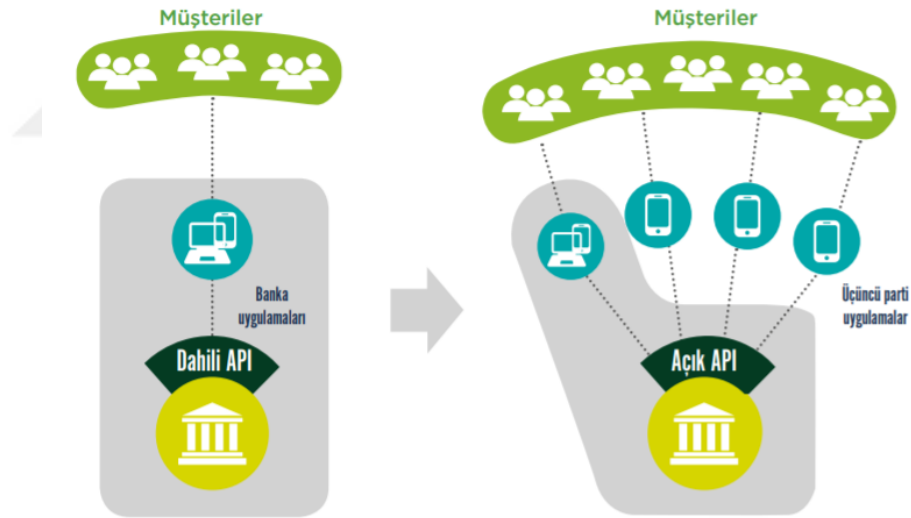
Şekil 1.24. Hesap Bilgileri Hizmet Sağlayıcısı Modeli

1.3. Açık Bankacılık

Açık bankacılık, en temel anlatımı ile üçüncü taraf hizmet sağlayıcılara (yani TPP'lere), bankacılık müşterilerinin bilgisi ve izni dâhilinde finansal müşteri verilerine erişimlerine ve işlem gerçekleştirebilmelerine olanak sağlayan güvenli bir yöntem olarak tanımlanmaktadır. Bu yöntem sayesinde TPP'ler arasında şeffaflık ve rekabetin artırılması ve buna bağlı olarak finansal hayatın kolaylaştırılması hedeflenmiştir (Menikgama, 2020).

Bankalar; mevduat hesabı yönetmek, kredi vermek, müşteriler arası ödemelere aracı olmak gibi en temel işlevlerini müşterilerine geleneksel kanallarından hâlihazırda sunmaktadırlar. Geleneksel olarak şube kanalı üzerinden sunulan bu hizmetler, teknoloji ve güvenlik önlemleri müsaade ettikçe ATM, çağrı merkezi, internet bankacılığı ve mobil bankacılık kanalları gibi diğer kanallar üzerinden de müşterilerin kullanımına açılmıştır. Bununla birlikte son kullanıcı beklentilerindeki artış ve endüstrideki dijitalleşme eğilimleri nedeniyle PSD2 gibi yeni bankacılık düzenlemeleri ortaya çıkmıştır. Bu doğrultuda geleneksel bankacılık hizmetlerine ek olarak TPP'ler üzerinden sağlanan bankacılık servislerinin ve uygulamalarının sayısı giderek artmaktadır (Petrović, 2020) zira bankacılık müşterileri, geleneksel olarak kullanageldikleri bankacılık hizmetleri yerine TPP'ler tarafından geliştirilip sunulan ve cep telefonları gibi mobil cihazlar üzerinden birkaç tıkla erişebilen daha pratik bankacılık hizmetlerine talep göstermektedirler. Özellikle müşterilerin birden fazla banka ile çalıştıkları durumlarda farklı bankalardaki hesaplarının birleştirilmiş (yani konsolide edilmiş) hallerini bir banka

veya TPP tarafından geliştirilen tek bir uygulama üzerinden görebilmeleri müşterilerin iş verimliliğini artırmaktadır. Bu nedenle genel çerçevesi PSD2 yönergesi ile ortaya konulan pek çok *Açık Bankacılık API'si* günümüzde bankalarca TPP'lerin ve dolayısıyla son kullanıcıların (yani bankacılık müşterilerinin) kullanımına sunulmaktadır. Bu bağlamda Şekil 1.25'te *Geleneksel Bankacılık* ve *Açık Bankacılık* kavramları temsili olarak ifade edilmektedir. Bu Şekilden anlaşılabileceği üzere *Geleneksel Bankacılık Servisleri* bankaların kendi *dâhili API'lerini* kullanan mobil ve/veya internet uygulamaları üzerinden sunulurken; Açık Bankacılıktaki servisler açık API'ler üzerinden TPP'lerin kullanımına ve dolayısıyla son kullanıcıların (yani bankacılık müşterilerinin) kullanımına sunulmaktadır. Bir başka deyişle bankalar tarafından sunula gelen geleneksel temel bankacılık hizmetleri, bankalar tarafından Açık Bankacılık minvalinde endüstri standardı güvenlik tedbirleri ile korunan Açık Bankacılık API'leri çerçevesinde TPP'lerce geliştirilen üçüncü parti uygulamalara ve bu uygulamalar üzerinde de son-kullanıcılara (müşterilere) açılmaktadır.



Şekil 1.25. Geleneksel Bankacılık ve Açık Bankacılık (Anonim, 2019 (3))

Açık Bankacılık yaklaşımı başlangıçta bankalar tarafından üstlenilen bir inisiyatifken bugün dünyanın farklı coğrafyalarında, finans sektöründe rekabet ortamını güçlendirmek amacıyla, kanunlar tarafından zorunlu tutulan bir yaklaşıma dönüşmektedir. Avrupa Birliği tarafından Açık Bankacılık standartları ortaya koymak amacıyla oluşturulan PSD2 yönergesinin ardından Asya ülkeleri ve bankaların farklı düzenlemeler ortaya koyması ile birlikte Açık Bankacılık finans sektöründe hızla yöntem

Türkiye’de ise Garanti, Yapı Kredi, Kuveyt Türk, Albaraka Türk, İş Bankası ve Denizbank API’lerinin bir kısmını internet üzerinden erişime açmışlardır. “Bir API” adlı firmanın Türkiye’de açık bankacılık konusunda yaptığı bazı çalışmalar sonucunda 2023 Temmuz ayı itibariyle API sayısı en yüksek bankanın Şekil 1.27’de görüldüğü gibi Aktif Bank’tır. Aktif Bank’ı sırasıyla Intertech ve Fibabanka takip etmektedir (Anonim, 2023 (2)).



Şekil 1.27. Yıllara Göre Bankaların API Sayılarındaki Değişim Tablosu (Anonim, 2023 (2))

1.4. Application Programming Interface (API-Uygulama Programlama Ara Yüzü)

Bu bölümde gerek *Açık Bankacılığın* gerekse *Mikro Servis Mimarisinin* pratikte uygulanabilir kılınmasında önemli bir yeri olan API teriminin kavramsal tanımından

başlayıp, kısa tarihçesine, endüstride karşılaşılabilecek türlerine ve güncel tasarım tekniklerine değinilmektedir.

1.4.1. API tanımı

Application Programming Interface (Uygulama Programlama Ara Yüzü-API) ifadesinin kısaltılmış hali olan API terimi bir yazılım uygulaması ile nasıl etkileşimde bulunulacağını tarif eden bir mekanizmadır. Bir başka deyişle API, bir yazılım uygulaması tarafından sağlanan işlevlere erişimi soyutlayan bir yapı ortaya koymaktadır. Bu bakımdan yazılım mühendisliğinde API'ye aynı zamanda *soyut programlama ara yüzü (Abstract Programming Interface)* de denilmektedir (Reddy, 2011). Bir örnekle ifade edilecek olursa A bilgisayar programının kendi bünyesinde bulunmayan ancak B programınca sağlanan veri ve işlevlere erişimi B programınca sağlanan API üzerinden mümkün olabilmektedir. Bu durum Şekil 1.28'de temsili olarak gösterilmektedir. Bu Şekilden de anlaşılabileceği üzere API bir bakıma A programı ile B programı arasında soyut (abstract) bir iletişim ve kullanım sözleşmesi tesis eder (Reddy, 2011).

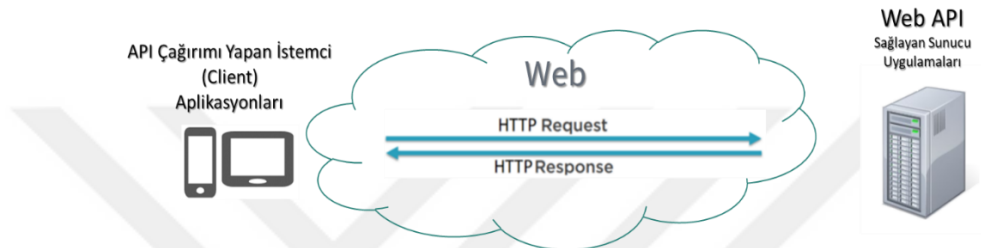


Şekil 1.28. A Programının B Programınca Sunulan İşlevlere B Programı API'sini Çağırarak Erişmesi

API'ler sayesinde yazılım geliştiricilerin ihtiyaç duydukları spesifik yazılım özelliklerini sil-baştan yazmalarına gerek kalmaz. Bunun yerine yazılım geliştiriciler, bu spesifik özellikleri halihazırda barındıran başka yazılım bileşenlerinin API'lerini kendi geliştirdikleri yazılım uygulamaları bünyesinde çağırarak ihtiyaç duydukları özellikleri kendi uygulamalarının bünyesine dahil etmiş olurlar. Bu bakımdan API'ler, *dağıtık bir yazılım sisteminin (distributed software system)* parçalarını oluşturan yazılım bileşenlerinin birbiriyle gevşek bağımlılık (loosely-coupled) ilişkisi içerisinde iletişimde bulunmalarına imkân tanıyan temel bir mekanizma oluşturmaktadır (Biehl, 2015).

Günümüzde API'lerin pek çoğu web üzerinden erişilmekte olup bu API'lerin kullanımları, *GET*, *POST*, *PUT* ve *DELETE* gibi standart *HTTP (Hyper-Text Transfer*

Protocol) protokolü metotlarının bir ağ (network) üzerinden çağırımına dayanmaktadır. Bu bakımdan günümüz yazılım endüstrisinde web üzerinden çağırılan API'lere **Web API** denilmektedir (Biehl, 2015). *Web API'lerinde* bir sunucunun üzerindeki uygulama ya da uygulamalar tarafından sağlanan veri ve işlemlere web tabanlı *HTTP request*'i (istemi) gönderilerek ulaşılır ve karşılığında uygulamadan bir *HTTP response* (cevabı) dönlür. Dolayısıyla aplikasyonların uygulamalarca sağlanan API'lere erişimi bu aplikasyonlar ile API'lerin konakladığı sunucular arasındaki *HTTP protokolü* esaslı mesajlaşma üzerine kurgulanır. Bu durum aşağıda Şekil 1.29'da görölmektedir:



Şekil 1.29. HTTP Protokolü Esaslı Web API Çağırımı

1.4.2. API'nin tarihçesi

API terimi önceleri uygulama programları (Application Programs-AP) olarak adlandırılan ve yalnızca son kullanıcıya dönük olan bilgisayar programları için bir ara yüz manasında kullanılmıştır. Ancak zamanla bu anlam genişlemiş ve donanım ara yüzlerinin de dâhil olduğu pek çok yazılım ara yüzünü kapsamıştır.

Terim olarak API ilk kez 1968'de "Uzak Bilgisayar Grafikleri için Veri Yapıları ve Teknikleri" adlı bir konferans makalesinde kullanılmıştır. API terimi bu makalede hususen bir bilgisayar grafiği programının sistemin geri kalanı ile donanım bağımsız olarak etkileşimde bulunabilmesi amacıyla kullanılmıştır (Cotton ve ark., 1968). Bunun akabinde API teriminin 1974 yılında veri tabanı alanında "İlişkisel ve Ağ yaklaşımları: API Karşılaştırması" başlıklı makalede kullanımı görölmüştür (Date ve Codd, 1974).

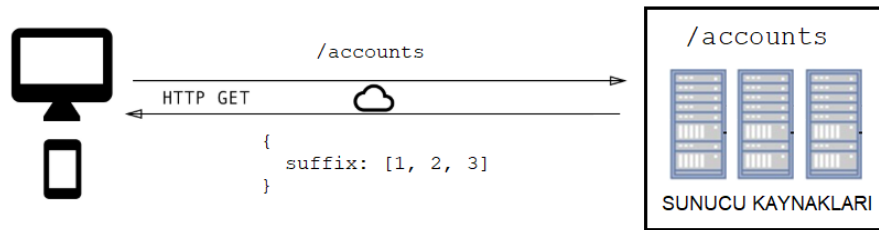
80'li yıllarda bilgisayar ağlarının yaygınlaşmasıyla API kavramının anlamı Uzak Yordam Çağırımı (Remote Procedure Call-RPC) teknikleri kapsamında genişlemiş ve coğrafi olarak ayrıık bilgisayar ağları üzerinden yazılım bileşenlerine erişim sağlayan bir mekanizma manasında kullanılmıştır (Birrell ve Nelson, 1984). Bu kapsamda *Common Object Request Broker (CORBA)*, *Distributed Common Object Model (COM)* ve *Java Remote Method Invocation (Java RMI)* adlarında alternatif yazılım teknolojileri ortaya

çıkıştır (Hirsch, 2007). Ayrıca son 10 yılda RPC konusunda yeni gelişmeler yaşanmış ve *gRPC* ve *Apache Thrift* gibi RPC tabanlı modern API geliştirme teknolojileri ortaya çıkmıştır (Abernethy, 2019; Du ve ark., 2018).

2000’li yıllarda internetin yaygınlaşması ile birlikte web API kavramının kullanımı yaygınlaşmıştır. Özellikle 2000 yılında Roy Fielding’in doktora tezi çalışması kapsamında geliştirdiği *Representational State Transfer (REST)* tekniği (Fielding, 2000) günümüzde web API’lerinin tasarımında baskın teknik haline gelmiştir (Lane, 2019).

1.4.3. REST ve GraphQL

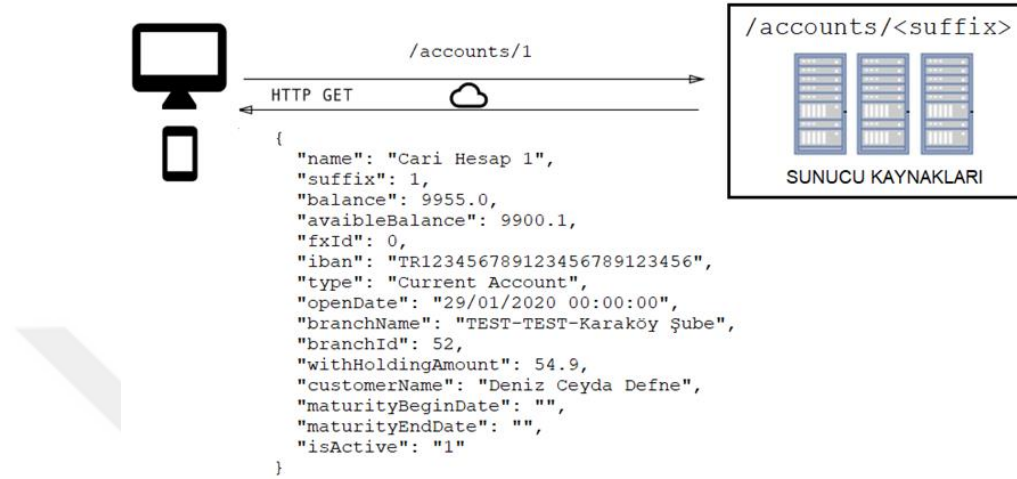
Representational State Transfer (REST) teknolojisi web API’lerin geliştirilmesinde, günümüzde endüstride yaygın olarak kullanılan ve PSD2 şartnamesi kapsamında da kabul gören bir teknolojidir. Bununla birlikte Facebook tarafından geliştirilen GraphQL teknolojisinin, daha az sayıda network çağırımı gerektirmesi (tek çağırıda birkaç request’i kapsaması) ve daha anlamlı (expressive) ara yüz tasarlama imkânı tanınması nedeniyle uzun vadede REST teknolojisinin yerini alması ve endüstride dominant olması beklenmektedir. GraphQL teknolojisinin avantajları aşağıda sırasıyla Şekil 1.30, 1.31 ve 1.32 ile temsil edilen örnek REST API çağırımları üzerinden açıklanmaktadır.



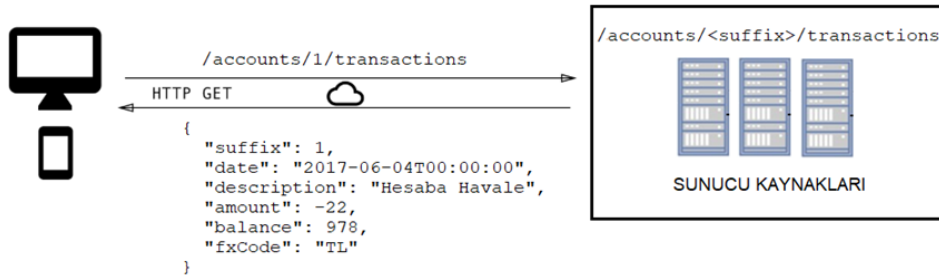
Şekil 1.30. Hesap Ek No’larını (Suffix) Getiren REST API Çağırımı

Bu örnek çağırımlarda bir bankacılık müşterisinin hesap hareketlerini sorgulamak için sunucu kaynağına 3 ayrı REST çağırısı yapılmaktadır. Bu çağırımlardan birincisi Şekil 1.30’da görüldüğü gibi bir müşterinin mevcut hesaplarının listesini çekerken ikinci çağırım ise Şekil 1.31’de görüldüğü gibi mevcut hesaplardan birine ait hesap ek no’larının (suffixes) listesini çekmektedir. Diğer yandan Şekil 1.32’de görülen üçüncü çağırım ise belirli bir hesap numarasının belirli bir ek no’su kapsamında gerçekleşen hesap hareketlerinin listesini çekmektedir. REST teknolojisi ile üç ayrı network sorgusu

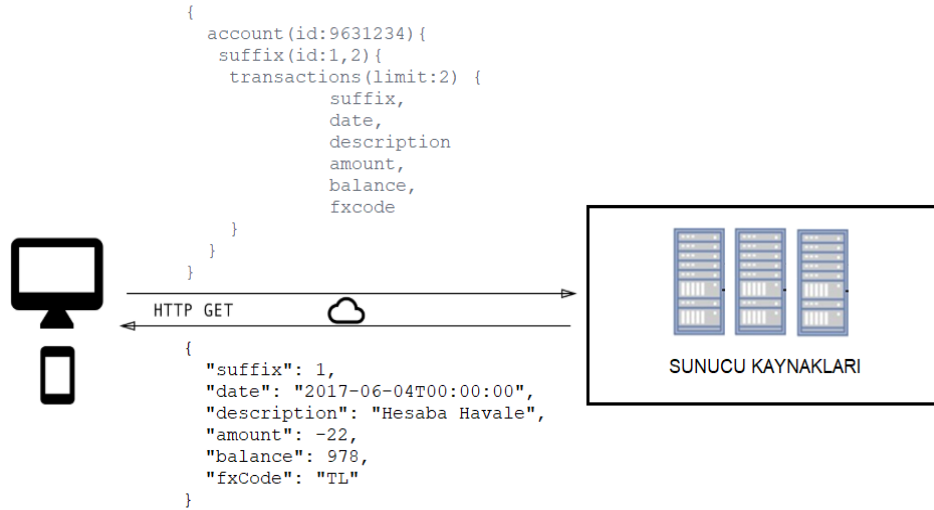
gerektiren bu yapı, GraphQL teknolojisinin kullanılması durumunda Şekil 1.33'te görüldüğü gibi tek bir API çağırımıyla daha verimli bir şekilde gerçekleştirilebilir. Bu nedenle GraphQL teknolojisinin uzun vadede istemci-API iletişiminde dominant yöntem olması öngörülmektedir.



Şekil 1.31. Belirli bir Hesabın Belirli bir Ek No'suna Ait Hesap Bilgilerini Getiren REST API Çağırımı



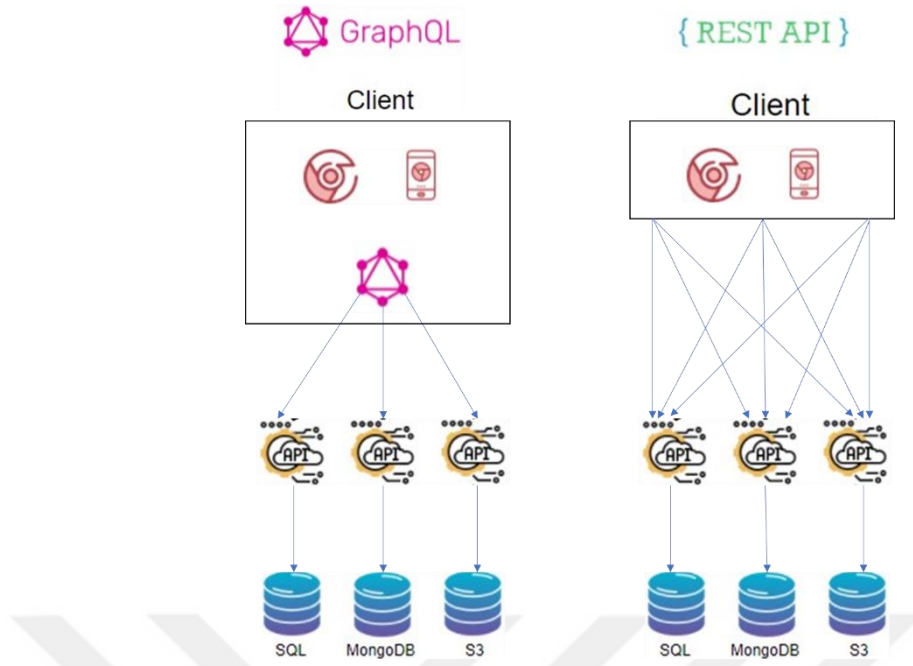
Şekil 1.32. Hesap Hareketlerini Getiren REST API Çağırımı



Şekil 1.33. Hesap Hareketlerini Getiren GraphQL Tabanlı API Çağırımı

1.4.4. GraphQL teknolojisinin endüstriyel kullanım örnekleri

GraphQL, API tasarlamak ve kullanmak için ortaya konmuş bir API tasarım tekniğidir. Bu yöntem, istemci uygulamaları tarafından (clients) ihtiyaç duyulan veri ihtiyaçlarının direk olarak istemci uygulama kapsamında belirlenip karşılanmasına olanak tanınmasıyla ön plana çıkmaktadır. Nitekim Şekil 1.34'teki GraphQL başlığı altındaki temsili resimden anlaşılacağı üzere API çağırımları GraphQL tarafından soyutlandığından (abstract edildiğinden) istemciler kendi ihtiyaçları doğrultusunda duydukları farklı nitelik ve nicelikteki verileri tek seferde daha akıllıca çekebilir. Dolayısıyla ağ üzerinden sunucu kaynaklarına yapılması gereken çağırımlar ve aktarılması gereken veriler minimal düzeyde tutulabilir. Bu durum özellikle düşük bant genişlikli ağlarda çalışan istemcilerin performansını iyileştirmeye yardımcı olur zira daha az ağ trafiği hasıl olur.



Şekil 1.34. GraphQL ve REST Teknolojilerinde API Çağırım Karmaşıklığının Temsili Resmi

Diğer yandan REST teknolojisini esas alan API’lerin istemci tarafındaki veri gereksinimlerinin sunucu tarafında ele alınıp tasarlanması gereklidir. Buna bağlı olarak Şekil 1.34’teki “REST API” başlığı altında işaret edilen temsilde görüldüğü gibi daha çok API çağırımı ve dolayısıyla ağ trafiği hasıl olur ve istemcilerin sunucu kaynaklarından (Örnek: SQLServer, MongoDB veya Amzon S3 gibi veri tabanları) bağımsız olarak tasarlanması güçleşir. Dolayısıyla gerek sunucu gerekse network kaynaklarının verimsiz kullanımı gerçekleşir. Bu bakımdan GraphQL teknolojisinin endüstride kullanımı giderek yaygınlaşmaktadır (Akalın, 2018).

Aşağıdaki başlıklarda GraphQL teknolojisinin farklı sektörlerdeki kullanım alanlarına örnekler sunulmaktadır.

1.4.4.1. GitHub

Git adlı sürüm kontrol sistemini barındıran bulut tabanlı bir servis olan GitHub açık kaynak kodlu yazılım projelerinin geliştirilip saklandığı bulut tabanlı bir servistir. GitHub, kaynak kodların geliştiricilerce paylaşımını kolaylaştıran ve bu suretle yazılım ekiplerinin paralel olarak çalışıp çevik (agile) usuller çerçevesinde yazılım geliştirmelerine olanak tanıyan bir web sitesidir. GitHub, kaynak kodların saklanıp paylaşılması kapsamındaki işlevlerini gerçekleştirmek için yazılım geliştiricilere REST

teknolojisi esaslı API'ler sunmuştur. Bununla birlikte aşağıdaki paragraflarda ifade olunan bazı nedenler dolayısıyla REST API'lerine ilaveten GraphQL API'lerini de hizmete açmıştır.

GitHub'ın REST tabanlı API'ler dolayısıyla yaşadığı sorunlardan en önemlisi ölçeklenebilirliktir (scalability). GitHub resmî sitesinde verilen bilgiye göre GitHub'ın veri tabanı çağırımlarının %60'ı REST API'lerinden kaynaklı olmaktadır. Bu durumun altında yatan temel nedenin istemcilerin veri gereksinimi ihtiyaçlarındaki farklılıklar olduğu söylenebilir zira bu farklılıkları karşılamak adına tek bir işlevi gerçekleştirmek için bile zaman zaman birçok farklı sunucu kaynağından veri dönen farklı API'lerin (Örneğin validasyon API'leri) farklı sayı ve sırada çağırılması gerekmektedir. Bu bakımdan GitHub REST API'lerinin uzun vadede sunucu kaynakları üzerindeki verimsiz kaynak tüketimi sistemin bütününe ölçeklenebilirliğini güçleştirmiştir. Diğer yandan farklı sıra ve sayıda çağrılan REST API'lerine erişimlerin yetki kontrolünün de yapılması zorlaşmıştır. Bu nedenlerle GitHub GraphQL API'lerine geçmektedir (Daigle, 2016).

1.4.4.2. Courier

Courier firması, kurumsal firmaların müşterilerine ürünleriyle ilgili Short Message Service (SMS) veya e-posta bildirimi yapmalarına ve bu bildirimlerin dağıtım durumunu kontrol etmelerine olanak tanıyan bir bilişim altyapısı sunmaktadır. Bu altyapı, kurumsal firmaların müşterilerine yönelik kampanyalarını yönetmelerine ve ürünlerini müşteri taleplerine göre Şekillendirmelerine olanak tanıyan analitik servisler sunmaktadır.

Courier, halihazırda kurumsal müşterilerine sunduğu servislerin pek çoğunu REST API'leri üzerinden sağlamaktadır. Kurumsal müşterilerin REST API'lerine entegre olması her ne kadar kolay da olsa REST tekniğinin esnek olmayan bir veri sorgulama yapısı müşterilerin anlık değişebilen analitik sorgu ihtiyaçlarını karşılamakta sıkıntı oluşturmıştır. Bu sıkıntının altında yatan esas neden REST API'lerinin beklediği ve sunduğu input/output veri yapılarının statik olmasıdır. Bir başka deyişle kurumsal müşterilerin geliştirdiği istemci uygulamaları analitik gereksinimlerini karşılamak adına statik veri yapıları sunan Courier REST API'lerini farklı sayı ve sırada tekrar tekrar çağırılmaktadır. Bu durum gereksiz/verimsiz network çağırımlarına neden olmakta ve istemcilerinin işlevlerini yerine getirmekte geç kalmalarına ve müşteri kaybetmelerine

neden olmaktadır. Bu nedenle Courier bünyesinde REST API'lerin GraphQL API'lerine evrilmesi süreci başlatılmıştır (Carney, 2020).

1.4.4.3. New York Times Gazetesi

New York Times, 18 Eylül 1851'de kurulmuş olan ve New York şehrinde basılan günlük bir Amerikan gazetesidir. Ülkenin önde gelen gazetelerinden sayılmaktadır. Günümüzde mobil aplikasyon ve web siteleri üzerinden farklı içerikte dünya genelinden haberler yayınlamaktadır. Bu aplikasyon ve web sitesi haber kaynakları toplamak ve uygun formatta okuyucuya sunmak için farklı önyüz teknolojileri kullanmakta, çok sayıda farklı REST API'sine çağırım yapmakta ve hatta kişiye özel içerik göstermek için okuyucularının kimlik tespitini (identification) yapmaktadır. Gerek içerik çekmekte kullanılan gerekse kimlik tespitinde kullanılan API'lerin çokluğu çok sayıda verimsiz network çağırımına neden olmakta ve bu durum haber içeriklerinin müşterilere sağlanmasını geciktirmektedir. Gazete buna bağlı olarak API alt yapısını GraphQL teknolojisine geçirmeye karar vermiştir (Taylor, 2017).

1.4.4.4. Banco Bilbao Vizcaya Argentaria (BBVA) Bankası

GraphQL'in kullanımın bankacılık sektöründe de kullanımı başlamıştır. Madrid-İspanya merkezli çok uluslu bir banka olan BBVA bu konuda başı çekmektedir. BBVA'nın GraphQL kullanmayı seçmesindeki ilk neden REST'in aksine GraphQL'in oluşabilecek her yeni veri ihtiyacı için yeni bir API yazımını gerektirmiyor olmasıdır. Bunun altında yatan neden GraphQL API'lerini çağırım sırasında yazılan sorgu parametrelerinin farklı ihtiyaçlar doğrultusunda esnetilmeğe açık olmasıdır. Burada esneklikle ifade edilmek istenen husus aynı API çağırımını farklı içerikte zenginleştirilmiş sorgular yaparak kullanılabilme kabiliyetidir.

BBVA GraphQL'i tercih etmesindeki ikinci neden ise API geliştirme zamanının REST'e göre oldukça kısa olmasıdır. Nitekim BBVA iddia'sına göre GraphQL sayesinde API geliştirme süresi toplamda 2 aydan 3 hafta dolaylarına inmiştir (Martinez, 2019).

1.4.4.5. Coursera Eğitim Platformu

Coursera, Stanford Üniversitesi'nden Bilgisayar Bilimleri profesörü Andrew Ng ve Daphne Koller tarafından kurulmuş, çeşitli alanlarda ve konularda kitlesel web tabanlı eğitim veren bir açık öğretim platformudur.

Coursera'nın REST API'lerine yaklaşımı, her bir kaynak grubu için bir API grubu (yani eğitimler için bir API grubu, eğitmenler için bir API grubu, ders notları için bir API grubu vb.) oluşturmaktır. REST'in önyüzle sunucu (backend) sistemleri ayrıştırmada (separation of concerns) getirdiği kolaylık sayesinde bu API grupları ilk etapta kolayca geliştirilmiştir. Ancak, Coursera platformunun sunduğu dersler ve katılımcıların sayısı arttıkça API'lerin ölçeklenmesi zorlaşmış performans sorunlarıyla karşılaşmıştır zira birçok sayfada işlenmesi gereken verileri almak için API'ler üzerinden sunucuya dört veya beş gidiş-dönüş yapılması gerekmiştir. Bu durum Coursera mühendislerine REST API'lerini bütünüyle GraphQL API'lerine dönüştürmeyi düşündürmüş olsa da REST API'lerin devasa sayısı bütün API'lerin GraphQL'e dönüşümünü çok maliyetli kılmıştır. Bu nedenle REST API'lerini bütünüyle kaldırmak yerine araya GraphQL esaslı proxy API'ler yazmak suretiyle halihazırdaki API'lerdeki değişiklikleri minimal düzeyde tutmuşlardır (Kane, 2020).

1.5. Açık Bankacılık API Standartları

Açık bankacılık, bir müşterinin tek bir erişim noktası kullanarak farklı bankaların ürünlerini ve hizmetlerini tek bir yerden kullanabilmesini sağlayan standartlar bütünü olarak ifade edilebilir. Burada tek bir erişim noktası ile işaret edilmek istenen kavram bir üçüncü taraf hizmet sağlayıcı (Third Party Provider (TPP)) ve bu sağlayıcı tarafından sağlanan bilgisayar yazılımıdır (yani günümüz şartlarında mobil cihazlarda çalışan bir uygulamadır).

Açık bankacılık standartları, TPP'lerin bankalarla iş birliği yaparak ortaya koyabilecekleri hizmet sağlama yöntemleri ve ürünlerindeki çeşitlilikten kaynaklanabilecek karmaşayı ortadan kaldırma amacıyla Avrupa Birliği tarafından Ödeme Hizmetleri Yönergesi (PSD2, Yönerge (AB) 2015/2366) kapsamında ortaya konmuş standartlar bütünüdür. PSD2, güçlü müşteri kimlik doğrulaması (Strong Customer Authentication), üçüncü taraf hizmet sağlayıcı (TPP), hesap bilgileri ve ödeme başlatma hizmetlerinin tanımlarını ve kurallarını belirtir (Parker B., 2019). Bu bakımdan

PSD2, bankalarla TPP’ler arasındaki etkileşimlerde bulunması gereken açık bankacılık ekosistemi ögelerini (Örnek: Ödeme servisi hizmet sağlayıcıları, hesap hizmeti ödeme sağlayıcıları gibi) ortaya koyar ve bu ögelerin görevlerini standardize eder. Bununla birlikte PSD2 yönergeleri teknik minvalde bu ögelerin birbirleri ile nasıl (yani hangi Uygulama Programlama Ara Yüzleri (API’ler) ile) etkileşime gireceği konusunda açık bir standart ortaya koymaz. Bu nedenle dünyada açık bankacılık ekosistemindeki ögelerin API’ler üzerinden etkileşimini standardize edici konseyler ve/veya araştırma grupları ortaya çıkmış ve bu suretle PSD2 yönergeleri çerçevesinde Açık Bankacılık API’lerinin oluşturulmasına gayret gösterilmiştir. Bu gayretler neticesinde ortaya çıkan açık bankacılık API standartlarından günümüzde dünya genelinde en yaygın olarak kullanılanları Almanya menşeli Berlin Grubu Standartları ile İngiltere menşeli ve UK Açık Bankacılık Standartlarıdır (UK Open Banking Standards). Bundan sonraki bölümdeki amacımız bu iki standardın genel yapısını ifade edip aralarındaki temel farklılıkları ortaya koymaktır.

1.5.1. Açık Bankacılık Standartlarının Kullanım Alanları

Bu bölümde, PSD2 yönergesi çerçevesinde hazırlanan “Berlin Group API” ve “Open Banking UK API” standartları kullanım alanları bakımından incelenip karşılaştırılmaktadır. Open Banking UK API Standartları “Ödeme Başlatma”, “Hesap Bilgisi” ve “Açık Veri Servisi” kullanım alanlarını desteklemektedir. Berlin Grubu API Standartları ise “Ödeme Başlatma” ve “Hesap Bilgisi” kullanım alanlarını desteklemektedir (KPMG, 2017). Kullanım alanındaki bu farklılıklar Tablo 1.2’de özetlenmektedir. Bu bölümün takip eden alt başlıklarında bu kullanım alanları minvalinde değerlendirilen servislerin ortaya koyduğu işlevler açıklanmaktadır.

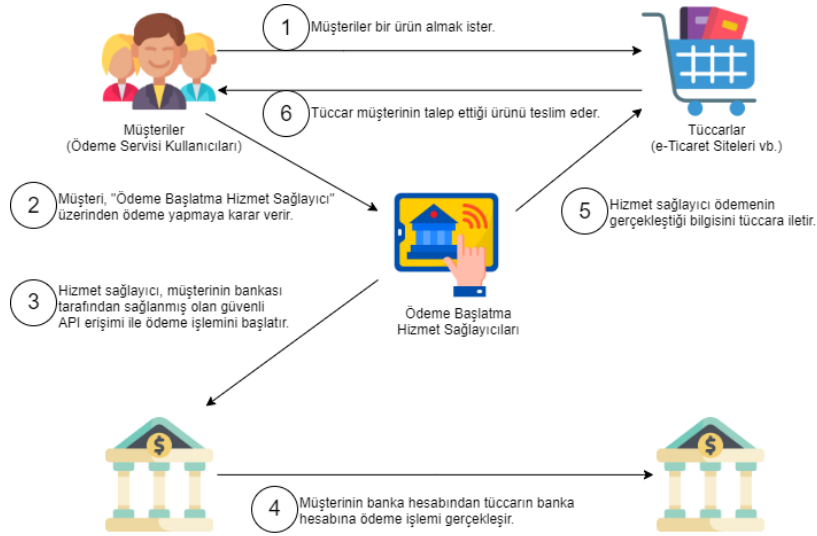
Tablo 1.2. API Standartlarında Kullanım Alanları

API Standartlar	Ödeme Başlatma	Hesap Bilgisi	Açık Veri Servisi
Berlin Grubu API Standartları	X	X	
UK API Standartları	X	X	X

1.5.1.1. Ödeme başlatma servisi

Ödeme başlatma servisi, web üzerinden gerçekleşen ticari işlemlerle alakalı ödeme işlemlerinin gerçekleştirilmesini sağlayan servis veya servisler grubu olarak ifade edilebilir. Bu servisler Ödeme Başlatma Servisi Sağlayıcısı niteliğinde olan TPP'ler tarafından sağlanmakta olup TPP'lere bu nitelik merkez bankası gibi bir devlet otoritesi tarafından gerçekleştirilen değerlendirme süreci sonrasında verilir. Türkiye'de bu süreç 6493 sayılı Ödeme ve Menkul Kıymet Mutabakat Sistemleri, Ödeme Hizmetleri ve Elektronik Para Kuruluşları Hakkında Kanun çerçevesinde husule gelmektedir.

Ödeme servislerinin kullanımında Şekil 1.29'da görüldüğü gibi müşteriler, tüccarlar, üçüncü parti hizmet sağlayıcılar (TPP'ler) ve bankalar olmak üzere 4 temel unsurdan söz edilebilir. Bu unsurlardan ödeme işlemini başlatan taraf müşteri; işlemi gerçekleştirecek taraf TPP, işlemle ilgili ödemeyi alacak olan taraf tüccar; ödemeyi transfer edecek unsur ise müşterinin bankasıdır. Bu unsurlardan ilki olan müşteriler bir ürünü TPP'nin sağlayacağı bir aplikasyon (Örnek: Trendyol gibi bir e-ticaret sitesi) üzerinden seçerek bir tüccardan almak isterler. Sonrasında seçtikleri ürünleri aplikasyon tarafından görüntülenen sepetlerine ekleyerek ödeme işlemini baştırılar. Bu aşamada aplikasyon müşteri adına banka ile iletişime geçer. Bu iletişimin sağlanması için en temel şart aplikasyon ile banka (ya da bankaların) yazılım altyapıları arasında güvenliği tesis edilmiş bir web tabanlı iletişim mekanizmasının kurulmasıdır. Bu mekanizmanın günümüz endüstrisindeki karşılığı Uygulama Programlama Ara Yüzüdür (Application Programming Interface-API). Dolayısıyla aplikasyon banka tarafından sağlanan ödeme servisi API'lerini kullanarak müşteri adına sepetteki ürünlerin satın alınması için gerekli finansal prosedürü başlatır. Bunun akabinde ilgili müşterinin bankası tarafından tüccarın banka hesabına ürünlerin tutarı yatırılır ve yatırma işleminin gerçekleştiği bilgisi Tüccar'a TPP tarafından iletilir. Bu sürecin tamamlanması ile müşterinin talep ettiği ürünü alması sağlanmış olur. Bu süreç Şekil 1.35'te görülmektedir.



Şekil 1.35. Ödeme Başlatma Hizmeti Genel Yapısı

Açık bankacılık standartlarında hizmet sağlama API'lerinin geliştirilmesi ile bu API'ler üzerinden güçlü kimlik doğrulaması esaslı (strong customer authentication) güvenli iletişimin sağlanması günümüzde bankaların sorumluluğundadır (Premchand A., Choudhry A., 2018). Bu kapsamda UK standartları aşağıdaki ödeme türlerini sağlayan API standartları ortaya koyar (Eroglu, 2017):

- Tek Ödeme
- Planlanmış Ödemeler
- Bekleyen Emirler
- Dosya Ödemeleri

Berlin grubu tarafından ortaya konan standartlar UK standartlarına göre biraz daha genişletilmiş API standartları ortaya koymaktadır. (Eroglu, 2017):

- Tek Ödeme
- Planlanmış Ödemeler
- Bekleyen Emirler
- Dosya Ödemeleri
- Tek Ödeme
- İleri Tarihli Tek Ödemeler
- Toplu Ödemeler
- Yinelenen Ödemeler
- Ödeme İptali

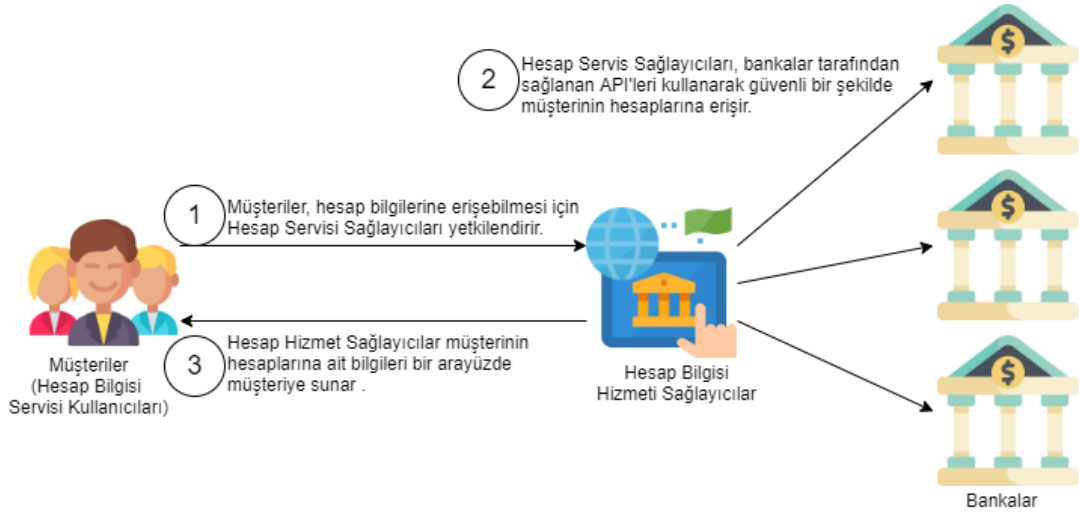
Yukarıda belirtilen listelerden de anlaşılabileceği üzere Berlin Grubu Standartlarında verilen API'ler UK standartlarında verilen API'leri kapsarken ek bazı API'leri de barındırmaktadır. Bu bakımdan Berlin gurubu standartlarındaki ödeme hizmetleri API'leri daha zengindir. İki standart arasındaki belli bazlı farklılıklardan en dikkat çekenleri şu şekilde ifade edilebilir:

- UK API standartlarında ödemenin iptali ile ilgili bir standart belirtilmemiştir.
- Berlin Group standartlarında ödeme iptali için ilgili API standartları verilmiştir.
- Toplu ve tekrarlı işlemlerin yapılmasına Berlin Grubu API'lerinde yer verilirken; UK standartları API'lerinde yer verilmemiştir.

1.5.1.2. Hesap bilgisi servisi

Hesap bilgisi servisi, müşterilerin bankalarındaki bakiye bilgisi ve hesap hareketleri gibi hesaplarıyla ilgili bilgilere erişebilmelerini sağlayan servis ya da servisler grubunu ifade eder. Hesap bilgisi servisleri Hesap Bilgisi Servis Sağlayıcısı olan TPP'ler tarafından sağlanır. TPP'ler, müşterilerinden banka hesaplarına erişmek için güçlü kimlik doğrulaması (strong customer authentication) esaslı elektronik rıza aldıktan sonra müşterilerin bankalarındaki hesap bilgilerine hesap bilgisi servislerini kullanarak erişebilirler. Web tabanlı bir network bağlantısı üzerinden gerçekleştirilen bu servis erişiminin gerçekleşmesi için TPP'lerin bankalar ile API entegrasyonu yapmaları gerekmektedir (Premchand A., Choudhry A., 2018).

Hesap bilgisi servislerinin gerçekleştirilmesindeki süreçte yer alan unsurlar sırasıyla müşteriler, üçüncü parti hizmet sağlayıcılar (TPP'ler) ve bankalar olarak tasavvur edilebilir. Bu unsurlar Şekil 1.30'da görüldüğü gibi işleyen bir süreç üzerinden birbirleri ile etkileşimde bulunurlar. Bu etkileşimlerin gerçekleştirilmesindeki en temel unsur olan müşteriler, banka hesaplarını ve bu hesaplarında oluşan para hareketlerini görmek için TPP'ler tarafından sağlanan uygulamaları kullanırlar. Müşterilerin bu uygulamaları kullanarak yaptığı talep sonrasında TPP uygulamaları müşterilerin bankaları tarafından sağlanan API'leri kullanarak güvenli bir şekilde müşterinin hesaplarına erişirler. Bankalardan alınan veriler TPP uygulamasının ara yüzü üzerinden müşterilere sunulur. Bu durum Şekil 1.36'da görülmektedir.



Şekil 1.36. Hesap Bilgisi Servisi Genel Yapısı

UK API Standartlarında, aşağıda listelenen hesap servislerini sunan API'ler standartlaştırılmıştır (Noctor, 2018; Michael, 2017):

- Hesap Detayları
- Hesap Bakiyeleri
- Hesap Hareketleri
- Hesap Lehtarları
- Hesap Özetleri
- Hesapla İlişkili Doğrudan Borçlar
- Hesapla İlgili Bekleyen Emirler
- Hesapla İlgili Planlanmış Ödemeler
- Hesap İçin Mevcut Teklifler

Berlin Grubu API Standartlarında ise şu servisleri sunan API'ler standartlaştırılmıştır (Noctor, 2018; Michael, 2017):

- Hesap Detayları
- Hesap Bakiyeleri
- Hesap Hareketleri

Hesap Bilgisi Servisleri kapsamında UK API Standartları Berlin Grubu Standartlarına göre daha kapsamlıdır. Bununla birlikte her iki standartta da müşterilerin

hesaplarını, bakiyelerini ve bakiye hareketlerini görüntülemelerini sağlayan API standartlarına yer verilmiştir. Diğer yandan müşterilerin ileri tarihli borçları ile bu borçları ile ilişkili lehtarları ve hesapları üzerindeki ileri tarihli emirleri ve ödemeleri görüntüleyebilmeleri için gerekli API Standartlarına UK API standartlarında yer verilirken; Berlin Grubu standartlarında bunlara yer verilmemiştir.

1.5.1.3. Açık veri servisi

Açık veri servisi şube yeri belirleme, Automated Teller Machine (ATM) yeri belirleme gibi direk olarak bankaların kendisi ile ilgili olan verilerin paylaşılmasını kapsayan servis ya da servisleri ifade eder. Bu bağlamda UK Standartları bankacılık ürünlerini keşfetmeyi sağlamaya yönelik bir dizi açık bankacılık veri bankacılık servisi ile bu servisleri çağırma olanak tanıyan API standartları ortaya koymuştur. Bununla birlikte Berlin Grubu API Standartları açık veri servislerini kapsamaz. UK standartları kapsamında sunulan açık veri servisleri aşağıdaki gibi listelenebilir:

- Şube Yeri Belirleme
- ATM Yeri Belirleme
- Kişisel Cari Hesaplar
- Ticari Cari Hesaplar
- İşletme (Kobi) Kredileri
- Kredi Kartları

1.5.2. Veri alanları

Ödeme Sistemleri Direktifi (PSD2), bankalar ve TPP'ler arasında kullanılacak API servislerinde standart veri alanları oluşturulmasını gerektirmektedir. Örneğin, ABC isimli bir TPP'nin geliştirdiği bir uygulama bir müşterinin hesaplarının bulunduğu X ve Y bankalarındaki hesap bilgilerini çekerken gerek X bankasına gerekse Y bankasına standart bir yapıda ve birbirinin aynı sorgulama parametrelerini kullanabilmelidir. (Eroglu, 2017).

1.5.3. Onay modelleri

Hesap Bilgisi Hizmet Sağlayıcısı TPP'ler ile Ödeme Başlatma Hizmet Sağlayıcısı TPP'lerin müşteri hesaplarına güvenli bir şekilde erişmesini sağlayacak bazı standart onay modellerinin sunulması gerekmektedir. Bu onay modelleri yetkisiz kişilerin hesaplara erişmesini engelleyici bir parametreler barındırabilmelidir (Perng, 2019). Bu amaçla gerek Berlin Grubu standartlarında gerekse UK API Standartlarında, müşteri adına otomatik ödeme başlatabilmesi amacıyla müşteriden rıza alıcı onay modelleri hakkında standartlar oluşturulmuştur. Özellikle Berlin Grubu API Standartlarında bu onay modellerinin belirli bir süre geçerli olması için ilave bazı standart veri yapıları da ortaya konmuştur (Eroglu, 2017).

1.5.4. Mesaj biçimleri

TPP'lerle bankalar arasında gerçekleştirilecek API trafiğinin parametre alan adları ve bu alanların veri türleri belirli bazı standartlara haiz olmalıdır. Bu bağlamda UK API Standardında yalnızca ISO20022'ye dayalı olan ve Java Script Object Notation (JSON) formatında veri alanları kullanılmaktadır. Berlin Grubu API standardında ise ISO20022 dışında alternatif endüstri standartları sunulmaktadır.

1.5.5. Kimlik doğrulama

Kimlik doğrulama, müşterilerin hesabına erişim sağlayacak kişilerin gerçekten müşteri olduğunun doğrulamasını sağlamayı amaçlar. Bir başka ifadeyle müşterilerin hesaplarına yetkisiz kişilerin erişimini engellemeyi amaçlar (Barbasura, 2021). Bu doğrultuda kullanılacak bazı yöntemler mevcuttur. Bu yöntemlere örnek olarak şunlar verilebilir:

- Müşterinin bildiği bir şey (Örnek: Şifre, PIN numarası)
- Müşterinin sahip olduğu bir şey (Örnek: Token, smard card, ATM kartı)
- Müşteriye ait biyometrik tanılama (Parmak izi, ses tanıma, retina taraması gibi)

Diğer yandan güçlü kimlik doğrulama (strong customer authentication) için bu yöntemler tek başına yeterli değildir zira tek başına kullanıldıklarında kimlik doğrulamayı

zafiyete uğratmaktadırlar. Örneğin biyometrik doğrulama ile birlikte müşteri no ve şifrenin de alınması gerekmektedir (Loesch S., 2018).

Güçlü kimlik doğrulama metotlarının uygulanabilmesi için kullanılan üç farklı yöntem şu şekildedir:

- Yönlendirmeli (Redirection) Güçlü Kimlik Doğrulama
- Ayrık Güçlü Kimlik Doğrulama
- Gömülü Güçlü Kimlik Doğrulama

Open Banking UK API Standardı “Yönlendirme (redirection)” yaklaşımı aracılığıyla güçlü kimlik doğrulama (GKD) yapmayı desteklemektedir. Yönlendirmeli kimlik doğrulamada müşteri bankanın bu doğrulama için hazırlanan sayfasına yönlendirilir. Müşteri açılan sayfa üzerinden müşteri numarası ve şifresini girerek bankanın müşterisi olduğunu doğrular. Berlin Grubu API Standartlarında ise yönlendirme yaklaşımına ek olarak “Ayrık GKD” ve “Gömülü GKD” yaklaşımları da sunulmaktadır.

1.5.6. Kullanıcı deneyimi

Bankalar tarafından gerek hesap bilgisi hizmet sağlayıcısı TPP'lere gerekse ödeme başlatma hizmet sağlayıcısı TPP'lere sunulacak olan API'lerin özelliklerinin bulunduğu dokümantasyonların paylaşılması TPP'lerin bankalarla entegrasyonunu sağlamak adına önem arz etmektedir zira dokümantasyon sayesinde TPP'ler müşterilerine neyi nasıl sunacaklarını daha iyi bilerek müşterilerine daha iyi kullanıcı deneyimi sunabilirler (Eroglu, 2017).

Bu bağlamda UK Standartları, API'lerin kullanımına onay verme, çift faktörlü kimlik doğrulama (2FA) ve hesap bilgisi/ödeme hizmetlerinin dokümantasyonunu kullanıcı deneyimini artırıcı minvalde kolaylaştıracak standartlar ortaya koymuştur. Bununla birlikte Berlin Grubu standartlarında her bankanın kendi kullanıcı deneyimini kendinin tasarlamasına izin veren daha esnek bir dokümantasyon yapısı ortaya konmuştur.

1.5.7. Hesap bilgilerine erişim rıza (Consent) uç noktaları (API'leri)

Hesap bilgisi hizmet sağlayıcısı olan TPP'ler müşterilerinin hesap bilgilerine erişebilmek için onlardan elektronik ortamda rıza (consent) almalıdırlar. Müşteriler,

TPP'ler için kendileri rıza oluşturabileceği gibi oluşturulan bir rızayı da istedikleri zaman iptal edebilmelidirler. Ayrıca rıza işlemlerinin gerçekleştirilmesi için gerekli olabilecek API'ler bankalar tarafından geliştirilmeli ve belirli bazı standartlara haiz olmalıdır (Eroglu H.,2017). UK standartlarında ve Berlin Grubu standartlarında hesap bilgisi sorgulamak için geliştirilen rıza API'leri ile alakalı standartlarda farklılıklar mevcuttur. UK standartlarındaki rıza API'leri (başka bir deyişle uç noktaları) Tablo 1.3'te ve 1.4'te de görüldüğü gibi Berlin Grubu rıza API'lerine göre daha sınırlıdır.

Tablo 1.3. Open Banking UK Rıza Uç Nokta Standartları (Gyara, 2019)

Uç Nokta URL	Zorunluluk	Metot	Güvenlik Türü	Açıklama
account-access-consents	Zorunlu	POST	Client Credentials	Rıza Oluşturma
account-access-consents/{ConsentId}	Zorunlu	GET	Client Credentials	Rıza Sorgulama
account-access-consents/{ConsentId}	Zorunlu	DELETE	Client Credentials	Rıza Silme

Tablo 1.4. Berlin Group Rıza Uç Nokta Standartları (Anonim, 2023 (3))

Uç Nokta URL	Zorunluluk	Metot	Güvenlik Türü	Açıklama
/consents	Zorunlu	POST	Authorization Code	Rıza Oluşturma
/consents/{consentId}	Zorunlu	GET	Authorization Code	Rıza Sorgulama
/consents/{consentId}	Zorunlu	DELETE	Authorization Code	Rıza Silme
/consents/{consentId}/status	Zorunlu	GET	Authorization Code	Rıza Durumu Sorgulama
/consents/{consentId}/authorizations	Zorunlu	GET	Authorization Code	Rıza Yetkisi Sorgulama
/consents/{consentId}/authorizations	Zorunlu	POST	Authorization Code	Rıza Yetkisi Oluşturma
/consents/{consentId}/authorizations/{authorizationId}	Zorunlu	GET	Authorization Code	
/consents/{consentId}/authorizations/{authorizationId}	Zorunlu	PUT	Authorization Code	

1.5.8. Hesaplar (Accounts) uç noktaları (API'leri)

Hesap bilgisi hizmet sağlayıcı TPP'leri, sırasıyla bakiye, işlem hareketleri, ödeme emirleri ve taksit bilgileri gibi müşterilerin hesaplarına ait bilgilere bankalar tarafından geliştirilip sağlanacak olan API'ler üzerinden erişebilmelidir. Bu kapsamda bankaların geliştireceği API'ler belirli standartlara haiz olmalıdır (Eroglu H.,2017). UK API

standartlarında ve Berlin Grubu API standartlarında hesap bilgisi sorgulamak için geliştirilen API'ler farklılık göstermektedir. Aşağıda Tablo 1.5'te ve 1.6'da UK ve Berlin Group tarafından geliştirilen Hesap API'leri listelenmektedir.

Tablo 1.5. Open Banking UK Hesap Bilgileri Uç Nokta Standartları (Michael, 2017 (1))

Uç Nokta URL	Zorunluluk	Metot	Güvenlik Türü	Açıklama
/accounts	Zorunlu	GET	Authorization Code	Hizmet sağlayıcının yetkisinin olduğu tüm hesapların listesini verir.
/accounts/{AccountId}	Zorunlu	GET	Authorization Code	Belirli bir hesap numarası için hesap bilgilerini verir.
/balances	Tercihe Bağlı	GET	Authorization Code	Hizmet sağlayıcının yetkisinin olduğu tüm hesapların bakiye bilgisini verir.
/accounts/{AccountId}/balances	Zorunlu	GET	Authorization Code	Belirli bir hesap numarası için bakiye bilgisini verir.
/transactions	Tercihe Bağlı	GET	Authorization Code	Hizmet sağlayıcının yetkisinin olduğu tüm hesaplara ait işlem hareketleri bilgisini verir.
/accounts/{AccountId}/transactions	Zorunlu	GET	Authorization Code	Belirli bir hesap numarası için işlem hareketlerini verir.
/beneficiaries	Tercihe Bağlı	GET	Authorization Code	Hizmet sağlayıcının yetkisinin olduğu tüm hesaplara ait lehtarların bilgisini verir.
/accounts/{AccountId}/beneficiaries	Koşullu	GET	Authorization Code	Belirli bir hesap numarası için lehtar bilgisini verir.
/direct-debits	Tercihe Bağlı	GET	Authorization Code	Hizmet sağlayıcının yetkisinin olduğu tüm hesaplara ait otomatik ödeme bilgilerini verir.
/accounts/{AccountId}/direct-debits	Koşullu	GET	Authorization Code	Belirli bir hesap numarası için otomatik ödeme bilgisini verir.
/standing-orders	Tercihe Bağlı	GET	Authorization Code	Hizmet sağlayıcının yetkisinin olduğu tüm hesaplara ait açık emir bilgilerini verir.
/accounts/{AccountId}/standing-orders	Koşullu	GET	Authorization Code	Belirli bir hesap numarası için açık emir bilgisini verir.
/products	Tercihe Bağlı	GET	Authorization Code	Hizmet sağlayıcının yetkisinin olduğu tüm hesaplara ait kullanılan ürünlerin bilgisini verir.
/accounts/{AccountId}/product	Koşullu	GET	Authorization Code	Belirli bir hesap numarası için kullanılan ürünlerin bilgisini verir.
/offers	Tercihe Bağlı	GET	Authorization Code	Hizmet sağlayıcının yetkisinin olduğu tüm hesaplara ait teklifler bilgisini verir.
/accounts/{AccountId}/offers	Koşullu	GET	Authorization Code	Belirli bir hesap numarası için teklifler bilgisini verir.
/party	Koşullu	GET	Authorization Code	Hizmet sağlayıcının yetkisinin olduğu tüm hesaplara ait hesap sahipleri bilgisini verir.
/accounts/{AccountId}/party	Koşullu	GET	Authorization Code	Belirli bir hesap numarası için hesap sahipleri bilgisini verir.

/accounts/{AccountId}/parties	Koşullu	GET	Authorization Code	
/scheduled-payments	Tercihe Bağlı	GET	Authorization Code	Hizmet sağlayıcının yetkisinin olduğu tüm hesaplara ait planlanan ödemeler bilgisini verir.
/accounts/{AccountId}/scheduled-payments	Koşullu	GET	Authorization Code	Belirli bir hesap numarası için planlanan ödemeler bilgisini verir.
/statements	Tercihe Bağlı	GET	Authorization Code	Hizmet sağlayıcının yetkisinin olduğu tüm hesaplara ait ekstre bilgisini verir.
/accounts/{AccountId}/statements/{StatementId}/transactions	Koşullu	GET	Authorization Code	Belirli bir hesap numarası için ekstre detay hareketler bilgisini verir.
/accounts/{AccountId}/statements/{StatementId}/file	Tercihe Bağlı	GET	Authorization Code	Belirli bir hesap numarası ve belirli ekstre numarası için dosya şeklinde ekstre verir.
/accounts/{AccountId}/statements/{StatementId}	Koşullu	GET	Authorization Code	Belirli bir hesap numarası ve belirli bir ekstre numarası için ekstre bilgisini verir.
/accounts/{AccountId}/statements	Koşullu	GET	Authorization Code	Belirli bir hesap numarası için ekstre bilgisini verir.

Tablo 1.6. Berlin Group Hesap Bilgileri Uç Nokta Standartları (Anonim, 2023 (3))

Uç Nokta URL	Zorunluluk	Metot	Güvenlik Türü	Açıklama
accounts	Zorunlu	GET	Authorization Code	Hizmet sağlayıcı-nın yetkisinin olduğu tüm hesapların listesini verir.
accounts?withBalance	Tercihe Bağlı	GET	Authorization Code	Hizmet sağlayıcı-nın yetkisinin olduğu tüm hesapları bakiye bilgisi ile birlikte verir.
accounts/{account-id}	Zorunlu	GET	Authorization Code	Belirli bir hesap numarası için hesap bilgilerini verir.
accounts/{accountid}?withBalance	Tercihe Bağlı	GET	Authorization Code	Belirli bir hesap numarası için hesap bilgilerini bakiye bilgisi ile birlikte verir.
accounts/{accountid}/balances	Zorunlu	GET	Authorization Code	Belirli bir hesap numarası için hesap hakkında ayrıntılı bakiye bilgisi verir.
accounts/{accountid}/transactions	Zorunlu	GET	Authorization Code	Belirli bir hesap numarası için işlem hareketlerini verir.
accounts/{accountid}/transactions?withBalance	Tercihe Bağlı	GET	Authorization Code	Belirli bir hesap numarası için işlem hareketlerini bakiye bilgisi ile birlikte verir.
accounts/{accountid}/transactions/{transaction Id}	Tercihe Bağlı	GET	Authorization Code	Belirli bir hesap numarası ve işlem numarası için detaylı işlem bilgisini verir.

1.5.9. Ödeme başlatma (Payment Initiation) uç noktaları

Ödeme başlatma hizmet sağlayıcısı olan TPP'ler, ödeme hizmetini kullanacak bankacılık müşterisi adına ödeme işlemi başlatabilir ve başlatılan ödeme hizmeti ile ilgili durumu sorgulayabilir. TPP'lerin bu ödeme başlatma ve durum sorgulama işlemlerini müşteri adına gerçekleştirmeleri için bu hizmetleri sunan ve bankalar tarafından geliştirilecek olan API'lere erişimleri gerekmektedir. Bankalarca geliştirecek bu API'lerin belirli bazı standartları olmalıdır (Eroglu H.,2017). UK API standardında ve Ayrıca Berlin Grubu API standardında ödeme hizmeti başlatmak için geliştirilen API'ler UK standartlarına göre daha zengindir. Aşağıda Tablo 1.7 ve 1.8'de UK ve Berlin Grubu tarafından ortaya konulan Ödeme Başlatma API'leri listelenmektedir.

Tablo 1.7. Open Banking UK Ödeme Başlatma Uç Nokta Standartları (Michael, 2017 (2))

Uç Nokta URL	Zorunluluk	Metot	Güvenlik Türü	Açıklama
/domestic-payments	Zorunlu	POST	Authorization Code	Hizmet sağlayıcı bir yerel ödemeyi başlatmak için kullanır.
/domestic-payments/{DomesticPaymentId}/payment-details	Tercihe Bağlı	GET	Client Credentials	Hizmet sağlayıcı bir yerel ödeme ile ilgili detaylı bilgiye erişmek için kullanır.
/domestic-scheduled-payments	Koşullu	POST	Authorization Code	Hizmet sağlayıcı yerel planlı bir yerel ödemeyi başlatmak için kullanır.
/domestic-scheduled-payments/{DomesticScheduledPaymentId}	Zorunlu	GET	Client Credentials	Hizmet sağlayıcı planlı bir yerel ödemenin durumunu sorgulamak için kullanır.
/domestic-scheduled-payments/{DomesticScheduledPaymentId}/payment-details	Tercihe Bağlı	GET	Client Credentials	Hizmet sağlayıcı planlı bir yerel ödeme ile ilgili detaylı bilgiye erişmek için kullanır.
/domestic-standing-orders	Koşullu	POST	Authorization Code	Hizmet sağlayıcı bir yerel ödeme emri başlatmak için kullanır.
/domestic-standing-orders/{DomesticStandingOrderId}	Zorunlu	GET	Client Credentials	Hizmet sağlayıcı planlı bir yerel bir ödeme emrinin durumunu sorgulamak için kullanır.
/domestic-standing-orders/{DomesticStandingOrderId}/payment-details	Tercihe Bağlı	GET	Client Credentials	Hizmet yerel ödeme emri ile ilgili detaylı bilgiye erişmek için kullanır.
/international-payments	Koşullu	POST	Authorization Code	Hizmet sağlayıcı bir uluslararası ödemeyi başlatmak için kullanır.
/international-payments/{InternationalPaymentId}	Zorunlu	GET	Client Credentials	Hizmet sağlayıcı bir uluslararası ödemenin durumunu sorgulamak için kullanır.
/international-payments/{InternationalPaymentId}/payment-details	Tercihe Bağlı	GET	Client Credentials	Hizmet sağlayıcı bir uluslararası ödemenin detaylı bilgisine erişmek için kullanır.
/international-scheduled-payments	Koşullu	POST	Authorization Code	Hizmet sağlayıcı uluslararası planlı bir ödemeyi başlatmak için kullanır.

/international-scheduled-payments/{InternationalScheduledPaymentId}	Zorunlu	GET	Client Credentials	Hizmet sağlayıcı uluslararası planlı bir ödemenin durumunu sorgulamak için kullanır.
/international-scheduled-payments/{InternationalScheduledPaymentId}/payment-details	Tercihe Bağlı	GET	Client Credentials	Hizmet sağlayıcı uluslararası planlı bir ödeme ile ilgili detaylı bilgiye erişmek için kullanır.
/international-standing-orders	Koşullu	POST	Authorization Code	Hizmet sağlayıcı uluslararası bir ödeme emri oluşturmak için kullanır.
/international-standing-orders/{InternationalStandingOrderPaymentId}	Zorunlu	GET	Client Credentials	Hizmet sağlayıcı uluslararası bir ödeme emrinin durumunu sorgulamak için kullanır.
/international-standing-orders/{InternationalStandingOrderPaymentId}/payment-details	Tercihe Bağlı	GET	Client Credentials	Hizmet sağlayıcı uluslararası bir ödeme emrinin detaylı bilgisini sorgulamak için kullanır.
/file-payments	Koşullu	POST	Authorization Code	Hizmet sağlayıcı bir dosya ödemesi oluşturmak için kullanır.
/file-payments/{FilePaymentId}	Zorunlu	GET	Client Credentials	Hizmet sağlayıcı bir dosya ödemesinin durumunu sorgulamak için kullanır.
/file-payments/{FilePaymentId}/report-file	Koşullu	GET	Client Credentials	Hizmet sağlayıcı sorgulanan bir dosya ödemesinin indirmek kullanır.
/file-payments/{FilePaymentId}/payment-details	Tercihe Bağlı	GET	Client Credentials	Hizmet sağlayıcı bir dosya ödemesinin detaylı bilgisini sorgulamak için kullanır.

Tablo 1.8. Berlin Group Ödeme Başlatma Uç Nokta Standartları (Anonim, 2023 (3))

Uç Nokta URL	Zorunluluk	Metot	Güvenlik Türü	Açıklama
payments/{paymentproduct}	Zorunlu	POST	Authorization Code	Hizmet sağlayıcı bir ödemeyi başlatmak için kullanır.
payments/{paymentproduct}/{paymentId}	Zorunlu	GET	Authorization Code	Hizmet sağlayıcı bir ödeme hizmetinin detaylı bilgisini sorgulamak için kullanır.
payments/{paymentproduct}/{paymentId}/status	Zorunlu	GET	Authorization Code	Hizmet sağlayıcı bir ödeme hizmetinin durumunu sorgulamak için kullanır.
bulk-payments/{paymentproduct}	Tercihe Bağlı	POST	Authorization Code	Hizmet sağlayıcı bir toplu ödemeyi başlatmak için kullanır.
bulk-payments/{paymentproduct}/{paymentId}	Zorunlu	GET	Authorization Code	Hizmet sağlayıcı bir toplu ödeme hizmetinin detaylı bilgisini sorgulamak için kullanır.
bulk-payments/{paymentproduct}/{paymentId}/status	Zorunlu	GET	Authorization Code	Hizmet sağlayıcı bir toplu ödeme hizmetinin durumunu sorgulamak için kullanır.
Periodic-payments/{paymentproduct}	Tercihe Bağlı	POST	Authorization Code	Hizmet sağlayıcı periyodik olarak tekrar eden bir ödemeyi başlatmak için kullanır.
periodicpayments/{paymentproduct}/{paymentId}	Zorunlu	GET	Authorization Code	Hizmet sağlayıcı periyodik olarak tekrar eden bir ödemenin detaylı bilgisine erişmek için kullanır.
periodicpayments/{paymentproduct}/{paymentId}/status	Zorunlu	GET	Authorization Code	Hizmet sağlayıcı periyodik olarak tekrar eden bir ödemenin durumunu sorgulamak için kullanır.

{paymentservice}/{paymentproduct}/{paymentId}/authorisations	Zorunlu	POST	Authorization Code	Hizmet sağlayıcı yetkilendirme alt kaynağı oluşturmak için kullanılır.
{paymentservice}/{paymentproduct}/{paymentId}/authorisations	Zorunlu	GET	Authorization Code	Oluşturulan tüm yetkilendirme alt kaynakları kimliklerinin listesine erişmek için kullanılır.
{paymentservice}/{paymentproduct}/{paymentId}/authorisations/{authorisationId}	Zorunlu	PUT	Authorization Code	Hizmet sağlayıcı tarafından gerekli olması durumunda yetkilendirme kaynağındaki verileri güncellemek için kullanılır.
{paymentservice}/{paymentproduct}/{paymentId}/authorisations/{authorisationId}	Zorunlu	GET	Authorization Code	Hizmet sağlayıcı güçlü kimlik doğrulamaya ilişkin yetkiyi sorgulamak için kullanılır.
{paymentservice}/{paymentproduct}/{paymentId}	Tercihe Bağlı	DELETE	Authorization Code	Hizmet sağlayıcının başlatılan ödemeleri iptal etmesi için kullanılır.
{paymentservice}/{paymentproduct}/{paymentId}/cancellation-authorisations	Tercihe Bağlı	POST	Authorization Code	Hizmet sağlayıcının başlatılan ödemeleri iptal etmesi için gereken yetkiyi almak için kullanılır.
{paymentservice}/{paymentproduct}/{paymentId}/cancellation-authorisations	Tercihe Bağlı	GET	Authorization Code	İptal edilen deme hizmetlerinin listesini verir.
{paymentservice}/{paymentproduct}/{paymentId}/cancellationauthorisations/{authorisationId}	Zorunlu	PUT	Authorization Code	Gerekli olması durumunda iptal yetkilerinde güncelleme yapmak için kullanılır.
{paymentservice}/{paymentproduct}/{paymentId}/cancellationauthorisations/{authorisationId}	Zorunlu	GET	Authorization Codes	İptal yetkisine ait güçlü kimlik doğrulamanın durumunu sorgulamak için kullanılır.

1.6. Monolit Mimari

1.6.1. Tanım

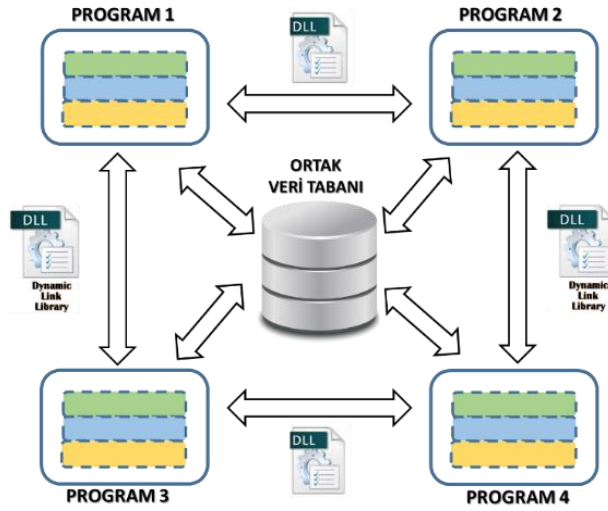
İstemci-sunucu mimarisine göre geliştirilmiş olan yazılımlar genel itibarıyla üç temel katman üzerine oturtularak geliştirilirler (Fowler, 2014). Sunum, uygulama ve veri erişim katmanları olarak adlandırılan bu katmanlardan *Sunum Katmanı*, son kullanıcının uygulamayla etkileşimde bulunduğu katmanı temsil eder. Bu katmanın temel amacı, kullanıcıya veri görüntülemek ve ondan veri toplamaktır. Bu katman, bir mobil uygulama, web tarayıcısı ya da masaüstü uygulaması tarafından direkt erişilebilen bir ara-yüz katmanıdır. Diğer yandan *Uygulama Katmanı* ise sunum katmanı çağırımlarını alarak işleyen ve bu doğrultuda ihtiyacı duyulan verileri veri katmanından çeken katman olarak tasavvur edilir ve genellikle Python, Java, Perl, PHP veya Ruby gibi yüksek

seviyeli programlama dilleri kullanılarak geliştirilir. Son olarak *Veri Erişim Katmanı* ise uygulama katmanı tarafından işlenen bilgilerin depolandığı ve yönetildiği arka-uç katmanını temsil eder. Bu katman bünyesinde, PostgreSQL, MySQL, MariaDB, Oracle, DB2, Informix veya Microsoft SQL Server gibi ilişkisel bir veri tabanı yönetim sistemi veya Cassandra, CouchDB veya MongoDB gibi ilişkisel olmayan (NoSQL) bir veri tabanı sistemi kullanılabilir (IBM, 2020).

Yazılım mühendisliği terminolojisinde monolit terimi, çok katmanlı yazılım mimarisinde mevcut bulunan sunum, uygulama ve veri erişim katmanlarının tamamının aşağıda Şekil 1.37’de görüldüğü tek bir yekpare katmanda toplandığı bir yazılım sistemini ifade etmek için kullanılır (Villamizar, 2015; Fowler, 2014). Bu yapıda yazılım sistemini oluşturan sunum, uygulama ve veri erişim katmanlarının tamamı Şekil 1.37’de görüldüğü gibi tek bir bilgisayar programı içinde barındırabileceği gibi bir bilgisayar ağı üzerinde dağıtık durumda birden fazla bilgisayar programını barındıran bir yapıda da olabilir. Dağıtık monolit (distributed monolith) olarak adlandırılacak olan bu ikinci durum aşağıda Şekil 1.38’de temsili olarak ifade edilmektedir. Bu şekilden anlaşılabileceği gibi monolit yazılımı oluşturan programlar (bir başka deyişle bileşenler) birbirlerine verdikleri kütüphane referansları (yani **Dynamically Linked Library-DLL**) bakımından ve veri tabanı üzerinde ortaklaşa kullandıkları saklı yordamlar (yani **stored procedure’ler**) ve tablolar bakımından sıkı sıkıya bağımlıdırlar (yani tightly-coupled yapıdadırlar).



Şekil 1.37. Üç Katmanlı Monolit Mimari



Şekil 1.38. Dağıtık Monolitlerden Oluşan bir Yazılım Sisteminin Kütüphane (DLL) ve Veri Tabanı Bakımından Bağımlılıkları

Günümüzdeki pek çok bankanın bilişim altyapısı bu tür dağıtık monolit yazılım sistemlerinden oluşmaktadır. Şelale (waterfall) modeline göre geliştirilmiş olan bu büyük ölçekli ve hantal yapıdaki dağıtık monolitlerin günümüzün bulut bilişim tabanlı yazılım mimarisine uyum sağlayabilmesi ve buna bağlı sektör ihtiyaçlarını karşılayabilmesi için modernize edilmesi gereklidir. Bununla birlikte monolit yazılım sistemlerinin modernize edilmesi son derece maliyetli ve uzun bir süreç haline dönüşebilir zira bu tür sistemlerin yapısında birbiriyle içe içe girmiş olan ve bağımlılıkları yüksek (coupled) çok sayıda yazılım bileşeninin mevcuttur. Bu bileşenlerin birinde yapılacak bir değişiklik sistemin bütününde yapılması gereken hataya açık, pahalı ve uzun sürebilecek değişiklik silsilesine neden olacaktır (Bucchiarone, 2018; Gouigoux ve ark., 2017).

1.6.2. Monolitlerin avantajları ve dezavantajları

Monolit mimaride, geliştirme, yaygınlaştırma, ölçeklendirme, iş yükü, karmaşıklık ve teknoloji kullanımı gibi parametrelere bağlı olarak değişen avantajlar ve dezavantajlar mevcuttur. Bu avantajlar ve dezavantajlar Tablo 1.9 ve 1.10'da belirtilmektedir (Richardson, 2019; Bucchiarone, 2018):

Tablo 1.9. Monolit Mimari Avantajları (Richardson, 2019; Bucchiarone, 2018)

GELİŞTİRME	Basitliği ve popülerliği sebebi ile herhangi bir yazılımcı tarafından kolaylıkla benimsenebilir.
	Bu mimarinin kullanımına yönelik çok sayıda geliştirme ortamı, platform sağlayıcı uygulamalar ve yardımcı araçlar hâlihazırda mevcuttur.
	Sıfırdan bir uygulamanın geliştirme ve yaygınlaştırma işleminde hız ve kolaylık sağlar.
YAYGINLAŞTIRMA	Monolit sistemlerde geliştirmeler tek bir yekpare uygulama üzerinde yapılmaktadır. Bu nedenle yapılacak bütün yeni geliştirmeler bu yekpare sisteme dâhil edilmek suretiyle yaygınlaştırma tek hamlede tamamlanmış olur.
ÖLÇEKLENDİRME	Aynı yazılım uygulamasının farklı numuneleri ayrı bilgisayarlara yüklenip bir yük dengeleyici (load-balancer) arkasında çalıştırılabilir. Bu durum iş yükünün aynı uygulamanın birden fazla numunesi üzerine eşit şekilde dağıtılmasına ve dolayısıyla aynı uygulamadan daha fazla verim alınmasına olanak tanır.

Tablo 1.10. Monolit Mimari Dezavantajları (Richardson, 2019; Bucchiarone, 2018)

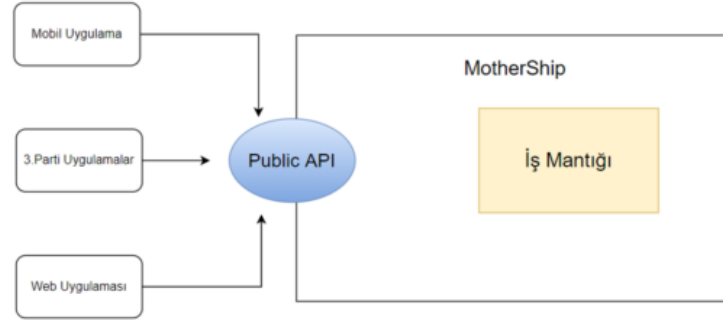
KOD	Yazılım tek bir kod tabanında geliştirildiği için yazılımı oluşturan bileşenlerin (yani modüllerin) büyümesi veya artması ile beraber kodun kendisinin ve bağımlılıkların yönetilmesi zorlaşır. Modüller arasındaki belirleyici sınırlar kaybolur ve kodun karmaşıklığı artar.
	Paylaşılmış kod, kütüphane ve veri tabanı kaynakları üzerinde geliştirme gerçekleştirildiği için monolit sistem bünyesinde oluşacak basit bir değişiklik bile bütün sistemin durmasına ve hizmet verememesine neden olabilir.
YAYGINLAŞTIRMA	Uygulamanın herhangi bir yerinde yapılan bir değişiklik için her seferinde uygulamanın yeni bir sürümünün oluşturulması gerekmektedir.
	Yaygınlaştırma sırasında monolit yazılım sistemi bütün bileşenleriyle topyekûn olarak duraklatılmakta ve akabinde yeni derlenen kod düzenlemeleri geçilmektedir. Bu topyekûn duraklama sırasında monolit sisteme bağlı çalışan başka bazı sistemlerin olumsuz etkilenmesi ve dolayısıyla onların da bütünüyle durdurulması gerekebilmektedir. Bu durum ciddi zaman ve iş gücü israfına neden olabilmektedir.
ÖLÇEKLENDİRME	Yapılan ölçeklendirme işlemi sırasında bütün yekpare monolit bazında topyekûn bir ölçeklendirmeye gidilir. Bu da gereğinden fazla kaynak yönetimine neden olur.
TEKNOLOJİ	Monolit uygulama tek bir platform üzerinde tek bir programlama dili ile yazıldığı için değişen gereksinimlere karşılayabilecek yeni bir teknolojinin uygulamaya kazandırılması zordur. Aynı platformda ve aynı dilde bir teknoloji kazandırılacak olsa bile bunun yan etkilerinin olma ihtimali birçok servisin aynı uygulamada çalışmasından dolayı oldukça yüksektir.
	Platforma ve programlama diline bağımlılık söz konusu olduğu için pek çok yeni teknoloji entegre edilemez.

1.6.3. Örnek monolit uygulamalar

1.6.3.1. SoundCloud

SoundCloud, web üzerinden müzik ve ses dosyalarını ücretsiz olarak dinlemeye, indirmeye ya da yüklemeye izin veren Berlin merkezli bir çevrimiçi web uygulamasıdır. Bunun yanında birçok web sitesi ve ürünle entegrasyonu olup, müzik ve ses dosyalarının görüntülenmesi, çalınması veya indirilmesi gibi ürün ve servisleri de vardır (Anonim, 2021(9)).

Hâlihazırda SoundCloud'ın altyapısında yazılım ve teknoloji olarak Scala, Clojure ve JRuby kullanılmıştır. Bununla birlikte ürün ilk kez piyasaya sürüldüğü zamanlarda monolitik bir Ruby on Rails uygulaması olarak geliştirilmiştir. Ruby on Rails açık kaynak kodlu bir web uygulama geliştirme çatısıdır (Anonim, 2021(10)). Bu monolit uygulamaya Şekil 1.39'da görüldüğü gibi *MotherShip* adı verilmiş olup müziklerin dağıtımı, işlenmesi, verinin yönetilmesi, kayıtların alınması gibi birçok işlemi geliştirmektedir. *MotherShip* üzerinde ayrıca *Public API* adını verdikleri ve üçüncü parti uygulamaların entegrasyonuna imkân tanıyan bir API servisi sunulmaktadır. (Anonim, 2021(9)).



Şekil 1.39. SoundCloud Monolit Mimarisi

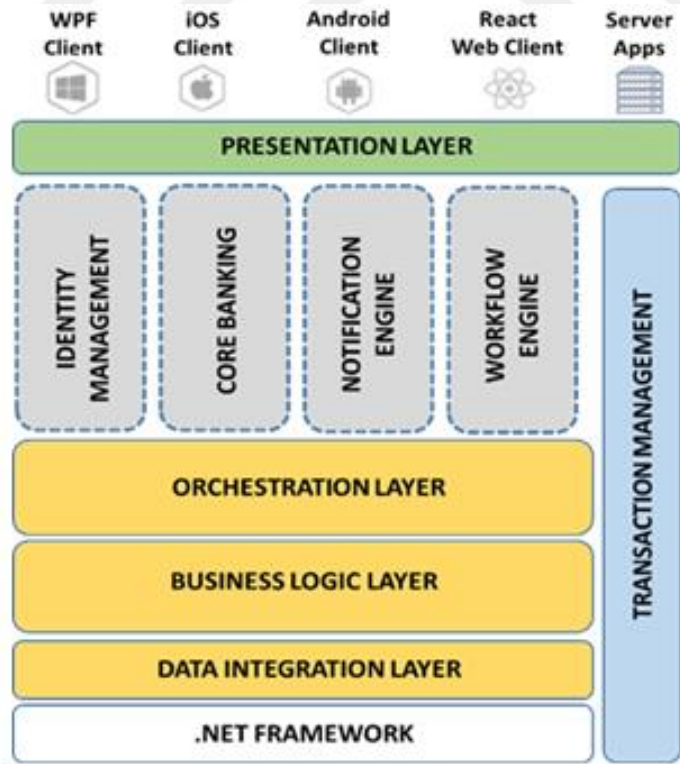
Bu monolit yapıyı kullanan kişi sayısı ilk başlarda 100 binlerle ifade edilirken uygulamanın şu an 40 milyon civarında kayıtlı kullanıcısı bulunmaktadır (Anonim, 2021(9)). 2007'de kurulan SoundCloud'ın müşteri sayısı arttıkça müziklerin işlenmesi istemci uygulamalarına dağıtılması gibi ağır işlemlerin tek bir yekpare sunucu uygulaması tarafından gerçekleştirilmesi ciddi performans ve ölçeklendirme sorunlarına neden olmuştur. SoundCloud teknik ekibi sistemi topyekûn ölçeklendirme için birçok araç geliştirmiştir ancak bu denli büyük bir uygulamanın asıl probleminin mimari olduğu

tespit edildikten sonra ekip *Mikro Servis Mimarisine* geçmeyi uygun görmüştür (Calçado, 2014).

1.6.3.2. Business Oriented Architecture (BOA)-Kuveyt Türk

Kuveyt Türk 1989 yılından beri Türkiye'de Katılım Bankası olarak faaliyet göstermektedir. Şu an yaklaşık 1,4 milyon müşteriye hizmet vermektedir. Banka kendi yazılımlarını kendi AR-GE merkezi bünyesinde 2010 yılından beri geliştirmektedir.

Kuveyt Türk AR-GE merkezi, Bankacılık Odaklı Mimari (Business Oriented Architecture-BOA) adını verdikleri monolit bir altyapı Şekil 1.40'da görüldüğü gibi çok katmanlı bir yapıda dağıtık bir monolit olarak (distributed monolith) oluşturmuştur. Bankacılık alanında mevcut bulunabilecek finansal yazılım bileşenleri bu monolit mimari içerisinde çalışmakta ve bütün istemci uygulamalara bu altyapıdan hizmet verilmektedir. BOA'nın iç bileşenleri, çoğunlukla Windows Communication Foundation (WCF), Windows Presentation Foundation (WPF) ve .NET gibi Microsoft teknolojilerini kullanılarak geliştirilmiştir (Anonim, 2021(11)).



Şekil 1.40. BOA Monolit Mimarisi

1.6.3.3. Uber

Uber, Amerikan menşeli bir uluslararası ulaşım ağı şirkettir. Araç çağırma, yemek teslimi (Uber Eats), paket teslimi, kurye, yük taşımacılığı gibi hizmetleri mobil uygulaması ve web uygulaması üzerinden vermektedir. 28 Mayıs 2015 itibarıyla, Uber dünya çapında 58 ülkede ve 300 şehirde kullanılmaktadır.

Uber'in teknolojik altyapısı Şekil 1.41'de görüldüğü gibi ilk olarak monolit mimari esaslı olarak tasarlanmıştır. Bu mimaride belirtilen *Rest API* bileşeni araç sürücüleri ile müşterilerin kullandığı Uber mobil uygulamasından gelen istekleri karşılamaktadır. Bu istekler Şekil 1.41'de görüldüğü gibi 3 farklı adaptör ile ödeme sistemleri, rezervasyon ve bilgilendirme gibi yazılım bileşenlerine ulaştırılmaktadır. Bu bileşenlerin hepsi tek bir veri tabanından beslenerek işlemleri gerçekleştirmektedir.



Şekil 1.41. Uber Monolit Mimarisi

Tek bir kod tabanına ve veri tabanına sahip olan bu monolit altyapı Uber'in temel iş sorunlarına ilk zamanlarında çözüm getirmiştir ancak müşteri sayısının artması ile beraber hizmet türleri ve sayısı da artmış ve buna bağlı olarak uygulama büyümüş ve yönetimi oldukça zor bir hal almıştır. Uygulamanın büyümesiyle ekipler de büyümüş ve

aynı mimari üzerinde kod yazan yazılımcı sayısı artmıştır. Bu artış ve büyüme birçok zorluğu da beraberinde getirmiştir. Bu zorlukların başında yapılan herhangi bir geliştirmede uygulamanın yeniden derlenmesi ve testlerinin topyekûn tekrardan yapılması gelmiştir. Ayrıca kod tabanındaki büyüme ile beraber kod üzerindeki değişikliklerin sistemi geri kalanını kötü etkileme riski artmıştır. Bu problemlerin çözümüne yönelik olarak 2015 itibari ile Uber mikro servis mimari dönüşümünü başlatmıştır (Kappagantula, 2018; Reinhold, 2016).

1.7. Mikro Servis Mimarisi

Bu bölümde Mikro Servis Mimarisinin tanımı ve karakteristik özellikleri hakkında genel bilgiler verilmektedir.

1.7.1. Tanım

Mikro Servis terimi ilk olarak 2011 yılında Martin Fowler tarafından geliştirilmiştir (Fowler, 2014). Bununla birlikte bu yaklaşım farklı isimler altında daha öncesinde de kullanılagelmiştir. Örneğin Netflix'te, ince taneli Servis Odaklı Mimari (SOA) adı altında çok benzer bir mimari kullanılmıştır (Wang, 2013).

Günümüzde Mikro Servis Mimarisi, her biri temel düzeyde (atomik) işleve haiz olan ve birbirinden bağımsız olarak çalışabilen mikro servislerin geniş çaplı bir yazılım sistemini oluşturmak için bir arada koordineli olarak çalıştıkları bir dağıtık sistem mimarisini ifade etmek için kullanılmaktadır. Bir başka deyişle bu mimari Şekil 1.42'de temsilen gösterilen ve bir bisikletin farklı işlevler gören fakat ahenk içinde çalışan bileşenleriymiş gibi bir yazılım sistemini bağımsız ancak koordineli bileşenlerden oluşacak şekilde parçalayarak geliştirmeye ve yaygınlaştırmaya imkân tanımakta ve bu şekilde sistemin bütününde artan karmaşıklığı yönetmeyi kolaylaştırmaktadır (Dragoni, 2017).

Mikro Servis Mimarisi, günümüzde bakımı yapılabilir ve kolaylıkla ölçeklenebilir yazılımların geliştirilmesini sağlayan bir eğilim haline gelmiştir zira yazılımı oluşturan bileşenler arasında gevşek bağımlılığı (loosely coupling) ve yüksek uyumu (high cohesion) sağlamayı mümkün kılmaktadır. Bu yaklaşım, yazılımın güncel ihtiyaçları karşılama bağlamındaki sürdürülebilirliğini ve ölçeklenebilirliğini mümkün kılmaktadır (Dragoni, 2017).



Şekil 1.42. Mikro Servis Mimarisi Benzetimi

1.7.2. Mikro Servislerin karakteristik özellikleri

Mikro Servis Mimarisinin herkes tarafından kabul edilmiş kırmızı çizgileri yoktur. Ancak *Mikro Servis Mimarisi* ile geliştirilmiş uygulamaların benzer karakteristik özelliklere sahip olduğu söylenebilir. Bu bölümde bu karakteristik özelliklerden bahsedilecektir.

1.7.2.1. Yazılımın iş kuralları çerçevesinde bileşenlere ayrılması

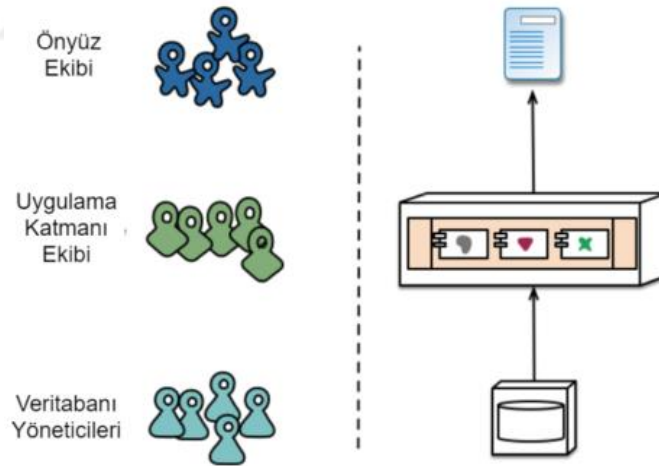
Yazılım endüstrisinde ilk zamanlardan günümüze kadar geliştirilen hemen hemen bütün sistemlerde önce bileşenlere yani modüllere ayrıştırma gerçekleştirilip ardından ayrıştırılıp geliştirilen bu modüllerin bütünleştirilmesi çerçevesinde hareket edilmiştir. Monolit mimaride de benzer iş kuralları tek bir modül etrafında toplanır ve bu modül sahipleri geliştirmelerini ayrı ayrı yaparlar. Monolit sistem bu modüllerin birleşmesiyle meydana gelir. Bu bileşenlere ayırma ve sonra birleştirme işlemi, yazılım geliştirme süreçlerinde tekrar tekrar yapılmasına rağmen bileşen kavramının tam bir tanıma oturtulması ve sınırlarının net olarak çizilmesi güçtür. *Mikro Servis Mimarisinin* ortaya çıkışı ile birlikte bileşen tanımı netleşmiş ve bağımsız bir şekilde değiştirilebilenve geliştirilebilen her yazılım birimine bileşen yani *mikro servis* adı verilmiştir.

Mikro Servis Mimarisinde bileşenler yani *Mikro servisler* uzak yordam çağırımı (Remote Procedure Call - RPC) ile iletişim kurulabilen yazılım bileşenleri olarak tasavvur

edilebilir. Bunun altında yatan neden mikro servislerin yekpare bir monolit gibi tek bir program olarak ve tek bir konak bilgisayarda çalıştırılmanın aksine bir bilgisayar ağı üzerindeki farklı konak bilgisayarlarda tıpkı bir bisikletin farklı parçalarıymış gibi koordineli ama bağımsız olarak çalıştırılma prensibine göre tasarlanıyor olmalarıdır (Fowler, 2014).

1.7.2.2. Organizasyon yapısının iş kuralları etrafında Şekillendirilmesi

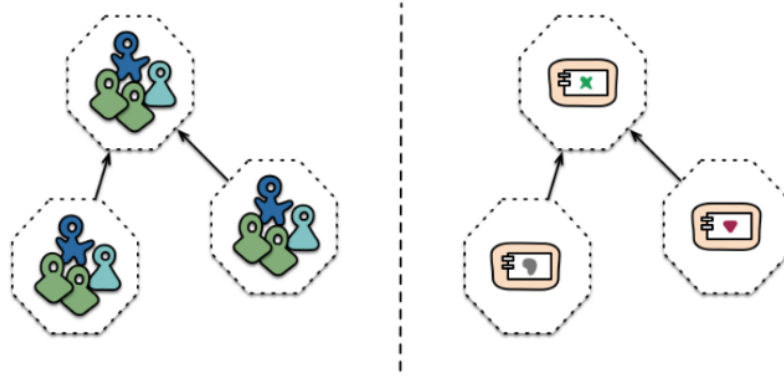
Bir monolit uygulamanın geliştirilmesi yapılırken yazılım ekipleri genellikle teknik bakımdan Şekil 1.43'te görüldüğü gibi önyüz, uygulama katmanı ve veritabanı katmanı gibi teknik olarak ayrılmış bileşenlerden sorumlu farklı gruplara ayrılmaktadırlar. Monolit mimaride bu ayrılmış bileşenler minvalinde karşılaşılabilecek sıkıntılar, ekipler arasında da benzer şekilde gün yüzüne çıkabilmektedir. Nitekim Merlin Conway 1968 yılında yaptığı çalışmada organizasyonel yapılar ile sistemsel yapıların birbirine benzer şekilde oluştuğunu söyleyerek bu duruma işaret etmiştir. Bu durum günümüzde literatürde halen *Conway's Law* olarak adlandırılmaktadır (Conway, 1968).



Şekil 1.43. Monolit Mimaride Organizasyon Yapısının Dağılımı

Mikro Servis Mimarisindeki yazılım bileşenleri yani *mikro servisler* yazılımın işlevini gerçekleştireceği iş alanı (yani business domain'i) çerçevesindeki iş kuralları ve mantığı ile bire bir örtüşecek şekilde ayrıştırıldıkları için ekipler Şekil 1.44'te temsili olarak ifade edildiği gibi iş alanı ile birebir örtüşen doğal bir yapıda organize olunurlar. Bu bağlamda her ekip belirli iş kurallarının (business rules) ve iş alanı nesnelerinin

(business domain objects) etrafında Şekillenmektedir. Bu iş kurallarına ait geliştirilecek uygulamaların bütün sorumluluğu ilgili ekibe verilmektedir. Dolayısıyla ilgili ekip, kullanıcı ara yüzünden veri tabanı katmanına kadar olan bütün bileşenlerden sorumlu hale gelmektedir (Bakshi, 2017).



Şekil 1.44. Mikro Servis Mimarisi Organizasyonel Dağılımı

1.7.2.3. Projelerin yerini ürünlerin alması

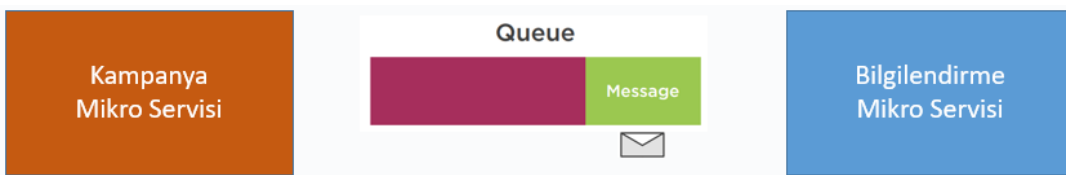
Bir yazılım projesinin geliştirme sürecinde teknik ekipler yazılımı geliştirmekte ve tamamlandığında bakımı için başka bir ekibe veya organizasyona devretmektedir. Ardından bu projeyi oluşturan ekipler dağıtılmaktadır. Mikro Servis mimarisinde bu yaklaşımdan kaçınılmaktadır. Bunun yerine bir ekibin ömrü boyunca tek bir ürüne sahip olması gerektiği fikri ortaya konulmaktadır. Bu fikrin ilham kaynağı ise Amazon firmasında ortaya konulan “*Sen inşa et, sen çalıştır*” prensibidir (Queue, 2006). Bu prensipte yazılım geliştiricilerin, yazılımın ürün haline gelinceye kadar ki sürecinde ürün ile daha fazla iletişim kurması sağlanmaktadır. Geliştiriciler, son kullanıcı destek işlerinin de bir kısmını zorunlu olarak üstlenmeleri gerektiğinden son kullanıcılarla daha fazla temas halinde olmaktadır. Dolayısıyla kullanıcıların isteklerini ve gereksinimlerini karşılamak için doğru adımların atılması minvalinde geliştiricilerin daha bilinçli ve uyanık olmaları sağlanmaktadır (Fowler, 2014).

1.7.2.4. Mikro Servisler arası iletişim

Mikro Servis Mimarisinde geliştirilen servislerin diğer servisler ile gevşek bağımlı (loosely-coupled) ve kendi içlerinde de yüksek uyumlu (high-cohesive) olmaları hedeflenmektedir. Bu ilke *mikro servislerin* bağımsız (yani otonom) olmaları kadar

koordineli olarak çalışmaları için de lüzumlu olan bir ilkedir. Bu bağlamda her bir mikro servis aldığı istemi (yani request'i) kendi iş mantığı çerçevesinde işleyip dönecek şekilde tasarlanmaktadır. Bunu mümkün kılmak için mikro servisler arasında iki tür iletişim yöntemi kullanılabilir. Bu yöntemler birincisi *Representational State Transfer (REST)* teknolojisi esaslı geliştirilebilecek olan *Uygulama Programlama Ara-yüzleri (Application Programming Interface'ler-API'ler)* üzerinden sağlanacak senkron (eş zamanlı) bir iletişim yöntemidir. Bu ilk yöntemde bir mikro servis başka bir mikro servisi API aracılığı ile network üzerinden anlık olarak çağırır ve cevap yine network üzerinden anlık olarak döner.

Mikro servisler arasında kullanılabilecek ikinci genel iletişim yöntemi ise asenkron (eş zamansız) iletişim olarak adlandırılabilir olan veri yolu (yani event-bus) temelli iletişimdir. Bu iletişimde hizmet veren mikro servis hizmet talebini yapan diğer mikro servislere döneceği cevabı anlık olarak vermek yerine belirli bir gecikme ile döner. Bu tür iletişim hususen anlık sonuç beklemeyen ve “*Fire & Forget (Tetikle ve Unut)*” olarak adlandırılan mikro servislerin çağırımında kullanılmaktadır (Sill, 2016). Buna örnek olarak aşağıda Şekil 1.45'te temsili olarak gösterilen kampanya ve bilgilendirme mikro servisleri verilebilir. Burada kampanya mikro servisi kampanya mesajlarını göndermek için bilgilendirme mikro servisine asenkron çağırımlar yapmaktadır. Bu çağırımlar bir kuyruk üzerinden bilgilendirme servisine aktarılmakta olup çağırımı yapan kampanya mikro servisi mesajın gönderilip gönderilmediğine dair bir cevap beklentisi olmadan yığınlar halinde mesaj gönderme yapmaktadır. Bu bakımdan kampanya mikro servisi “*Fire & Forget (Tetikle ve Unut)*” servislerine bir örnek olarak verilebilir.



Şekil 1.45. Mikro Servisler Arası Asenkron İletişim

1.7.2.5. Teknoloji bağımsız geliştirme

Monolit uygulamalarda uygulamaya ait bütün bileşenler tek bir platform üzerinde geliştirilmek zorundadır. Bu durum ihtiyaca göre doğru aracı ve programlama dilini

seçebilmek konusunda geliştiricilere hiçbir olanak seçenek sunmamaktadır. Deneyimler bu yaklaşımla geliştirilen uygulamalarda geliştiricilerin dar bir alana sıkıştığını ve doğru çözümlerin teknoloji bağımlılığından dolayı uygulanamadığını göstermektedir.

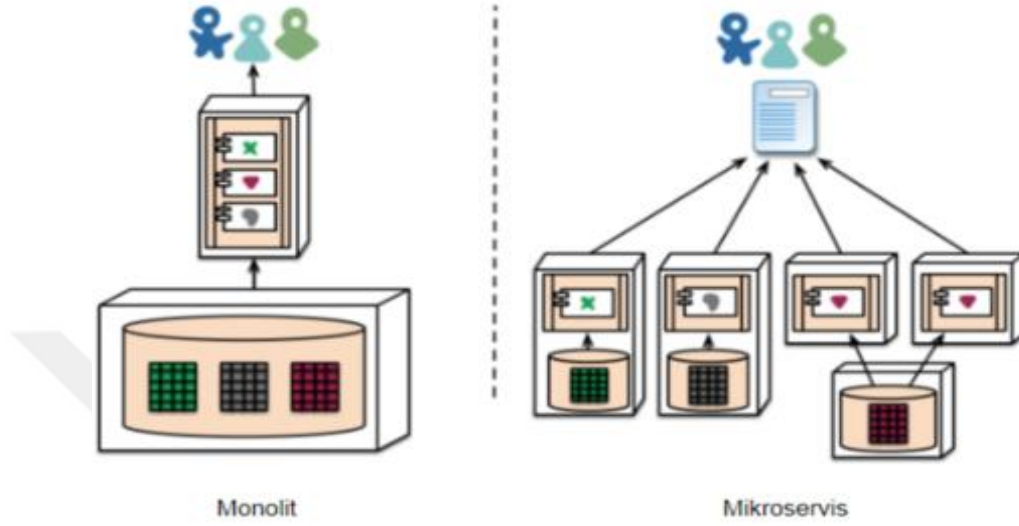
Mikro Servis mimarisinde ise yazılım bağımsız ama koordineli bileşenlerin ahengi şeklinde tasarlandığı için teknoloji bağımlılığı olmadan gereksinime göre her bir bileşen (yani mikro servis) istenilen teknoloji ya da yöntemle geliştirilebilmektedir. Arzu edilen veri modeline göre veri tabanı, uygulama katmanındaki yüke göre de programlama dili seçilebilmektedir.

Yazılımın mikro servislere ayrılması ile beraber ortaya çıkabilecek bir diğer durum da kod tekrarlarıdır. Birbirinden bağımsız geliştirme yapan ekipler aynı problemleri çözmek için aynı geliştirmeleri yapabilir hale gelmektedir. Monolit uygulamalarda bu durum belli standartlar koyularak çözülmektedir. Bu standartlarda, herkes tarafından kullanılacak kod bloklarının belirli bir projede toplanması gibi kurallar bulunmaktadır. Mikro servis geliştiren ekipler ise bu durumlarda açık kaynak mantığında ilerlemektedir. Bir başka deyişle herkes tarafından kullanılacak kod parçaları sadece o spesifik mikro servisle ilgili geliştirmede (implementasyonda) bırakılmayıp diğer ekiplerinde kullanımına açık kütüphane olarak sunulmaktadır. Böylece ekipler benzer çözümler için geliştirilen benzer yöntemleri tekrar tekrar kullanarak zaman ve emek kazanımı sağlamaktadırlar (Fowler, 2014).

1.7.2.6. Merkezileştirilmiş veri yönetimi

Monolit uygulamalarda veri tabanı tek olduğu için, içerisinde oluşturulan veri modelleri arasında kavramsal uyumsuzluklar oluşabilmektedir. Örneğin, kurumsal bir monolit sistemdeki bir bileşen bir kullanıcı için bir anlam ifade derken; başka bir kullanıcı için başka bir anlam ifade edebilmektedir. Bu anlam farklılıklarını mikro servis mimarisinde yönetebilmek için monolit mimarideki gibi tek bir veri tabanı üzerinde bütün farklılıkları tutmak yerine her bir mikro servis özelinde ayrık bir veri tabanı oluşturulması söz konusu olmaktadır. Bununla birlikte mikro servis mimarisi esaslı bir sistemde bu veri tabanı ayrıştırılması doğru yapılmadığı takdirde birbirleriyle ilişkili veri modelleri farklı veri tabanları üzerinde tutulacağından veriye erişim karmaşıklaşmakta ve sistemin genel yönetimi zorlaşmaktadır. Mikro servis mimarisinde bu durumun çözümü için Alan Tabanlı Tasarım (Domain Driven Design – DDD) tekniği uygulanmaktadır. Bu tasarım şeklinde karmaşık iş mantığı birbirleriyle ilişkili parçalara bölünmekte ve bu parçalar

arasında bir ilişki haritası çıkartılmaktadır. İş mantığı bu ilişki haritasına göre Şekil 1.46'da görüldüğü gibi gerektiğinde parçalanmakta gerektiğinde de birleştirilmektedir. İçeride oluşturulacak veri modelleri de kolaylıkla iş alanına göre düzenlenebilmekte ve anlam karışıklığına yol açacak sebepler ortadan kaldırılmaktadır (Furda, 2018).



Şekil 1.46. Monolit Mimariden Mikro Servis Mimarisine Geçişte Veri Tabanı Ayırımı

1.8. Mikro Servis Tasarım Kalıpları

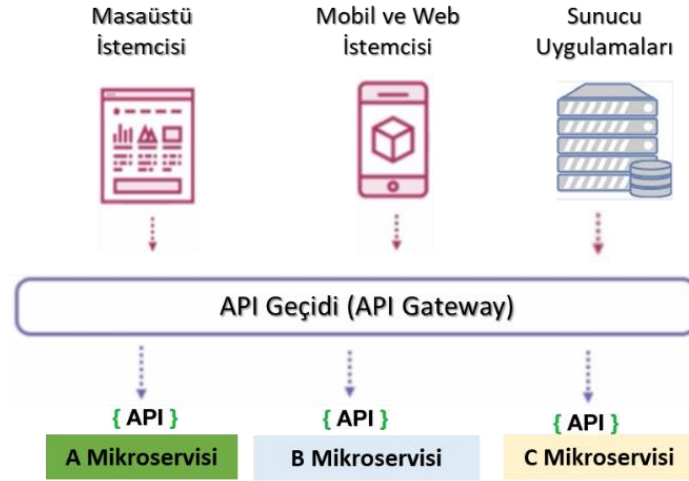
Yazılım sistemlerinin tasarımında sürekli karşılaşıla gelen sorunların çözümüne yönelik edinilmiş tecrübelerinin bir birikimi olan tasarım desenleri, tak-kullan şeklinde bir çözüm kolaylığı getirmemekle birlikte çoğu zaman yazılımcıya sorunların nasıl giderilebileceği yönünde yol gösterici olması nedeniyle yazılım geliştirme sürecini hızlandırmaktadır.

Mikro servis mimarisinde servislerin dayanıklılığını sağlamak monolit mimariye göre daha güçtür çünkü mikro servisler arası iletişim dağıtık olarak gerçekleşmekte ve bir transaction (işlem) için bile servisler arası yoğun network trafiği oluşmaktadır. Servisler arasındaki bu yoğun iletişim işlemlerin gerçekleşmesinde kesinti riski doğurmaktadır. Bu bölümün geri kalanında mikro servisler arası kesintileri minimize edecek yapısal tasarım desenlerinden hususen doktora çalışmamızla en ilişkili olanlarına değinilmiştir (Richardsson, 2019).

1.8.1. API Geçidi (API Gateway) tasarım kalıbı

Günümüzün modern yazılım sistemlerini oluşturan modüller (yani bileşenler) arasındaki hatlar Uygulama Programlama Ara Yüzleri (API'ler) aracılığı ile tanımlanmaktadır. Bu durum yazılımı oluşturan bileşenlerin lego parçaları gibi ihtiyaç durumunda tekrar tekrar kullanılabilmesi için gerekli bir stratejidir (Bloch, 2006).

Aynı strateji Mikro Servis Mimarisi (MSA) kapsamında da değerlendirilebilmektedir zira MSA'da her mikro servis, sistemin bütününde var olması gereken temel bir işlevi (fonksiyonu) ihtiva eder ve bu işlev mikro servisin sunduğu API üzerinden diğer mikro servislerin ve istemci uygulamalarının hizmetine sunulur (Richardson, 2019; Fritzsche, 2019). İstemci uygulamalarına örnek olarak mobil istemciler (Android veya iOS uygulamaları gibi), masaüstü istemcileri ve zamanlanmış görevler (scheduled tasks) yerine getiren sunucu uygulamaları verilebilir. İstemci uygulamalar, mikro servisleri herhangi bir aracı olmadan direk API'lerini kullanarak çağırabilirler. Bununla birlikte, mikro servisler tarafından sağlanan API'lerin ayrıntı düzeyi (yani implemente ettikleri işlevlerin sayısı- granülite), genellikle istemciler tarafından beklenen ayrıntı düzeyinden farklıdır. Başka bir deyişle, istemciler genellikle daha geniş bir işlev kümesini kapsayan bir API'yi tek seferde çağırarak ihtiyaç duydukları işlevlerin tek hamlede yapılmasını beklerler. Bununla birlikte mikro servis API'leri daha kısıtlı bir işlev düzeyi sunarlar zira mikro servislerin her biri yalnızca temel bir işlevi (atomic function) yerine getirme konusunda uzmanlaşmıştır. Dolayısıyla istemcilerin beklediği işlev kümesini çalıştırmak için birden çok API'nin birden fazla network çağırımı gerektirecek şekilde çağırılmaları gerekmektedir. Bu gereksinimi gerçekleştirmek amacıyla MSA'da *API Geçidi Tasarım Kalıbı (API Gateway Design Pattern)* uygulanmaktadır. Bu tasarım kalıbında mikro servisler ve istemciler arasında *API Geçidi (API Gateway)* olarak adlandırılan bir aracı bileşen yer almaktadır. Bu aracı bileşen istemcilere işlev bakımından daha zengin (yani işlevce şişman bir granülite seviyesinde) API'ler sunmakta ve bu API'lerin sunduğu işlevleri yerine getirirken ilgili münferit mikro servis API'lerini uygun sırada istemci adına çağırıp istemciye cevap dönmektedir. Bu bakımdan, *API Geçidi*, mikro servislerin önünde bir sınır bileşen görevi görerek istemci uygulamaların mikro servisleri doğru ayrıntı düzeyine sahip iyi tanımlanmış bir API aracılığıyla çağırmasını sağlayan API odaklı, merkezi bir kontrol bileşeni görevi görmektedir (Xu ve ark., 2019; Zhao ve ark., 2018; Lu ve ark., 2017; Montesi ve Weber, 2016) Bu durum Şekil 1.47'de temsili olarak ifade edilmektedir.

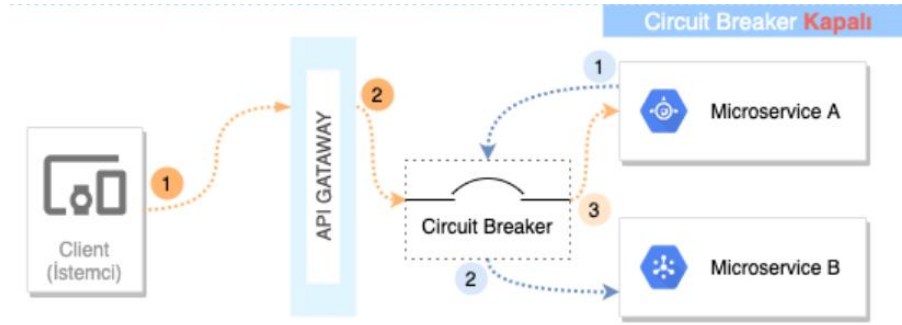


Şekil 1.47. API Geçidi Üzerinden Mikro Servis API'lerinin Çağırımı

1.8.2. Devre kesici (Circuit Breaker) tasarım deseni

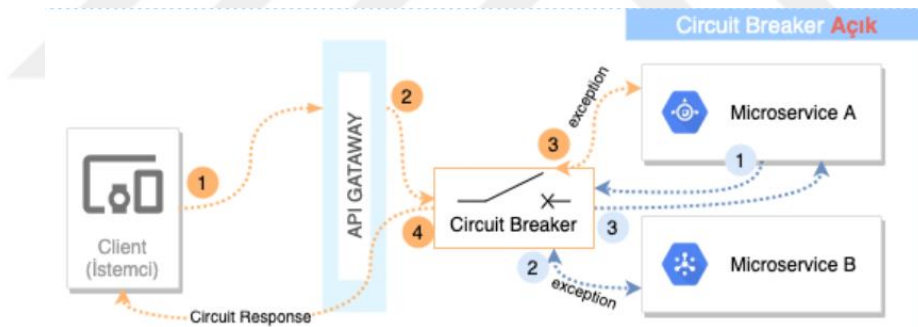
Circuit Breakers deseni, elektronik devrelerdeki devre kesici şalt cihazlarından esinlenerek geliştirilmiş bir yöntemdir. Devre kesiciler, için sistemde meydana gelen yüksek akım veya kısa devre gibi bir devreye zarar verecek aşırı yük durumlarında devreye gelen akım geçişini durdurucu bir işlev gösterirler. Mikro servisler arasındaki iletişimde de benzer sorunlar oluşabilmektedir. Örneğin bir mikro servise gönderilen işlemlerin gerçekleştirilmesi veri tabanı erişiminin kesilmesi ya da RAM belleğin yetersiz kalması gibi durumlarda başarısızlıkla sonuçlanabilir. Başarısız gerçekleşen işlemlerin sayısı belirli bir adedi aştığında devre kesici bir yazılım bileşeni devreye girerek mikro servise giden network çağırımlarını keser ve bu suretle sistemin belirli bir ögesine aşırı yüklenilmesini engeller.

Devre kesicilerin üç durumu vardır. Bu durumlar: Açık (Open), Kapalı (Closed) ve Yarı-Açık (Half-Open) olarak ifade edilebilir. Bu durumlardan kapalı durum, devre anahtarının kapalı olması ve akım geçişine izin verilmesi durumunu ifade eder. Kapalı durumda mikro servisler normal işleyişlerini gerçekleştirdikleri için devre kesici mikro Şekil 1.48’de görüldüğü gibi servisleri sadece monitör eder. Dolayısıyla kullanıcıdan gelen istekler tıpkı bir elektrik akımı gibi kapalı devre üzerinden akarak bir ve iki nolu kollar üzerinden akarak A ve B mikro servislerine ulaşır. Servislerden hiçbiri hata almadığı için devre kesici kapalı durumunu korumaya devam eder.



Şekil 1.48. Devre Kesici Bileşen Kapalı Durumdayken

Açık durum ise devre anahtarının açık olmasına ve akım geçişine izin vermemesine benzetilebilir. Bu durum Şekil 1.49’da örneklenmektedir. Buradan anlaşılacağı gibi istemci tarafından yapılan isteklere (turuncu bağlantı) A mikro servisi hata (yani exception) döndürmektedir. Servisin hata döndürmeye devam etmesi durumunda belirlenen bir eşik değeri aşılmış ise devre kesici açık duruma geçecek ve A servisine yapılacak yeni istekler artık devre kesici ’de (circuit breaker) kesilecek ve daha önceden belirlenen sabit cevaplar istemciye döndürülecektir.

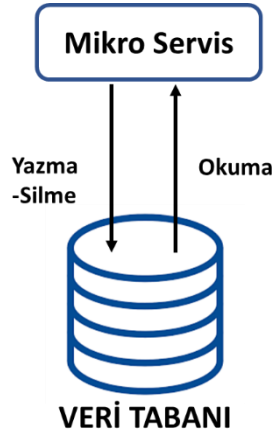


Şekil 1.49. Devre Kesici Bileşen Açık Durumdayken

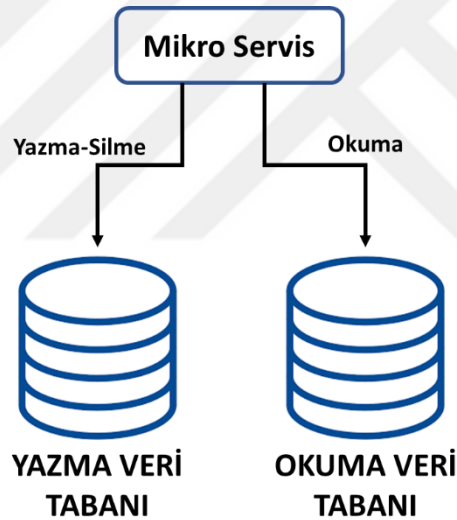
1.8.3. CQRS (Command Query Responsibility Segregation) tasarım deseni

Monolit mimaride, veri tabanı gibi veri kaynakları üzerinde yapılan işlemlere kısaca CRUD (Create-Read-Delete-Update/Yazma-Silme-Okuma) aksiyonları denilmektedir. Geleneksel monolit mimaride CRUD aksiyonları genel itibariyle Şekil 1.50’de görüldüğü gibi aynı tek bir veri modeli (yani veri tabanı) üzerinde uygulanmaktadır. CRUD aksiyonlarının tek bir veri modeli üzerinde uygulanışı küçük çaplı monolit sistemlerde memnuniyet verici sonuçlar doğrulanabilmektedir. Bununla birlikte daha karmaşık veri modeline sahip büyük çaplı sistemlerde CRUD aksiyonlarının

tek bir veri modeli üzerinde direk uygulaması performans ve ölçekleme sorunlarına neden olmaktadır (Richardson, 2019).



Şekil 1.50. CRUD Aksiyonlarının Tek bir Veri kaynağı Üzerinde Uygulanışı



Şekil 1.51. CQRS Tasarım Deseninde Yazma ve Okuma Veri Kaynaklarının Fiziksel ve Mantıksal Ayırıştırılması

Şekil 1.50’de görüldüğü gibi tek kaynaklı bir veri tasarımında, bir veritabanı hem okuma hem de yazma etkinliğinin hedefidir. Komut-Sorgu (Command-Query) sorumluluk ayrıştırması olarak ifade edilebilecek olan CQRS tasarım deseninde okuma ve yazma aksiyonları Şekil 1.51’de görüldüğü gibi çeşitli veri kaynakları arasında ayrıştırılır ve bu suretle uygulamaların yazma ve okuma davranışları hem fiziksel hem de mantıksal olarak ayrıştırılır. Bu sayede her veri kaynağı için okuma yazma gereksinimlerine uygun bir veri tabanı teknolojisi kullanılabilir. Örneğin, yazma

verilerini depolamak için PostgreSQL veya SQL Server gibi ilişkisel bir veri tabanı sistemi tercih edilebilir. Öte yandan okunan verileri depolamak ve görüntülemek için MongoDB gibi bir belge tabanlı veri tabanı sisteminin kullanımı tercih edilebilir. Bu durum sistemin zaman içinde daha iyi gelişmesine ve ölçeklenmesine olanak tanır (Richardson, 2019).

1.8.4. SAGA tasarım deseni

SAGA, mikro servisler arasında birden fazla işlemin sistematik bir şekilde işlenerek veri tutarlılığının sağlanmasını amaçlayan bir tasarım desni ortaya koyar. SAGA, transaction'ların (işlemlerin) yönetiminde event'lerden (olaylardan) yararlanmaktadır. SAGA, mikro servislerdeki her bir işlemin yönetimi için olay (event) olarak adlandırılan mesajlardan yararlanmaktadır. Başlatılan bir işlemin tamamlanması sonrasında işlemin gerçekleştiğini belirtmek amacıyla bir mesaj yayınlanarak diğer mikro servis ya da servislerin tetiklenmesi sağlanır. İşlemler, mikro servisler arasındaki birbirini takip eden bu mesaj ve olay dizileri ile takip edilmektedir. Birbirini takip eden işlemlerde, bir mikro serviste oluşabilecek hata durumunda önceki yapılan tüm işlemler geri alınmaktadır (roll-back edilmektedir) (Richardson, 2019).

SAGA kalıbının kullanımına örnek olması açısından bu başlık altında E-Ticaret sitelerindeki satın alma işlemi incelenmiştir. E-Ticaret sitesi üzerinden bir müşterinin bir ürünü alma süreci birbirini takip eden sıralı işlemlerden oluşmaktadır. Müşterinin siparişi vermesi için ödeme servisine istek atılmaktadır. Aynı istek, siparişi gönderecek pazarlama birimi tarafından da alınmaktadır. Müşteri başlatılan istekte ödeme işlemini yapmaması halinde diğer başlatılan hizmetlerde geri alınmaktadır. SAGA bu yönetimi sebebiyle veri tutarlılığını sağlamaktadır (Kanber, 2021).

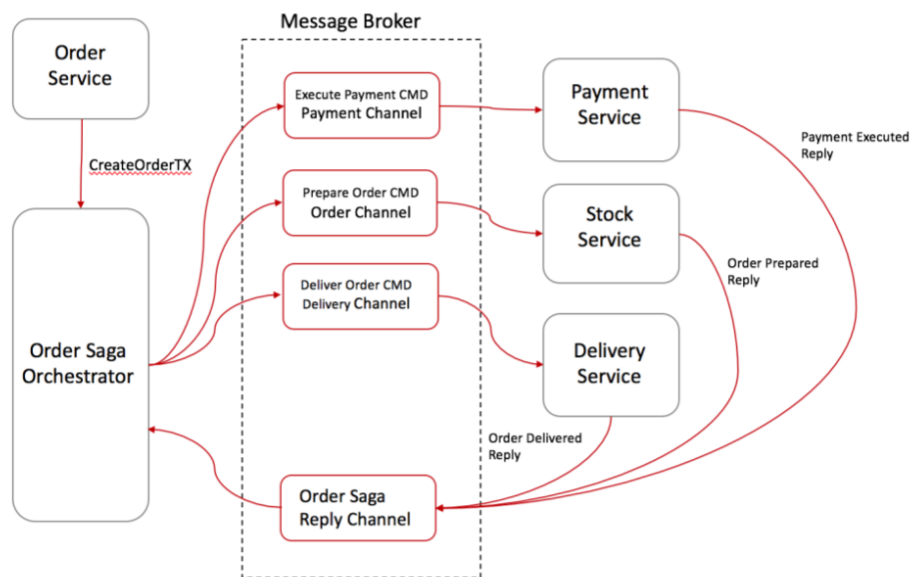
SAGA tasarım desenin orkestrasyon ve koreografi esaslı olmak üzere iki varyantı mevcuttur. Bu bölümün takip eden alt başlıklarında bu varyantlara değinilmektedir.

1.8.4.1. Orkestrasyon esaslı SAGA

Orkestrasyon esaslı yaklaşımda tüm işlemleri (transaction'ları) sevk ve idare bir SAGA orkestrasyon mikro servisi mevcuttur. Bu servis, kendisine abone olan tüm tüketici mikro servislere neyi ne zaman yapacaklarını dikte eden mesajlar gönderir (Richardson, 2021; Anonim, 2021 (14)).

Şekil 1.52’de bir e-ticaret senaryosu üzerinden orkestrasyon esaslı SAGA deseni örneklenmektedir. Bu örnek kapsamında bir e-ticaret siparişinin alınıp işleme konulduğu pozitif senaryo aşağıdaki adımlarda izah edilen şekilde işler (Kanber, 2021):

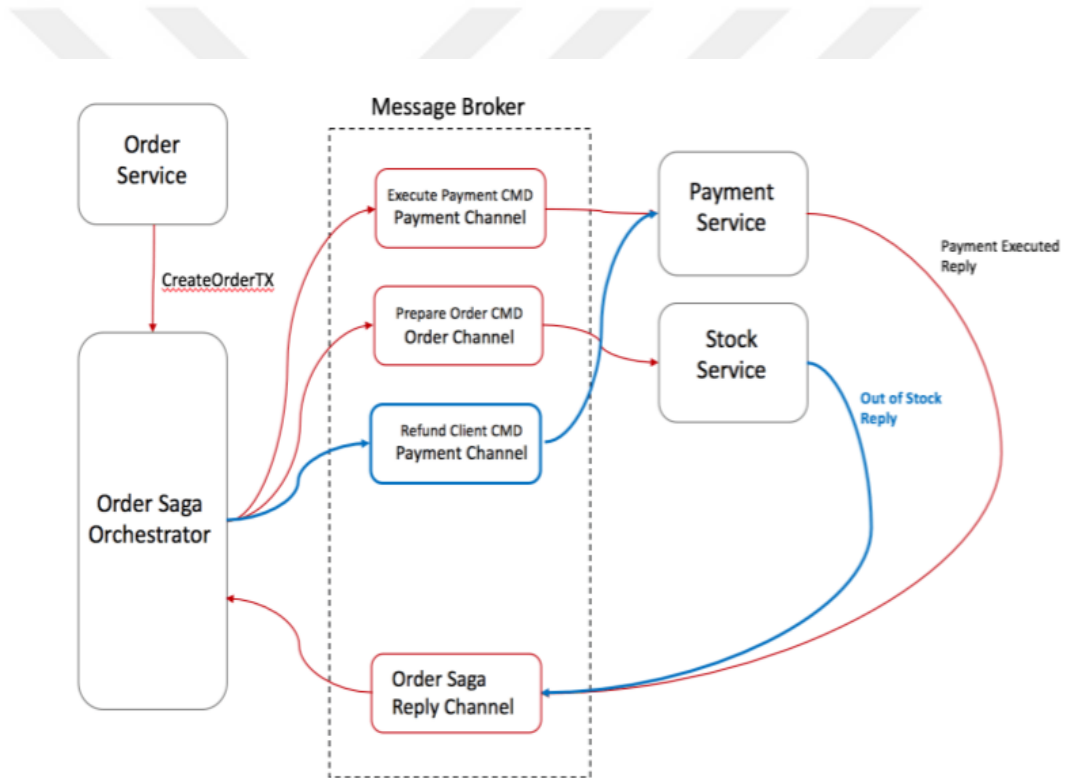
- i. Order service’i (sipariş mikro servisi) pending (askıda) durumunda siparişi oluşturur ve orchestrator bileşene ORDER_CREATED bilgisini içeren bir mesaj gönderir. Bunun akabinde orchestrator sipariş oluşturma işlemini başlatır.
- ii. Orchestrator, EXECUTE_PAYMENT komutunu payment service’ine (ödeme servisine) göndererek ödeme işlemini tetikler. Bunun akabinde payment servisi siparişe ilişkili ödeme işlemini gerçekleştirir ve orchestrator servisine ödemenin gerçekleştirildiği bilgisini içeren bir mesaj döndürür.
- iii. Orchestrator, UPDATE_STOCK komutunu stock service’ine göndererek stok adedindeki güncelleme işlemini başlatır. Bunun akabinde stock service’i sipariş kapsamındaki ürün ya da ürünlerin stok adetlerini düşürür ve stok bilgisinin güncellendiği yönündeki bilgilendirme mesajını orchestrator servisine döndürür.
- iv. Orchestrator, ORDER_DELIVER komutunu shipment service’ine (yani kargo servisine) göndermek suretiyle sipariş kapsamındaki ürün ya da ürünlerin gönderim sürecini başlatır. Shipment servisi sipariş gönderildiğinde bu doğrultudaki bir bilgilendirme mesajını orchestrator servise döndürür.



Şekil 1.52. Bir E-Ticaret Sistemi Kapsamındaki Mikro Servislerin Orkestrasyon Esaslı SAGA Deseni Çerçevesinde Etkileşim Senaryosu (Kanber, 2021)

Şekil 1.52’de örneklenen senaryoda negatif durumlar yani başarısızlık durumları meydana gelebilir. Böyle bir duruma örnek olarak sipariş kapsamındaki ürünlerin stokta mevcut olmaması verilebilir. Bu tür durumlarda orchestrator servisi bir siparişin iptali için gerekli akışın ne olduğu bilgisine sahiptir. Dolayısıyla sipariş işlemlerini geri almak için kendisine abone olmuş tüm mikro servislere geri alma (roll-back) işlemini koordine edici direktifleri ardı ardına gönderir. Bu durumu örnekleyen bir senaryo Şekil 1.53’te görülmekte olup gerçekleşen roll-back adımları şu şekilde olabilir (Kanber, 2021):

- i. Stock servisi Orchestrator’a OUT_OF_STOCK bilgisini gönderir.
- ii. Orchestrator işlemin başarısız olduğu bilgisini alıp rollback işlemini başlatır.
- iii. SAGA Orchestrator abone olan tüm service’leri çağırılabilir ancak, abone olan servisler orchestrator’ü çağırılmaz.



Şekil 1.53. Orkestrasyon Esaslı SAGA Deseni Çerçevesinde Roll-back (Geri alma) Senaryosu İşleyişi (Kanber, 2021)

Orkestrasyon esaslı SAGA tasarım deseninin avantajları ve dezavantajları şu şekilde sıralanabilir (Kanber, 2021):

- i. Distributed transaction yönetimini merkezileştirir.
- ii. Implemente etmek ve test etmek kolaydır.
- iii. Rollback yönetimi basittir.

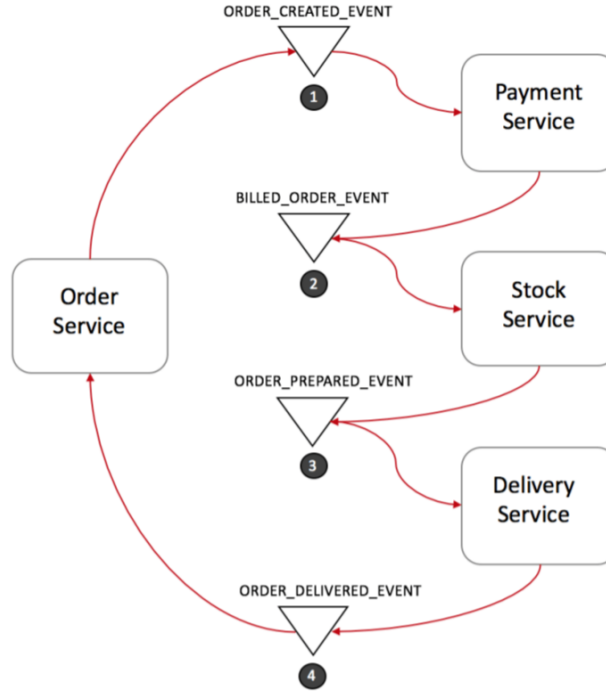
- iv. Yeni step'ler eklendiğinde işlem karmaşıklığı choreography yaklaşımına göre daha az kompleksleşir.
- v. Birbirini bekleyen işlemleri bu yaklaşım ile yönetebilirsiniz.
- vi. Fazladan bir service'i yönetmeniz gerektiği için sisteminizdeki infrastructure karmaşıklığı artacaktır.

1.8.4.2. Koreografi esaslı SAGA

Bu yaklaşımda merkezi bir yönetici yoktur. Her servis işlemi tamamlar ve bir event (mesaj) yayınlar. Bir sonraki servis ilgili event'i alıp normal işleyiş sürecine devam eder (Richardson, 2021; Anonim, 2021 (14)).

Şekil 1.54'te bir e-ticaret senaryosu üzerinden koreografi yaklaşımı örneklenmektedir. Bu örnek senaryoda gerçekleşen adımlar aşağıdaki gibi özetlenebilir (Kanber, 2021):

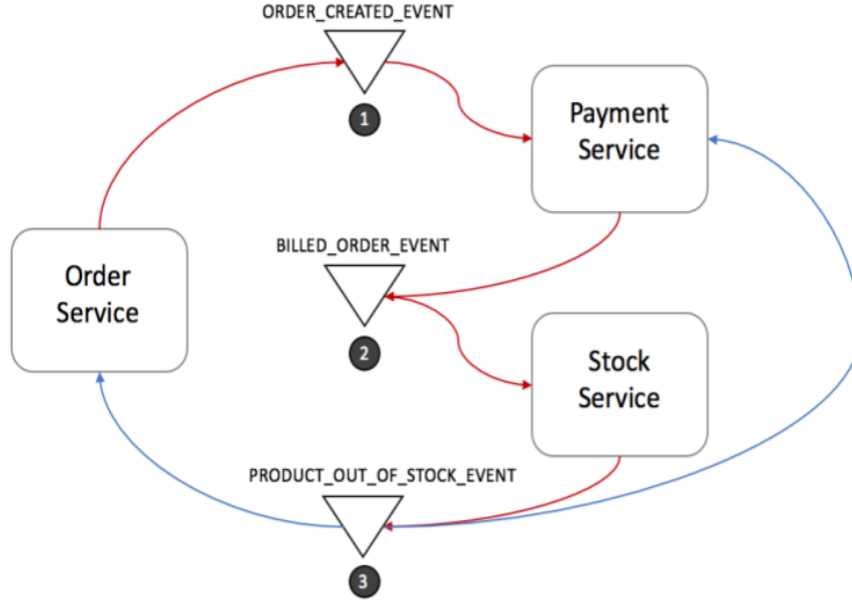
- i. Order (sipariş) mikro servisi siparişi oluşturur ve pending (askıda) statüsünde ORDER_CREATED_EVENT mesajını yayınlar.
- ii. Payment (ödeme) mikro servisi ORDER_CREATED_EVENT'ini mesajını ve ödeme işlemini gerçekleştirip BILLED_ORDER_EVENT mesajını yayınlar.
- iii. Stock (stok kontrol) mikro servisi BILLED_ORDER_EVENT'ini dinler ve stok güncellemesini yapar ve ardından ORDER_PREPARED_EVENT mesajını yayınlar.
- iv. Shipment (kargo) mikro servisi ORDER_PREPARED_EVENT'ini dinler ve kargo işlemlerini gerçekleştirir ve ardından ORDER_DELIVERED_EVENT mesajını yayınlar.
- v. Son olarak, order mikro servisi ORDER_DELIVERED_EVENT'ini dinler ve siparişin durumunu tamamlandı durumuna çekerek süreci sonlandırır.



Şekil 1.54. Bir E-Ticaret Sistemi Kapsamındaki Mikro Servislerin Koreografi Esaslı SAGA Deseni Çerçevesinde Etkileşim Senaryosu (Kanber, 2021)

Şekil 1.54'te ifade edilen senaryo içerisindeki her adımda sipariş durumu güncellenmektedir. Bu senaryo içerisindeki her adımda sipariş mikro servisinin (order service) mesajlarını dinleyen bir tüketici servis bulunmaktadır. Bu tüketici servisler sipariş durumlarını güncellenmektedir. Dolayısıyla senaryonun işleyişi sırasında içerisinde bir hata oluşması durumunda geri-alma (rollback) işlemi için gerekli yeni mesajlaşmaların tetiklenmesi gerekmektedir. Şekil 1.55'te resmedilmekte olan bu geri alma işlemleri kapsamında gerçekleştirilecek adımlar şu şekilde ifade edilebilir (Kanber, 2021):

- i. Stock servisi PRODUCT_OUT_OF_STOCK_EVENT mesajı yayınlar.
- ii. Order ve Payment servisleri bu mesajı dinler ve payment servisi ödemeyi geri alır.
- iii. Order service'i order'ın statüsünü Failed'a çeker.



Şekil 1.55. Koreografi Esaslı SAGA Deseni Çerçevesinde Roll-back (Geri alma) Senaryosu İşleyişi (Kanber, 2021)

Koreografi SAGA kalıbının **avantajları** ve **dezavantajları** şu şekilde sıralanabilir:

- i. Basit, anlaşılması kolay ve event'i tetikleyen servisler birbirinden ayrı olduğu için var olan bir sisteme üzerine entegre edilmesi kolaydır.
- ii. Süreç 2 ila 4 adım gibi kısa adımlar içeriyorsa daha tercih edilebilir olmaktadır.
- iii. Hangi servisin hangi mesajları dinlediğini takip etmek, akış adımları arttıkça daha karmaşık hale gelebilmektedir. Ayrıca servislerin birbirine abone olmaları gerektiğinden döngüsel bağımlılıklar olabilmektedir.
- iv. Son olarak, sistemi tam anlamıyla test etmek için tüm service'lerin çalışır durumda olması gerektiğinden test yapmak zor olabilmektedir.

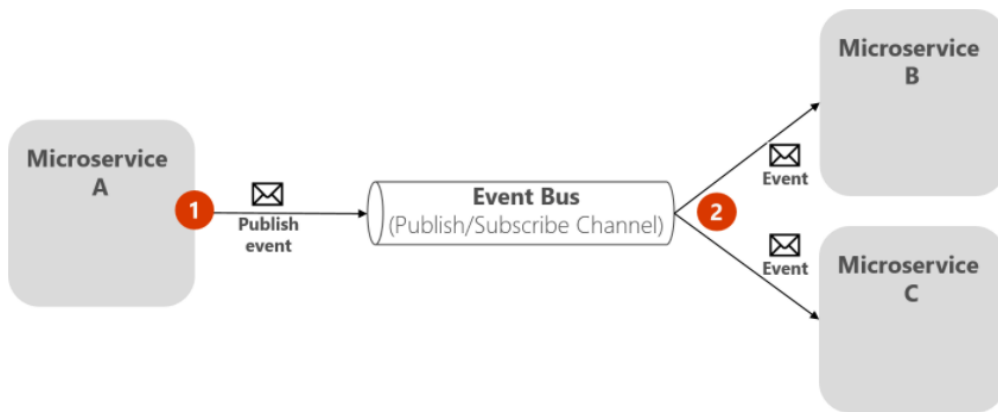
1.9. Event-Bus

Bir yazılım sisteminin bileşenlere ayrıştırılması, yazılım tasarımının en önemli aşamalarından biridir. Bu mühim aşamayı başarmanın bir yolu, bileşenler (servisler) arasında eş zamansız (asenكرون) bir iletişim yolu sağlayan mesajlaşma sistemlerini kullanmaktır. Bu bağlamda event-bus, yazılım bileşenlerinin birbirinden habersiz olarak (yani eş zamansız – asenكرون olarak) bir mesajlaşma kanalı üzerinden iletişim kurmasına olanak sağlayan bir mesajlaşma sistemi olarak ifade edilebilir (Torre ve ark., 2021).

Event-bus terminolojisiinde mesajı event-bus’a koyan yazılım bileşenine yayıncı (publisher) servis; mesajı event-bus’tan alan bileşen ya da bileşenlere ise abone (subscriber) servis ya da servisler denilmektedir. Bu bakımdan event-bus, yayıncı servislerle abone servisler arasında bir tür publish/subscribe (yayıncı/abone) mesaj kanalı tesis etmektedir. Genel olarak web üzerinden işleyen bu kanalda el değıştiren mesaja bir olayın vuku bulmasına neden olması hasebiyle kısaca “event” yani vukuat adı verilmektedir.

Bir event-bus mesajlaşma sisteminde yayıncı bir servis, hangi abone servisin mesajı alacağını umursamadan event-bus’a bir mesaj bırakır. Bu mesajın belirli bazı nitelikleri vardır. Event-bus, bu niteliklere abone olmuş diğ er servislere bu mesajı otomatik olarak iletir. Bu şekilde, servisler birbirine bağımlı olmadan (asenkron) iletişim kurabilirler. Ayrıca, bir servisin yapısı diğ er servislerden bağımsız olması itibarıyla servislerin tasarımı ve yapısı birbirlerine haber vermeksizin değıştirilebilir. Yapısı değıştirilen bileşen gönderilen ve alınan mesajları anladığı sürece diğ er bileşenler üzerinde hiçbir değışiklik gerektirmeyecektir. Bu bakımdan event-bus gevşek bağımlı (loosely-coupled/asenkron) dağıtık sistemlerin geliştirilmesinde kritik bir öneme sahiptir ve scrum gibi çevik (agile) esaslı proje geliştirmeyi kolaylaştırır (Bellemare, 2020).

Günümüzde modern yazılım bileşenlerinin mikro servis olarak tasarlanması nedeniyle event-bus genel olarak mikro servisler arasında gevşek bağımlı (loosely-coupled) bir asenkron iletişim hattının kurulmasında kullanılmaktadır. Bu durum aşağıda Şekil 1.56’da temsili olarak ifade edilmektedir. Bu Şekilden de anlaşılabilceğı üzere **A mikro servisi** bir event-bus’a bir “publish event” mesajı yayınlar. Event-bus bu mesajı mesajın haiz olduğı niteliklere abone olmuş olan diğ er mikro servislere (yani B ve C **mikro servislerine**) anlık olarak iletir (Torre ve ark., 2021).



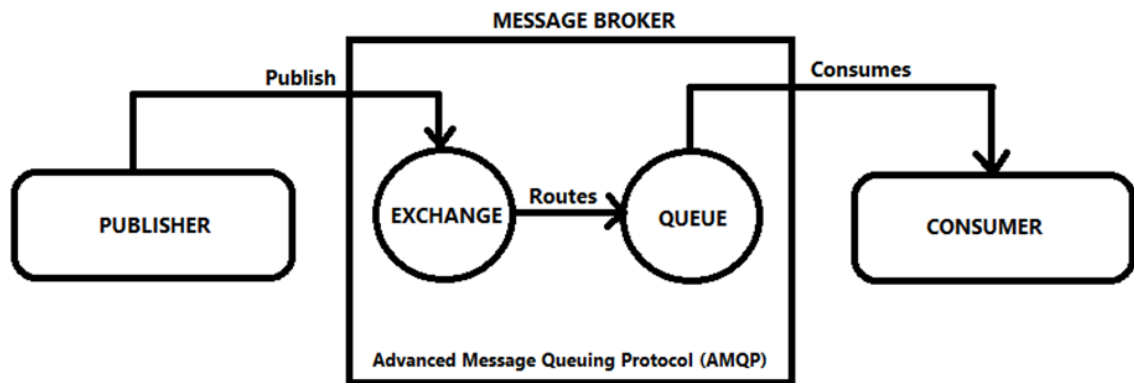
Şekil 1.56. Event-Bus Genel İşleyiş Yapısı (Torre ve ark., 2021)

Günümüz endüstrisinde firmaların farklı ihtiyaçlarına göre özelleştirilebilen ve event-bus geliştirmek için tasarlanmış olan mesajlaşma altyapı sistemleri mevcuttur. Bu altyapı sistemleri, kurumların ihtiyaç duyacağı mesajlaşma sistemlerinin karmaşıklığını yazılım geliştiriciden soyutlayarak yazılım geliştiricinin uygulama mantığının geliştirilmesine odaklanmasına yardımcı olurlar. Message-Broker/Message Queue olarak da adlandırılan bu mesajlaşma çerçevelerinin en yaygın olarak kullanılanları RabbitMQ ve Apache Kafka'dır. Bu bölümün geri kanalında bu iki altyapı sistemi hakkında yaptığımız araştırmamız ve kullanım tercihimiz hakkında bilgi verilmektedir.

1.9.1. RabbitMQ

RabbitMQ da Apache Kafka gibi yazılım bileşenleri (servisleri) arasında asenkron iletişim kurulmasına olanak tanıyan bir mesajlaşma altyapısı sunan bir sistemdir. Ölçeklenebilir yapısı ve kurulum kolaylığı nedeniyle RabbitMQ günümüzde en yaygın kullanılan mesajlaşma sistemidir (Bellemare, 2020).

Şekil 1.57'de görüldüğü gibi RabbitMQ temelinde Publisher ve Consumer olarak adlandırılabilir yayıncı ve abone olarak da ifade edebileceğimiz iki aktör ile birlikte Exchange ve Queue enstrümanlarından mevcuttur. Burada Publisher RabbitMQ mesaj kuyruğuna mesaj gönderen yani bir başka deyişle mesaj üreten servistir/uygulamadır. Publisher mesajı publish ettikten sonra ilgili mesajı Exchange karşılayacaktır. Exchange, kendisine belirtilen route ile ilgili mesajı kuyruğa (Queue) yönlendirir (Roy, 2017).



Şekil 1.57. RabbitMQ Genel İşleyiş Yapısı (Roy, 2017)

Kuyruğa yönlendirilen mesajın nasıl kuyruğa gideceği Exchange içerisindeki routedan öğrenilir. Ardından kuyruğa gelen mesajlar Queue’de sıralanır. Queue ilk giren ilk çıkar mantığında (FIFO) çalışan bir mekanizmaya sahiptir. Kuyruktaki mesajları tüketen/alan servis/uygulama ise Consumer’dır. Consumer herhangi bir dille geliştirilebilir olduğundan dolayı yazılım dilinden bağımsız bir uygulamadır. Tüm bu süreçte RabbitMQ AMQP (Advanced Message Queuing Protocol) adlı protokolü kullanmakta olup bu protokol üzerinden faaliyetlerini gerçekleştirmektedir (Roy, 2017).

RabbitMQ mesajlaşma sisteminin kullanımı aşağıdaki koşullarda tercih sebebi olabilir (Johansson ve Dossot, 2020):

- Producer’lar ve consumer’lar arasında farklı iletişim protokolleri kullanılmak isteniyorsa (Örnek: AMQP, STOMP, MQTT protokolleri).
- Yüksek performansa ihtiyaç varsa
- Yüklenmesi ve kurulumu kolay olan bir mesajlaşma altyapısı aranıyorsa.
- Mesajların consumer’lara ulaştırılmasının garanti edilmesi arzu ediliyorsa.
- Daha az sayıda mesajlaşma varsa (Dolayısıyla RabbitMQ yoğun mesajlaşma gerektiren sektörler (Örnek: Bankacılık)) için tercih edilmeyebilir.

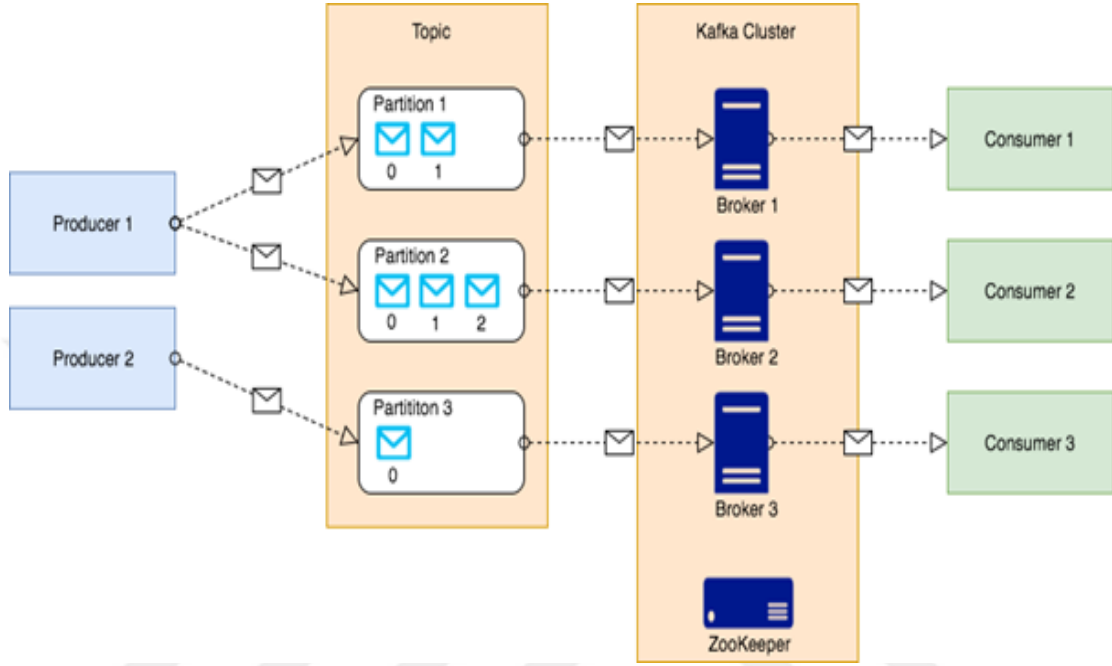
RabbitMQ mesajlaşma sistemin dezavajları şu şekilde sıralanabilir (Johansson ve Dossot, 2020):

- Yüksek hacimde veri işleme potansiyeli olan bir mesajlaşma sistemi değildir. Dolayısıyla bankalar gibi yoğun mesaj trafiğinin olabileceği kurumlarca kullanımı uygun olmayabilir.
- Consumer’ların sürekli çevrimiçi (online) durumda oldukları kabul edilir. Dolayısıyla eğer mesaj bir consumer’a iletilemezse mesaj beklemede olarak işaretlenir.
- Mesajların kalıcı (persistent) olarak saklanması arzu ediliyorsa RabbitMQ persistent modda çalıştırılmalıdır. Aksi takdirde RabbitMQ restart edildiğinde kuyruktaki bütün mesajlar kaybolacaktır.

1.9.2. Apache Kafka

Kafka, dağıtık bir yazılım sistemini oluşturan bileşenlerin yayınla/abone (publish/subscribe) tekniği esaslı olarak iletişim kurmasına olanak tanıyan hata toleranslı

ve ölçeklenebilir bir mesajlaşma sistemi oluşturmak için gerekli altyapıyı sunan bir sistemdir. 2011 yılında LinkedIn bünyesinde geliştirildikten sonra 2012 yılında Apache yazılım vakfı bünyesinde bir açık kaynak kodlu projeye dönüştürülmüştür. Java ve Scala programlama dillerinde yazılmıştır (Anonim, 2017).



Şekil 1.58. Event-Bus Genel İşleyiş Yapısı

Kafka altyapısında Şekil 1.58’de görüldüğü gibi mesaj gönderen uygulama/servis **producer** (üretici) olarak adlandırılırken; mesaj alan uygulama/servis ise **consumer** (tüketici) olarak adlandırılmaktadır. Bu yapıda mesajlar bazı kategoriler altında gruplanırlar. Üreticiler, **topic** olarak adlandırılan bu kategorilere mesajlarını bırakırlar. Tüketiciler, bu topiclerin bir veya daha fazlasından mesaj (yani veri) çekebilir. Topicler yönetimi kolaylaştırmak amacıyla **partition** denilen parçalara bölünerek cluster üzerinde fault-tolerance amaçlı olarak dağıtık tutulabilmektedir (Koutanov, 2020).

Tek bir kafka sunucusuna **Broker** adı verilmektedir. Bir başka deyişle Kafka Cluster’ındaki her bir ögeye **Broker** adı verilir. Cluster içerisinde yer alan öğelerden biri lider (master) olarak tanımlanır ve bütün okuma ve yazma işlerinden sorumludur. Diğer öğeler ise takipçi olarak tanımlanır ve tüketici servisler (consumer’lar) ile benzer şekilde çalışırlar. Eğer lider öge çalışmazsa takipçi öğelerden birisi lider öge olarak görevi devir alır. Ayrıca Kafka altyapısında herhangi bir kesinti olması durumunda üretici ve tüketici servisleri bu konuda bilgilendiren ve Kafka sistemi bileşenlerinin sağlıklı bir şekilde

ayakta kalmasını kontrol eden ZooKeeper adında ilave bir sistem de mevcuttur (Koutanov, 2020).

Kafka hem çevrimdışı hem de çevrimiçi mesaj tüketimi için uygun bir altyapı sunar. Bir başka deyişle Kafka mesajları, veri kaybını önlemek için diskte tutulur ve gerektiğinde abone servislere dağıtılmak üzere çoğaltılır. Bu durum yayıncı servisler ile abone servisler arasındaki network iletişimi kopsa bile mesajların saklanmasına ve network iletişimi geri geldiğinde mesajların abonelere teslim edilmesine olanak tanır. Bu bakımdan Kafka hata toleranslı (fault-tolerant) bir mesajlaşma sistemidir (Bejeck, 2018).

Kafka, gerçek zamanlı veri akışlarını işlemek için geliştirilmiş bir platformdur. Bu bakımdan düşük gecikmeli mesaj teslimini sağlar ve makine arızalarının varlığında hata toleransı garantisi verir. Çok sayıda farklı yayıncı ve abone bileşeni idare etme yeteneğine sahiptir. Ayrıca çok hızlıdır, saniyede 2 milyon yazma gerçekleştirir ve tüm verileri diskte tutar. Bu, esasen tüm yazmaların işletim sisteminin (RAM) önbelleğine ve oradan da direk olarak bir ağ soketine veri aktarımı anlamına gelir. Bu nedenle ço verimli ve performanslı çalışır (Bejeck, 2018).

Kafka'nın avantajları genel olarak aşağıdaki gibi sıralanabilir (Koutanov, 2020):

- **Güvenilirlik (Reliability):** Kafka'da mesajlar bölümlenip çoğaltılarak paçalar halinde birden fazla abone servise iletilebilir. Mesajların iletimi hataya (yani veri kaybına) karşı dayanıklıdır. Bir başka deyişle oluşabilecek bir network sorunu nedeniyle eğer mesajlar ya da mesajların parçaları abonelere ulaştırılamazsa Kafkanın dahili veri tabanında saklanarak network bağlantısı gelir gelmez aboneye teslim edilir. Dolayısıyla Kafka sistem hatalarına karşı dayanıklıdır (yani fault-tolerant yapıya haizdir).
- **Ölçeklenebilirlik (Scalability):** Kafka, aşırı mesaj yükü altında kesinti olmadan kolayca ölçeklenebilir yani sistem kesintisi yaşanmadan kapasitesi artırılabilir.
- **Dayanıklılık (Durability):** Kafka, mesajların diske olabildiğince hızlı şekilde kaydedilmesini sağlayan dağıtılmış taahhüt günlüğünü (distributed commit log) yapısını kullanır, bu nedenle mesaj kayıplarına karşı dayanıklıdır.
- **Performans:** Kafka hem mesajları yayınlamak hem de abone olmak isteyen yazılım bileşenleri için yüksek performanslı bir mesaj saklama altyapısı sunar. Dolayısıyla büyük miktarlarda (yani TB seviyesinde) mesaj saklansa bile istikrarlı bir performans ortaya koyar.

Apache Kafka mesajlaşma sisteminin kullanımı aşağıdaki koşullarda tercih sebebi olabilir (Koutanov, 2020):

- Tüketicilerin mesaj almak için anlık olarak broker'ları tetiklemesi gerekiyorsa.
- Mesajlar kaybolmamalıysa (Persistent olmalıysa).
- Bankacılık sistemleri gibi birim zamanda yüksek hacimde verinin işlenmesi gerekiyorsa (yani yüksek throughput/verim gereksinimi varsa).
- Sistemin aşırı yüklenmesi durumunda kesintiye yol açmadan kolayca anlık ölçeklenebilirlik gerekiyorsa.

Apache Kafka mesajlaşma sistemin dezavantajları şu şekilde sıralanabilir:

- Üreticiler ve tüketiciler arasında Native Kafka protokolü dışında tek bir iletişim protokolü kullanılır. IN'da olduğu gibi AMQP, STOMP, MQTT protokolleri desteklenmez. Dolayısıyla AMQP gibi özelleşmiş mesajlaşma network protokolleri kullanılamaz.
- Microsoft'un .net platformu ile geliştirilen servislerin Kafkaya entegre edilmesi için üçüncü parti geliştiricilerce geliştirilmiş kütüphanelere ihtiyaç vardır.

Doktora çalışmamız kapsamında yüksek verimlilikte (throughput) veriyi işleme potansiyeli, mesajları saklaması özelliğinden kaynaklı fault-tolerant yapısı ve kolay ölçeklenebilirliği nedeniyle Apache Kafka'nın kullanımı tercih edilmiştir.

1.10. Önbellekleme (Caching)

Günümüzde günlük hayatta gerçekleştirilen rutin işler yazılım sistemleri ile modellenmek suretiyle otomatikleştirilmektedir. Bu şekilde kullanıcıların günlük hayatta yaptığı işleri daha hızlı şekilde yapmaları sağlanmış olmaktadır. Geliştirilen yazılım sistemlerini kullanan kişi sayısı artsa da sistemlerde genel bir yavaşlamanın olmaması ve kullanıcılara hızlı cevap vermesi arzu edilir. Önbellekleme (caching) teknikleri bu arzunun gerçekleşmesine olumlu katkı sağlar. Bununla birlikte çok sık erişilmeyen dataları önbellekleme doğru değildir zira istek yapan kullanıcıya yanlış veri gönderimi gerçekleşebilir (Carlsson, 2013).

Bu bağlamda önbellekleme sık kullanılan verileri kaydetme için geliştirilen bir teknik olarak tanımlanabilir ve şu ihtiyaçlar kapsamında kullanımı önerilir (Silva, 2015):

- Sistem kullanıcılarına hızlı cevap vermek.
- Daha iyi kullanıcı deneyimi sağlamak.
- Veri tabanına doğru gereksiz network trafiğine ve dolayısıyla bant genişliği kullanımına neden olabilecek veri trafiğine engel olmak.
- Sisteme bir kişi de girse 1000 kişi de girse aynı kullanıcı deneyimi sağlamak.

Ön belleklemenin gerçekleştirilmesinde iki tür stratejik yaklaşım mevcuttur (Burns, 2018):

- On-Demand:** Verilerin talep edildiği anda önbelleklenmesidir.
- Prepopulation:** Uygulama ayağa kalkar kalkmaz ancak istek gelmeden önce verilerin öbelleklendiği stratejidir.

Diğer yandan ön belleklemede iki tür önbellek ömrü söz konusudur

- Absolute Time:** Verilen süre dolduğunda veri önbellekten bütünüyle silinir.
- Sliding Time:** Verilen süre içerisinde önbellek verisine erişilirse, önbelleğin geçerlilik süresi “sliding time” süresi kadar daha uzatılır Bununla birlikte sadece bu özellik kullanılırsa her zaman eski veri kullanıcıya dönülebilir. Bu arzu edilmeyen durumu engellemek için sliding time’la birlikte absolute time da kullanılır.

Bunun dışında ön bellekleme kullanılabilecek iki somut yöntem mevcuttur. Bunlardan ilkinde ön belleklenecek veri yazılım sistemini çalıştıran sunucunun ram belleğinde (In Memory Caching) tutulur. Dolayısıyla önbellekte tutulabilecek veri boyutu sunucunun RAM bellek kapasitesi ile sınırlıdır. Bellekte ön bellekleme yöntemi yazılım uygulamasından sadece tek bir sunucuda çalışan tek bir numune mevcutsa hiçbir sorun yaşanmadan kullanılabilir. Ancak uygulamanın birden fazla sunucuda çalışan birden fazla numunesi mevcutsa ve bu numunelere bir “load balancer” yönlendirme yapıyorsa uygulamaya istek yapan bir kullanıcı farklı zamanlarda farklı sunuculara yönlendirilebilir. Bu durumda kullanıcıya uygulamanın çalıştığı farklı sunuculardan farklı zamanlarda farklı önbellek verileri döneceğinden kullanıcı her istek yapışında farklı bir veri görür. Bu tutarsızlığı engellemek adına “load balancer” üzerinde “sticky session”

özelliđi aktifleřtirilir. Bu özellik sayesinde uygulamaya istek yapan kullanıcı belirli bir süre boyunca (önbelleđin geçerlilik süresi dolana dek) devamlı aynı uygulama numunesine (yani aynı sunucuya) yönlendirilir Dağıtık ön bellekleme yönteminde, önbellek verileri uygulamanın çalıştırıldığı sunucunun ram belleğinde tutulmaz. Önbellek bunun yerine ayrı bir önbellek sunucusunda tutulur. Dağıtık ön bellekleme yönteminin avantajları řu şekilde ifade edilebilir (Burns, 2018):

- Veri tutarsızlığının önüne geçilmiş olur.
- Uygulamanın çalıştırıldığı bilgisayar yeniden başlatıldığında önbellek verileri silinmez.

Dağıtık ön bellekleme yönteminin dezavantajları ise řu şekilde ifade edilebilir (Burns, 2018):

- Ayrık bir ön bellek sunucusundan önbellek verilerinin çekiliyor olması dağıtık ön belleklemenin bellekte ön bellekleme yöntemine göre daha yavaş bir yaklaşım olmasına neden olabilir.
- Kullanımı bellekte ön belleklemeye göre daha zordur zira ayrık önbellek sunucusu ile network iletişimini koordine etmek için uygulamada ilave kodlama gereklidir.

Eđer bir yazılım sistemi birden fazla sunucu üzerinde çalışıyorsa bellekte ön bellekleme yerine dağıtık ön belleklemenin kullanımı verimlilik sağlanması açısından daha doğru olacaktır zira sunuculardan birinde arıza olması durumunda o sunucunun yerini alacak yeni bir sunucu önbellek verilerine erişebilecek ve sistemin genel işleyiři kaldığı yerden devam edebilecektir. Mikro servis mimarisinin doğası geređi uygulamalar (yani mikro servisler) çok sayıda sanal sunucu üzerinde çalışıyor olacağından bu mimaride dağıtık ön bellekleme (distributed caching) yaklaşımının kullanımı benimsenmektedir (Bellemare, 2020). Dolayısıyla bu doktora çalışmamız kapsamında da dağıtık bellekleme yaklaşımının kullanımı uygun görölmüştür.

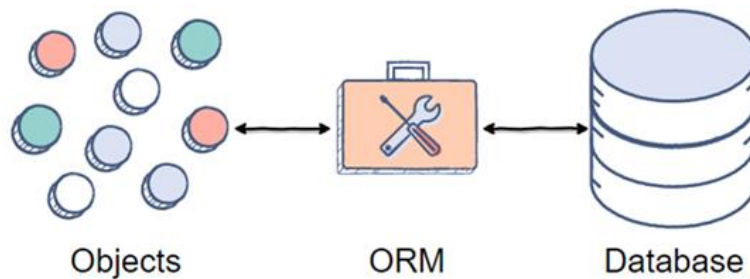
Dağıtık ön bellekleme için endüstride genel olarak hızlı okuma/yazma kapasitesine sahip NoSQL veri tabanı sistemi tercih edilmektedir. Dağıtık ön bellekleme kapsamında endüstride kullanılan veri tabanı sistemlerinden en yaygın olanı Remote Dictionary Service (Redis) olarak adlandırılan bir NoSQL veri tabanı sistemi olup veriyi anahtar/deđer çiftleri (key-value pairs) şeklinde tutan bir yapıya sahiptir. C dilinde

yazılmış olan Redis veriyi bellekte tuttuğu için çok anahtar (key) esaslı hızlı okuma ve yazma yapabilmektedir (Tavares ve ark., 2015). Dolayısıyla hızlı olması ve BSD lisanslı açık kaynak kodlu sistem olması gibi avantajları nedeniyle doktora çalışmamız kapsamında Redis teknolojisinin kullanımı uygun görülmüştür.

1.11. ORM (Object Relational Mapping)

ORM (Object Relational Mapping), nesne yönelimli programlama dillerinde uyumsuz tipler arasındaki verilerin dönüştürülmesi için kullanılan bir programlama tekniğidir. ORM kullanımıdaki esas amaç, bir sınıf örneği olarak alınan nesne katmanından, veri tabanı tablosundaki kayıtlara eşleme sağlamaktır.

Nesne yönelimli programlama ile ilişkisel veri tabanları arasında önemli farklılıklar bulunmakta ve nesnelerin veri tabanlarıyla doğrudan uyumluluğunu engelleyen özellikleri bulunmaktadır. Nesneler geçicidir ve çalışma zamanında örneklenirken ilişkisel modeldeki veriler kalıcıdır. Nesnelerin, veri tabanlarının aksine sabit bir şeması bulunmamaktadır. Nesnelerin şemasının bulunmamasından dolayı ilişkisel veri tabanlarıyla etkileşim kurmak için bir dil (SQL) kullanılmaktadır. ORM araçlarıyla, nesne yönelimli programlama için bir ara yüz sağlayarak nesne ve veri tabanı arasındaki uyumsuzluğun giderilmesi sağlanmaktadır. Bu sayede uygulamanın kendisinde SQL manipülasyonu için doğrudan bir yeteneğe ihtiyaç duyulmamakta ve sadece ORM araçları kullanılmaktadır (Colley ve ark., 2018). Şekil 1.59'da uygulamanın ORM'e bağlı olduğu görülmektedir. Uygulama veri tabanı ile direk olarak herhangi bir bağlantısı bulunmamaktadır. Uygulama tarafında oluşan nesneler üzerinden veri tabanı işlemleri yapıldığında ORM tarafından bu işlem SQL ifadesine dönüştürülmektedir. SQL ifadesine dönüştürme işlemine Code Generating işlemi denilmektedir. Tüm ORM araçları aynı işlemi yapmaktadır.



Şekil 1.59. Uygulama ve Veri Tabanı Arasındaki ORM Bağlantısı (Edpresso Team, 2021)

ORM kullanmanın avantajları şunlardır:

- Nesne tabanlı programlama standartlarına uygun olarak kod yazma imkânı sağlanmaktadır.
- Minimum SQL bilgisi ile veri tabanı işlemleri yapma imkânı tanımaktadır.
- Veri tabanı platformu bağımlılığı bulunmamaktadır.
- Kod yazma süresini kısaltmaktadır.
- Kod okunabilirliğini arttırmaktadır.

ORM kullanmanın dezavantajları şunlardır:

- Veri tabanı, nesneler üzerinden modellendiği için nesneler arasında bağ bulunmaktadır. Bilinçsiz yapılan müdahaleler ile bağlantıyı bozma durumu oluşabilmektedir.
- Yeni başlayacak yazılım geliştiricileri için yazım kuralları (syntax) farklı gelebilmektedir.
- Karmaşık veri işlemleri için SQL sorguları kadar rekabetçi değildir. (HARLEY, J., 2011).

Aşağıdaki alt başlıklarda günümüzde endüstride yaygın olarak kullanılan iki ORM kütüphanesi olan Dapper ve Entity Framework'ten bahsedilmektedir.

1.11.1. Dapper Mikro ORM

Dapper, Stackoverflow ekibi tarafından açık kaynak kodlu olarak geliştirilmiş bir micro ORM'dir. Dapper'ın geliştirilmesindeki temel amaç performans ve kullanım kolaylığı sağlanmasıdır. Nuget paketi olarak indirilebilen Dapper, yazılım projelerinde kütüphane olarak kolaylıkla kullanılabilir. Ayrıca Dapper kullanarak SQL DML (Data Manipulation Language) işlemleri yapılabilir. Dapper'ın kullanım alanları incelendiğinde özellikle yoğun trafik alan e-ticaret projelerinde performans kaygılarından dolayı Entity Framework vb. gibi ORM araçları tercih edilmemektedir. Bu noktada Dapper gibi Micro ORM araçları ön plana çıkmaktadır. Şekil 1.60'da görülen performans testi sonuçlarından Dapper'ın diğer rakiplerinden önde olduğu anlaşılmaktadır.

Dapper, hafif (Light-weight) yapısı sayesinde tür dönüşümü yapmadan, arka planda konfigürasyon ya da xml dosyası bulundurmadan basit bir şekilde

kullanılabilmektedir. Bununla beraber Dapper ile direkt olarak SQL sorguları uygulama içerisinde çalıştırılabilmektedir. Bu sayede üst düzey performans sağlanmaktadır. SQL sorgularının direkt olarak yazılabilmesi SQL injection saldırılarına zemin hazırlayabilir gibi görünse de Dapper’da bu duruma önlem almak adına dinamik parametreler adında yapılar geliştirilmiştir. Aynı zamanda direkt olarak saklı yordamlar (Stored Procedure) ile çalışmak da mümkündür (BORSOFT, 2016).

Method	Duration
Hand coded (using a SqlDataReader)	47ms
Dapper ExecuteMapperQuery	49ms
ServiceStack.OrmLite (QueryById)	50ms
PetaPoco	52ms
BLToolkit	80ms
SubSonic CodingHorror	107ms
NHibernate SQL	104ms
Linq 2 SQL ExecuteQuery	181ms
Entity framework ExecuteStoreQuery	631ms

Şekil 1.60. ORM Araçları Performans Analiz Sonuçları

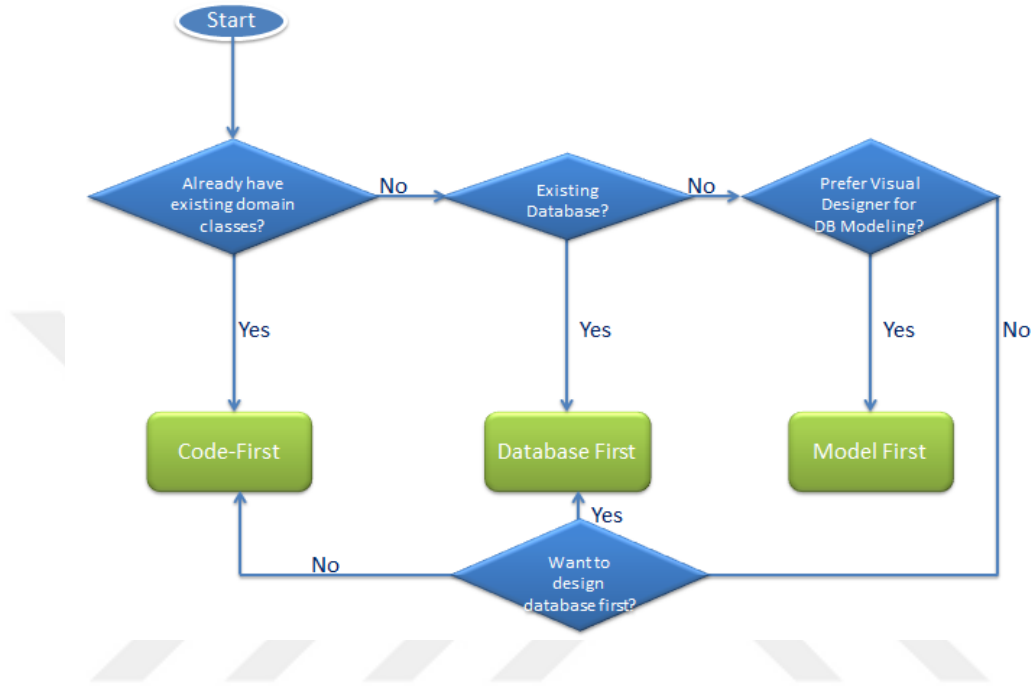
1.11.2. Entity Framework

Entity Framework, ilişkisel veri tabanı ile nesne yönelimli programlama arasında köprü görevi gören bir ORM aracıdır. Microsoft tarafından geliştirilmiştir. Kompleks SQL sorgularından ziyade pratik şekilde CRUD işlemlerinin yapılmasını sağlamaktadır. Entity Framework temelde ADO.Net altyapısını kullanmaktadır. Nuget paketi olarak indirilip projeye entegre edilebilmektedir (Aslan, 2019). Entity Framework kullanımı ile alakalı 3 temel yaklaşım bulunmaktadır:

- i. **Model First:** Visual Studio üzerinde boş bir model dosyası eklenerek veri tabanının bu model üzerinde tasarlanabilmesi sağlanmalıdır. Derleme anında verilen script dosyası veri tabanını oluşturmaktadır.
- ii. **Database First:** Önceden oluşturulmuş veri tabanını projeye model olarak bağlayarak gerekli sınıflar Entity Framework tarafından oluşturulmaktadır.

- iii. **Code First:** Sınıflar Visual Studio ortamında oluşturulup veri tabanı bu sınıflardan türetilmektedir.

Şekil 1.61’de bu üç temel yaklaşımın seçiminde kullanılabilecek adımları ifade eden için bir akış diyagramı görülmektedir.



Şekil 1.61. Entity Framework Temel Yaklaşımları (Alagöz, 2018)

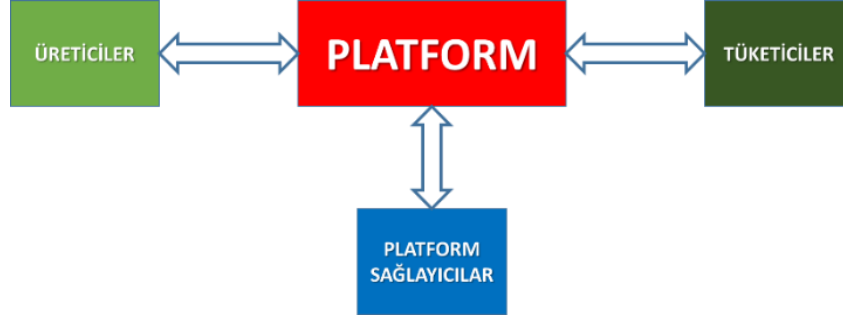
Entity Framework kullanımının avantajları şu Şekildedir:

- i. Veri tabanı bağımlılığı kalkmaktadır.
- ii. Sade ve basit SQL sorguları sayesinde veri tabanı performansı artmaktadır.
- iii. Yazılım geliştiricilerinin veri tabanı üzerinde çalışma yapmadan sadece uygulama üzerine odaklanmasına olanak sağlamaktadır.
- iv. Kod yazma süresini kısaltarak daha az zamanda daha çok iş yapmayı sağlamaktadır (Sezer, 2019).

1.12. Tez Çalışmasının Amacı

İktisadi manada platform terimi, bir ürün ya da ürün grubunun sermaye sahipliğini üstlenmeden bu ürünlerin üretici ve tüketici kesimler arasında alım-satımına olanak tanıyan bir ticari ekosistem olarak ifade edilebilir (Constantinides ve ark., 2018; Giesen,

2017; Kabakova ve ark., 2016). Böyle bir ekosistemi hazırlayıp sunan kurum ya da kuruluşlara platform sağlayıcı denilmektedir. Aşağıda Şekil 1.62’de iktisadi bir platformu oluşturan ekosistem ögeleri temsili olarak görülmektedir.



Şekil 1.62. İktisadi Platform ve Ögeleri

Dijital iktisadi platformların teknik manada oluşumunda FinTek yazılım uygulamalarının bankaların yazılım altyapısına entegrasyonu önemli bir gerekliliktir zira üretici ve tüketici kesimler arasındaki dijital para akışı bankacılık lisansına sahip olmayan FinTeklere değil doğrudan bankalara bağlıdır. Bu entegrasyonun sağlanması amacıyla dünya genelinde Açık Bankacılık (Open Banking) adı altında gerçekleştirilen standartlaştırma çalışmaları mevcuttur (Gozman ve ark., 2018). Bu çalışmalar kapsamında bankacılık yazılım sistemlerine erişimin, güvenliği tesis edilmiş ve standartlaştırılmış Uygulama Programlama Ara-yüzleri (Application Programming Interface’ler-API’ler) aracılığı ile sağlanması amaçlanmıştır. Bu doğrultuda Avrupa Birliği tarafından 2015’te Ödeme Hizmetleri Şartnamesi (Payment Services Directive–PSD2) adı altında bir yasa ve buna bağlı yönerge çıkarılmıştır (Anonim, 2015). Türkiye Büyük Milleti Meclisi’nce 1 Ocak 2020 tarihi itibarıyla yürürlüğe konulan 7192 numaralı kanun değişikliği ile birlikte PSD2 uyumu kapsamında Türkiye’deki ilk yasal düzenlemeler yapılmış olup özel ya da kamusal bütün Türkiye Cumhuriyeti Bankalarının PSD2 uyumluluğunun sağlanması zorunlu kılınmıştır.

PSD2 şartnamesi kapsamında oluşturulan standartlar, bankaların sağladığı API servislerine erişim hakkında üst düzey (high-level) standartlar ortaya koymakta iken; API’lerin kullandığı yazılım altyapısının nasıl tasarlanıp geliştirileceği konusunda teknik bir standart ortaya koymamaktadır (Cortet ve ark., 2016).

Diğer yandan günümüzde pek çok bankanın yazılım altyapısı Bulut Bilişimin yaygınlaşmasından önceki dönemde monolit mimari esaslı geliştirilmiş bir yapı arz

etmektedir (Megargel ve ark., 2020). Bu monolit bankacılık sistemlerinin (Örnek: Main-frame bankacılık sistemleri) günümüzün bulut-yerlisi yazılım geliştirme yaklaşımında kullanılan “Kubernetes” ve “Container” gibi donanım bağımsız sanallaştırma teknolojilerine üzerinde çalışacak şekilde modernize edilmesi oldukça maliyetli ve güç bir iştir (Fritzsche ve ark., 2019; Chen ve ark., 2017; Fowler ve Lewis, 2014). PSD2 şartnamesinin yalnızca bankacılık API’lerinin şekli unsurlarını standartlaştırması ve bankacılık sistemlerinin nasıl tasarlanıp geliştirileceği konusunda teknik bir standart ortaya koymaması bu monolit sistemlerin nasıl modernize edilebileceği konusunda ayrıca bir belirsizlik ortamı oluşturmaktadır. Bu bakımdan PSD2 standartlarına ek olarak monolit bankacılık yazılım sistemlerinin modern bulut mimarisi esaslı platformlarda yaygınlaştırılmasına imkân tanıyacak altyapısal bir çözüme de ihtiyaç vardır. Sanallaştırma teknolojilerinde ve bulut bilişimdeki gelişmeler doğrultusunda ortaya çıkmış olan *Mikro Servis Mimarisi (MSA)* bu ihtiyacı giderebilecek bir dağıtık sistem mimarisi ortaya koymakta olup Amazon, Airbnb, Netflix ve Uber gibi firmalar ile Emirates NBD¹, Bank Monzo UK², Solaris Bank³ ve Bank Canada (CIBC)⁴ gibi bankalarca benimsenmiş ve daha pek çok kurum tarafından da adaptasyonu gerçekleştirilmiştir (Jeremy, 2020; Currie, 2017).

Buradan hareketle, doktora çalışmamız kapsamında çözüm bulmayı amaçladığımız problem (Research Question) şu şekilde ifade edilebilir: “*Bir bankanın sahip olduğu monolit yapıdaki temel bankacılık sistemi, Açık Bankacılık Standartlarına uyumlu bir bankacılık platformuna Mikro Servis Mimarisi temelinde nasıl dönüştürülebilir?*”.

Bu soruya cevap bulma amacı doğrultusunda bankaların monolit yazılım sistemlerinin Mikro Servis Mimarisi temelinde PSD2 uyumlu bir açık bankacılık sistemine dönüştürülmesi için gerekli yazılım ihtiyaçları araştırılmış, tekniklerin/modellerin/tasarım kalıplarının uygulanabilirliği tespit edilmiş ve bu doğrultuda bir mimari model ortaya konmuştur. Bu mimari modelin doğruluğu bir

¹ <https://www.redhat.com/rhdc/managed-files/rh-emirates-nbd-case-study-f16724cs-201904-en.pdf>

² https://www.theregister.com/AMP/2020/03/09/monzo_microservices/

³ <https://www.solarisbank.com/blog/solarisbank-core-banking-beginning-of-a-journey/>

⁴ <https://nordicapis.com/creating-a-microservices-framework-at-cibc-a-case-study/>

TÜBİTAK 1505 üniversite-sanayi iş birliği projesi kapsamında Kuveyt Türk Katılım Bankası'nın sağladığı donanım kaynakları kullanılarak gerçekleştirilen bir vaka çalışması kapsamında test edilmiştir. Bu kapsamda Kuveyt Türk Bankası bünyesinde kullanılmakta olan monolit yapıdaki “*Business Oriented Architecture (BOA)*” adlı çok kanallı bankacılık yazılım altyapısının Mikro Servis Mimarisi temelinde PSD2 uyumlu bir bankacılık yazılım altyapısına dönüştürülmesine zemin teşkil edecek mikro servis mimarisi temelli bir alt yapı tasarlanıp geliştirilmiştir. Bu alt yapı *Open Banking Architecture (OBA)* olarak adlandırılmıştır.

1.13. Tez Dokümanı Organizasyonu

Bu tez dokümanı aşağıda özetlenen bölümler halinde organize edilmiştir:

Bölüm 1- Giriş: Bu bölümde doktora çalışmamızın araştırma konusu ile ilgili konularda genel bilgiler verildikten sonra projenin amacına ve özgün yönüne değinilmektedir.

Bölüm 2 – Kaynak Araştırması: Bu bölümde doktora araştırma konumuzla ilgili önceki çalışmaların ana fikirleri, metotları ve sonuçları hakkında genel bilgiler verilmektedir.

Bölüm 3 – Materyal ve Yöntem: Bu bölümde doktora çalışmamız kapsamında ele aldığımız problem ile bu problemin çözümü için ortaya koyduğumuz hipoteze değinilmektedir. Bu kapsamda ayrıca yaptığımız araştırmanın düzenlenme şekli ile kullanılan bilimsel yöneme de değinilmektedir.

Bölüm 4 – Bankacılık İşlev Sınıfları: Bu bölümde bankacılık servislerinin açık bankacılık standartlarını yansıtacak şekilde işlev sınıflarına ayrıştırılması sağlanmış ve bu suretle bir bankanın dijital hizmetlerini açık bankacılık standartlarında nasıl organize edebileceği ortaya konmuştur.

Bölüm 5 – Tasarım ve Geliştirme Çalışmaları: Bu bölümde doktora çalışmamızla ilişkili TÜBİTAK projesi kapsamında yaptığımız vaka çalışması sırasında gerçekleştirdiğimiz tasarım ve geliştirme çalışmaları ile ilgili bilgiler verilmektedir. Ayrıca geliştirme çalışmaları üzerinde icra ettiğimiz yük testlerinin nasıl gerçekleştirildiğine ve sonuçlarına değinilmektedir.

Bölüm 6 – Dağıtık SAGA İşlemlerinin Aktör Modeli Esaslı Yönetimi: Bu bölümde doktora çalışmamızın en can alıcı konusu ve akademik katkısını oluşturan dağıtık SAGA işlem (transaction) yönetimi konusundaki buluşumuzun içeriği açıklanmaktadır.

Bölüm 7 – Sonuçlar: Bu bölümde yaptığımız çalışmalar genel olarak özetlendikten sonra ortaya konulan mimari yapının yük testi sonuçları değerlendirilmekte ve mikro servis mimarisinin sahada uygulanışı noktasında genel değerlendirmemiz sunulmaktadır.



2. KAYNAK ARAŞTIRMASI

Bu bölümde doktora çalışmamız kapsamında gerçekleştirdiğimiz kaynak araştırmasına yer verilmektedir. Buradaki kaynak araştırmamız sırasıyla FinTekler, Açık Bankacılık/API Teknolojileri, Mikro Servis ve Monolit Mimarileri ve Mikro Servis Mimarisi Temelli Sistemlerde Dağıtık SAGA İşlemlerinin (Transactionlarının) Koordinasyonu konulu başlıklar altında organize edilmiştir.

2.1. FinTek Konulu Çalışmalar

Literatürde FinTek ile ilgili çalışmalar incelendiğinde FinTek’lerin tarihsel gelişimi, FinTek ekosistemleri, iş modelleri, kullanılan teknolojiler ve yapılan yatırımlar konularında çalışma ve makalelere yer verilmiştir. Bu çalışmalara örnek olarak şunlar verilebilir:

Demirdöğen (2019) tarafından yapılan çalışmada dünyadaki FinTek regülasyonları ve işlerliği incelenmiştir. Yağcı (2018) finansal teknolojilerin ekonomi politikasını FinTek ve Bitcoin kapsamında değerlendirmiştir. Arner ve ark. (2015), FinTek’lerin son 150 yıldır yaşadığı dönüşümü ve bu konudaki regülasyonları analiz etmişlerdir. Lee ve Shin (2018) FinTek’lerin tarihsel gelişimini ortaya koyarak, FinTek ekosistemini, iş modellerini ve yatırım türlerini incelemiştir. Pollari (2016), FinTek’lerin finansal hizmetler sektörü ve iş modelleri üzerindeki etkisini ve yatırım eğilimlerini analiz etmiş ve finansal kurumlar için stratejik zorluk ve fırsatları araştırmıştır. Puschmann (2017), finansal hizmetler sektörünün dijitalleşmesi ile FinTek’ler arasındaki ilişkiyi incelemiştir. Marginingsih (2019) konvansiyonel bankacılıkta; Baidhowi (2018) ise İslami bankacılıkta finansal teknoloji kullanımının etkilerini SWOT analizi ile değerlendirmişlerdir. Çağıl ve Candemir (2020) tarafından Türk Bankacılık Sektörünün Finansal Teknoloji Şirketleri (FinTek’ler) kaşısındaki durumu adlı çalışmada A’WOT yöntemi ile analiz yapılmış ve Türk bankaları için FinTek’lere yönelik uygulanabilecek stratejiler önerilmiştir.

2.2. Açık Bankacılık ve API Teknolojileri Konulu Çalışmalar

Açık Bankacılığın gelişmesi ile birlikte literatürde birçok çalışma yapılmış ve makaleler yazılmıştır. Açık bankacılığın yapı taşı-temelini oluşturan API'ye de bu çalışmalarda yer verilmiştir. Bu çalışmaların bazılarına aşağıda yer verilmektedir:

Wang, Ma, Dai ve Imran (2020), bir açık bankacılık sistemini, Nudge Teorisi'ni esas alan ve blockchain tabanlı bir veri gizliliğine dayalı bir model geliştirerek açıklamaya çalışmışlardır.

Araluze (2020), İspanya'da müşterilerin açık bankacılık modeline ihtiyaçlarının olup olmadığı, bankacılık sistemindeki değişikliğin pazardaki rekabetten mi yoksa müşteri ihtiyaçlarından mı kaynaklandığını ortaya koyan bir çalışma gerçekleştirmişlerdir. Çalışmada müşteri ihtiyaçlarının pazardaki rekabetten önce geldiği görülmüştür.

Taştan ve Saruhan (2020), açık bankacılığın Türk hukukunda ve Avrupa Birliği hukukundaki gelişimini 6698 sayılı Kişisel Verilerin Korunması Kanununu temel alan yedi hukuki problemi ele almıştır.

Rastogi ve ark., (2020) yapmış oldukları çalışmada bir anket yöntemi kullanarak topladıkları verileri en küçük kareler yöntemine göre analiz etmişler ve bu doğrultuda açık bankacılık ile ekonomik kalkınma arasında bir ilişki olduğunu tespit etmişlerdir. Açık bankacılık hizmetlerinin bilinirliğindeki ve kullanımındaki artışa paralel olarak ekonomik kalkınmayı artacağı belirtilmiştir.

Hsieh (2019) tarafından yapılan çalışmada Tayvan'da açık bankacılığın gelişimini Açık veri, açık API ve açık ekosistem olarak ifade edilen unsur üzerinden incelemiştir.

Aksoy ve Bilgel (2019) yaptıkları çalışmada Türk bankacılık sektöründe uygulama programlama ara yüzlerinin (API) altyapısı çerçevesinde FinTeklerle gerçekleştirilebilecek stratejik hedefleri tartışmış ve bu doğrultuda ortaya konulabilecek yeni ürünlerin teknolojik gelişmeye olan katkısını değerlendirmişlerdir.

Sivathanu (2019), açık bankacılığın Hindistan'daki etkilerini 945 müşteri üzerinde uygulanan bir anket ile "Teknoloji Hazırlık ve Kabul" modeli bağlamında incelemiştir. Anket sonuçları, katılımcıların açık bankacılık uygulamaları konusunda iyimser olduklarını ancak güvenlik riskinin katılımcılar üzerinde etkili olan negatif bir unsur olarak yer aldığını ortaya koymuştur.

Kartal ve Zengin (2019), API uygulamalarının bankacılık sektörüne etkilerine yönelik bir çalışma yapmışlardır. Bankacılık istemine entegre edilecek yeni kanallarla

bankaların finansal hizmet dağılımının artacağı ve dağıtım maliyetlerinde düşüşler yaşanacağını ortaya koymuşlardır.

Kolobova ve ark., (2018), açık bankacılığın Rusya'daki gelişimini incelemiştir. Çalışmada gelişimin yavaş olduğu belirtilmiştir. Bu gelişimin yavaş olmasının sebepleri olarak güvenlik riski ve bankalar arası rekabetin yüksek olması gösterilmiştir.

Omarini (2018), bankalardaki FinTek ve açık bankacılık uygulamalarını BBVA örneği üzerinden incelemiştir. Geleneksel bankaların, açık bankacılık uygulamalarından ve FinTek'lerden yararlanmaları gerektiği çalışmada vurgulanmıştır. Müşteri odaklı çalışmaların ve doğru yatırımların FinTek sektörüne olan talebi arttıracığı belirtilmiştir.

Zachariadis ve Özcan (2017), API kavramını açıkladıktan sonra Avrupa Birliği ve İngiltere'deki bu kavramın açık bankacılık kapsamındaki uygulamalarını istatistik ve mülakat yöntemleriyle incelemiştir.

Açık bankacılık sistemine ilişkin hukuk kapsamında yeterli sayıda çalışma olduğunu söylemek pek mümkün değildir. Bununla birlikte müşteri bazlı olarak adlandırılabilen bazı çalışmalar PSD2'yi tanıtip yükümlülüklerden genel olarak bahsetmektedir (Klynveld Peat Marwick Goerdeler [KPMG], 2017; Deloitte, 2017). Diğer yandan nitelik bağlamında önem arz eden iki çalışmanın altını çizmek gerekir. Bunlardan ilki olan European Commission, PSD2'nin uygulanması için belirlenen teknik standartları ve bunların daha güvenli ve yenilikçi elektronik ödeme servisleri için etkisini ele almıştır. Avrupa Bankacılık Federasyonu (European Banking Federation-[EBF]) ise PSD2 bağlamında rehber niteliği taşımaktadır (Anonim, 2015). Son olarak konu finans boyutundan ziyade salt hukuk boyutuyla Gün (2019) tarafından ele alınmış ve gerek Avrupa'daki gerekse Türkiye'deki etkileri ve geleceği incelenmiştir.

API teknolojileri ve mikro servis mimarisi kapsamında yaygın olarak API geçitleri (API Gateways), devre kesiciler (circuit breakers) ve hizmet keşif araçları (service discovery agents) gibi mekanizmalar hakkında da literatür çalışmaları mevcuttur. Bu çalışmalar, yalnızca bu mekanizmaların işlevsel tasarımını değil; aynı zamanda kalitesini etkileyen kullanılabilirlik, birlikte çalışabilirlik, performans, ölçeklenebilirlik ve güvenlik gibi işlevsel olmayan yönlerini de içermektedir. Bu bağlamda yapılan başlıca çalışmalar aşağıdaki gibidir:

Zhao ve arkadaşların yaptıkları çalışmada, yük dengeleme ve otomatik istek yönlendirme gibi mikro servis mimarisinde API Geçitleri tarafından gerçekleştirilen işlevlerin uygulanması kapsamında geliştirme verimliliğini artırıcı önerilerde bulunmuşlardır (Zhao ve ark.,2018).

Montesi ve Weber tarafından yapılan bir çalışmada, API geçidi, devre kesici ve hizmet keşfi gibi mikro serviste bulunan ortak mekanizmalar için farklı dağıtım stratejilerini/modelleri analiz edilmiştir. (Montesi ve Weber,2016).

Xu ve arkadaşlarının yaptığı çalışma sonucunda, güvenli bir kimlik doğrulama mekanizması oluşturmak için bir uç bilgi işlem platformunu bir API geçidi ile entegre etmek için bir mikro servis güvenlik aracı önerilmiştir (Xu ve ark.,2019).

Lu ve arkadaşları, IoT'deki “thing”'in bir mikro servis olarak temsil edildiği IoT sistemleri için bir mikro servis mimari önermişlerdir. Yazarlar, bir API geçidi aracılığıyla birbirine bağlanabilen birlikte çalışabilir ve güvenli IoT servisleri oluşturmak için bir mikro servis yaklaşımının kullanılabileceğini savunmuşlardır (Lu ve ark., 2017).

Shafabakhsh ve arkadaşları, performans verimliliği ve kullanılabilirliği açısından RabbitMQ, gRPC ve REST gibi eş zamansız ve eş zamanlı mikro servisler arası iletişim teknolojilerini incelemiştir (Shafabakhsh ve ark., 2020). Fernandes ve arkadaşları tarafından yapılan çalışmada REST API'lerinin performansı gelişmiş mesaj kuyrukla protokolü (AMQP) ile karşılaştırılmıştır. Çalışmanın sonuçları, AMQP'nin büyük miktarda verinin işlenmesini gerektiren senaryolar için REST'ten daha iyi performans gösterdiğini ortaya koymuştur (Fernandes ve ark., 2013).

IBM'de Ueda ve arkadaşları tarafından yürütülen başka bir çalışmada, özellikle mikro servis mimarilerini çalıştırmak için optimize edilmiş bir altyapı önerilmiştir. Bu altyapının iki farklı versiyonu hayata geçirilmiştir. Birinci versiyonu monolitik bir sistem olarak; ikinci versiyonu ise mikro servis mimarisine dayalı olarak geliştirilmiştir. Akabinde bu iki versiyon karşılaştırılmış ve mikro servisler arası iletişim doğru kurgulanmadığı takdirde, mikro servis mimarisi tabanlı versiyonun monolitik versiyondan daha düşük performans gösterebileceği belirtilmiştir (Ueda ve ark., 2016).

Dragoni ve arkadaşları bir Danimarka Bankasına ait monolit bir sistemi mikro servis mimarisi tabanlı bir sisteme dönüştürmek için bir vaka çalışması gerçekleştirmiştir. Bu dönüşüme dayanarak, yazarlar ortaya çıkan MSA tabanlı sistemi orijinal monolitik sistemle karşılaştırmışlar ve mikro servis mimarisi tabanlı sistemin çok daha iyi ölçeklenebilir olduğu sonucuna varmışlardır (Dragoni ve ark., 2017).

Genel API tasarımı ile ilgili olarak, Afonso ve arkadaşları, daha iyi API tasarımı gerçekleştirmek için tasarımcılar ve programcılar arasındaki iletişimi geliştirmeye yönelik birleşik bir semiyotik ve bilişsel yöntem sunmuşlardır (Afonso ve ark., 2012).

Lindman ve arkadaşları, API tasarımcılarının iş ve teknik kaygılarla ilgili tasarım kararlarını daha iyi organize etmelerine yardımcı olmak için bir API çerçevesi sunmuşlardır (Lindman ve ark., 2020).

Kullanılabilirlik gibi nitel API tasarım kriterleri ilgili çalışmalar da literatürde mevcuttur. Örneğin, Stylos ve arkadaşları, özellikle API'lerdeki kullanılabilirlik problemlerini belirlemek ve önlemek için API tasarım seçeneklerinin kullanılabilirliği ile ilgili genel bir teknik önermiştir (Stylos ve ark., 2006). Daha pratik içerikli benzer bir çalışma Piccioni ve arkadaşları tarafından yapılmıştır. Burada yazarlar, esneklik ve kullanılabilirlik açısından kalıcılık kitaplığı API'sini kullanırken programcıların davranışını sistematik olarak gözlemlemişlerdir (Piccioni ve ark., 2013). Bu alanda Rauf ve arkadaşları tarafından yapılan başka bir çalışmada kullanılabilirlik değerlendirmesi için hangi API faktörlerinin hangi API geliştirme aşamalarında kullanıldığını belirleyen için bir literatür araştırması yapılmıştır (Rauf ve ark., 2019).

2.3. Mikro Servis ve Monolit Mimari Konulu Çalışmalar

Literatürde mikro servis mimarisi ile ilgili 2012 yılından günümüze kadar çok sayıda çalışma yer almaktadır. Mikro servis mimarisi ile ilgili çalışmalar da monolitik mimari konusuna da yer verilmiştir. Bu çalışma ve makalelere şunlar örnek verilebilmektedir:

Terim 2012 yılında (Lewis, James; Fowler, 2014) tarafından popülerleştirilmiştir. Ancak, akademiye yoğun ilgi 2014 yılında ivme kazanmaya başlamıştır (Balalaie, Heydarnoori ve Jamshidi, 2016).

Escobar ve ark. (2016) tarafından yapılan çalışmada eskiden kalan (legacy) sistemlerin mikro servisler yoluyla modernleştirilebilmesi ile ilgili bir yaklaşım sunulmuştur. Yaklaşım iki önemli adımdan oluşmaktadır. İlk adım, eskiden kalan sistemi anlamaya yardımcı olan görselleştirme modelleri üretmektir. Bu adımda eskiden kalan sistemin güçlü ve zayıf bileşenlerini gösteren bir model oluşturulmaktadır. İkinci adımda ise bu model oluşturan elementlerinin aralarındaki bağımlılık ilişkisine dayanarak, gevşekçe bağlanmış bileşenleri ayrı mikro servislere toplayan bir dizi modelleme yapılmaktadır.

Abdullah ve ark. tarafından yapılan bir çalışmada eskiden kalan sistemlerin (legacy systems) mikro servislere dönüştürülmesinin performans ölçümü ve ölçeklenebilirliği incelenmiştir (Abdullah ve ark., 2019). Yazarlar, otomatik olarak

ayrıştırılmış eskiden kalan bir sistemin, manuel olarak ayrıştırılmış eskiden kalan bir sistemden daha iyi performans gösterdiğini ve daha iyi ölçeklenebildiğini iddia etmişlerdir. Bu nedenle, yazarlar, eskiden kalan sistemlerin mikro servislere otomatik olarak ayrıştırılması için bir yöntem sunmuştur. Yöntem, verilen bir monolitik sistemin kaynak tanımlayıcılarını (URI) kullanarak teknik, bir dizi URI tanımlamaktadır. Tanımlanmış olan bu URI'ler otomatik olarak farklı mikro servislere eşleştirilmektedir. Sonrasında her mikro servis, performansı en üst düzeye çıkarmak için uygun sanal makine türü kullanılarak dağıtılmaktadır. Veri tabanı katmanı bu çalışmada dikkate alınmamıştır.

Liv ve ark. tarafından yapılan bir diğer çalışmada monolitik sistemlerin mikro servislere ayrıştırılması için veri akışı temelli tekniklere dayanan, yarı otomatik bir yöntem geliştirilmiştir (Li ve ark., 2019). Bu yöntemde ilk olarak monolitik sistemin kullanım senaryosunun tanımlaması ile iş mantığını açıklanması yapılmaktadır. Bunu; veri akışı diyagramlarının oluşturulması ve bunu veri akışı diyagramının bir işlem-veri deposu versiyonunun elle geliştirilmesi takip etmektedir. Son olarak hem veri akışı hem de işlem-veri deposu diyagramları aday mikro servisler oluşturmak için otomatik olarak kullanılmaktadır. Bununla birlikte bu çalışma, monolitik sistemin ayrışmasının ötesine geçmemektedir.

Kazanavicius ve Mazeika tarafından yapılan bir çalışmada eskiden kalan monolitik bir sistemden mikro servis sistemine olan farklı dönüşüm yöntemlerini özetleyen bir çalışma yapılmıştır (Kazanavicius ve Mazeika, 2019). Bu çalışma, farklı taşınma yöntemlerini özetlemeden önce hem monolit hem de mikro servis sistemleriyle ilgili genel farklılıkları, avantajları ve dezavantajları ortaya koymuştur. Akabinde mevcut dönüşüm teknikleri yeniden düzenleme ve yeniden inşa etme şeklinde iki gruba ayrılarak incelenmiştir. Yeniden düzenleme, mevcut monolitlerin, uygulamayı sıfırdan yeniden oluşturmaya gerek kalmadan tek tek mikro servislere ayrılmasını ifade ederken, yeniden inşa etme, tüm sistemleri sıfırdan geliştirmeyi ifade etmektedir. Yazarlar, eskiden kalan sistemlerde kullanılan boyut, karmaşıklık ve teknolojilerin bu sistemleri ya yeniden inşa etme ya da yeniden düzenleme için bir aday haline getirdiğini savunmaktadırlar.

Mikro servisler hakkındaki mevcut kaynakların çoğu, sıklıkla bulut bilişim ile ilişkilidir. Örneğin Guo ve ark. tarafından yapılan bir çalışmada bulut için mikro servis tabanlı uygulamaların geliştirilmesi ve konuşlandırılması ele alınmıştır (Guo ve ark., 2016). Singh ve ark., tarafından yapılan çalışmada mikro servislerin bulut üzerinde dağıtımında konteynerlerin kullanımı üzerinde durulurken (Singh ve Peddoju, 2017) Gan

ve Delimitrou tarafından yapılan başka bir çalışmada mikro servislerin bulut üzerindeki etkisi incelemiştir (Gan ve Delimitrou, 2018). Zhen ve wei tarafından yapılan çalışmada bulut bilişim için mikro servis tabanlı uygulamaların teknik ve ekonomik analizleri yapılmıştır (Zheng ve Wei, 2018). Villamizar ve ark. tarafından yapılan çalışmada ise mikro servis uygulamalarının buluta dağıtımının faydaları ve zorlukları tartışılmıştır (Villamizar ve ark., 2015).

Mikro servislerin, Nesnelerin İnterneti (IoT) konusunda da akademik çalışmalar mevcuttur. Bu kapsamda akıllı bir şehir inşa etmek için mikro servislerin kullanımını öneren bir çalışma yapılmıştır (Krylovskiy ve ark., 2015) Sun ve ark., IoT'nin uygulanışında ilgili farklı gereksinimlerin, birlikte çalışabilirliğin, özelleştirmenin ve ölçeklenebilirliğin zorluklarını çözmek için mikro servislere dayalı bir açık ortam önerisinde bulunmuşlardır (Sun ve ark. 2017). Jarwar ve ark. tarafından yapılan bir diğer çalışmada web nesneleri içerisinde mikro servislerin kullanılarak IoT ortamında yeniden kullanılabilirlik prensibinin geliştirilmesi üzerinde durulmuştur (Jarwar ve ark., 2018). AlMasri tarafından yapılan çalışmada da IoT uygulamalarının işlevsel olmayan nitelikleri incelenerek mikro servislerin genel hizmet kalitesinin geliştirilmesi için detaylı bir çerçeve çizilmiştir (AlMasri, 2019).

Mikro servis uygulamalarının güvenliği konusunda da akademik çalışmalar literatürdeki yerini almıştır. Örneğin, Yarygina ve Bagge tarafından yapılan çalışmada güvenlik sorunlarının taksonomisi ve mikro servis iletişimini güvence altına alan bir çerçeve sunulmaktadır (Yarygina ve Bagge, 2018). Ayrıca Yu ve ark. tarafından yapılan bir çalışmada mikro servislerin güvenliğinin tesisi konusunda karşılaşılabilecek hususlar ve hususlar kapsamında karşılaşılabilecek sorunları önleme kapsamında bir çözüm önerisi sunulmuştur (Yu ve ark., 2017).

Son olarak mikro servislerin performansı da araştırmacılar için önemli bir konu teşkil etmektedir. Heinrich ve ark., mikro mimarisinin tesisinde mevcut tekniklerin yetersiz olduğuna vurgu yaparak mikro servis mimarisi çalışmalarının performansı ile ilgili daha fazla araştırmaya ihtiyaç duyulduğuna vurgu yapmışlardır (Heinrich ve ark., 2017). Niu ve ark., mikro servis mimarisinde performansı artırmak için bir yük dengeleme algoritması önermiştir (Niu ve ark., 2018). Ayrıca Lloyd ve ark. tarafından yapılan bir çalışmada sunucusuz bir bulut platformunda mikro servislerin performansını etkileyen faktörler incelenmiştir (Lloyd ve ark., 2018). Fan ve ark., mikro servislerde veri tutarlılığının artırılmasına yönelik bir metodoloji geliştirmiştir ((Fan ve ark., 2018).

Knoche ve Hasselbring tarafından yapılan bir çalışmada ise mikro servis mimarisinin benimsenmesine yönelik engeller irdelenmiştir (Knoche ve Hasselbring, 2019).

2.4. Mikro Servis Mimarisi Temelli Sistemlerde Dağıtık SAGA İşlemlerinin (Transactionlarının) Koordinasyonu Konulu Çalışmalar

Mikro servisler tarafından gerçekleştirilen dağıtık işlemlerin (distributed transactions) koordine edilmesi maksadıyla kullanılagelen “*Software Automation, Generation, and Administration- (SAGA)*” adlı tasarım deseni (design pattern) konusunda yaptığımız literatür araştırması bu konuda yapılmış yayınların sayısının oldukça sınırlı düzeyde olduğunu göstermiştir.

Bu konuda yapılmış en yeni çalışmalardan biri Daraghmi ve ark. (2022) tarafından yapılmıştır. Bu çalışmada yazarlar, mikro servislerin veri tabanı üzerinde geri dönüşü olmayan yanlış işlem yapmasını engellemek için işlemleri direk olarak veri tabanında işlemek yerine mikro servislerin belleği üzerinde SYNC edilen bir in-memory veri yapısı üzerinde işlemek suretiyle SAGA akışları arasındaki izolasyon eksikliği sorununu çözebilecek bir transaction yönetim stratejisi ortaya koymuşlardır. Bu stratejide bir SAGA akışında görev alan mikro servisler başarılı olarak gerçekleştirdikleri veri tabanı değişikliklerini gerçek veri tabanı yerine ön-bellek olarak adlandırılan bir in-memory veri yapısı üzerinde işlemektedir. Gerçek veri tabanı üzerindeki commit yalnızca ön-bellekteki tüm işlemler başarıyla tamamlandığında gerçekleştirilmektedir. Bu suretle farklı SAGA akışlarının gerçekleştirdiği veri operasyonlarına bağlı oluşabilecek veri tutarsızlıklarının önüne geçilmektedir. Tez çalışmamız kapsamında geliştirdiğimiz ve bölüm 6’da ortaya koyduğumuz çözüm bu çalışmayla oldukça benzer bir içeriğe sahiptir ancak bizim çözümümüz aktör modelini esas almakta olup bir ön bellek ihtiyacı gerektirmemesi bakımından çok daha verimli olup yönetimi daha kolaydır.

Zuckmental ve ark. (2022), veri odaklı semantik yapılar kullanmak suretiyle mikro servislerin koordinasyonunu sağlamaya çalışmıştır. Bununla birlikte bu çalışma farklı SAGA akışlarının gerçekleştirdiği veri operasyonlarına bağlı oluşabilecek veri tutarsızlıklarını önleyememiştir. Buna karşın tez çalışmamız kapsamında geliştirdiğimiz ve bölüm 6’da açıkladığımız yöntem veri tutarsızlıklarının oluşumunu kati suretle engellemektedir.

Kristianto ve Zahra (2021), SAGA tasarım desenindeki koreografi ve orkestrasyon varyantlarının performans analizini yapmış ve koreografi varyantının

orquestrasyona kıyasla daha iyi bir yanıt süresi, verim (throughput) ve işlemci (CPU) kullanımına sahip olduğu sonucuna varmıştır. Bu çalışma SAGA deseni hakkında detaylı analizler yapmasına rağmen farklı SAGA akışlarının veri okuma/yazma tutarsızlıkları kapsamında bir çözüm ortaya koymamaktadır.

Laigner ve ark. (2021), mikro servis esaslı bir altyapı üzerinde event-driven (olay-esaslı) bir işlem yönetim yaklaşımının gerekli olduğu görüşünü savunmuş ve gelecekteki tasarım ve araştırma çalışmalarının bu alanda ilerleyeceğini dile getirmişlerdir. Bununla birlikte bu çalışma SAGA deseniyle direk olarak ilgili değildir.

Kindson ve Martinek (2020), mikro servisler üzerinde gerçekleşen dağıtık işlemlerin performansının iyileştirmesi üzerine bir çalışma yapmış ve bu kapsamda işlem performansını iyileştirmek için mikro servisler arasında tamponlanmış bir serialization yaklaşımı önermişlerdir. Elde ettikleri sonuçlar, işlem hatası kurtarma süresinde %60'lık bir iyileşme elde edildiğini göstermiştir. Bununla birlikte bu çalışma da direk olarak SAGA deseniyle ilgili değildir.

Malyuga ve ark. (2020), özellikle SAGA desenindeki orkestratör varyantı için iyileştirilmiş zaman ve bellek özelliklerine sahip hataya dayanıklı bir çözüm önermiştir. Bununla birlikte, bu çalışma bölüm 6'da ele aldığımız farklı SAGA akışları arasındaki izolasyon eksikliği konusunu ele almamıştır.

Zhang ve ark. (2019), mikro servisler tarafından oluşturulan işlemlerin yönetimi için deterministik veri tabanı teknolojilerinden ve iyimser eşzamanlılık kontrol protokolünden yararlanan GRIT adlı bir sistem önermişlerdir. Bu çalışma, mikro servis mimarisi tabanlı sistemlerdeki dağıtık işlemlerin yönetimi konusuna SAGA desenine odaklanmadan yaklaşmış olması bakımından tez çalışmamız kapsamında bölüm 6'da sunduğumuz çalışmadan farklıdır.

Rudrabhatla (2018), SAGA tasarım desenini kapsamındaki koreografi ve orkestrasyon varyantlarının performansını karşılaştırmak için nicel bir analiz yapmış ve koreografi deseninin orkestrasyona kıyasla performans açısından çok daha hızlı olduğu sonucuna varmıştır. Bununla birlikte bu çalışma da direk olarak da SAGA deseniyle ilgili değildir.

3. MATERYAL VE METOT

Bu bölümde doktora çalışmamız kapsamında ele aldığımız problem ile bu problemin çözümü için ortaya koyduğumuz hipoteze değinilmektedir. Bu kapsamda ayrıca yaptığımız araştırmaların düzenlenme şekli ile kullanılan bilimsel yöneme de değinilmektedir.

3.1. Problemin Tanımı

Günümüzdeki pek çok bankanın yazılım altyapısı monolit olarak adlandırılan yazılım sistemlerinden oluşmaktadır. Monolit sistemlerin yazılım altyapısı *Bulut Bilişimin* yaygınlaşmasından önceki dönemden kalan eski teknolojiler (Örn: Cobol gibi) kullanılarak geliştirilmiş olup sunum, uygulama ve veri erişim katmanlarının hepsini içine alan yekpare bir yapı olarak inşa edilmiştir (Megargel ve ark., 2020). Bu yekpare yapı tek bir kod tabanı üzerinde geliştirilmiş olup veri yönetimini yine tek bir veri-tabanı sistemi üzerinden sağlamaktadır. Genel itibariyle *şelale (waterfall)* yazılım geliştirme yöntemine göre geliştirilmiş olan bu yekpare yapının, günümüzün bulut-yerlisi (cloud-native) yazılım geliştirme yaklaşımında kullanılan “Kubernetes” ve “Container” gibi donanım bağımsız sanallaştırma teknolojilerine uyum sağlaması ve bu teknolojiler üzerinde yaygınlaştırılması oldukça güçtür. Bunun altında yatan temel neden bu yekpare yapıyı oluşturan yazılım bileşenlerinin kullandığı devasa büyüklükteki ortak kod ve veri tabanını parçalara ayırmadaki güçlüklerdir. Monolitler bu bakımdan günümüzün açık bankacılık standartlarını esas alan modern bankacılık yazılım sistemlerinin ihtiyaçlarını görmekten uzak olup bankalar gibi geniş çaplı yazılım altyapısına sahip kurumlara ciddi bakım maliyetleri çıkarmakta ve yeni teknolojilerin ve açık bankacılık standartlarının adaptasyonunu zorlaştırmaktadır (Fritzsche ve ark., 2019; Gai ve ark., 2018; Kalske ve ark., 2018; Chen ve ark., 2017; Fowler ve Lewis, 2014).

Diğer yandan açık bankacılık ekosisteminde önemli bir yer teşkil eden FinTek’ler ile bankalar arasında PSD2 standartlarında açık bankacılık API’leri üzerinden erişim sağlanması önem arz etmektedir zira bankaların daha çok FinTek girişimiyle entegrasyon kurarak Fintek ekosistemindeki kazanımlarını artırabilmelerinde açık bankacılık API’leri katalizör görevi görecektir (Gozman ve ark., 2018; Premchand ve Choudhry, 2018). Bu bakımdan monolit yapıdaki bankacılık altyapı sistemlerinin PSD2/Açık Bankacılık standartlarında bir bankacılık platformuna dönüşümü FinTek ekosisteminde güçlü bir

yere sahip olmak isteyen her banka için ciddi bir gerekliliktir. Buradan hareketle tez çalışmamız kapsamında çözüm bulmaya çalıştığımız **problem** şu şekilde ifade edilebilir:

“Bir bankanın sahip olduğu monolit yapıdaki temel bankacılık sistemi, Açık bankacılık standartlarına uyumlu bir bankacılık platformuna Mikro Servis Mimarisi temelinde nasıl dönüştürülebilir?”

3.2. Hipotez

Günümüzde pek çok bankanın yazılım altyapısında Bulut Bilişimin yaygınlaşmasından önceki dönemde geliştirilmiş olan ve nispeten eski teknoloji ürünü (Örn: Cobol gibi) monolit yazılım sistemleri kullanılmaktadır (Megargel ve ark., 2020). *Mikro Servis Mimarisi (Micro Services Architecture- MSA)* ortaya koyduğu çözümler bakımından günümüzün modern bulut bilişim platformları üzerinde çevik geliştirme yöntemleri kullanılarak geliştirilip çalıştırılabilecek yazılım sistemlerinin geliştirilmesine imkân tanınması bakımından monolit mimarinin dezavantajlarını ortadan kaldıracabilecek modern bir yazılım mimarisi ortaya koymaktadır (Fowler ve Lewis, 2014).

Diğer yandan PSD2 kapsamındaki Açık Bankacılık Standartlarının adaptasyonunda açık bankacılık API’lerinin bankalarca sağlanabilmesi bankalarla FinTek’lerin uyum içerisinde çalışabilmesini gerektirmektedir zira bu uyum sayesinde Fintekler, Bankalar ve bankacılık müşterileri üçgeninde işleyen Fintek ekosisteminin gelişip büyümesi mümkün olacaktır (Gozman ve ark., 2018; Premchand ve Choudhry, 2018). Bununla birlikte Açık Bankacılık API’lerinin bankalarca geliştirilmesinde bölüm 1.6.3’te belirtilen avantajlarından dolayı GraphQL teknolojisinin kullanımı tercih edilmelidir. Zira bu teknoloji sayesinde daha az API çağırımı ile daha büyük yoğunlukta bankacılık işlevi gerçekleştirilebilir. Bu doğrultuda tez çalışmamız kapsamındaki **hipotezimiz** şu şekilde ifade edilebilir:

“Bankaların monolit yazılım sistemlerinin PSD2/Açık Bankacılık standartlarında bir Bankacılık Platformuna dönüşümünün sağlanmasında Mikro Servis Mimarisini temel alan bir mimari modeli uygulanabilir”.

3.3. Metot

İngilizcede **Case-Study** olarak adlandırılan **Vaka Çalışması** metodu gerçek hayatta karşılaşılan veya karşılaşılmı ihtimali olan problemlerin çözümü için ortaya

konulan bir bilimsel yöntem olarak tanımlanabilir (Noor, 2008). Vaka çalışması metodunun *keşif niteliğinde*, *tanımlayıcı* ve *açıklayıcı* olmak üzere 3 farklı varyasyonu mevcuttur. Bu varyasyonlardan *keşif niteliğindeki vaka çalışmaları*, gerçek dünya ile birebir ilişkili problemlerin formüle edilmesi ve bu problemlere çözüm öneren hipotezlerin test edilmesi amacıyla kullanılabilir. Buna karşın *tanımlayıcı vaka çalışması* olarak adlandırılan vaka çalışması yöntemi genellikle piyasaya sürülen yeni bir ürünün ya da servisin durumu ve akıbeti ile alakalı açıklayıcı bir rapor düzenlenmesi gerektiğinde kullanılabilir. Diğer yandan *açıklayıcı vaka çalışması* olarak adlandırılabilen vaka çalışması türü ise organizasyonlardaki iş süreçlerinin (business processes) açıklanması amacıyla kullanılabilir (Yin, 1993).

Hipotezimiz kapsamında ortaya koyacağımız mimari model ile onu doğrulayacak nihai sisteme olan ihtiyaç gerçek endüstriyel ihtiyaçlardan doğduğu için doktora çalışmamızda *vaka çalışması yönteminin* kullanılması uygun görülmüştür. Bununla birlikte doktora çalışmamız kapsamındaki amacımız, gerçek endüstriyel bir ihtiyaçtan doğan bir problemin formülasyonu ile bu problemin çözümü kapsamındaki hipotezimizin doğrulanması şeklinde değerlendirilebileceğinden doktora çalışmamız kapsamında uyguladığımız yöntem özellikle *keşif niteliğinde bir vaka çalışması* olarak nitelendirilmelidir.

Buradan hareketle hipotezimizde ortaya koyduğumuz mimari modelin doğruluğu Kuveyt Türk Katılım Bankası ile TÜBİTAK ve Selçuk Üniversitesi Bilgisayar Mühendisliği Bölümü arasında imza olunan bir TÜBİTAK 1505 üniversite-sanayi iş birliği projesi kapsamında Kuveyt Türk’ün sağladığı kaynaklar kullanılarak uygulamalı bir vaka çalışması kapsamında gösterilmiştir. Bu vaka çalışmasında Kuveyt Türk Bankası bünyesinde halihazırda kullanılmakta olan monolit yapıdaki **“Business Oriented Architecture (BOA)”** adlı temel bankacılık yazılım altyapısı, **mikro servis mimarisi** esaslı bir açık bankacılık yazılım altyapısına evrilmiştir. Bu doğrultuda uygulamalı olarak ortaya koyduğumuz mikro servis mimarisi tabanlı yazılım platformu **Açık Bankacılık Mimarisi (Open Banking Architecture- OBA)** olarak adlandırılmış olup bu mimarinin tasarımında ve geliştirilmesinde takip etmiş olduğumuz iş kelimeleri ana hatlarıyla aşağıdaki gibidir:

- 1) OBA tarafından dış kullanıma açılacak olan bankacılık servislerinin açık bankacılık standartlarında bankacılık işlev sınıflarına ayrıştırılmak suretiyle tespit edilmesi ve servisler bünyesinde sağlanacak Application Programming Interface’lerin (API’lerin) tespit edilmesi (bkz. Bölüm 4).

- 2) OBA'ya evirilecek olan monolit BOA mimarisinin incelenmesi ve bu mimarideki sorunların ortaya konması (bkz. Bölüm 5).

Bu ikinci kalem kapsamında şu konular ele alınmıştır:

- a) Monolit Mimari nedir? Dezavantajları nelerdir?
- b) BOA monoliti nedir? Yapısı nasıldır?
- c) Dağıtık BOA Monolitleri nedir ve nasıl işlerler?
- d) BOA'da kullanılagelen yaygınlaştırma (deployment) stratejileri nelerdir?

- 3) OBA'nın mikro servis mimarisi çerçevesinde genel mimari unsurlarının ve temel altyapı bileşenlerinin (yani OBA bileşenlerinin) tespit edilip bu bileşenlerin sorumluluklarının ve kendi aralarındaki etkileşimlerinin tasarlanması ve bu tasarımın vaka çalışması kapsamında uygulanması (bkz. Bölüm 5). Bu üçüncü kalem kapsamında şu konular ele alınmıştır:

- a) Açık Bankacılığın ortaya çıkış süreci, genel felsefesi ve tanımı
- b) İstemci tiplerinin (yani son kullanıcı tiplerinin) sınıflandırması
- c) Temel altyapı bileşenleri ve aralarındaki etkileşimler
- d) Yaygınlaştırma stratejisi
- e) OBA mimarisine kuş bakışı
- f) Diğer OBA bileşenleri
- g) OBA'da jurnal (API istek/cevap (request-response)) içeriklerinin tutulmasında kullanılan mimari

- 4) OBA'da mikro servisler arası dağıtık SAGA işlemlerinin (Distributed SAGA Transactions) yönetilmesi kapsamında ortaya konulan buluşun açıklaması (bkz. Bölüm 6).

Bu dördüncü kalem kapsamında şu konular ele alınmıştır:

- a) SAGA tasarım desenin güncel durumu
- b) Tezimiz kapsamında ortaya konulan aktör modeli esaslı dağıtık SAGA işlem yönetiminin bilinen SAGA varyantlarına göre getirdiği kazanımların izahı.
- c) Tezimiz kapsamında ortaya konulan aktör modeli esaslı dağıtık SAGA işlem yönetiminin mimari yapısının ve çalışma mantığının detaylı izahı.
- d) Aktör modeli esaslı dağıtık SAGA işlem yönetiminin sanayiye uygulanma biçimi

Yukarıdaki 4 iş kaleminden birincisi Bölüm 4’te; ikincisi ve üçüncüsü Bölüm 5’te; dördüncüsü ise Bölüm 6’da ele alınmıştır.

Bu tez dokümanının geri kalanında aksi belirtilmedikçe “Banka” kelimesi ile kastedilmek istenen Kuveyt Türk Katılım Bankasıdır.



4. BANKACILIK İŞLEV SINIFLARI

Bankacılık işlevleri, bir bankanın şube, bankamatik, internet ve mobil gibi hizmet kanalları üzerinden müşterilerine sağladığı bankacılık hizmetleri olarak ifade edilebilir. Bu bölümde doktora çalışmamız kapsamında uygulamalı olarak ortaya koyduğumuz mikro servis mimarisi tabanlı yazılım platformu olan ve **Açık Bankacılık Mimarisi (Open Banking Architecture- OBA)** olarak adlandırılan yapı kapsamında sağlanacak bankacılık servislerinin açık bankacılık standartlarını yansıtacak şekilde işlev sınıflarına ayrıştırılması gerçekleştirilmiştir. Bu sınıflandırma kapsamında ortaya konulan bankacılık işlevlerinin her biri bir veya daha fazla Application Programming Interface (API) olarak tasavvur edilebilecek olup aşağıdaki gibi sınıflandırılabilir:

- Hesap ve Mevduat İşlevleri Sınıfı
- Para Transferi İşlevleri Sınıfı
- Kredi/Finansman İşlevleri Sınıfı
- Kredi Kartı İşlevleri Sınıfı
- Ödeme İşlevleri Sınıfı
- Genel Bankacılık İşlevleri Sınıfı

Açık Bankacılık/Payment Services Directive2 (PSD2) standartlarındaki bankacılık işlev sınıfları ile bu sınıflar altında gruplanan işlevler aşağıda Tablo 4.1 ve Tablo 4.2’de görüldüğü gibi sınıflandırılabilir.

Tablo 4.1. Bankacılık İşlev Sınıfları (a)

Hesap ve Mevduat İşlevleri	Para Transferi İşlevleri	Kredi/Finansman İşlevleri
Hesap Bilgileri	Param Birimleri Listesi	Kar Payı Hesaplama
Hesap Hareketleri	IBAN’a Para Transferi	Finansman Hesaplama
Hesap Bakiyesi	Hesaba Para Transferi	Kredi Vade Oranları Listesi
	Döviz Alış	Kredi/Finansman Listesi
	Döviz Satış	Kredi/Finansman Bilgileri
	Döviz Alım-Satım Hareketleri Listesi	Kredi/Finansman Taksitleri
	Döviz Kur Listesi	Fon İsteği
	Kıymetli Maden Alış	
	Kıymetli Maden Satış	
	Kıymetli Maden Kur Listesi	
	Para Transfer Detayları (Dekontlar)	

Tablo 4.2. Bankacılık İşlev Sınıfları (b)

Kredi Kartı İşlevleri	Ödeme İşlevleri	Genel Bankacılık İşlevleri
Kart Detayları	Yurt İçi Ödeme Yapma	ATM Lokasyonları
Kart Listesi	Uluslararası Ödeme Yapma	Şube Lokasyonları
Kart Taksitleri Listeleme	Planlı Yurt içi Ödemeler Listesi	Bağış Listesi
Bekleyen Provizyonların Listelenmesi	Uluslararası Ödemeler Listesi	VIP SMS Gönderme
Sanal Kart Oluşturma	Planlı Uluslararası Ödemeler Listesi	Toplu SMS Gönderme
Sanal Kart Silme		PUSH Bildirimi Gönderme
Sanal Kart Bilgisi		Bankacılık Ürün Bilgileri
Sanal Kart Listesi		
Sanal Kart Limit Güncellemesi		

Aşağıdaki alt başlıklarda Tablo 4.1 ve 4.2’de gruplanan bankacılık işlev sınıfları ile bu sınıflar özelindeki işlevler kapsamında yer alan aktörlerin etkileşimlerine değinilmektedir.

4.1. Hesap ve Mevduat İşlevleri Sınıfı

Bu sınıf, HHS’de hesabı bulunan ve erişim rızası alınmış bir veya daha fazla kişinin hesapları ile ilgili bakiye, işlem hareketleri gibi bilgilere erişimi sağlayan işlevleri kapsar. Bu işlevler şunlardan oluşabilir:

- Hesap Bilgileri
- Hesap Hareketleri
- Hesap Bakiyesi

Hesap ve mevduat işlevleri özelinde genel olarak aşağıda listelenen 4 unsur ya da aktörden söz edilebilir:

- Ödeme Hizmeti Kullanıcısı:** Bankacılık müşterisini ifade eder.
- Hesap Bilgisi Hizmet Sağlayıcı:** Hesap bilgisini sunan tüzel kişiyi (Örnek: Papara, Vomsis gibi FinTek firmaları) ifade eder.
- Hesap Hizmeti Sağlayıcısı Yetkilendirme Servisi:** Türkiye Cumhuriyet Merkez Bankası (TCMB) ya da Bankacılık Düzenleme ve Denetleme Kurumu (BDDK) gibi resmi bir otoriteyi temsil eder.
- Hesap Hizmet Sağlayıcı Kaynak Sunucusu:** Hesap hizmetleri sunan API sunucu bilgisayarını temsil eder.

Yukarıda listelenen aktörlerin etkileşimini örnekleyen sıralama (sequence) diyagramı aşağıda Şekil 4.1, 4.2 ve 4.3'te görülmektedir. Bu diyagramlarda sırasıyla hesap bilgisi temini, hesap hareketleri, hesap bakiyesi sorgulama işlevleri ile ilgili aktörlerin etkileşimleri ifade edilmektedir.



Şekil 4.1. Hesap Bilgisi Edinme İşlevi Etkileşimleri

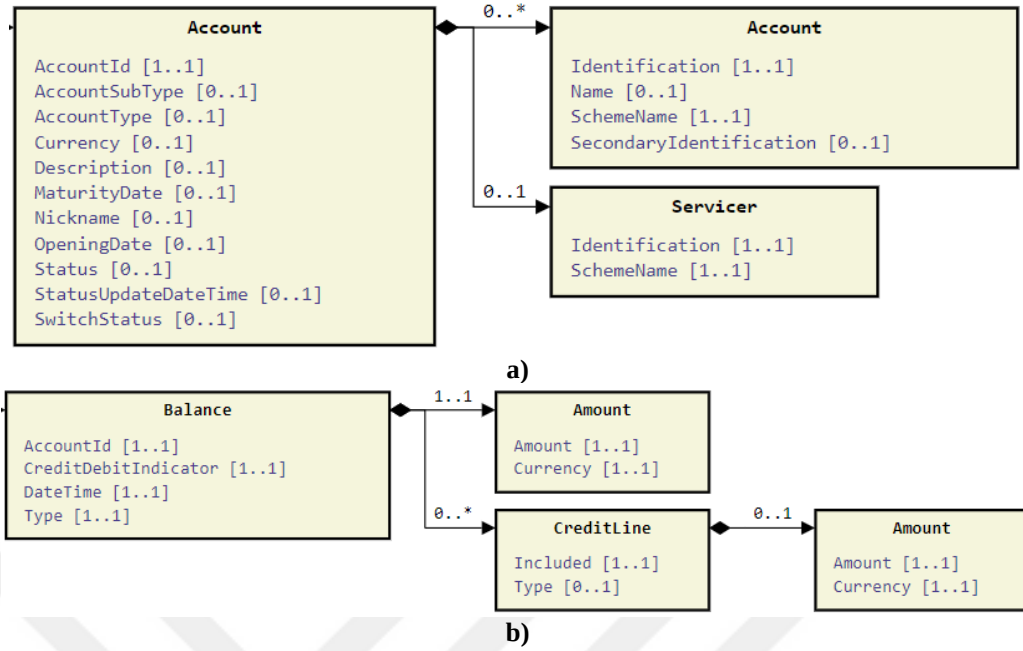


Şekil 4.2. Hesap Hareketleri Sorgulama Etkileşimleri



Şekil 4.3. Hesap Bakiyesi Sorgulama Etkileşimleri

Hesap ve Mevduat işlevleri kapsamında yukarıdaki sıralama diyagramları ile temsil edilen bankacılık etkileşimlerinin UK Açık Bankacılık standartlarına göre ele alınması gereken domain nesnelerini belirten sınıf diyagramları (class diagrams) Şekil 4.4'te görülmektedir.



Şekil 4.4. a) Hesap Etkileşimleri ile ilgili Domain Nesnelerini Gösteren UML Class Diyagramı
b) Bakiye Etkileşimleri ile ilgili Domain Nesnelerini Gösteren UML Class Diyagramı

4.2. Para Transfer İşlevleri Sınıfı

Bu sınıftaki bankacılık işlevleri, bir Hesap Hizmetleri Sağlayıcı (HHS) nezdinde (Örnek: Papara, Trendyol gibi bir FinTek/TPP bir HHS'dir) ödeme hesabı bulunan bir bankacılık müşterisi adına onun rızası dahilinde belirli bir IBAN veya hesap numarasına havale ve EFT marifetiyle para transferi gerçekleştirmeyi sağlayan işlevlerdir. Burada Papara, Vomsis gibi bir Fintek (Third Party Provider (TPP) – Üçüncü Taraf Sağlayıcı) bir HHS olarak tasavvur edilebilir. Dolayısıyla Banka aynı zamanda bir HHS kullanıcısı olan müşterisi için HHS üzerinden para transferi işlevlerini sağlayan işlevler sunar. Bu işlevlere örnek olarak şunlar verilebilir:

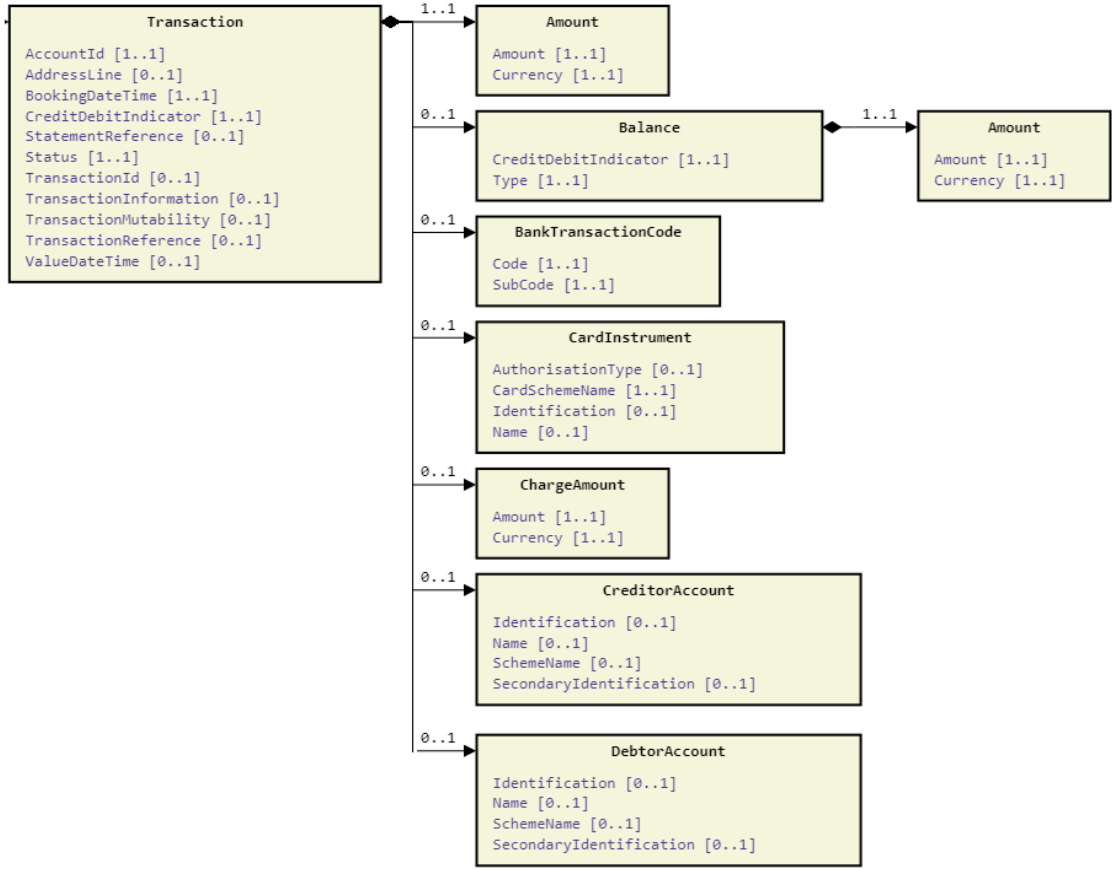
- Para Birimleri Listesi
- IBAN'a Para Transferi
- Hesaba Para Transferi
- Döviz Alış
- Döviz Satış
- Döviz Alım-Satım Hareketleri Listesi
- Döviz Kur listesi
- Para Transferi Detayları (Dekontlar)

- Kıymetli Maden Alış
- Kıymetli Maden Satış
- Kıymetli Maden Kur Listesi

Şekil 4.5'te para transferi işlevlerine örnek olması amacıyla IBAN'a para transferi ile ilgili aktörlerin etkileşimlerini temsil eden bir sıralama diyagramı görülmektedir. Ayrıca transfer etkileşimlerin UK Açık Bankacılık standartlarına göre ele alınması gereken domain nesnelerini içeren class diagramı Şekil 4.6'da görülmektedir.



Şekil 4.5. Para Transferi Sırasında Meydana Gelebilecek Etkileşimler



Şekil 4.6. Para Transferi Domain Nesnelerini Temsil eden Class Diyagramı

4.3. Kredi/Finansman İşlevleri Sınıfı

Bu sınıf, HHS'de hesabı bulunan bir bankacılık müşterisi için kâr payı hesaplama veya finansman hesaplama gibi işlevleri kapsar. Bu işlevlere örnek olarak şunlar verilebilir:

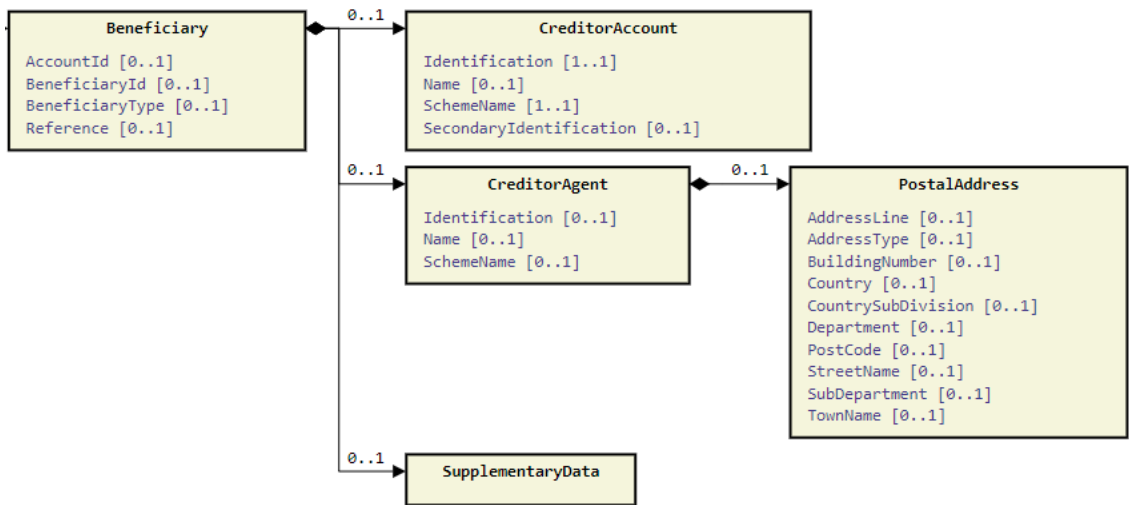
- Kar Payı Hesaplama
- Finansman Hesaplama
- Kredi Vade Oranları Listesi Çıkarma
- Kredi/Finansman Listesi Çıkarma
- Kredi/Finansman Bilgileri Çıkarma
- Kredi/Finansman Taksitleri ile İlgili Bilgi Sağlama
- Fon Ücretleri Listesi Sağlama
- Fon Onaylama
- Fon İsteği

Şekil 4.7’de kredi/finansman işlevlerine örnek olması amacıyla fon isteği işlevi ile ilgili aktörlerin etkileşimlerini temsil eden bir sıralama diyagramı görülmektedir.



Şekil 4.7. Fon Talebi ile İlişkili Aktör Etkileşimleri

Kredi-finance işlemleri kapsamında UK Açık Bankacılık standartlarına göre ele alınması gereken domain nesnelerini içeren sınıf diyagramı (class diagram) Şekil 4.8’de görülmektedir.



Şekil 4.8. Kredi/Finansman Domain Nesnelerini Temsil eden Class Diyagramı

4.4. Kredi Kartı İşlevleri Sınıfı

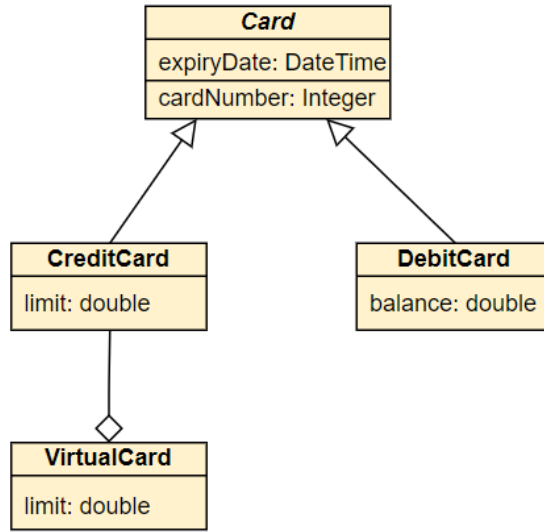
Bu sınıf kapsamındaki işlevler, HHS'de hesabı bulunan bir bankacılık müşterisi için kredi kartı bilgilerine veya kredi kartı hareketlerine erişim sağlayan bankacılık işlevleridir. Bu işlevlere örnek olarak şunlar verilebilir:

- Kart Detay Bilgisi Talebi
- Kart Listesi
- Kart Taksitleri
- Bekleyen Provizyonlar
- Sanal Kart Oluşturma
- Sanal Kart Silme
- Sanal Kart Listesi ve Bilgisi
- Sanal Kart Limit Güncellemesi

Şekil 4.9'da kredi kartı işlevlerine örnek olması amacıyla Kart Detay Bilgisi işlevi ile ilişkili etkileşimleri temsil eden bir sıralama diyagramı görülmektedir. Ayrıca Şekil 4.10'da kredi domain modelimiz görülmektedir.



Şekil 4.9. Kart Detay Bilgisi ile İlişkili Aktör Etkileşimleri



Şekil 4.10. Para Transferi Domain Nesnelerini Temsil eden Class Diyagramı

4.5. Ödeme İşlevleri Sınıfı

Bu sınıftaki işlevler, HHS'de ödeme hesabı bulunan ve erişim rızası alınmış bir bankacılık müşterisinin adına ödeme işlemi gerçekleştirmeyi sağlayan işlevlerdir. Bu işlevlere örnek olarak şunlar verilebilir:

- Yurt içi Ödeme Yapma
- Uluslararası Ödeme Yapma
- Yurt İçi Ödemeler Listesi
- Planlı Yurt İçi Ödemeler Listesi
- Uluslararası Ödemeler Listesi
- Planlı Uluslararası Ödemeler Listesi

Şekil 4.11’de ve 4.12’de ödeme işlevlerine örnek olması adına sağlanmış olan ve ödeme başlatma ve planlı ödeme bilgi isteği işlevleriyle ilgili etkileşimleri temsil eden sıralama diyagramları görülmektedir.



Şekil 4.11. Ödeme Başlatma ile İlişkili Aktör Etkileşimleri



Şekil 4.12. Planlı Ödeme Bilgi İsteği İşlevi ile İlişkili Aktör Etkileşimleri

4.6. Genel Bankacılık İşlevleri Sınıfı

Bu sınıftaki işlevler, ATM cihazlarının veya banka şubelerinin bulunduğu yerlerin listesini sağlayan bilgilendirme amaçlı işlevlerdir. Bu işlevlere örnek olarak şunlar verilebilir:

- Bankacılık Ürün Bilgileri
- ATM Konumları
- Şube Konumları
- Bağış Listesi
- VIP SMS Gönderme
- Toplu SMS Gönderme
- PUSH Bildirimi Gönderme

Şekil 4.13'te genel bankacılık işlevlerine örnek olması adına sağlanmış olan ve bankacılık ürün bilgileri işlevi ile ilgili aktörlerin etkileşimlerini temsil eden bir sıralama diyagramı görülmektedir.



Şekil 4.13. Bankacılık Ürün Bilgileri Listesini Getirmek için Meydana Gelen Etkileşimler

4.7. Bölüm Özeti

Bu bölümde doktora çalışmamız kapsamında uygulamalı olarak ortaya koyduğumuz mikro servis mimarisi tabanlı Açık Bankacılık Mimarisi (Open Banking Architecture- OBA) bünyesinde sağlanan bankacılık servislerinin açık bankacılık standartlarını yansıtacak bir şekilde işlev sınıflarına ayrıştırılması sağlanmış ve bu suretle bir bankanın dijital hizmetlerini açık bankacılık standartlarında nasıl organize edebileceği ortaya konmuştur. Bu sınıflandırma kapsamında ortaya konulan bankacılık işlevlerinin her biri sahada açık bankacılık standartlarına uyumlu bir veya daha fazla Application Programming Interface (API) olarak hizmete açılacaktır.



5. TASARIM VE GELİŞTİRME ÇALIŞMALARI

Bu bölümde, doktora tez çalışmasının hipotezinde ifade olunan Mikro Servis Mimarisi temelli mimari modelin tasarımı ve uygulaması ile ilgili bilgiler sunulmaktadır. Söz konusu modelin sahada uygulanarak doğrulanması amacıyla Kuveyt Türk bünyesinde halihazırda kullanılmakta olan monolit yapıdaki “**Business Oriented Architecture (BOA)**” adlı bankacılık yazılım altyapısının mikro servis mimarisi esaslı bir açık bankacılık yazılım altyapısına evrimi için bir vaka çalışması gerçekleştirilmiştir. Bu çalışma vesilesiyle ortaya koyduğumuz mikro servis mimarisi temelindeki açık bankacılık yazılım altyapısı **Açık Bankacılık Mimarisi (Open Banking Architecture-OBA)** olarak adlandırılmakta olup bu bölümde bu altyapı hakkında şu konular ele alınmaktadır:

- i. OBA’ya evrilecek olan monolit BOA mimarisinin incelenmesi ve bu mimarideki sorunların ortaya konması.

Bu kapsamda şu konular ele alınmıştır:

- a) Monolit Mimari nedir? Dezavantajları nelerdir?
- b) BOA monoliti nedir? Yapısı nasıldır?
- c) Dağıtık BOA Monolitleri nedir ve nasıl işlerler?
- d) BOA’da kullanılagelen yaygınlaştırma (deployment) stratejileri nelerdir? Dezavantajları nelerdir?

- ii. OBA’nın mikro servis mimarisi çerçevesinde genel mimari unsurlarının ve temel altyapı bileşenlerinin (yani OBA bileşenlerinin) tespit edilip bu bileşenlerin sorumluluklarının ve kendi aralarındaki etkileşimlerinin tasarımı.

Bu kapsamda şu konular ele alınmıştır:

- a) Açık Bankacılığın ortaya çıkış süreci, genel felsefesi ve tanımı.
- b) İstemci tiplerinin (yani son kullanıcı tiplerinin) sınıflandırması.
- c) Temel OBA altyapı bileşenleri ve aralarındaki etkileşimlerin modellenmesi.
- d) OBA altyapı bileşenlerinin yaygınlaştırma stratejisinin tespit edilmesi.
- e) Genel OBA mimarisinin ortaya konması.
- f) Diğer OBA bileşenlerinin işlevlerinin ve kullanımlarının modellenmesi.

- g) OBA’da jurnal (API istek/cevap (request-response)) içeriklerinin tutulmasında kullanılan mimarinin modellenmesi.
- iii. OBA bileşenlerinin bir araya getirilerek uçtan uca test edildiği PoC çalışmaları.
- iv. Yük Testi Çalışmaları

5.1. Monolit Mimari Nedir?

Yazılım mühendisliğinde yaygın olarak karşılaşılan tasarım problemlerine çözüm getirmek amacıyla ortaya konulan mimari yaklaşımlar mevcuttur. Bu yaklaşımlardan en geleneksel olan Monolit Yazılım Mimarisinde (Monolithic Architecture) bir yazılım sistemini oluşturan tüm bileşenler Şekil 5.1’de temsili olarak gösterilen yap-boz benzeri bir yapıda yekpare bir yazılım sistemini oluşturmak üzere kenetlenerek bir araya getirilirler.



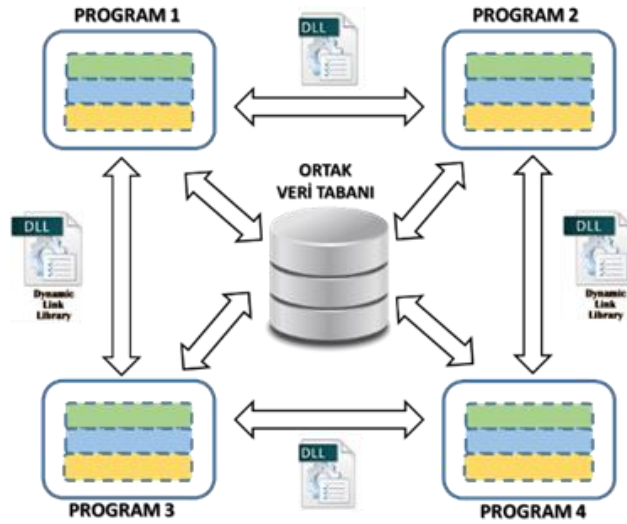
Şekil 5.1. Monolit Mimarinin Temsili Görünümü

Monolit mimaride oluşturulan bir yazılım sistemi tek tip bir geliştirme teknolojisini barındıran tek bir kod tabanı üzerinde ve yine tek bir veri tabanını kullanacak şekilde yekpare bir yapı olarak geliştirilir. Daha teknik bir ifadeyle monolit mimari, çok katmanlı yazılım mimarisinde mevcut bulunan sunum, uygulama ve veri erişim katmanlarının tamamının aşağıda Şekil 5.2’de görüldüğü tek bir yekpare katmanda toplayan bir mimariyi ortaya koyar (Villamizar ve ark., 2015; Fowler ve Lewis, 2014).



Şekil 5.2. Yekpare bir Monoliti oluşturan yazılım katmanları

Monolit bir yazılım sistemi, sunum, uygulama ve veri erişim katmanlarının tamamını gibi tek bir bilgisayar programı içinde barındırabileceği gibi bir bilgisayar ağı üzerinde birden fazla bilgisayar programını barındıracak şekilde bir yapı da arz edebilir. Dağıtık monolit (distributed monolith) olarak adlandırılabilir olan bu ikinci durum aşağıda Şekil 5.3'te temsili olarak görülmektedir. Bu Şekilden anlaşılabilir olduğu üzere monolit bir yazılımı oluşturan programlar (bir başka deyişle bileşenler) birbirlerine verdikleri kütüphane referansları (yani Dynamically Linked Library-DLL) bakımından ve veri tabanı üzerinde ortaklaşa kullandıkları saklı yordamlar (yani stored procedure'ler) ve ilişkili tablolar bakımından sıkı sıkıya bağımlıdırlar (yani tightly-coupled yapıdadırlar). Bu bakımdan monolit yazılım altyapısı büyüdükçe sistemin genel karmaşıklığını yönetmek zorlaşmaktadır. Ayrıca monolit altyapı bileşenlerinin genel olarak tek tip bir teknoloji ile geliştirilmiş olması yeni teknolojilere adaptasyonu uzun vadede güçleştirmektedir (Chen ve ark., 2017; Richardson, 2019).



Şekil 5.3. Dağıtık Monolitlerden Oluşan bir Yazılım Sisteminin Kütüphane (DLL) ve Veri Tabanı Bakımından Bağımlılıkları

Diğer yandan monolit bir yapı üzerinde yapılacak bir güncelleştirme veya kapasitesinin artırımı (yani ölçeklenme–scale edilme) nispeten uzun süren yaygınlaştırma süreçlerine neden olabilmektedir. Örneğin aylık gerçekleşen ve 2-3 saat sürebilen genel yaygınlaştırma süreçleri yaşanabilmektedir.

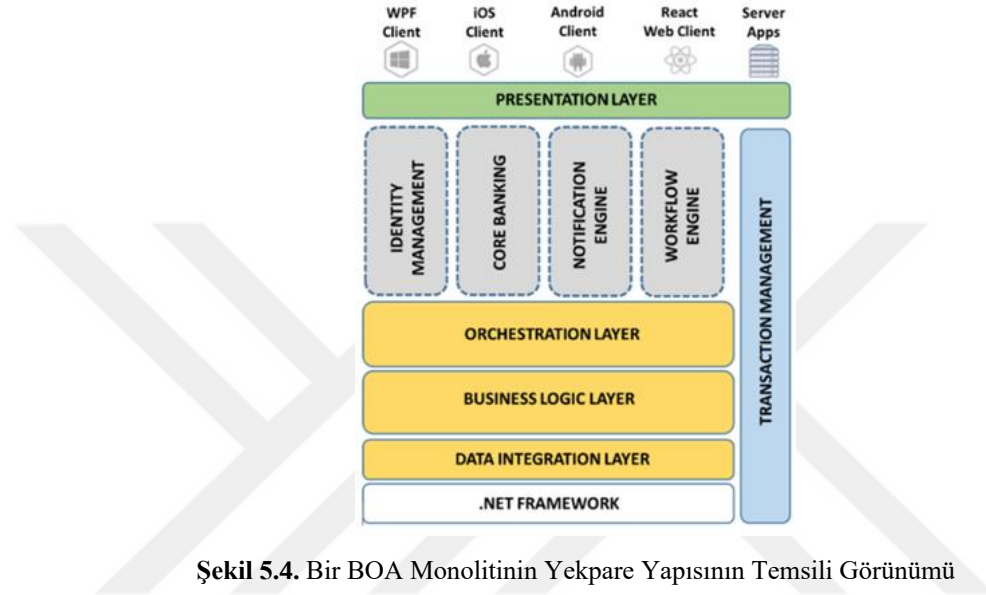
5.2. Monolit BOA Mimarisine Genel Bakış

Kuveyt Türk Katılım Bankası bünyesinde, Bankacılık Odaklı Mimari (Business Oriented Architecture-BOA) olarak adlandırılan monolit bankacılık sistemi 2010 yılından beri Kuveyt Türk Bünyesinde kullanılan çok kanallı bir bankacılık sistemidir.

5.2.1. Bir BOA monolitinin anatomisi

BOA bünyesindeki bankacılık servislerinin uygulama mantığı genel olarak monolit yazılım bileşenleri tarafından gerçekleştirilmektedir. Temelinde Model-View-Controller (MVC) tasarım desenini esas alarak geliştirilen bu bileşenlerin istemci tarafındaki önyüz uygulamalarında WPF/React/Swift/Java gibi teknolojiler kullanılırken bankacılık işlevlerini yerine getiren asli altyapısal yazılım bileşenleri C# dilinde geliştirilmiş olup uygulama sunucuları (application servers- AppServers) üzerinde koşarlar. Bu bağlamda Şekil 5.4'te bir BOA monolitinin yekpare yapısı temsili olarak görülmektedir. Buradan anlaşılacağı üzere presentation katmanında yer alan istemci

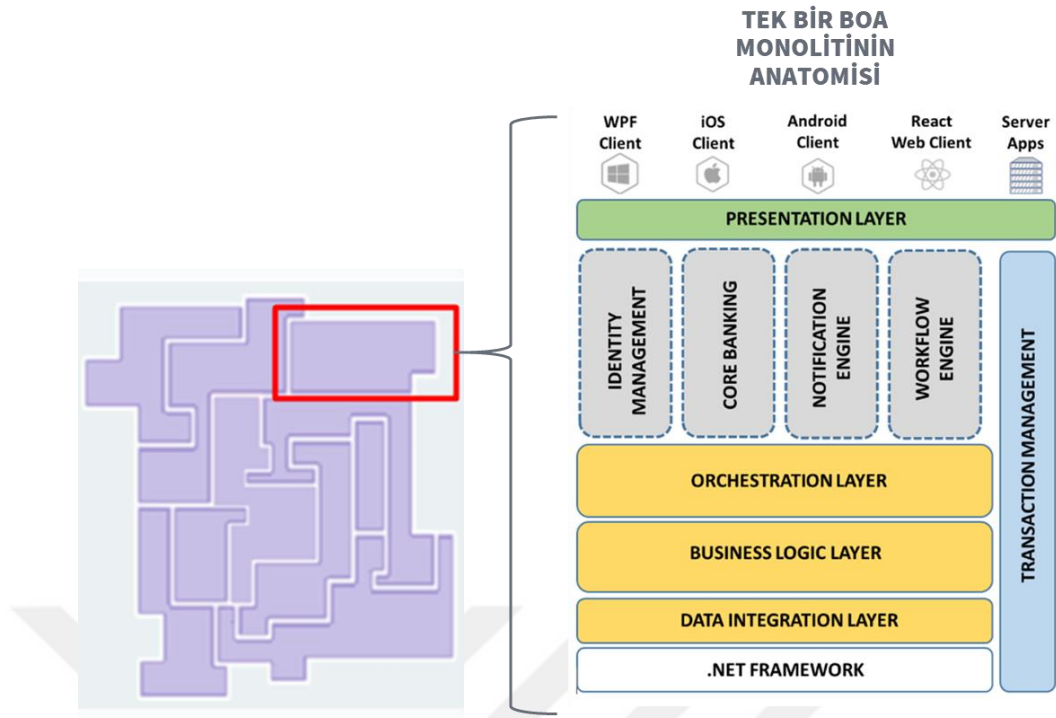
(client) uygulamaları MVC deseni esasları doğrultusunda sırasıyla orchestration, business ve data integration katmanlarında konumlandırılan bankacılık yazılımlarını Remote Procedure Call (RPC) network çağırımları yapmak suretiyle işletirler. Buna ek olarak düzeyde uçtan uca çalışan bilgilendirme motoru (notification engine), iş akışı motoru (workflow engine) ve transaction management gibi yardımcı uygulamalar/kütüphaneler de uçtan uca bütünlüğü sağlayıcı özellikler mevcuttur.



Şekil 5.4. Bir BOA Monolitinin Yekpare Yapısının Temsili Görünümü

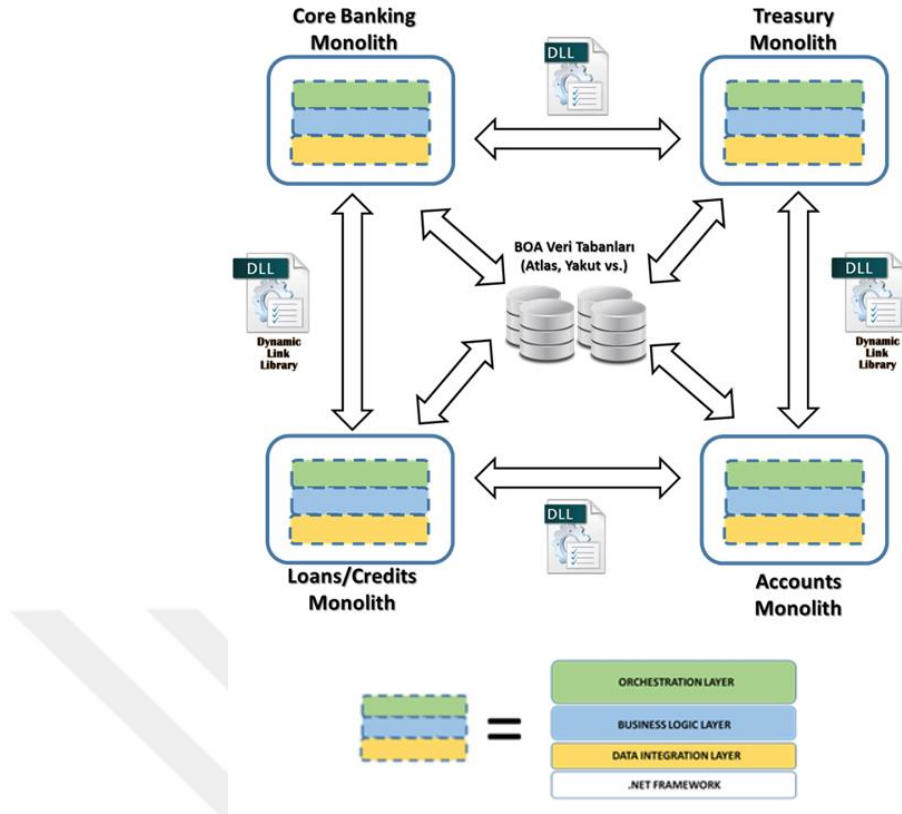
5.2.2. Dağıtık BOA monolitleri

BOA altyapısı kendi içinde monolitlerden oluşan bir dağıtık sistemdir (distributed system). Şekil 5.1’de görüldüğü gibi yap boz benzeri bir yapı arz eden bu sistemdeki her bir parça esasen bir BOA monoliti olup bu durum Şekil 5.5’te tasvir edilmektedir. Burada görüldüğü gibi genel yap-boz benzeri BOA sisteminde kırmızı çerçeve ile vurgulanan tek parça gibi parçaların her biri MVC desenine göre tasarlanmış bir BOA monolittir.



Şekil 5.5. Tek bir BOA Monolitinin Genel BOA Sistemindeki Tek bir Yap-Boz Parçasına Benzetimi

BOA monolitleri, kullandıkları kütüphane referansları (yani DLL'ler) ile veri tabanı üzerinde ortaklaşa kullandıkları saklı yordamlar (yani stored procedure'ler) ve bu yordamların ilişkili olduğu tablolar bakımından birbirlerine sıkı sıkıya bağımlıdırlar (yani tightly-coupled yapıdadırlar). Bu durum aşağıda Şekil 5.6'da temsili olarak görülmektedir. Örneğin ana bankacılık işlevlerinden sorumlu monolit (core banking monolith), hazine monoliti (treasury monolith) üzerinden faydalanacağı işlevleri kullanabilmek için hazine monoliti tarafından sağlanan process DLL'lerini kullanmaktadır. Bu DLL'ler sayesinde hazinenin business katmanı erişim mantığını kendi uygulama katmanına katmakta ve ilgili veri tabanlarına erişmektedir.

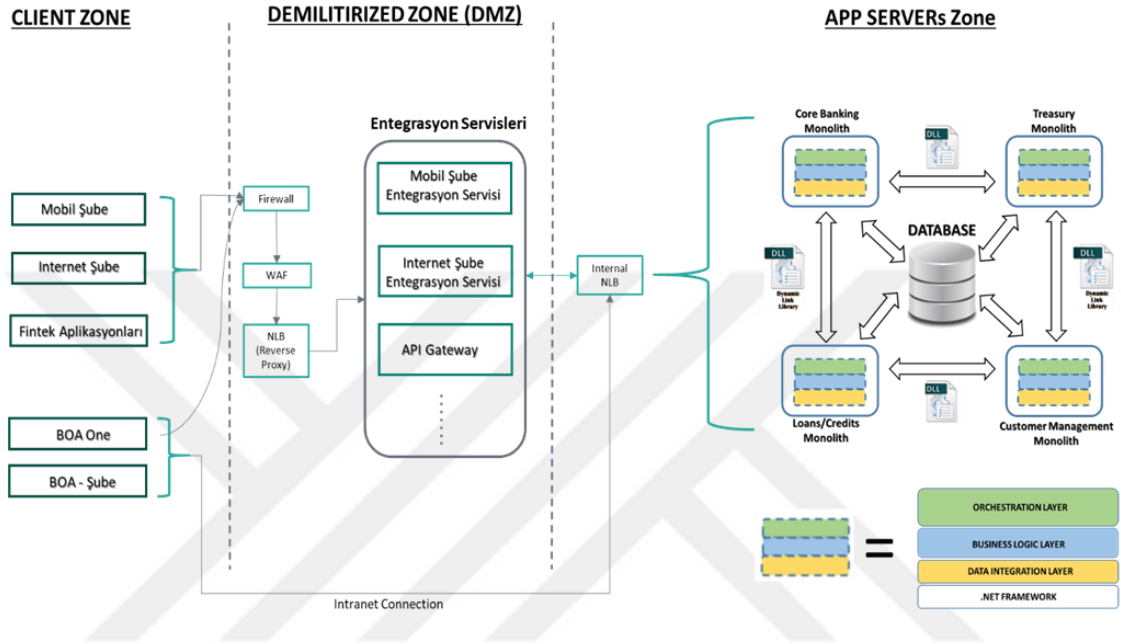


Şekil 5.6. BOA Monolitlerinin Dağıtık bir Sistem Olarak Kenetlenmiş Temsili Görünümü

Distributed BOA altyapısı her ne kadar güçlü bir bankacılık altyapısı sunsa da monolit yapısından kaynaklı bazı dezavantajlara sahiptir. Örnek olarak sistemdeki genel karmaşıklığın (application complexity) yönetimindeki zorluk, yeni teknolojilerin adaptasyonundaki zorluklar, çevik (agile) geliştirme yöntemlerini esas alan ekip çalışmasının kazanımındaki güçlükler, sistem kapasitesinin yatayda ölçeklenmesindeki zorluk ile buna bağlı olarak artan sistemin geneline binen yükü kaldırmada ki zorluklar verilebilir.

Dağıtık BOA monolitleri network üzerinde yaygınlaştırıldığında Şekil 5.7’de görüldüğü gibi ayrı network bölümleri (network zones) üzerinde koşarlar. Burada en solda “client zone” bölümünde public internet üzerinden erişim sağlanan mobil şube, internet şube ve FinTek uygulamaları gibi istemciler görülmektedir. Bu uygulamalardan iş mantıkları kapsamında alınan network çağırımları (yani BOA requestleri) DMZ bölümünde bulunan firewall, Windows Application Firewall (WAF) ve Network Load Balancer (NLB) üzerinden geçtikten sonra internal NLB marifetiyle “App Servers Zone” olarak isimlendirilen ve BOA monolitlerinin koştuğu uygulama sunucularını barındıran bölüme yönlendirilirler. Burada DMZ bölümünde görülen “Mobil Şube Entegrasyon Servisi”, “İnternet Şube Entegrasyon Servisi” ve “API Gateway” gibi yazılım bileşenleri

aslında kendi içinde birer monolit olup kurumun gerek iç gerekse dış networklerde bulunan istemcilerle uygulama sunucularında koşan altyapı monolitleri arasındaki iletişimi ve dolayısıyla entegrasyonu sağlamaktadırlar. Bu nedenle DMZ bölümündeki servisler entegrasyon monolitleri olarak adlandırılırken; “APP Servers Zone” bölümündeki monolitler de altyapı monolitleri olarak adlandırılmaktadır.



Şekil 5.7. BOA Monolitlerinin Dedike Network Bölümlerinde Koşumu

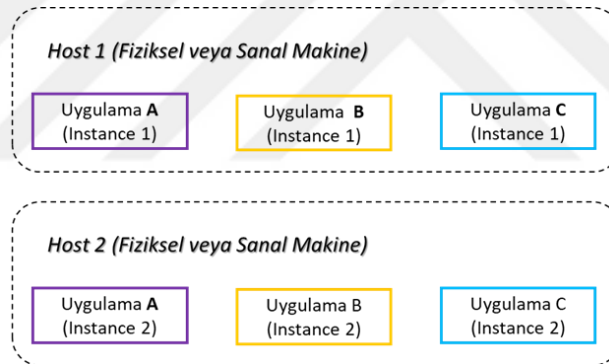
Şekil 5.7’de ayrıca “client zone” bölümünde bankanın lokal ağında koşan BOAOne istemcileri ile WPF istemcileri de görülmektedir. Bu istemciler genel müdürlük personeline kullanılır ve idari işlerin yürütüldüğü ekranları temsil etmektedirler. Bu iki istemci grubu bankanın lokal ağında (yani intranet) çalıştığı için network çağırımlarını direk olarak “APP Servers Zone” bölümüne içerdeki internal load balancer üzerinden yapabilmektedirler.

5.2.3. Dağıtık BOA monolitlerini yaygınlaştırma stratejileri

OBA mimarisinin kazanımlarını ve farklarını ilerleyen bölümlerde doğru izah edebilmek amacıyla bu alt bölümde BOA’da kullanılagelen deployment (yaygınlaştırma) stratejileri açıklanmaktadır.

5.2.3.1. Her bir host'ta birden çok uygulama barındırma yaygınlaştırma stratejisi

BOA kapsamında uygulanan ilk deployment stratejisi literatürde “Multiple Application Instances per Host” olarak ifade edilmektedir. Şekil 5.8’de görülen bu stratejide yazılım uygulamalarının farklı kopyaları (yani instance’ları) yedekli olarak birden fazla fiziksel ya da sanal makine üzerine yüklenir. Bu stratejide işlemci, bellek, disk gibi I/O kaynakları uygulamalar arasında paylaşımlı olarak kullanılır. Ayrıca DLL gibi uygulama bağımlılıkları host’a direk olarak kopyalanabilir. Bu bakımdan bu stratejinin avantajları mevcuttur. Bununla birlikte uygulamalar arasında izolasyon yoktur. Dolayısıyla uygulamalara ihtiyaçları doğrultusunda farklı kaynak kapasitesi tahsis edilemez. Örneğin A uygulaması B uygulamasından daha fazla işlemci gücü gerektiren bir uygulama ise bu stratejide A uygulamasına daha fazla işlemci kaynağı tahsis edilemez. Bu durum kaynakların paylaşılabilir ancak verimsiz kullanımına neden olur (Richardson, 2016).

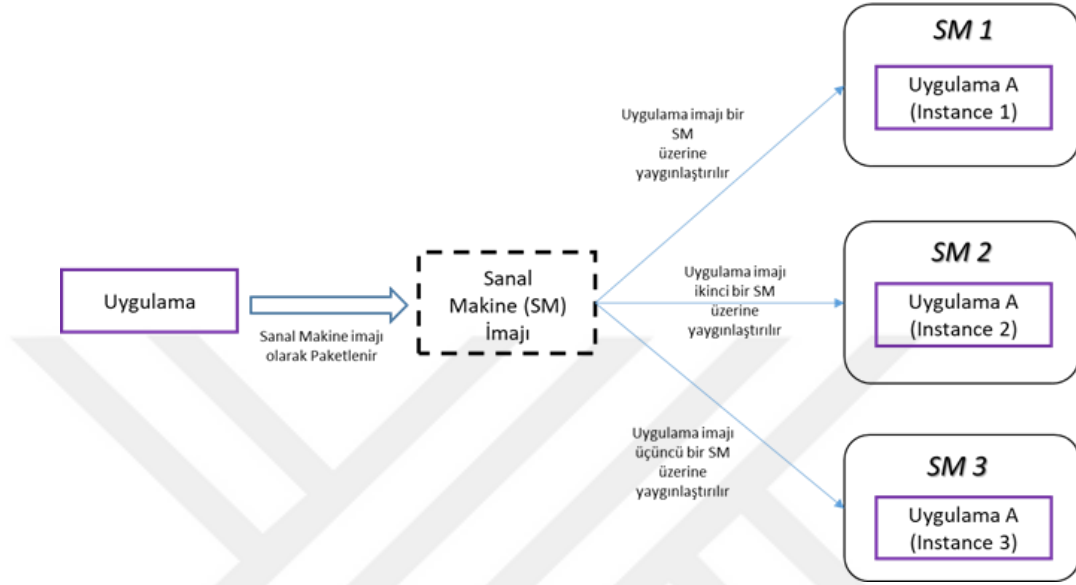


Şekil 5.8. Bir Host'ta Birden Çok Uygulama Barındıran Yaygınlaştırma Stratejisi (Richardson, 2016)

5.2.3.2. Her bir sanal makinede tek bir uygulama barındıran yaygınlaştırma stratejisi

BOA kapsamında kullanılabilen ikinci yaygınlaştırma strateji ise literatürde “Single Application Instance per Virtual Machine” olarak adlandırılmaktadır. Şekil 5.9’da görülen bu stratejide her bir uygulama bir sanal makine imajı olarak paketlenip dedike bir sanal makineye yüklenerek çalıştırılmaktadır. Bu nedenle her uygulama birbirinden izole olarak çalışmakta ve uygulamalara farklı kaynak ihtiyaçları doğrultusunda değişen kaynak tahsisi yapılabilir. Bununla birlikte sanal makinelerin boyutunun büyüklüğü nedeniyle network üzerinden transferi zordur. Bu

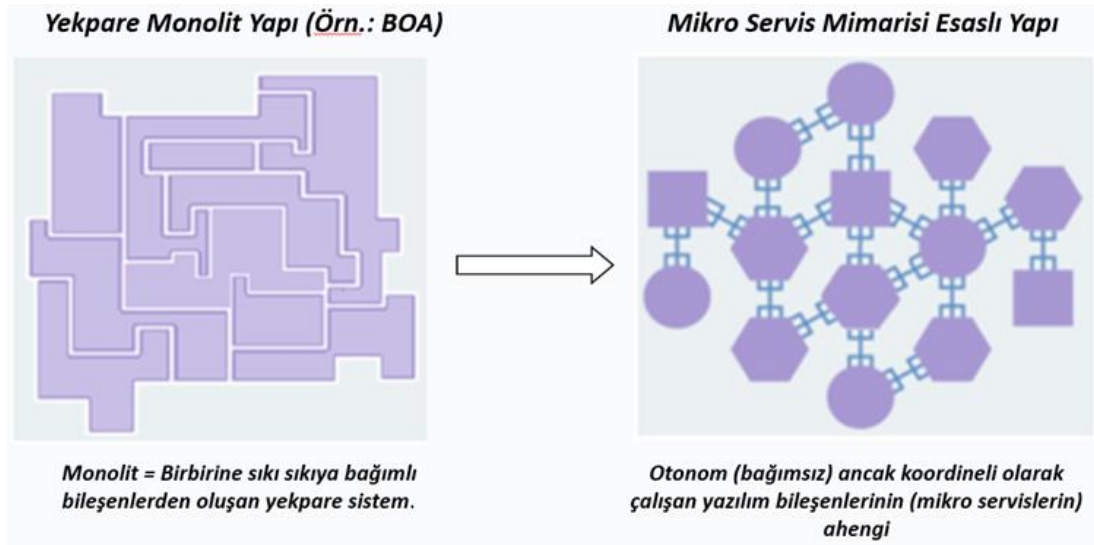
bakımdan uygulamaların yeni versiyonlarının paketlenmiş sanal makine imajı olarak iletilmesi zaman almakta ve bu bakımdan yeni uygulama versiyonun devreye alınmasında nispeten uzun sürebilecek sanal makine stop-redeploy-start süreleri oluşabilmektedir (Richardson, 2016).



Şekil 5.9. Bir Sanal Makinede Bir Uygulama Barındıran Yaygınlaştırma Stratejisi (Richardson, 2016)

5.3. Mikro Servis Mimarisi Nedir?

Mikro Servis Mimarisi (Micro Services Architecture- MSA) ortaya koyduğu çözümler bakımından monolit mimarinin dezavantajlarını ortadan kaldıracak modern bir yazılım mimarisidir (Fowler ve Lewis, 2014). Bu mimaride yazılım sistemini oluşturan bileşenler birbirinden bağımsız olarak geliştirilip test edilip yaygınlaştırılabilen otonom bileşenler olarak tasarlanmak suretiyle yazılım sisteminin bütünündeki karmaşıklığın yönetimi kolaylaştırılmaktadır. Bu bakımdan mikro servis mimarisinde, birbirine sıkı sıkıya kenetlenmiş yap-boz benzeri bir yapının yerini otonom ancak koordineli olarak çalışan bileşenlerin (yani mikro servislerin) ahengi alır (Villamizar ve ark., 2015; Richardson, 2019). Bu durum Şekil 5.10’da temsili olarak ifade edilmektedir.



Şekil 5.10. Monolit Yaklaşımdan Mikro Servis Esaslı Yaklaşım Dönüşüm

Şekil 5.10’da temsili olarak ifade edilmekte olan mikro servis mimarisine dönüşüm sonrasında monolit bir bankacılık yazılım altyapısı bir bisikletin farklı işlevler gören fakat ahenk içinde çalışan bileşenlerymiş gibi (bkz. Şekil 5.11) birbirinden ayrıık ancak koordineli olarak çalışan mikro servislere ayrılmaktadırlar. Bu bakımdan mikro servis mimarisi, monolit bir bankacılık sistemini “Böl ve Yönet” felsefesiyle parçalayarak geliştirmeye ve yaygınlaştırmaya imkân tanıyan ve bu şekilde monolit sistemin bütününde artan karmaşıklığı minimize etmeyi sağlayan modern bir yazılım mimarisi ortaya koymaktadır.



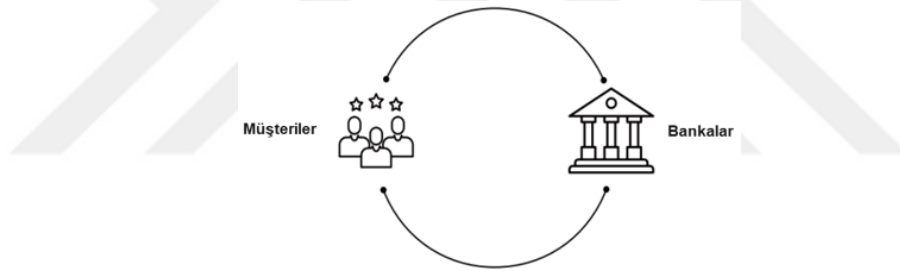
Şekil 5.11. Bağımsız Ancak Koordineli Çalışan Bisiklet Parçalarının Mikro Servislere Benzetimi

5.4. Açık Bankacılık Mimarisi – Open Banking Architecture (OBA)

Bu bölümde açık bankacılık ekosisteminin ortaya çıkışı ve bankaları nasıl etkilediğine kısaca değinildikten sonra doktora çalışmamızın vaka çalışması kapsamında mikro servis mimarisi temelinde tasarım ve geliştirme çalışmaları gerçekleştirilen OBA sisteminin alt yapı ögeleri izah edilmektedir.

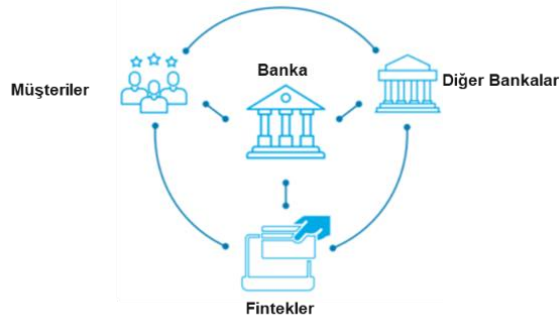
5.4.1. Açık bankacılığın ortaya çıkış süreci, genel felsefesi ve tanımı

Geleneksel manada bankacılık ekosistemi Şekil 5.12’de görüldüğü gibi bankalarla onların müşterileri arasındaki direk etkileşim üzerine kuruludur. Bu etkileşim müşterilerin fiziksel olarak şubede olması şeklinde gerçekleşirken 2000’li yıllar itibariyle internet teknolojilerindeki gelişmelerle birlikte bankaların web sayfaları üzerinden de gerçekleştirilebilir hale gelmiştir.



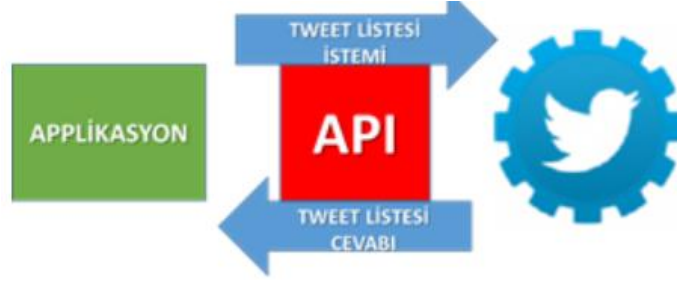
Şekil 5.12. Geleneksel Bankacılık Ekosistemi

Günümüzde mobil iletişim altyapısındaki meydana gelen hızlı gelişmeler doğrultusunda daha çok veri daha yüksek hızda mobil cihazlar üzerinden erişilebilir hale gelmiştir. Bu durum farklı sektörlerden (Turizm, emlak, giyim, otomobil vb.) mal veya hizmetin web üzerinden kolayca alınıp satılmasına imkân tanıyan yazılım uygulamaları üreten finansal teknoloji (FinTek) firmalarının ortaya çıkışına zemin hazırlamıştır. Buna bağlı olarak geleneksel manada anlaşılan bankacılık ekosistemi gelişerek Şekil 5.13’te görülen modern açık bankacılık ekosistemine evrilmiştir.



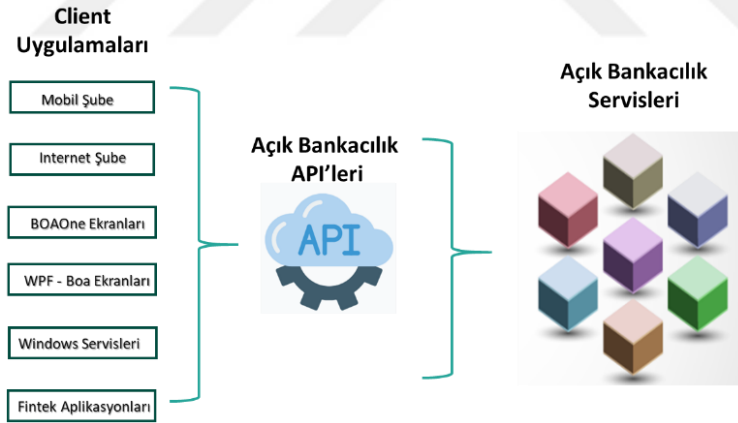
Şekil 5.13. Açık Bankacılık Ekosistemi

Şekil 5.13'te görülen açık bankacılık ekosistemi yalnızca bir banka ve onun müşterileri arasındaki direk ilişkinin ötesinde diğer bankaları ve FinTek'leri de içine alan bir yapı arz etmektedir. Bu yapıda müşteriler bankanın kendi müşterisi olduğu kadar FinTek'lerin ve hatta diğer bankaların da müşterisi de konumundadırlar. Buna bağlı olarak günümüzde bankalar, bilişim altyapılarını gerek kendi aralarında gerekse FinTek yazılımlarıyla daha çevik bir şekilde entegre ederek bu yeni ekosistemde söz sahibi olmayı amaçlamaktadırlar. Teknik boyuttaki bu entegrasyonun sağlanmasındaki en temel yapı taşı Application Programming Interface (API) olarak ifade edilen uygulama programlama ara yüzleridir. API, temelinde iki yazılım bileşeninin iletişim kurmasına olanak tanıyan kurallar ve protokoller bütünü olarak düşünülebilir. Bir başka deyişle API, bir yazılım uygulamasının başka bir yazılım uygulaması tarafından sağlanan özelliklere ya da servise web üzerinden akan birtakım network çağırımları yapmak suretiyle erişmesine izin veren bir yapıdır. Örneğin, cep telefonlarında çalışan bir uygulamanın iş mantığı gereği Twitter'daki bir kullanıcının yaptığı twitleri listelemesi ve twit metinlerini görüntülemesi için Twitter sunucularında çalışan yazılım servislerine internet üzerinden erişmesi gereklidir. Bu erişim Şekil 5.14'te görüldüğü gibi Twitter tarafından sağlanan API'ler ile mümkün olabilmektedir.



Şekil 5.14. Mobil uygulamaların Twitter servislerine yine Twitter tarafından sağlanan API’ler üzerinden erişimi

Bir uygulamanın Twitter servislerinden faydalanabilmesi için Twitter API’lerine erişmesi gerektiği gibi banka bünyesinde mevcut olan client uygulamalarının da (Örnek: Mobil şube) banka tarafından sağlanan açık bankacılık servislerinden (yani mikro servis altyapısından) faydalanabilmesi için yine banka tarafından sağlanan Açık Bankacılık API’lerine erişmesi gerekmektedir. Bu durum Şekil 5.15’te temsil edilmektedir. Bu şekilde client uygulamaları başlığı altında görülen BOAOne ekranları ile WPF-BOA ekranları bankanın genel müdürlük ve şube personeli tarafından kullanılan istemci (client) uygulamalarını temsil etmektedir.



Şekil 5.15. Client (İstemci) Uygulamalarının API’ler Vasıtasıyla Açık Bankacılık Mikro Servislerine Erişimi

Yukarıda verilen bilgiler ışığında Açık Bankacılığın terminolojik tanımı şu şekilde yapılabilir:

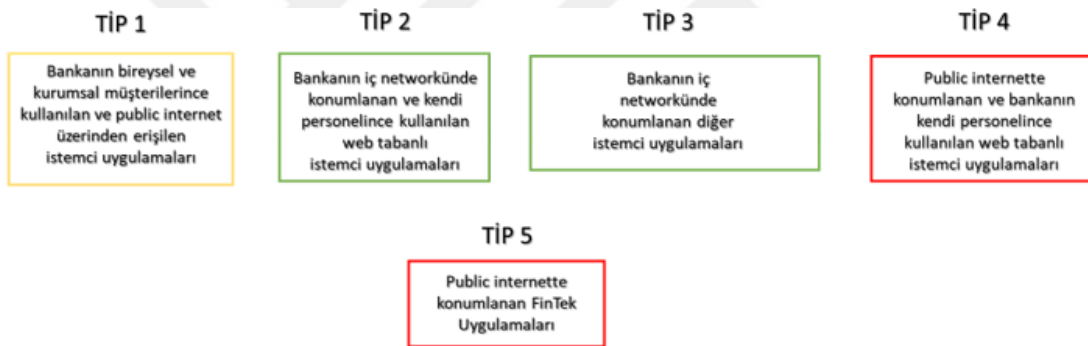
“Open Banking refers to a financial ecosystem where each banking capability is provided by clearly defined Application Programming Interfaces (APIs) and guidelines for improving customer experience” (Laplante ve ark. 2021).

Buradan hareketle doktora çalışmamızın vaka çalışması kapsamında mikro servis mimarisi esaslı tasarım ve geliştirme çalışmaları gerçekleştirilen **Open Banking Architecture (OBA) sisteminin vizyonu** tek cümleyle şu şekilde ifade edilebilir:

“Son kullanıcılara farklı istemci kanallarından (Örnek: Mobil şube, internet şube veya FinTek kanalı) aynı kullanım kolaylığını (customer experience) sağlayan bankacılık servislerinin API’ler şeklinde paketlenerek sunulmasına imkân tanıyan bir bankacılık altyapısını mikro servis mimarisi temelinde oluşturmak.”

5.4.2. OBA’da istemci kanallarının sınıflandırması

Önceki bölümde OBA vizyonu kapsamında işaret edilen istemci kanalları Şekil 5.16.’da görüldüğü gibi 5 mantıksal tipe ayrılabilirler. Bu tipler teknoloji ve kanal mantığı çerçevesinde oluşturulan mantıksal bir sınıflandırmayı ifade etmektedir:



Şekil 5.16. OBA’da İstemcilerin Teknoloji ve Kanal Bazlı Mantıksal Sınıflandırması

- **Tip 1:** iOS/Android tabanlı olarak banka bünyesinde geliştirilen mobil şube uygulaması ile internet bankacılığı web sitesi uygulamasını temsil eder.
- **Tip 2:** Bankanın kurumsal ağı üzerindiyken (ofis veya VPN) açılan ve banka personeline kullanılan istemci uygulamalarını temsil eder. Kuveyt Türk bünyesinde JavaScript (React) tabanlı geliştirilen ve BOAOne ekranları olarak adlandırılan web tabanlı istemci uygulamaları bu gruba girer.
- **Tip 3:** Kurumsal ağ üzerinde çalışan ve kurum ağına olmayı gerektiren istemci grubunu temsil eder. Örneğin periyodik olarak çalışan Windows servisleri, ATM/XTM istemci cihazları, call-center uygulamaları ve ERP uygulamaları ve masa üstü Windows uygulamaları bu gruba girer.

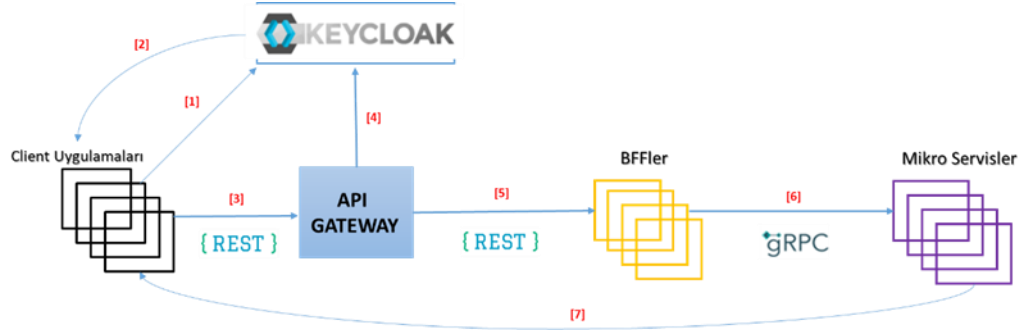
- **Tip 4:** Bankanın kurumsal ağı üzerinde değilken (Örnek: havalimanı, internet cafe) açılan ve public internet üzerinde konumlanan web-tabanlı istemcileri temsil eder. Tip 2’de ifade edilen Java-Script tabanlı BOAOne ekranları public internet üzerinden açıldığında bu grupta değerlendirilmektedir.
- **Tip 5:** FinTek’ler tarafından geliştirilen uygulamaları (Örnek: papara) temsil etmekte olup daima public internet üzerinden erişim sağlayan istemcilerdir.

5.4.3. OBA’nın temel altyapı bileşenleri

OBA altyapısında genel olarak Şekil 5.16’daki blok diyagramında görüldüğü gibi *API Gateway*, *Keycloak*, *BFF’ler* ve *mikro servisler* olmak üzere 4 temel unsur mevcuttur. Bu 4 unsur Şekil 5.16’daki sınıflandırma kapsamında yer alan client uygulamaları tarafından tetiklenen bir dizi aksiyonu icra ederler. Bu aksiyonlar Şekil 5.17’deki aksiyon kolları üzerinde temsil edilmekte olup şu şekilde açıklanabilir:

1. Client uygulaması (Örnek: mobil şube) keycloak’a karşı bir Representational State Transfer (REST) çağırımı yaparak kimlik denetleme (authentication) talebinde bulunur.
2. Keycloak, client uygulamasını OAuth protokolü kapsamındaki bazı erişim anahtarlarına göre (access credentials) doğrular ve akabinde client’a bir “Access Token” döner.
3. Client uygulaması Keycloak’tan aldığı access token’i dahil ettiği bir API request’ini (API isteğini) hazırlayıp API Gateway’e REST stilinde bir API çağırımı gönderir.
4. API Gateway, API request’ini alır access token’ı çeker ve keycloak’la iletişime geçerek bu access token’ın geçerli olup olmadığını doğrular. Burada keycloak’un token’in geçerli olduğu şeklinde bir yanıtı API Gateway’e döndüğünü varsayalım (Buradaki örnek kapsamında pozitif bir senaryo işlendiği varsayılmaktadır).
5. API Gateway, sorumlu BFF’in REST API’sine bir çağırım yapar (yani API request’i (istemi) gönderir).
6. BFF gerekli mikro servis ya da servisleri çağırarak ilgili bankacılık iş mantığını çalıştırır.

7. Servislerin işleri bitince BFF'e dönülen cevaplar tek bir API response'ına (cevabına) paketlenip API Gateway'e ve buradan da nihayetinde orijinal request'i yapan client uygulamasına network üzerinden dönülür.



Şekil 5.17. OBA'da Temel Altyapı Bileşenlerinin bir API Çağırımını (API Request) İşleyişi

Takip eden alt bölümlerde Şekil 5.17'de görülen temel altyapı unsurlarının gerçekleştirdikleri işlevler açıklanmaktadır.

5.4.3.1. Keycloak (Kimlik denetleme – Authentication bileşeni)

Keycloak⁵, Open Authentication (OAuth) Güvenlik Protokolü ⁶ çerçevesinde bulut-tabanlı uygulamalara yönelik kimlik doğrulama servisi sunan açık kaynak kodlu bir sistem olup Java dilinde geliştirilmiş olup geliştirme ve bakım çalışmaları Cloud Native Computing Foundation⁷ (CNCF) bünyesinde gerçekleştirilmektedir. OBA kapsamındaki istemci tiplerindeki uygulamaları kullanan son kullanıcıların (Örnek: Bireysel/kurumsal müşteri, şube personeli) kimlik doğrulamasını (end-user authentication'ını) yapar.

5.4.3.2. API gateway (API geçidi)

API Gateway, client uygulamaları tarafından yapılan API çağırımları sırasında alınan API request'lerini ilgili BFF'e sevk eder. BFF'lerden dönen API response'ını da

⁵ <https://www.keycloak.org/>

⁶ <https://oauth.net/>

⁷ <https://www.cncf.io/about/who-we-are/>

client uygulamalarına döner. Bu çift yönlü sevk sırasında her bir API Request-Response çifti ile alakalı bir jurnal kaydı oluşturarak bu kaydı Kafka Message Bus üzerinden ElasticSearch veri tabanına asenkron olarak kaydeder.

CNCF bünyesinde geliştirilen açık kaynak kodlu bir API Gateway çözümü olan APISIX'in⁸ kullanımı performans odaklı yapısı nedeniyle OBA kapsamında kullanımı benimsenmiştir.

5.4.3.3. Backend for frontend (BFF)

BFF'ler, mikro servis mimarisinde alınan API çağrılarının belirli aplikasyon mantıkları çerçevesinde gruplanması ve ön işleme tabii tutulması amacıyla kullanılan yazılım bileşenleridir (Newman, 2021). OBA kapsamındaki BFF'ler, Şekil 5.16'da görülen her bir istemci tipinden gelen fark API çağrılarının aplikasyon mantığının özelleştirilmesi için kullanılmaktadır. Daha açık bir ifadeyle API Gateway ile mikro servisler arasında konumlandırılan BFF'ler her bir istemci tipi özelinde yapılacak API yetki kontrolü (API Authorization) ve input validasyonu işlevlerini yerine getirirler. Bu bağlamda OBA'da kapsamında mütalaa edilen her bir istemci tipi (bkz. Şekil 5.16) için bir BFF mevcut olacak olup C# dilinde ve .Net 7 platformu üzerinde çalışacak şekilde geliştirilmiştir. Ayrıca her BFF'in REST standartları (Fielding, 2000) çerçevesinde yazılmış API'si olacaktır. Dolayısıyla API gateway bir BFF'e request sevk ederken aslında BFF'in REST API'sini çağırıyor olacaktır.

5.4.3.4. Mikro servisler

OBA kapsamındaki mikro servisler her biri temel bir bankacılık işlevini yerine getirecek şekilde tasarlanan yazılım bileşenleridir. BFF'ler belirli bir bankacılık iş mantığı kapsamında birden fazla mikro servisi senkron (yani anlık request/response esaslı) ya da asenkron (yani publish/subscribe modeli esaslı) çağırabilecektir. Senkron çağırım mikro servislerin gRPC (Goole Remote Procedure Call) ⁹API'leri üzerinden

⁸ <https://apisix.apache.org/>

⁹ <https://grpc.io/>

gerçekleştirilirken; asenkron çağırımlar ise Kafka Message Bus¹⁰ üzerinden gerçekleşmektedir. OBA kapsamında her bir mikro servis C#¹¹ dilinde ve .Net 7 platformu¹² üzerinde çalışacak şekilde geliştiriliyor olacaktır.

5.4.4 OBA bileşenlerinin yaygınlaştırma (deployment) stratejisi

OBA bileşenlerinin yaygınlaştırılmasında kullanılan temel teknoloji konteyner teknolojisidir. Bu teknolojiye uygulamalar işletim sisteminin (OS) kapsüllenmesiyle sağlanan bazı özelliklerden faydalanır. Bu özellikler, uygulamaların birer konteyner imajı içine paketlenerek birbirinden yalıtılmasını ve yalnızca ihtiyaçları kadar CPU, bellek ve disk gibi işletim sistemi kaynaklarını kullanmasını sağlar. Kaynak kullanımı gerektiğinde dinamik bir şekilde artırılıp azaltılabilir. Bu durum kaynak kullanımını daha verimli kılar (Bentaleb ve ark., 2022; Richardson, 2019). Bu nedenle, konteyner teknolojisinin kullanımı giderek popülerlik kazanmakta ve endüstride Docker¹³ ve Podman¹⁴ gibi konteyner teknolojilerinin sayısı giderek artmaktadır.

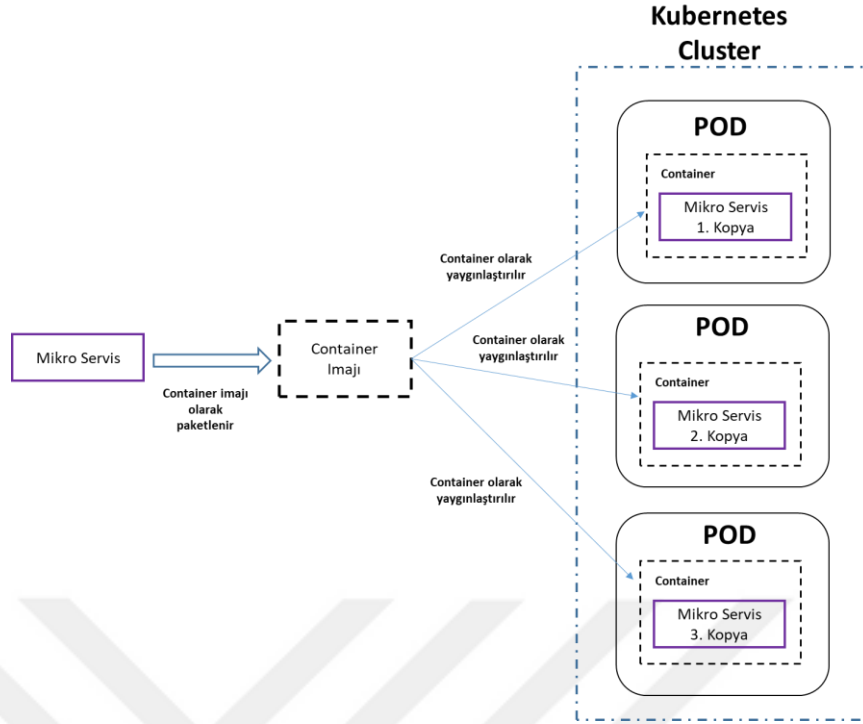
¹⁰ <https://kafka.apache.org/>

¹¹ [https://en.wikipedia.org/wiki/C_Sharp_\(programming_language\)](https://en.wikipedia.org/wiki/C_Sharp_(programming_language))

¹² <https://dotnet.microsoft.com/en-us/>

¹³ <https://www.docker.com/>

¹⁴ <https://podman.io/>



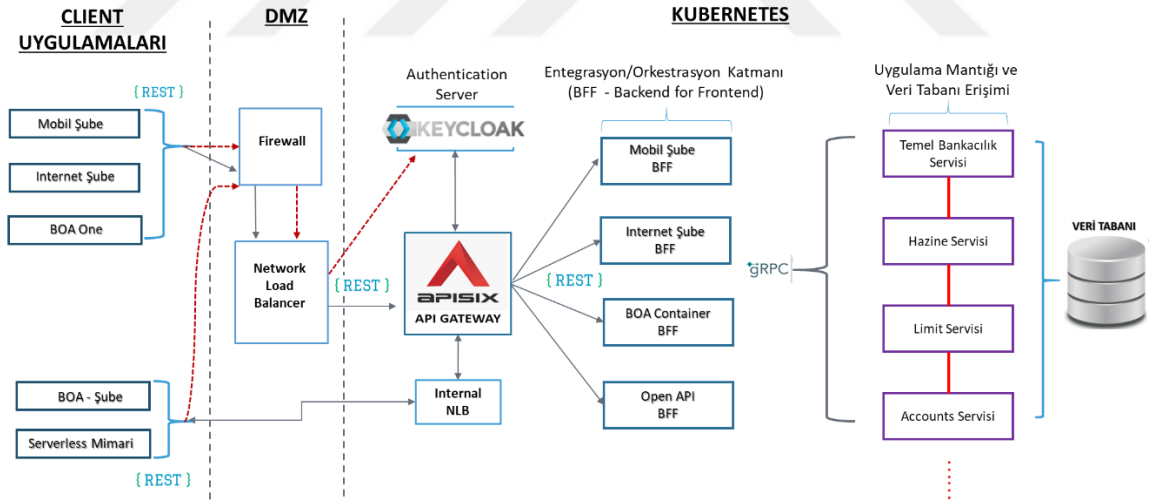
Şekil 5.18. OBA Mikro Servislerinin Konteyner (Container) Teknolojisi Tabanlı Yaygınlaştırma Stratejisi

OBA altyapısında kullanılan yaygınlaştırma stratejisi OBA bileşenlerinin birer konteyner imajı olarak paketlenildikten sonra Kubernetes¹⁵ olarak bilinen bir konteyner tabanlı bilgisayar ağı (yani cluster) üzerinde yaygınlaştırılması şeklinde kurgulanmıştır. Konteyner esaslı yaygınlaştırma stratejisi (Container-based Deployment Strategy) olarak adlandırılabilir. Bu yaygınlaştırma stratejisinde konteynerler Kubernetes cluster'larına özgü bir terim olan POD'lar üzerinde çalışmaktadır. Kubernetes terminolojisinde POD, depolama ve ağ kaynaklarını paylaşımlı olarak kullanan bir veya daha fazla konteyner barındıran mantıksal bir yaygınlaştırma birimi olarak tasavvur edilebilir. Örneğin OBA bünyesindeki bir mikro servis uygulamasının paketlenildiği konteyner imajları Şekil 5.18 görüldüğü gibi 3 kopya (instance) şeklinde bir POD'a deploy edilmek suretiyle yedekli olarak çalışmaktadır. Servislere gelen request yükü Kubernetes tarafından otomatik olarak POD'lara dağıtılmaktadır. POD'lardan birinin çökmesi/hata aması durumunda yeni bir POD içinde otomatik olarak yeni bir konteyner oluşturulmaktadır.

¹⁵ <https://kubernetes.io/>

5.4.5. OBA'ya kuş bakışı

OBA altyapısında bütün OBA bileşenleri birer konteyner içerisinde paketlenerek Şekil 5.19'daki kuşbakışı mimari görünümde görüldüğü gibi Kubernetes cluster'ı üzerinde yaygınlaştırılırlar. Mobil şube, internet şube gibi public internet üzerinde koşan client uygulamaları tarafından OBA bileşenlerine gönderilen network çağırımları öncelikle Demilitarized Zone (DMZ) katmanındaki firewall ve load balancer bileşenlerine ulaşır. Network çağırımları bu DMZ bileşenlerini geçtikten sonra Kubernetes cluster'ı sınırlarına girer ve authentication talebi barındırıyorsa Keycloak'a (Şekil 5.19'da kırmızı kesikli oklarla vurgulanmıştır); API çağırımı barındırıyorsa API Gateway'e yönlendirilirler (Şekil 5.19'da siyah kesikli oklarla vurgulanmıştır). Diğer yandan BOA ekranları gibi kurumsal ağ içerisinde kullanılan iç client uygulamalarına ait network çağırımları ise DMZ katmanına girmeden internal load balancer (internal NLB) aracılığı ile Keycloak veya API Gateway bileşenlerine ulaştırılır. Burada gerek Keycloak'a gerekse API Gateway'e yapılan bütün network çağırımları REST prensipleri (Fielding, 2000) çerçevesinde yapılan network çağırımları olarak düşünülmelidir.



Şekil 5.19. OBA Mimarisine Kuş Bakışı

Şekil 5.19 üzerindeki yapıda client uygulamalarından gönderilen REST çağırımlarının BFF'lere ve mikro servislere direk erişimi hiçbir şekilde mevcut değildir. Bu REST çağırımları BFF'lere ve mikro servislere ancak APISIX (API Gateway) üzerinden ulaşabilmektedir. Bu bakımdan API Gateway, API çağırımlarını sevk eden merkezi bir kontrol birimi olarak tasavvur edilmelidir. API Gateway ile BFF'ler

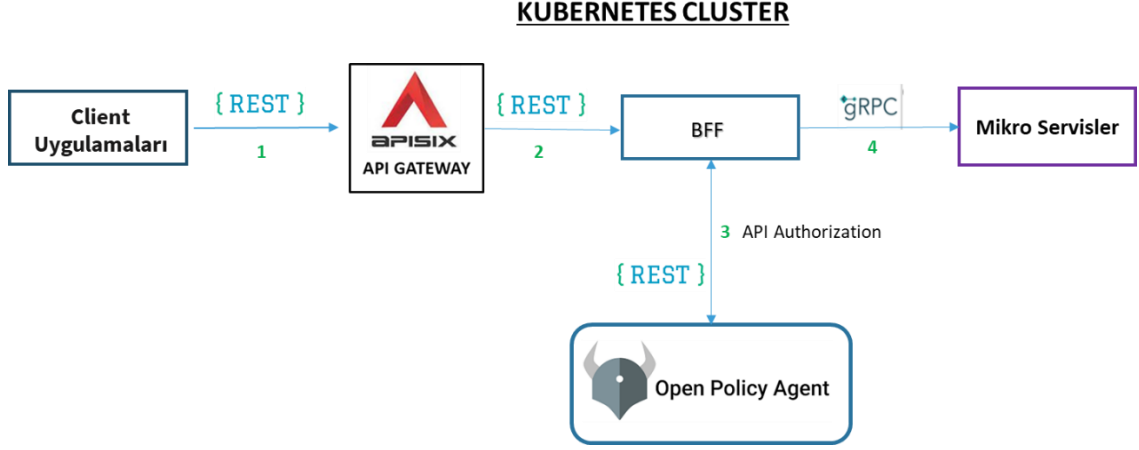
arasındaki network iletişimi REST prensipleri çerçevesinde gerçekleştirilirken; BFF'lerle mikro servisler arasındaki network iletişimi gRPC (Google Remote Procedure Call) çağırımları üzerinden sağlanmaktadır. Bir başka deyişle BFF'ler, mikro servisleri onların gRPC API'lerini çağırarak suretiyle işletmektedir. Bu bakımdan BFF'ler birer gRPC client işlevi görürken; servisler ise birer gRPC server işlevi görmektedir. Şekil 5.19 üzerindeki yapıda dikkat edilmesi gereken diğer bir husus da BFF'lerin adları ve sayısının Şekil 5.16'da belirtilen istemci tiplerini esas alacak şekilde ayarlanmış olduğudur.

5.4.6. Diğer OBA bileşenleri

5.4.6.1. Open Policy Agent (OPA)

Şekil 5.19'da görülen yapı aslında OBA mimarisinin basitleştirilmiş bir kuş bakışı görünümüdür. Buradaki Keycloak, API Gateway, BFFler ve mikro servislerden mütevellit olan temel bileşenlere ek olarak başka OBA bileşenleri de mevcuttur. Bu ilave bileşenlerden birincisi ve belki de en önemlisi Open Policy Agent (OPA)¹⁶ bileşenidir. Bu bileşen CNCF bünyesinde mezun olmuş açık kaynak kodlu bir üründür ve görevi API çağırımlarının müşteriye (yani hesap hamiline) ve istemci tipine göre filtrelenmesidir. Bir başka deyişle OPA, belirli bir müşteri no ve API özelinde authorization (yani API yetki kontrolü) yapmaktadır. OPA, Şekil 5.20'de görüldüğü gibi BFF'ler tarafından REST prensipleri çerçevesinde network çağırımı yapılarak çağırılmaktadır (bkz. Şekil 5.20, adım [3]). Burada bir BFF kendisine ulaştırılan her bir API çağırımı kapsamındaki müşteri numarasını ve API adını (yani API endpoint path) OPA'ya gönderir. OPA'da bu müşteri numarasına sahip müşterinin ilgili API'ye erişim yetkisi olup olmadığını kontrol ederek BFF'e bir cevap döner. BFF mikro servisleri ancak OPA'dan olumlu yanıt dönerse çağırır (Adım [4]). Aksi takdirde API Gateway'e HTTP401 unauthorized status kodunu içeren bir HTTP response'ı döner.

¹⁶ <https://www.openpolicyagent.org/>



Şekil 5.20. Bir BFF’in OPA üzerinden API Yetki Kontrolü (Authorization) Yapması

5.4.6.2. Consul ve Vault

Consul¹⁷, Hachicorp firması tarafından üretilmiş açık kaynak kodlu bir ürün olup OPA gibi CNCF bünyesinde büyümüş bir üründür. Consul BFF’lerin ve mikro servislerin aplikasyon parametrelerini merkezi olarak tutmaktadır. Kubernetes cluster’ındaki her bir BFF ve mikro servis ile ilgili aplikasyon parametreleri boot zamanında Consul marifetiyle appsettings.json adlı konfigürasyon dosyasına beslenmektedir. Consul’un aplikasyon parametrelerini appsettings dosyasına boot zamanında beslemesi Kubernetes bünyesinde “short-running” işler için özel olarak tasarlanmış bir konteyner olan init-container marifetiyle gerçekleştirilmektedir. BFF’ler ve mikro servisler boot edilmeden hemen önce consul-init-container, appsettings.json dosyasına ilgili uygulama parametrelerini kaydedip sonlanmaktadır. Bunun hemen akabinde BFF’ler ve mikro servisler boot olurken en güncel parametreleri appsettings.json dosyasından okumaktadırlar.

Vault¹⁸ da benzer şekilde Hachicorp firması tarafından üretilmiş açık kaynak kodlu bir ürün olup Consul gibi CNCF bünyesinden büyümüş bir üründür. Consul aplikasyon parametrelerinden sorumlu iken Vault ise veri tabanı bağlantısı için gerekli kullanıcı adı ve şifre gibi “secret” kabilinde düşünülebilecek parametrelerin tutulmasından ve BFF’lere ile mikro servislere beslenmesinden sorumludur. Vault da

¹⁷ <https://www.consul.io/>

¹⁸ <https://www.vaultproject.io/>

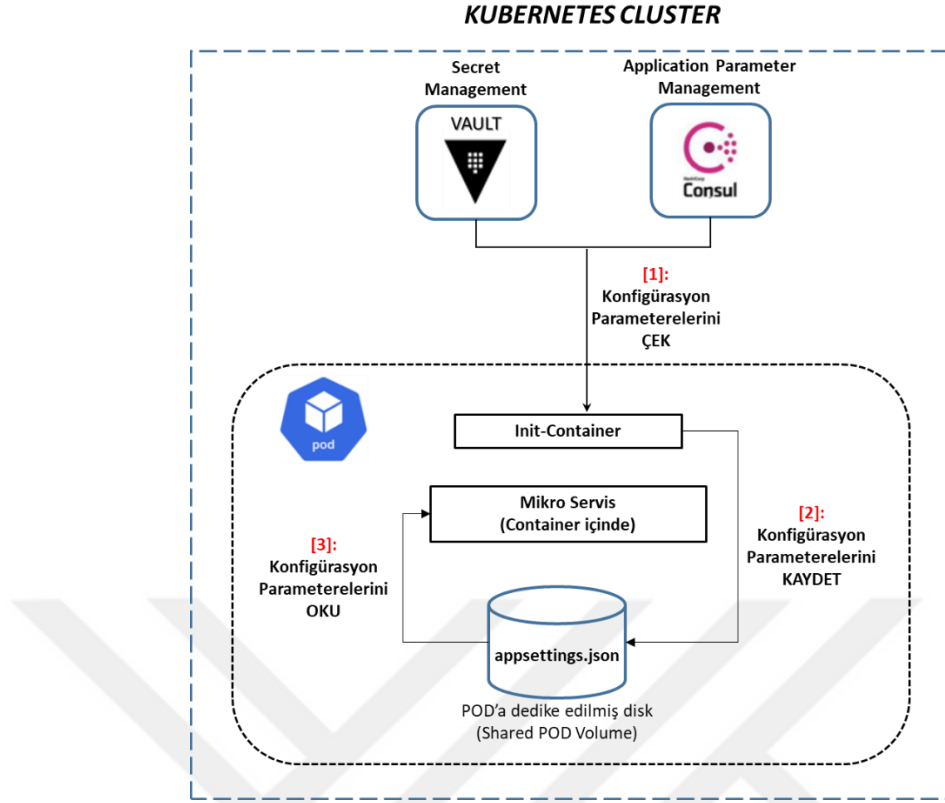
aynı consul gibi bir init-container marifetiyle appsettings.json dosyasını secret parametreleri ile besler.

Yukarıdaki paragraflarda Consul ve Vault kapsamında açıklanan yapının örnek bir işletimi aşağıda Şekil 5.21’de resmedilmiştir. Bu Şekil referans alındığında şu etkileşimler gerçekleşir:

- i. [1] numaralı adımda bir init-container, bir mikro servis özelindeki uygulama parametrelerini Consul’dan çekerken; Vault’tan da secret parametrelerini çeker.
- ii. [2] numaralı adımda init-container Consul ve Vault’tan çektiği parametreleri ilgili mikro servisin koşacağı Kubernetes POD’undaki disk alanına appsettings.json adlı bir dosyanın içine JavaScript Object Notation (JSON) formatında kaydeder. Bunun akabinde init-container hemen sonlanır.
- iii. İlgili mikro servis, kubernetes tarafından yönetilen bir POD’da konumlanan bir konteyner içerisinde boot edilir. Bu sırada mikro servis POD diskinde init-container tarafından ikinci adımda oluşturulmuş olan appsettings.json dosyasını [3] numaralı adımda okuyup genel işleyişine başlar.

Yukarıdaki örnek işletim her bir mikro servis ve BFF özelinde ayrı ayrı gerçekleşmekte olup geliştiricilerin direk olarak dahil olmasını gerektirmeyen otomatik bir süreçtir. Geliştiriciler ve network yöneticileri mikro servisler ve BFF’ler özelindeki uygulama parametrelerini sırasıyla consul ve vault tarafından sağlanan web tabanlı ara yüz (yani bir ekran) vasıtasıyla girerler. Bununla birlikte geliştiriciler kendi şahsi bilgisayarlarında çalışırken gerek BFF gerekse mikro servislerle alakalı aplikasyon ve secret parametrelerini Visual Studio¹⁹ içerisinde görüntüleyecekleri **appsettings.json** dosyası içerisinde manuel olarak JSON formatında yapıyor olacaktırlar ve bu durum Kubernetes’deki işleymden bağımsız olarak yalnız geliştirici bilgisayarlarında husule gelecektir.

¹⁹ <https://visualstudio.microsoft.com/>



Şekil 5.21. Consul ve Vault'un Mikro Servis Konfigürasyonunu besleyişi

5.4.6.3. Fluent Bit

FluentBit,²⁰ CNCF bünyesinde büyümüş bir ürün olup telemetrik verilerinin performanslı bir şekilde bir yazılım kaynağından toplanıp bir veri tabanına kaydetme konusunda uzmanlaşmış açık kaynak kodlu bir üründür.

FluentBit, 4 temel OBA bileşeni olan Keycloak, API Gateway (APISIX), BFF'ler ve mikro servisler tarafından oluşturulan aplikasyon log kayıtlarının performanslı bir şekilde toplanıp Elasticsearch²¹ veri tabanına kayıt edilmesi için OBA bünyesine dahil edilmiştir. FluentBit, bu aplikasyon log kayıtlarını esasen bu 4 OBA bileşeninin çalıştığı konteynerler bünyesindeki standart işletim sistemi stream'lerini izlemek suretiyle yapar. Bu stream'ler Unix türevi bütün işletim sistemlerinde mevcut olan ve işletim sistemi tarafından sağlanan **stdout** ve **stderr** stream'leridir. Stdout information ve debug

²⁰ <https://fluentbit.io/>

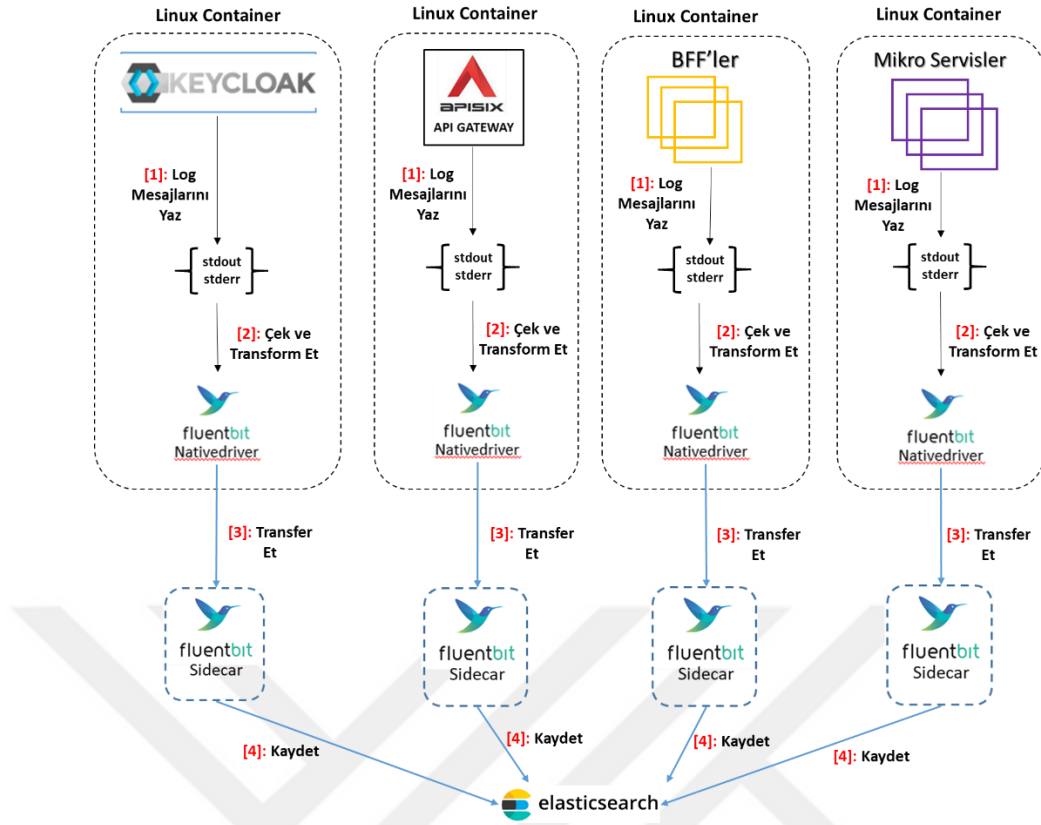
²¹ <https://www.elastic.co/>

loglarının aktığı bir işletim sistemi stream'i iken; stderr ise hatalı durumlarda oluşturulan error log kayıtlarının aktığı stream'dir.

4 temel OBA bileşeni olan Keycloak, API Gateway (APISIX), BFF'ler ve mikro servislerin her birinin üzerinde çalıştıkları konteynerler aslında Linux tabanlı imajlardan türetilmiş konteynerlerdir. Bu imajlar OBA özelinde yazılmış dockerfile'lar marifetiyle oluşturulmuş olup imajların oluşum zamanında FluentBit tarafından sağlanan bir native driver (yani bir sistem yazılımı) imajların bünyesine dahil edilmektedir. Bu kapsamda Şekil 5.22'de görülen FluentBit teknolojisi esaslı OBA altyapısında şu adımlar gerçekleşmektedir:

- i. OBA bileşenleri, herhangi bir aplikasyon logu ürettiğinde Şekil 5.22'de [1] numaralı adımlarda vurgulandığı gibi ilgili aplikasyon logu stdout ve stderr streamlerine Linux işletim sistemi tarafından aktarılır.
- ii. Bunun akabinde standart stream'leri konteyner içinde sürekli dinlemekte olan native driver gelen log kayıtlarını işleyip aynı POD içerisinde bulunan FluentBit side-car instance'larına JSON formatında periodik olarak transfer eder (bkz. Şekil 5.22'deki [2] numaralı adımlar).
- iii. FluentBit side-car instance'ları aldıkları log kayıtlarını ElasticSearch'e kaydeder (bkz. Şekil 5.22'deki [3] numaralı adımlar).

Mikro servis geliştiricisinin bakış açısıyla aplikasyon loglama işlemi .NET 7 platformu ile gelen standart **ILogger** objelerini kullanmak suretiyle yapılmaktadır. Bir başka deyişle BFF'lerin ve mikro servislerin C# sınıf yapıcılarında (class constructor'larında) dependency injection marifetiyle oluşturulan ILogger objeleri üzerindeki LogInformation ve LogError metotlarını çağırmak suretiyle programmatik olarak aplikasyon loglama gerçekleşmektedir. Geliştiricilerin işi burada bitmekte ve dolayısıyla aplikasyon loglarının ElasticSerach'e kadar olan yolculuğu Şekil 5.22'de görüldüğü şekilde geliştiricinin farkındalığı olmadan gerçekleşmektedir.



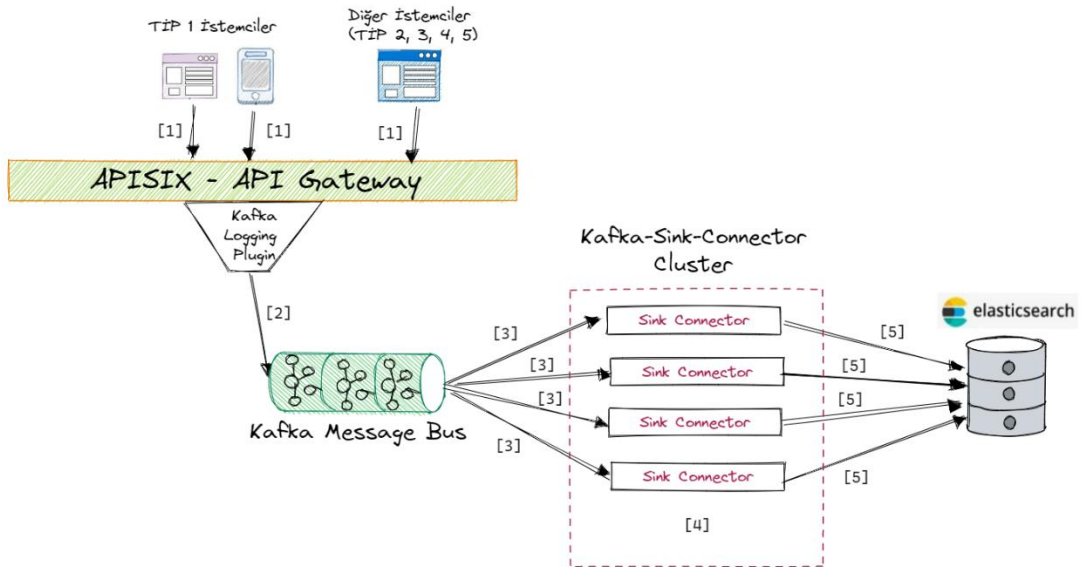
Şekil 5.22. OBA’da Uygulama Log Kayıtlarının FluentBit Teknolojisini Kullanarak İşlenmesi ve Kayıt Altına Alınması

5.4.7. OBA’da request-response log (jurnal) altyapısı

Daha önce Şekil 5.16’da ifade olunan bütün OBA client (istemci) uygulamalarından (yani Tip 1, 2, 3, 4 ve 5 istemciler) API Gateway’e (yani APISIX’e) gönderilen ve bu istemcilere dönülen bütün request-response çiftleri OBA minvalinde jurnal kaydı olarak nitelendirilmekte ve BDDK kararları gereği kayıt altına alınmaktadır. Bu amaçla OBA’da tasarlanan jurnal altyapısı Şekil 5.23’te görülmektedir. Bu şekil üzerinden bir jurnal kaydının alınması, işlenmesi ve kayıt altına alınması kapsamında şu adımlar gerçekleşmektedir:

- i. Şekil 5.23’te [1] numaralı adımlar kapsamında OBA client (istemci) uygulamalarından APISIX’e doğru bir bankacılık API’si çağırımı gerçekleştirilir (Örnek: Kur listesi dönen API ya da para transferi yapan API çağırılır). APISIX kendisine ulaşan API çağırımı kapsamında ilgili BFF üzerinden işi gerçekleştirecek mikro servis ya da servislerin işletimini sağlar ve sonuçta ilgili BFF’ten APISIX’e bir API response’ı (cevabı) döner.

- ii. APISIX API request-response çiftini JSON formatında ham bir jurnal kaydı olarak paketler ve kafka message bus'a iletir (bkz. Şekil 5.23'teki [2] numaralı adımlar). Paketlenen ham jurnal kaydının Kafka message bus'a iletilmesinde APISIX bünyesinde açık kaynak kodlu olarak mevcut olan APISIX Kafka Logging plugin ²² (tak-çalıştır bileşeni) kullanılmaktadır.
- iii. Kafka Message bus'a iletilen jurnal kayıtları daha sonra Kafka-Sink cluster'ımızdaki sink-connector'lar tarafından asenkron olarak Şekil 5.23'teki [3] numaralı adımlarda görüldüğü gibi toplanır.
- iv. Şekil 5.23'te görülen Kafka-Sink-Connector cluster'ındaki her bir sink connector içinde kendi yazdığımız Java tabanlı bir jurnal transformer bileşen çalışmaktadır. Bu transformer bileşen, sink-connector tarafından çekilen jurnal kayıtlarında maskeleyme ve hassas verilerin silinmesi (Örnek: T.C. Kimlik No, şifre) gibi bir takım manipülasyon işlemleri yapar (bkz. Şekil 5.23'teki [4] numaralı adım).
- v. Son olarak sink connector bileşenleri işlenmiş jurnal kaydını paralel olarak çalışarak Elasticsearch'e kaydetmektedir (Şekil 5.23'teki [5] numaralı adımlar).



Şekil 5.23. OBA'da Jurnal Kayıtlarının Toplanıp İşlenmesi ve Kayıt Altına Alınması

²² <https://apisix.apache.org/docs/apisix/plugins/kafka-logger/>

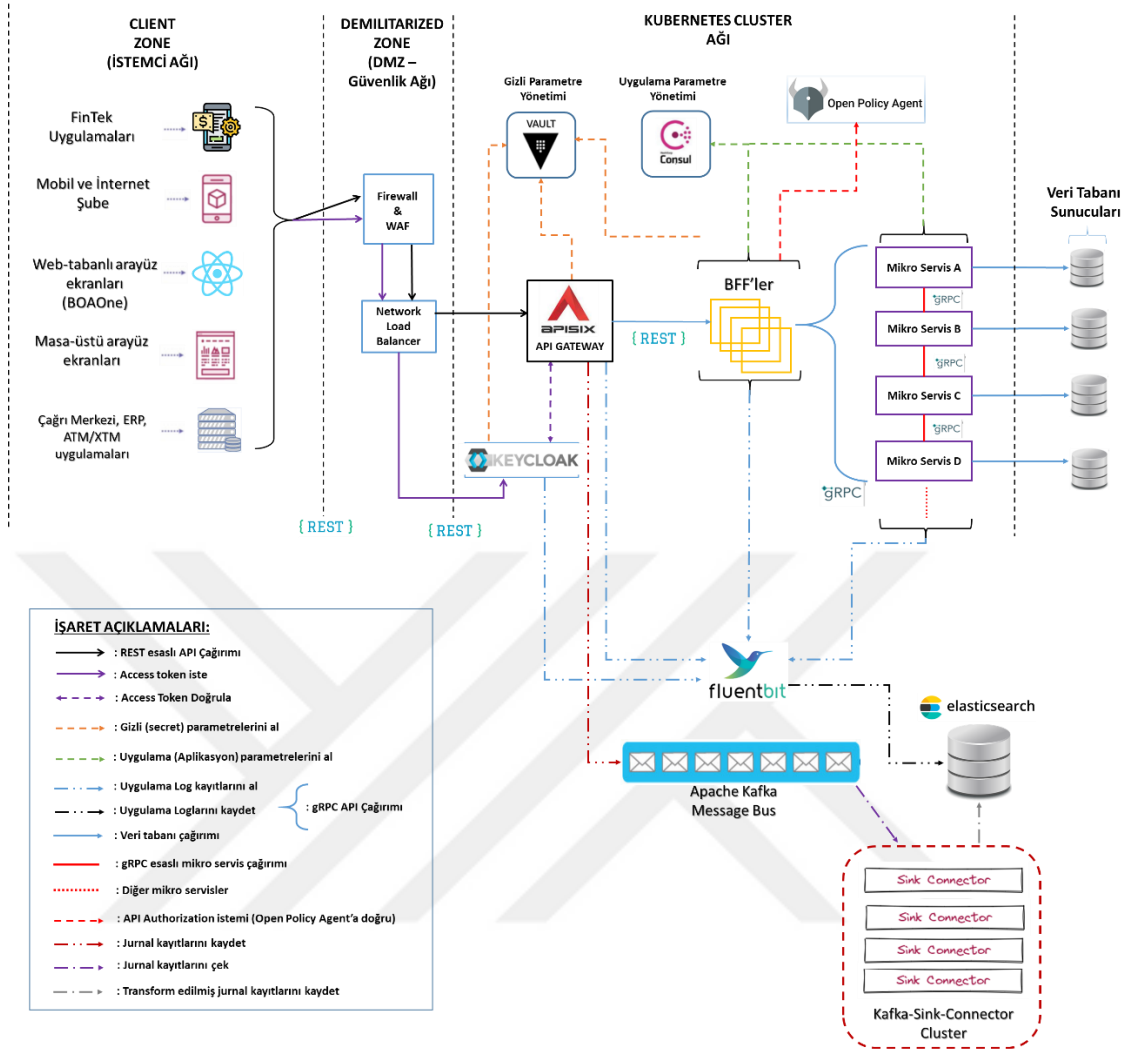
Buradaki Kafka-Sink-Connector²³ altyapısı Kafka Message Bus'ın orijinal geliştiricisi olan Confluent firması tarafından açık kaynak kodlu olarak üretilen bir streaming platformudur. Bu platform, Kafka Message Bus'tan performanslı bir şekilde veri okunup Elasticsearch gibi veri tabanlarına veri aktarımı konusunda uzmanlaşmıştır ve bu performanslı yapısı nedeniyle OBA jurnal altyapısı bünyesinde kullanımı benimsenmiştir. Kafka-Sink-Connector cluster'ı canlıda koşan OBA altyapısında 4 makineden oluşan dedike bir sanal ağ (cluster) üzerinde ve Kubernetes dışında konumlandırılmıştır. Kafka-Sink-Connector cluster'ı bilinçli olarak Kubernetes dışında konumlanmıştır zira yoğun input/output (I/O) işlemleri gerektirmektedir ve bu durum Kubernetes cluster'ının performansını olumsuz etkileyebilir. Diğer yandan Kafka-Sink-Connector cluster'ındaki sanal makinelerin her birinde Confluent firması tarafından yazılan Java kütüphanelerini kullanarak geliştirdiğimiz Java Transformer bileşenlerimiz çalışmaktadır. Daha önce de belirtildiği üzere bu transformer bileşenler jurnal kayıtlarında maskeleye ve hassas verilerin silinmesi (Örnek: T.C. Kimlik No, şifre) gibi bir takım manipülasyon işlemleri yapmaktadır.

5.4.8. OBA mimarisinin bütününe genel bakış

Sorumlulukları ve etkileşimleri önceki bölümlerde açıklanan OBA bileşenlerinin tamamının bir arada olduğu mimari yapının daha geniş bir perspektiften görünümü aşağıda Şekil 5.24'te sunulmaktadır.

²³ https://docs.confluent.io/platform/current/connect/kafka_connectors.html

<< OBA Mimarisinin Bütününe Genel Bakış >>



Şekil 5.24. OBA Mimarisinin Bütününe Genel Bakış
(Daha yüksek çözünürlüklü bir imaj için EK-1'e bakılması rica olunur.)

Şekil 5.24'te en sol bölümde daha evvel Şekil 5.16 ifade edilen istemci tipleri yer almaktadır. Bu istemciler sisteme genel erişim için Keycloak'tan access-token alır ve bu token ile APISIX (yani API geçidi) üzerinde tanımlı bankacılık API'lerini çağırırlar. Bu API'ler esasen daha önce Bölüm 4'te Tablo 4.1 ve 4.2'de listelenen bankacılık işlevleriyle birebir ilişkili API'lerdir. Bu bankacılık API'lerinden herhangi birinin çağırımı sonrasında aşağıdaki olaylar vuku bulur:

1. APISIX, istemciden gelen API çağırımı kapsamında alınan API request'inin (isteminin) header alanından access-token bilgisini okur ve bunu Keycloak ile iletişime geçerek doğrular. Bu suretle API çağırımının bilinen ve izin verilen bir müşteriden ve istemciden geldiği doğrulanır (yani kimlik kontrolü yapılmış olur).

2. APISIX, istemciden gelen API çağırımını bu çağırım ile ilişkilendirilmiş BFF'e (Örnek: Mobil şube BFF'i) yönlendirir.
3. İlgili BFF, OPA'ya *müşteri no (999111 gibi), istemci tipi (Şekil 5.16'daki tiplerden biri)* ve *API endpoint adı* (Örnek: /transfers/toIban) parametrelerini içerecek şekilde bir API authorization (yetki kontrolü) sorgusu yapar. OPA, bu sorgu kapsamında ilgili bankacılık müşterisinin API çağırımını yapan istemcinin tipi ve API endpoint'i özelinde erişim yetkisi olup olmadığını kontrol eder ve BFF bu konuda bir cevap döner. Bu süreç near real-time'da (gerçek zamanda) gerçekleşmektedir (ortalama 6 ms'de).
4. OPA'dan olumlu yanıt dönülürse BFF ilgili bankacılık işlevini yerine getirmek adına bir veya daha fazla mikro servis ya da servisleri çağırır. Burada eğer döviz alım/satım veya para transferi gibi transactional işlemleri kapsayan bir bankacılık işlevi söz konusu ise BFF, SAGA komut-orkestratör tasarım deseninin genişletilmiş bir varyantını tatbik etmek suretiyle birden fazla mikro servisi ardışık olarak çağırır. Bu SAGA varyantı bu doktora tezinin ana akademik katkılarından biri olup detayları 6. bölümde sunulmaktadır.
5. BFF, bankacılık işlevinin icrası kapsamında yaptığı mikro servis çağırımlarının sonucu olarak bir API response'ı üretir ve bunu APISIX'e döner.
6. APISIX istemciden gelen orijinal API request mesajı ile BFF'ten dönen cevabı JSON formatında paketleyerek bir jurnal kaydı oluşturur ve bu kaydı Kafka Message Bus'a iletir (stream eder).
7. Kafka Message Bus'ı izlemekte olan Kafka-Sink-Connector altyapımız 5. adımda Kafkaya stream edilen jurnal kaydını anlık olarak alır (consume eder).
8. Kafka-Sink-Connector'ların her birinin üzerinde bulunan transformer'lerden biri 6. adımda alınan jurnal kaydı üzerinde maskeleyme ve şifreleme işlemleri yapar (Örnek: Şifre, kredi kartı numarası, bakiye alanları maskelenir) ve maskelenmiş jurnal kaydını ElasticSearch veritabanında kayıt altına alır.
9. APISIX, 5. adımda kendisine BFF tarafından dönülen API mesajını istemi yapan istemciye döner. Burada 6'dan 9'a kadar olan adımlar paralel icra edilmektedir zira 6, 7 ve 8 numaralı adımlar APISIX tarafından asenkron olarak icra edilmektedir.

Yukarıda 5. adımda çağırılan mikro servislerin banka yazılımcıları tarafından geliştirilmesini kolaylaştırmak amacıyla C# dilinde ve .NET 7 runtime'ını kullanacak

şekilde kütüphaneler hazırlanmıştır. ‘Kaynak Kod Yapılanması’ adlı sonraki alt bölümde bu kütüphaneler hakkında detaylı bilgi sunulmaktadır.

5.4.9. Kaynak kod yapılanması

OBA kapsamında bankacılık mikro servisi geliştirecek banka yazılımcılarının temel kütüphane ihtiyaçlarını karşılamak ve dolayısıyla geliştiricilerin sadece mikro servislerin aplikasyon mantığına odaklanmasını sağlamak amacıyla vaka çalışmamız bünyesinde bir takım temel alt yapı projeleri geliştirilmiştir. .Net 7’de geliştirilen bu projeler mikro servis mimarisi esaslı bir bankacılık framework’u olarak düşünülebilir. Bu alt bölümde hususen bu framework’ü oluşturan projelere ve içeriklerine değinilmektedir.

OBA’da “*bir Visual Studio Solution’ı = bir GIT Respository*” olacak şekilde bir kod yapılanması benimsenmiştir. Bu bağlamda bankanın bütün geliştirme ekipleri tarafından doğrudan (yani kütüphane bağımlılığı olarak) veya dolaylı olarak kullanılacak projeleri barındıran İki temel Visual Studio solution’ı ve iki temel GIT alanı mevcuttur. Sırasıyla **OBA** ve **OBA.Core** olarak adlandırılan bu Visual Studio Solution’larının içeriği hakkında aşağıdaki ilk iki alt başlıkta bilgi verilmektedir. Son alt başlıkta ise uygulama geliştiricilerin esas alması gereken Visual Studio Solution/Proje isimlendirme standartlarımız belirtilmektedir.

5.4.9.1. OBA çatı proje içeriği

Çatı proje olarak da adlandırılan OBA adlı solution içerisinde Şekil 5.25’te görüldüğü gibi, APISIX, Keycloak ve IdentityAPI gibi temel altyapı unsurları ile bunlarla ilgili projeler, proje klasörleri ve konfigürasyon dosyaları yer almaktadır. Bu unsurların işlevleri kısaca şu şekilde izah edilebilir (Bkz. Şekil 5.25):

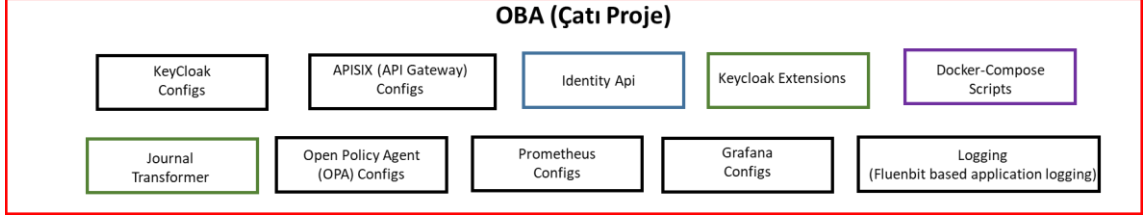
Kırmızı Çerçeve = GIT Repository = Visual Studio Solution

Siyah Çerçeve = Solution Klasörü

Mavi Çerçeve = Visual Studio C# Projesi

Yeşil Çerçeve = Java Projesi (IntelliJ)

Mor Çerçeve = Visual Studio Docker-Compose Betikleri



Şekil 5.25. OBA (Çatı Proje) İçeriği

OBA Çatı Proje İçeriği (Şekil 5.25):

- 1. Keycloak Configs:** Keycloak'un gerek Kubernetes ortamı ve gerekse local geliştirici bilgisayarlarında konfigüre edilmesi için gerekli dosyaları ve keycloak imajının oluşturulması için gerekli Dockerfile'ı içerir. Solution içerisinde şu klasör altında mevcuttur: src\Identity\KeyCloak\Server
- 2. APISIX (API Gateway) Configs:** API Gateway olarak kullandığımız APISIX ürününün konfigürasyon dosyalarını içerir. Solution içerisinde şu klasör altında mevcuttur: src\Gateway\APISIX
- 3. IdentityApi:** Keycloak'un veri tabanındaki kullanıcı verilerine ve ActiveDirectory kayıtlarına erişimi için geliştirdiğimiz bir .Net 7 web API projesidir. Kaynak kodlar, appsettings.json ve Dockerfile gibi öğeler barındırır. Solution içerisinde şu klasör altında mevcuttur: src\Identity\Services\IdentityIntegrationApi
- 4. Keycloak Java Extensions:** Keycloak'un davranışını OBA minvalinde genişletme etme amaçlı olarak yazdığımız Java kütüphanelerinin kaynak kodlarını barındırır (POM/Maven projesi olarak). Java SE 1.8 esaslı geliştirilmiş olup Keycloak'un bünyesine bir tür plugin (yani tak çalıştır bileşeni) olarak eklenmektedir. Solution içerisinde şu klasör altında mevcuttur: src\Identity\KeyCloak\Extensions
- 5. Docker-Compose Scripts:** OBA solution'ı altında yer alan ve burada izah olunan altyapı unsurlarının geliştirici bilgisayarlarında konteynerler olarak oluşturulup uçtan uca test edilmesine olanak tanıyan docker-compose betiklerini (script'lerini) içerir. Bu betikler src lokasyonu altında sırasıyla

docker-compose.yml ile docker-compose.override.yml dosyalarında yer almaktadır. Bu dosyaların içeriği EK-2’de sağlanmaktadır.

6. Journal Transformer: Bölüm 5.3.6’da izah edilen Journal transformer öğeleri için yaptığımız java geliştirmelerimizi barındırır. Bu geliştirmeler bir statik java kütüphanesi olarak (yani jar dosyası olarak) Kafka-Sink-Connector cluster’ımızdaki node’lara deploy edilmektedir (Azure Devops altyapısı tarafından derlenerek). Solution içerisinde şu klasör altında mevcuttur: src\KafkaConnect\KtJournalTransformator

7. Open Policy Agent (OPA) Configs: Bölüm 5.3.5.1’de izah olunan OPA bileşeninin konfigürasyon dosyalarını içerir. Solution içerisinde şu klasör altında mevcuttur: src\Opa\Policies

8. Prometheus Configs: OBA kapsamında aplikasyon metriklerini toplamak için kullandığımız bir bileşen olup bununla ilgili konfigürasyon dosyalarını barındırmaktadır. Solution içerisinde şu klasör altında mevcuttur: src\Prometheus

9. Grafana Configs: OBA altyapı metriklerini görsel olarak göstermek için kullanılan bir bileşen olup bununla ilgili konfigürasyon dosyalarını barındırmaktadır. Solution içerisinde şu klasör altında mevcuttur: src\Grafana\grafana_conf

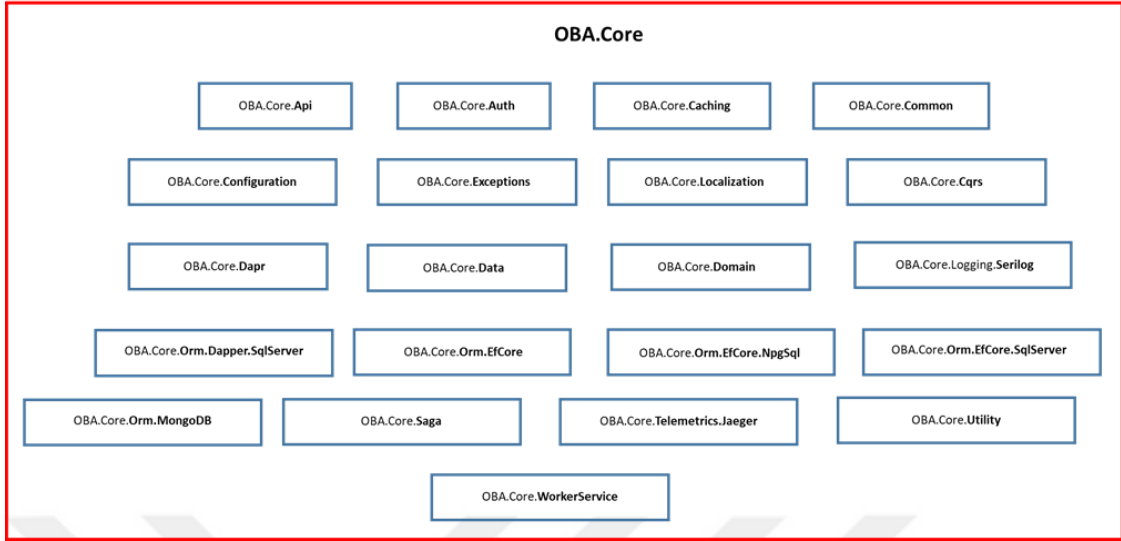
10. FluentBit Application Logging: Bölüm 5.3.5.3’te izah edilen ve application logging altyapımızın konfigürasyon dosyaları barındırılmaktadır. Solution içerisinde şu klasör altında mevcuttur: src\Logging\fluentd

5.4.9.2. OBA core solution içeriği

OBA.Core solution’ı OBA üzerinde mikro servis geliştirmelerini kolaylaştırmak için geliştirdiğimiz class library projeleridir. Şekil 5.26’da belirtilen bu projeler derlenerek nuget paketleri haline getirilmiştir (yani kaynak kod kütüphanesine dönüştürülmüştür). Banka uygulama geliştiricileri mikro servis geliştirmelerini bu nuget paketlerini kurumsal nuget repository’imiz üzerinden projelerine eklemek suretiyle yapacaklardır.

Kırmızı Çerçeve = GiT Repository = Visual Studio Solution

Mavi Çerçeve = Visual Studio C# Projesi



Şekil 5.26. OBA.Core Solution'ı ve Barındırdığı Class-Library Projeleri (.NET 7)

Şekil 5.26'da görülen projelerin içeriği ve amacı aşağıda açıklanmaktadır:

- 1. OBA.Core.Api:** Mikro servis geliştiricilerin hemen hemen bütün projelerinde kullanması gereken en temel OBA altyapı projesidir. Özellikle program.cs dosyalarından çağırılması gereken static extension metotlarını içermektedir. Bu metotlar OBA altyapısını geliştirici projelerine bir elbise gibi giydirmektedir (Örnek: AddInfrastructure, AddGrpcInfrastructure, AddEfCore).
- 2. OBA.Core.Auth:** BFF olarak geliştirilen projelerin OPA üzerinden API authorization yapabilmesi (yani OPA ile iletişime geçebilmesi) için gerekli middleware altyapısını barındırır. OBA.Core.Api tarafından nuget bağımlılığı olarak kullanılır bu nedenle geliştiricilerin nuget paketi olarak direk kullanımına açık değildir.
- 3. OBA.Core.Caching:** Önbellek yönetimi için kapsamındaki geliştirmelerimizi içerir. OBA.Core.Api tarafından nuget bağımlılığı olarak kullanılır bu nedenle geliştiricilerin nuget paketi olarak direk kullanımına açık değildir.
- 4. OBA.Core.Common:** Diğer projeler tarafından ortak olarak kullanılan bir projedir. Geliştiricilerin direk kullanımına açık değildir.
- 5. OBA.Core.Exceptions:** Exception yönetimi ile ilgilidir. Geliştiricilerin direk kullanımına açık değildir.

- 6. OBA.Core.Localization:** Constant mesaj tanımlarının bilimum servislere ve BFF'lere iletimi için kullanılmaktadır. Geliştiricilerin direk kullanımına açık değildir.
- 7. OBA.Core.Cqrs:** Veri tabanı erişiminde CQRS tasarım desenini kullanan altyapıyı sağlar. Core.api projesi tarafından nuget paketi olarak kullanılır.
- 8. OBA.Core.Orm.*:** Veri tabanı erişiminde kullanılacak olan Object Relational Mapping altyapısını sağlayan projelerdir.
- 9. OBA.Core.Saga:** SAGA altyapısını içerir. Geliştiriciler nuget paketi olarak bağladıktan sonra birkaç satırlık extension metodu çağırımı marifetiyle kullanacaklardır.
- 10. OBA.Core.Telemetrics.Jaeger:** Distributed tracing amaçlı geliştirmelerimizi içerir. Geliştiricilerin direk kullanımına açık değildir.
- 11. OBA.Core.Telemetrics.Utilities:** Utility kütüphanesidir.
- 12. OBA.Core.WorkerService:** Scheduled task altyapı geliştirmelerimizi içerir.

5.4.9.3. Solution/proje isimlendirme standartları

Banka bünyesinde OBA altyapısı üzerinde koşacak bankacılık mikro servisi geliştirecek yazılımcıların oluşturacağı Visual Studio Solution'larının ve projelerinin isimlendirilmesinde şu standartların kullanımı uygun görülmüştür:

1. Her Visual Studio Solution'ı bir GIT reposunda tutulacaktır (Solution = Repo)
2. Bir BFF implementasyonu barındıran solution kanal bazlı isimlendirilir (Örnek: OBA.Mobile.BFF):

OBA.[Channel].BFF (sln) :

- OBA.[Channel].BFF.Api (proje)
- Docker-Compose

3. Bir mikro servis implementasyonunu barındıran bir solution OBA.[Module].{Submodule}.Service şeklinde isimlendirilir. Submodule adı opsiyonel olup Module adı zorunludur. Solution içindeki projeler aşağıdaki gibi bir nizam çerçevesinde isimlendirilir:

OBA.[Module].{Submodule}.Service (sln) :

- OBA.[Module].{Submodule}.Service.Domain (proje)
- OBA.[Module].{Submodule}.Service.Infrastructure (proje)
- OBA.[Module].{Submodule}.Service.Api (proje)
- Docker-Compose

- Tests/OBA.[Module].{Submodule}.Service.UnitTest
- Tests/OBA.[Module].{Submodule}.Service.IntegrationTest

4. Bir utility implementasyonu barındıran bir solution OBA.[Module].Utility şeklinde isimlendirilir. Solution içindeki projeler aşağıdaki gibi bir nizam çerçevesinde isimlendirilir:

OBA.[Module].Utility (sln) :

- **OBA.**[Module].Utility (proje) (nuget paketlencek)
- Tests/ **OBA.**[Module].Utility.UnitTest
- Tests/ **OBA.**[Module].Utility.IntegrationTest

5. Bir WorkerService implementasyonu barındıran bir solution OBA.[Module].{Submodule}.WorkerService şeklinde isimlendirilir. Submodule adı opsiyonel olup Module adı zorunludur. Solution içindeki projeler aşağıdaki gibi bir nizam çerçevesinde isimlendirilir:

OBA.[Module].{Submodule}.WorkerService (sln) :

- **OBA.**[Module].Utility (proje) (nuget paketlencek)
- Tests/ **OBA.**[Module].Utility.UnitTest
- Tests/ **OBA.**[Module].Utility.IntegrationTest

5.5. Proof of Concept (PoC) Çalışmaları

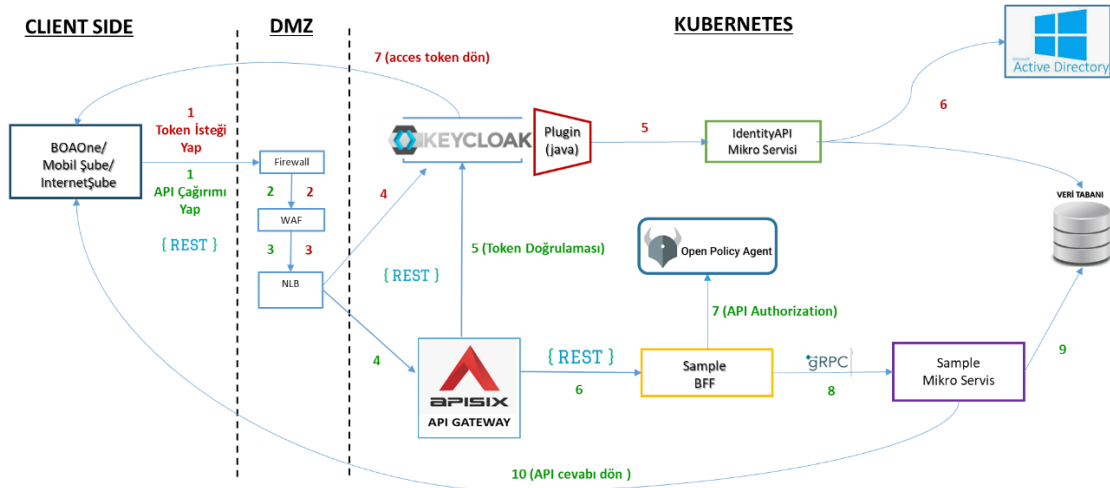
‘Kavram Kanıtı’, olarak ifade edilebilecek olan Proof of Concept (PoC) kavramı bir fikrin veya teoremin gerçek dünyada çalışabileceğini göstermek amacıyla kullanılan uygulamalı bir yöntemdir. Bu bölümde OBA altyapısını oluşturan bileşenlerin bir araya getirilerek uçtan uca test edildiği PoC çalışmalarımız hakkında bilgiler verilmektedir. Bu kapsamda önceki bölümde Şekil 5.24’te betimlenen genel OBA mimarisini gerçekleyen PoC çalışmaları yapılmış ve Kubernetes ortamında yaygınlaştırılmıştır (Rancher distribution’ı üzerinde). Bu PoC çalışmaları sırasıyla şu üç başlık altında ele alınmış olup takip eden alt bölümlerde bu PoC çalışmalarının içerikleri açıklanmaktadır:

- i. Mobil şube canlandırılmalı PoC çalışması
- ii. Mikro Servisler Arası Asenkron (Publish/Subscribe) İletişimi Tesis Eden PoC Çalışması

5.5.1. Mobil şube canlandırılmalı PoC çalışması

Bu PoC çalışmasında mobil şube tarafından yapılan güncel döviz kuru listeleme API çağırımının canlandırması yapılmıştır. Döviz kuru Listeleme API'si HTTP GET metodu kullanılarak çağırılan bir API'dir.

Bu PoC çalışmamız kapsamında Keycloak'un Kuveyt Türk hesap hamilleri (yani müşterileri) ile alakalı kullanıcı datalarını veri tabanından çekebilmesi için Java dilinde bir plugin bileşeni yazılmıştır ve Keycloak'un bünyesine bir statik Java kütüphanesi olarak dahil edilmiştir. Bu plugin Şekil 5.27'de Keycloak'un hemen sağına bitişik olan bir yamuk şekli içerisinde temsil edilmektedir. Keycloak, bu plugin'i bir tür REST client gibi kullanmak suretiyle yine kendi geliştirdiğimiz bir mikro servis uygulaması olan Identity API uygulamasına karşı REST çağırımları yapmaktadır. Identity API, plugin'den aldığı REST çağırımları kapsamında veri tabanından kullanıcı verilerini çekerek plugin'e ve dolayısıyla Keycloak'a dönmektedir. Keycloak, bu verilerle mobil şubeden aldığı bazı user-credential parametrelerini karşılaştırmak suretiyle mobil şubeye bir access-token dönüp dönemeyeceğine karar vermektedir. Keycloak'un karşılaştırması başarılı olursa mobil şubeye bir access-token dönlümlenmektedir. Bu token daha sonra mobil şube tarafından API çağırımı yaparken kullanılacaktır.



Şekil 5.27. Mobil Şube Canlandırılmalı PoC

Ayrıca OBA minvalinde gerçekleştirdiğimiz altyapı geliştirmelerimizi içeren bazı altyapı kütüphaneleri (yani OBA Core kütüphaneleri) oluşturmuştur. Detayları bölüm 5.4.9.2'de açıklanan bu kütüphaneler NuGet paketi olarak paketlenerek bu PoC

çalışmamız kapsamında kullanılmış ve bir adet örnek BFF ve mikro servis geliştirilmiştir. Yazılım geliştiriciler için örnek teşkil etmeleri bakımından bu BFF ve servise sırasıyla *Sample-BFF* ve *Sample-Mikro-Servis* olarak adlandırılmıştır. Sample-BFF ve Sample-Mikro-Servis Şekil 5.27'deki betimlemede sırasıyla sarı ve eflatun kutucuklar içerisinde vurgulanmaktadır.

Bu PoC kapsamında gerçekleşen bileşen etkileşimlerinin açıklamaları aşağıda 5.5.5.1 ve 5.5.5.2 başlıklarında adım adım açıklanmaktadır. Bu adımlar Şekil 5.27'de görülen kırmızı ve yeşil renklerle numaralandırılmış aksiyon kollarını kullanarak takip edilebilir. Kırmızı aksiyon kolları mobil şubenin Keycloak authentication'ı kapsamında yaptığı access-token alma çağırımlarını temsil ederken; yeşil renkli numaralar API Gateway'e karşı yapılan API çağırımlarını temsil etmektedir.

5.5.1.1. Mobil şube authentication adımları

Aşağıda listelenen adımlar, Şekil 5.27'de kırmızı renkli aksiyon numaraları ile vurgulanan ve mobil şubenin Keycloak authentication'ı kapsamında yaptığı access-token alma çağırımları kapsamında gerçekleştirilen aksiyonlardır:

1. Kırmızı renkte vurgulanan 1 nolu aksiyon adımında mobil şube keycloak'a doğru bir token talebinde bulunur. Bu talep aslında bir REST çağırımıdır ve DMZ katmanındaki Firewall'a ulaşır.
2. Request firewall'dan WAF'a iletilir
3. Request WAF'tan NLB'ye iletilir.
4. NLB request'i Keycloak'a ulaştırır.
5. Keycloak, plugin bileşenini kullanarak IdentityAPI'a kullanıcı verilerini çekmek için bir REST request'i gönderir.
6. IdentityAPI mikro servisi, banka veri tabanından kullanıcı verilerini çeker ve plugin'e ve dolayısıyla keycloak'a bir REST response'ı olarak döner.
7. Keycloak mobil şubeden aldığı bazı OAuth protokolü özelindeki parametrelerle Identity API'dan dönen kullanıcı verilerini karşılaştırır ve bir token verip vermeme kararı alır. Bu senorya kapsamında karşılaştırmanın olumlu sonuçlandığını varsayarsak Keycloak mobil şubeye bir access token döner.

5.5.1.2. Mobil şubenin döviz kuru API'sini çağırma adımları

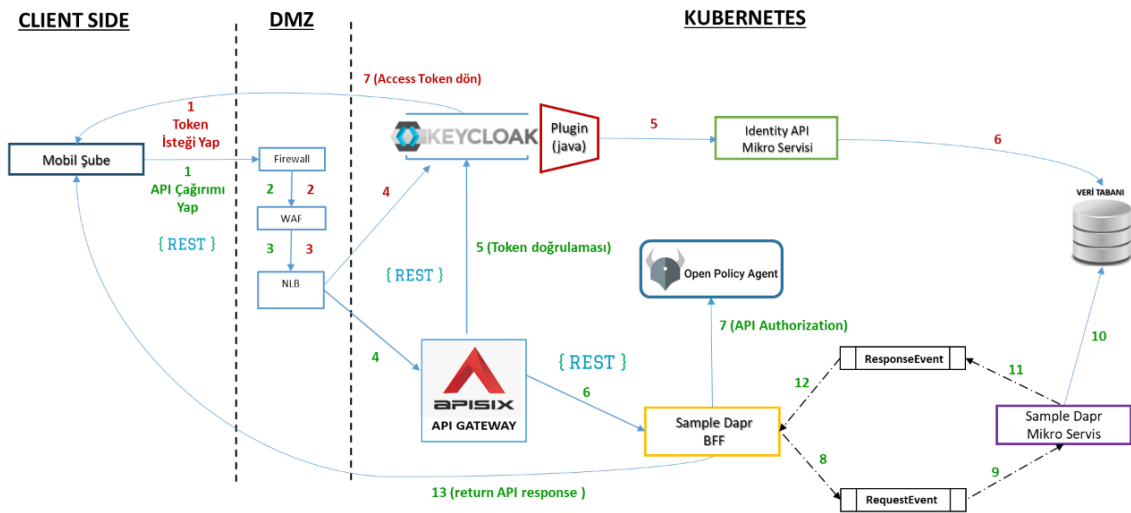
Aşağıda listelenen adımlar, Şekil 5.27'de yeşil renkli aksiyon numaraları ile vurgulanan ve mobil şubenin Keycloak authentication'ı kapsamında API Gateway'e karşı yapılan API çağırımları kapsamında gerçekleştirilen aksiyonlardır:

1. Yeşil renkte vurgulanan 1 nolu aksiyon adımı mobil şube, güncel kur listesini çekmek için APIGateway'e (yani APISIX'e) doğru REST tabanlı bir API request'i gönderir (API request'inin header'ında daha önce keycloak'tan alınan access-token'ı yerleştirmiş olarak). Bu çağırım ilk olarak DMZ katmanındaki Firewall'a ulaşır.
2. API Request'i Firewall'dan WAF'a iletilir
3. API Request'i WAF'tan NLB'ye iletilir.
4. API Request'i NLB'den çıkıp APISIX'e ulaşır.
5. APISIX, API requestinin header'ından access-token'ı çeker ve keycloak'a doğru bir token validasyon request'i gönderir (REST çağırımı yaparak). Keycloak bu token'ı mobil şubeye 7 no'lu kırmızı adımda vermişti. Keycloak bu tokenı doğruladığına dair (yani authentication yaptığına dair) bir REST response'ını APISIX'e döner.
6. APISIX, API Request'ini ilgili BFF'e (bu durumda Sample-BFF'e) sevk eder.
7. Sample-BFF bu aşamada mobil şubeyi kullanan Kuveyt Türk müşterisinin kur listeleme API çağırımını yapmaya yetkili olup olmadığını OPA'ya karşı kontrol eder (yani API authorization'ı yapar).
8. Sample-BFF, güncel kur listesinin oluşturulmasından sorumlu mikro servis'e (bu senaryoda sample mikroservis) bir gRPC request'i gönderir.
9. Sample mikro servis kurları veri tabanına sorgu atarak oluşturur. Sample-mikro-servis oluşturduğu kur listesini bir gRPC response mesajına paketleyerek sample-BFF'e döner. Sample BFF gRPC response'ını REST response'ına dönüştürür ve APISIX'e döner.
10. APISIX, API response'ını mobil şubeye dönmeden önce bu response'ın içeriğini (body ve header alanlarını) birinci adımda mobil şubeden aldığı orijinal API requesti'nin içeriği ile birleştirip JSON formatında ham bir jurnal kaydı oluşturur ve daha evvel 5.4.7 numaralı alt başlık altında anlatıldığı şekilde Elasticsearch'te kayıt altına alır.
11. APISIX, API response'ını (cevabını) mobil şubeye döner.

5.5.2. Mikro servisler arası asenkron (publish/subscribe) iletişimi tesis eden PoC çalışması

Bu PoC kapsamında önceki bölümdekine benzer şekilde mobil şube canlandırılmalı olarak kâr payı hesaplama API'sinin çağırımı örneklenmiştir. Diğer yandan diğer PoC çalışmalarından farklı olarak BFF ile mikro servis arasında Apache Kafka Message Bus kullanarak asenkron bir publish/subscribe iletişiminin tesisi gerçekleştirilmiştir. Söz konusu asenkron iletişimin tesis edilmesinde Distributed Application Runtime²⁴ (DAPR) olarak adlandırılan ve mikro servisler arası iletişim kapsamında yazılım geliştirmeyi kolaylaştıran bir yapının kullanımı benimsenmiştir. Bu bağlamda diğer PoC'lerden farklı olarak Sample-Dapr-BFF ve Sample-Dapr-Micro-Service adlarında bir BFF ve mikro servis uygulaması geliştirilmiş olup bu uygulamalar OBA kapsamında asenkron iletişimin nasıl tesis edilebileceğini örneklemektedirler.

Bu PoC kapsamında gerçekleşen bileşen etkileşimlerinin açıklaması aşağıda 5.5.2.1 ve 5.5.2.2 başlıkları altında adım adım belirtilmektedir. Bu adımlar Şekil 5.28'de görülen kırmızı ve yeşil renklerle numaralandırılmış aksiyon kollarını kullanarak takip edilebilir. Kırmızı aksiyon kolları mobil şube ekranının Keycloak authentication'ı kapsamında yaptığı access-token alma çağırımlarını temsil ederken; yeşil renkli numaralar API Gateway'e karşı yapılan API çağırımlarını temsil etmektedir.



Şekil 5.28. Asenkron İletişimi Örnekleyen PoC

²⁴ <https://dapr.io/>

5.5.2.1. Mobil şube authentication adımları

Aşağıda listelenen adımlar, Şekil 5.28’de kırmızı renkli aksiyon numaraları ile vurgulanan ve mobil şubenin Keycloak authentication’ı kapsamında yaptığı access-token alma çağırımları kapsamında gerçekleştirilen aksiyonlardır:

1. Kırmızı renkte vurgulanan 1 nolu aksiyon adımında mobil şube keycloak’a doğru bir token talebinde bulunur. Bu talep aslında bir REST çağırımıdır ve DMZ katmanındaki Firewall’a ulaşır.
2. Request firewall’dan WAF’a iletilir
3. Request WAF’tan NLB’ye iletilir.
4. NLB request’i Keycloak’a ulaştırır.
5. Keycloak, plugin bileşenini kullanarak Identity API mikro servisine kullanıcı verilerini çekmek için bir REST request’i gönderir.
6. IdentityAPI mikro servisi, banka veri tabanından kullanıcı verilerini çeker ve plugin’e ve dolayısıyla keycloak’a bir REST response’ı olarak döner.
7. Keycloak mobil şubeden aldığı bazı OAuth protokolü özelindeki parametrelerle Identity API’den dönen kullanıcı verilerini karşılaştırır ve bir token verip vermeme kararı alır. Bu senorya kapsamında karşılaştırmanın olumlu sonuçlandığını varsayarsak Keycloak mobil şubeye bir access-token döner.

5.5.2.2. Mobil şubenin döviz kuru API’sini çağırma adımları

Aşağıda listelenen adımlar, Şekil 5.28’de yeşil renkli aksiyon numaraları ile vurgulanan ve mobil şubenin Keycloak authentication’ı kapsamında API Gateway’e karşı yapılan API çağırımları kapsamında gerçekleştirilen aksiyonlardır:

1. Yeşil renkte vurgulanan 1 nolu aksiyon adımında mobil şube, kâr payı hesaplaması için APIGateway’e (yani APISIX’e) doğru REST tabanlı bir API request’i gönderir (API request’inin header’ında daha önce keycloak’tan alınan access-token’ı yerleştirmiş olarak). Bu çağırım ilk olarak DMZ katmanındaki Firewall’a ulaşır.
2. API Request’i Firewall’dan WAF’a iletilir
3. API Request’i WAF’tan NLB’ye iletilir.
4. API Request’i NLB’den çıkıp APISIX’e ulaşır.

5. APISIX, API requestinin header'ından access-token'ı çeker ve kecloak'a doğru bir token validasyon request'i gönderir (REST çağırımı yaparak). Keycloak bu token'ı mobil şubeye 7 no'lu kırmızı adımda vermişti. Keycloak bu token'ı doğruladığına dair (yani authentication yaptığına dair) bir REST response'ını APISIX'e döner.
6. APISIX, API Request'ini ilgili BFF'e (bu durumda Sample-Dapr-BFF'e) sevk eder.
7. Sample-Dapr-BFF bu aşamada mobil şubeyi kullanan Kuveyt Türk müşterisinin kâr payı hesaplama API çağırımını yapmaya yetkili olup olmadığını OPA'ya karşı kontrol eder (yani API authorization'ı yapar).
8. Sample-Dapr-BFF, güncel kâr payı hesaplaması için kafka message bus'a bir request event'i publish eder.
9. Kâr payı hesaplamasından sorumlu mikro servis (bu senaryoda sample-dapr-microservice), kafka message bus'tan 8. adımda gönderilen request event'i alır.
10. Sample-dapr-micro-service, veri tabanına sorgu atarak kâr payını hesaplar.
11. Sample-dapr-micro-service kâr payı hesabı sonucunu bir response-event mesajına paketleyerek yine kafka message bus'a bırakır.
12. Sample-Dapr-BFF, kafka message bus'tan response-event mesajını alır ve bunu bir REST response'ına paketleyerek APISIX'e döner.
13. APISIX, BFF'ten dönen response'ı mobil şubeye dönmeden önce bu response'ın içeriğini (body ve header alanlarını) birinci adımda mobil şubeden aldığı orijinal API requesti'nin içeriği ile birleştirip JSON formatında bir ham jurnal kaydı oluşturur ve daha evvel 5.4.7 numaralı altbaşlıkta anlatıldığı şekilde Elasticsearch'e kaydeder.
14. APISIX API response'ını mobil şubeye döner.

5.6. Yük Testi Çalışmaları

Mikro servis mimarisi temelli OBA altyapısı ile Kuveyt Türk Bankasında halihazırda kullanılan monolit BOA altyapısını performans verimliliği bakımından karşılaştırmak amacıyla her iki alt yapıya da yük testleri uygulanmış ve bu minvalde iki altyapı karşılaştırılmıştır. Bu bölümde bu karşılaştırmanın zeminin nasıl teşkil edildiği ve sonuçları açıklanmaktadır (Aydemir ve Başçiftçi, 2022 (1)).

Yük testlerini gerçekleştirirken Kuveyt Türk Bankası tarafından sağlanan iki ayrı test cluster'ı kullanılmıştır. Birinci test cluster'ı monolit BOA altyapısını barındırırken; ikinci cluster ise Şekil 5.27'de görülen mobil şube canlandırılmalı PoC çalışması kapsamında ortaya konulan OBA prototip altyapısını barındıran bir Kubernetes²⁵ cluster'ıdır. Bu Kubernetes cluster'ı hususen banka veri merkezinde kurulu olan ve Rancher²⁶ adlı Kubernetes varyantıdır. Her ikisi test cluster'ına da belirli hacimlerde yük tatbik edilerek ortalama istem işleme hacmi (average request throughput) ve ortalama cevap süresi (average response time) ölçülmüş ve bu kapsamda BOA ve OBA arasında bir performans karşılaştırması yapılmıştır.

Bu bölümün geri kalanında şu konular ele alınmıştır:

i. Monolit BOA altyapısının konumlandırıldığı test cluster'ının genel yapısı:

- a. Monolit BOA bileşenlerinin test cluster'ındaki genel yapılanması**
- b. Test cluster'ının bütününde kullanımı gerçekleşen işlemci ve bellek miktarları**

ii. OBA prototipinin yaygınlaştırıldığı Kubernetes cluster'ının genel yapısı:

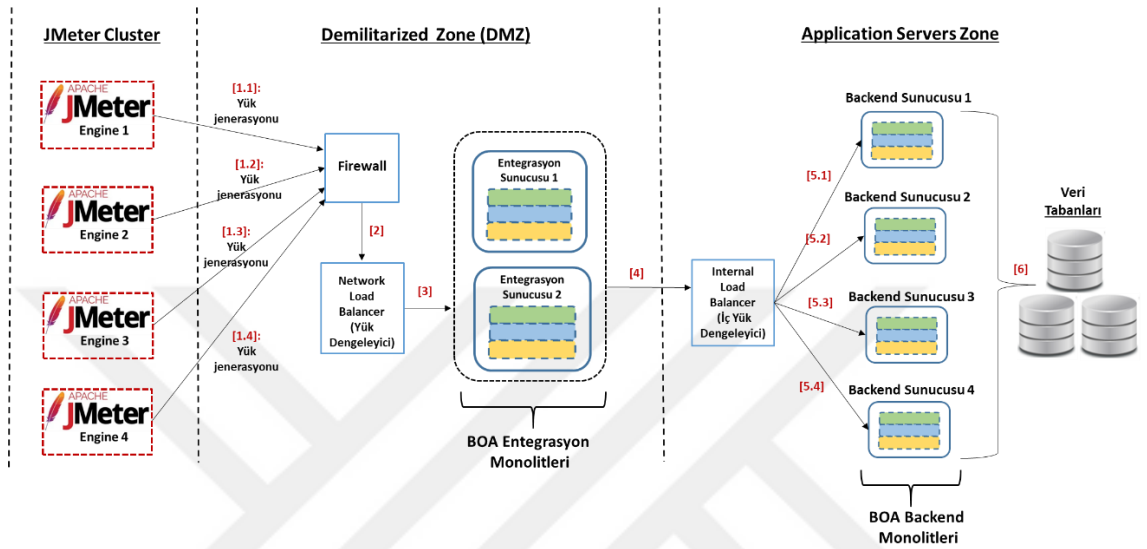
- a. Aşağıda listelenen OBA bileşenlerine dedike edilen donanım kapasiteleri:**
 - APISIX
 - Keycloak
 - OPA
 - Sample-BFF
 - Sample-Service
 - Vault ve Consul
 - b. Kubernetes cluster'ının bütününde kullanımı gerçekleşen işlemci ve bellek miktarları**
 - c. Jurnal altyapısına ayrılan donanım kapasitesi**
- iii. OBA üzerinde icra edilen yük testi konfigürasyonu**

²⁵ <https://kubernetes.io/>

²⁶ <https://www.rancher.com/>

5.6.1. Monolit BOA altyapısını barındıran test cluster'ının genel yapısı

Monolit BOA altyapısının yük testlerini gerçekleştirmek için Kuveyt Türk Bankası veri merkezinde halihazırda mevcut olan bir test cluster'ının kullanımı sağlanmıştır. Bu test cluster'ının genel yapısı aşağıda Şekil 5.29'da görülmektedir.



Şekil 5.29. BOA Test Cluster'ının Genel Yapılanması

Şekil 5.29 incelendiğinde en soldaki bölümde sanal API çağırımlarını üretmek amacıyla kullandığımız 4 adet JMeter²⁷ jeneratörü (JMeter Engine) görülmektedir. Bu 4 JMeter jeneratörün her biri bir sanal sunucu içinde konumlandırılmış olup birer yük jeneratörü işlevi görmektedirler. Burada yük ile ifade edilen şey birim zamanda jener edilen sanal kullanıcı (virtual user) sayısı olup her sanal kullanıcı birim zamanda birden fazla sanal API çağırımını (yani API istemini) jener edebilmektedir. JMeter cluster'ındaki 4 sanal makinenin her birine 16 core ve 32 GB bellek tahsisi yapılmıştır. Bu sanal makineler ortadaki DMZ bölümündeki sunuculara 5 gigabit ethernet bağlantısı ile bağlı durumdadırlar.

Şekil 5.29'da ortadaki DMZ bölümünde iki adet entegrasyon sunucusu görülmektedir. Bu entegrasyon sunucularının her birinde bir adet BOA entegrasyon monoliti koşturmaktadır. Diğer yandan şeklin sağ yanında ise 4 adet BOA backend sunucusu görülmektedir. Bu backend sunucularında birbirine DLL ve veri tabanı saklı yordamı

²⁷ <https://jmeter.apache.org/>

(stored procedure) kullanımı bakımından sıkı sıkıya bağımlı BOA monolitleri koşturmaktadır. Gerek entegrasyon sunucuları gerekse backend sunucuları daha önce bölüm 5.2.3.1’de BOA kapsamında açıklanan “*Her bir Host’ta Birden Çok Uygulama Barındırma Deployment Stratejisi*” adlı yaygınlaştırma stratejisinin tatbik edildiği sanal sunucular olup üzerlerinde birden fazla monolit BOA servisi koşturmaktadır. Burada bahsedilen 2 adet entegrasyon ve 4 adet backend sanal sunucusunun her birinin haiz olduğu kapasite Tablo 5.1’de görülmektedir. Bu tablo incelendiğinde BOA alt yapısını barındıran bu test cluster’ında toplamda 96 core (16 core x 6 SM) ve 96 GB bellek (16GB x 6 SM) boyutunda kaynak kullanımı yapıldığı sonucuna ulaşılmaktadır.

Tablo 5.1. Bir BOA Monolitinin Koştığı bir Sanal Makinanın Haiz Olduğu Konfigürasyon

Kaynak Tipi	Kapasite
Sanal İşlemci (vCPU)	16 Core
Bellek	16 GB
Disk Boyutu	24 GB SSD

Genel özellikleri yukarıdaki paragraflarda açıklanan bu BOA test cluster’ındaki BOA altyapısına JMeter jeneratörleri vasıtasıyla sırasıyla 100, 300, 600, 1000, 2500 ve 5000 sanal kullanıcı (Virtual User-VU) ile 5 dakika boyunca 6 farklı test senaryosu icra edilmiştir. Bu test senaryolarında, güncel döviz kurları listesini (Örnek: USD, EUR kurları) temin eden bir BOA bankacılık işlevi çalıştırılmıştır. Bu 6 test senaryosunun konfigürasyonu Tablo 5.2’de özetlenmektedir.

Tablo 5.2. BOA’ya Uygulanan 6 Ayrı Yük Testi Koşumlarının Kapsamı

Test Senaryo No	Sanal Kullanıcıların Sayısı	Test Süresi (dakika)	Maksimum Kullanıcı Adedine Yükselme Süresi – Ramp up Time (dakika)
1	100	5	1
2	300	5	1
3	600	5	1
4	1000	5	1
5	2500	5	1
6	5000	5	1

Tablo 5.2’de gösterilen her bir test senaryosu 1 dakikalık bir yükselme süresi ile başlatılmıştır. Buradaki yükselme süresi, toplam sanal kullanıcı sayısının maksimum sayıya ulaşması için geçen süreyi ifade eder. Örneğin 600 sanal kullanıcıyla 3. test

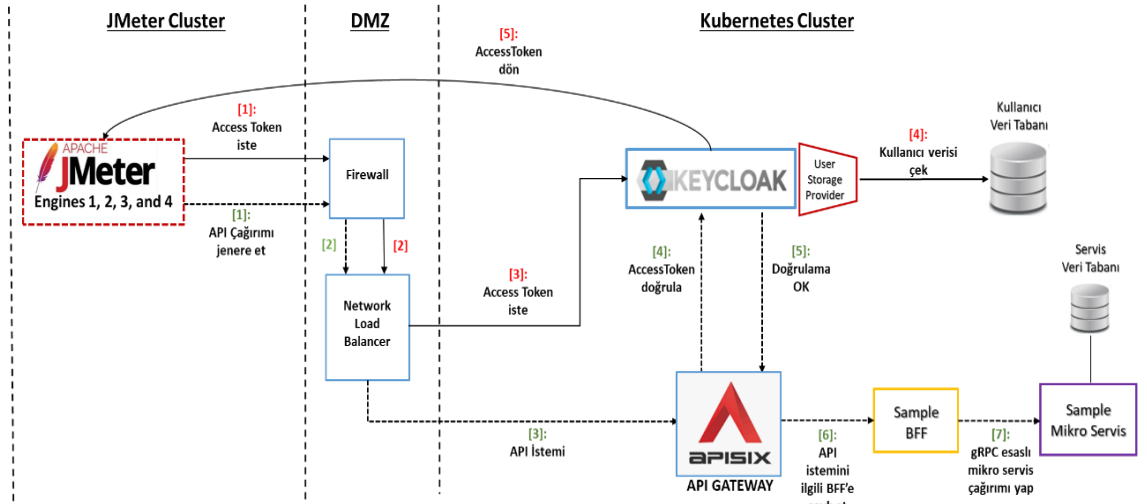
senaryosunda 600 sanal kullanıcının tamamının çağırım yapmaya başlaması 1 dakika sürmüştür.

Buna ek olarak, yükselme döneminde sanal kullanıcı sayısı üç adımda artmıştır. Örneğin, 3 numaralı test durumunda, testin yürütülmesinin ilk 20 saniyesinde yalnızca ilk 200 sanal kullanıcı başlatılmıştır. Daha sonra, sanal kullanıcı sayısı 20. ve 40. saniyeler arasında 400'e yükseltilmiş ve son olarak 40. ve 60. saniyeler arasında maksimuma (yani 600'e) yükseltilmiştir. Gerçek dünyada talep sayısının hemen maksimuma ulaşmadığı (örneğin, bankanın mobil uygulamasını kullanan bankacılık kullanıcılarının sayısının aniden değil, kademeli olarak arttığı) göz önünde bulundurulduğunda bu yapı testleri gerçeğe oldukça yakın hale getirmek için kasıtlı olarak kurulmuştur.

5.6.2. Mikro servis mimarisi esaslı OBA altyapısını barındıran test cluster'ının genel yapısı

OBA altyapısını test ettiğimiz cluster'ın yapılandırılmasında bölüm 5.3.3'te açıklanmış olan konteyner (container) teknolojisi tabanlı yaygınlaştırma stratejisi kullanılmıştır. Bu bağlamda OBA bileşenleri olan Keycloak, APISIX, BFF'ler ve mikro servisler Kuveyt Türk Bankası tarafından sağlanan bir Kubernetes cluster'ındaki POD'lar üzerinde koşan konteynerler içerisinde koşacak şekilde yaygınlaştırılmıştır (deploy edilmiştir).

Daha önce de belirtildiği gibi, Kubernetes terminolojisinde POD, depolama ve ağ kaynaklarını paylaşımlı olarak kullanan bir veya daha fazla konteyneri barındıran mantıksal bir yaygınlaştırma birimi olarak tasavvur edilebilir. Kubernetes cluster'ında Keycloak, APISIX, BFF'ler ve mikro servislerin her biri bir veya daha fazla POD üzerinde koşacak şekilde konumlandırılmıştır. Dolayısıyla bir OBA bileşenine hizmet eden POD'un sanal donanım kaynakları (Örnek: işlemci (core) sayısı, bellek boyutu) artan API çağırımları nedeniyle yükü kaldıramaz hale gelirse Kubernetes aynı OBA bileşenini barındıran başka özdeş bir ya da daha fazla POD'u cluster'a eklemek suretiyle altyapısını yatay olarak ölçeklendirmektedir (yani genişletmektedir). Bu suretle sistemin performansının sürekliliği sağlanmış olmaktadır.



Şekil 5.30. OBA Altyapısının Test Edildiği Test Cluster'ının Genel Yapılanması

Yukarıda verilen bilgiler ışığında OBA için banka bünyesinde konumlandırılan Kubernetes test cluster'ının genel yapısı Şekil 5.30'da görüldüğü gibidir. Burada en soldaki bölümde konumlandırılan JMeter jeneratörleri (her biri 16 core ve 32 GB bellekli) Keycloak'tan bir kereye mahsus acces-token almakta ve bu token ile 5 gigabit Ethernet bağlantısı üzerinden APISIX'e karşı API request (istem) bombardımanı yapmaktadırlar. Burada en sağdaki Kubernetes cluster'ı bölümünde mobil şube canlandırılmalı PoC çalışmamız kapsamında geliştirdiğimiz Sample-BFF ve Sample-Mikro-Servis görülmektedir. Dolayısıyla APISIX, JMeter jeneratörlerinden kendisine ulaşan API çağırımları neticesinde oluşan trafiği POD'lar içerisinde koşan Sample-BFF'e ve oradan da Sample-Mikro-Servise doğru sevk etmektedir. Burada Sample-BFF ile OPA arasındaki iletişim şekli sade tutmak adına açıkça belirtilmemiş olmasına rağmen gerçekte mevcut durumdadır. Bu yapı bize OBA altyapısının bütününe karşı yük testi uygulamamıza ve dolayısıyla OBA'nın genel performansını önceki bölümde açıklanan BOA test cluster'ındaki BOA alt yapısı ile karşılaştırmaya imkân tanımıştır.

Şekil 5.30'da en sağda konumlandırılan Kubernetes cluster'ında birer konteyner içerisinde paketlenerek POD'lar üzerinde koşturulan OBA bileşenlerinin (Keycloak, APISIX, Sample-BFF ve Sample-Mikro-Servis) her biri için bir veya daha fazla POD tahsisi yapılmış durumdadır. Bu POD'ların her birine ayrılmış donanım kaynaklarının boyutları takip eden alt başlıklarda izah edilmektedir.

5.6.2.1. APISIX kaynak konfigürasyonu

Şekil 5.30'da OBA bileşenlerinin konumlandırıldığı Kubernetes cluster'ında, performans gereksinimleri nedeniyle API Gateway (yani APISIX) için statik bir kapasite tahsisi gerçekleştirilmiştir. Bu statik tahsis kapsamında sayısı hep sabit kalacak şekilde 10 adet Kubernetes POD'u APISIX'e tahsis edilmiştir ve burada her bir POD'da bir APISIX uygulama konteyneri koşturmaktadır.

Tablo 5.3'te bu POD'lardan her birine APISIX özelinde tahsis edilebilecek minimum ve maksimum sanal işlemci ve bellek boyutları görülmektedir. POD'lar ilk oluşturulduğunda minimum kaynak tahsisiyle çalışmaya başlarlar. Bununla birlikte Kubernetes, bu 10 özdeş POD'un her biri özelindeki kaynak kullanımlarını düzenli olarak gözlemleyerek tahsis edilen kaynağı minimum ve maksimum kapasite aralığını geçmeyecek şekilde artırıp azaltabilir. Örneğin bir APISIX POD'u özelinde vCPU kullanımı ortalama 600m²⁸; bellek kullanımı da 700Mi²⁹ miktarını geçtiğinde Kubernetes otomatik olarak POD'un kaynaklarını 800m'lik vCPU'ya; belleğini de 1Gi'ye kadar artırabilir. Benzer şekilde ortalama kullanım azaldığında Kubernetes otomatik olarak o POD özelindeki kaynak kullanımlarını minimum düzeye indirebilir. Bu davranış Kubernetes'le birlikte gelen kaynakların daha verimli kullanımı amaçlı bir davranıştır.

Tablo 5.3. Tek bir APISIX POD'unun Kapasite Konfigürasyonu

Kaynak Türü	Minimum Kapasite	Maksimum Kapasite
Sanal İşlemci (vCPU)	400m ³⁰	800m
Bellek	500Mi ³¹	1Gi ³²

²⁸ m (millicpu) terimi, Kubernetes'de atomik boyutta bir sanal işlemci (vCPU) birimini temsil eder (Örnek: 500m yarım vCPU'ya eş değer; 1000m ise 1 vCPU'ya yani 1 fiziksel Core'a eş değerdir). Daha fazla bilgi için şu kaynak incelenebilir: <https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/>

²⁹ Mi (mebibyte) terimi, Kubernetes'de atomik bir sanal bellek birimidir. Örneğin 500Mi yaklaşık 500 megabyte'lık fiziksel belleğe eş değerdir. Benzer şekilde Gi (Gilllicpu) da bir sanal bellek birimi olup daha büyük bellek boyutunu ifade etmek için kullanılmaktadır. Örneğin 1 Gi yaklaşık 1 Gigabyte'lık fiziksel belleğe eş değerdir. Daha fazla bilgi için şu kaynak incelenebilir: <https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/>

³⁰ 1000m (millicpu) aşağı yukarı 1 sanal işlemciye (vCPU) eş değerdir.

³¹ 1Mi aşağı yukarı 1 Megabyte (MB) fiziksel belleğe eş değerdir.

³² Gi (Gilllicpu) terimi, bir sanal bellek birimi olup Mi'ye göre daha büyük bellek boyutunu ifade etmek için kullanılmaktadır. Örneğin 1 Gi yaklaşık 1 Gigabyte'lık fiziksel belleğe eş değerdir. Daha fazla bilgi için şu kaynak incelenebilir: <https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/>

5.6.2.2. Keycloak kaynak konfigürasyonu

Keycloak OBA'nın kimlik denetleme (authentication) bileşeni olup aynı APISIX gibi Şekil 5.30'da görülen Kubernetes cluster'ında POD'lar üzerinde yaygınlaştırılmaktadır. Bununla birlikte APISIX'in aksine Kubernetes cluster'ında Keycloak için sabit sayıda (yani statik) bir POD tahsisi yapılmamıştır. Bunun yerine Kubernetes ilk olarak Keycloak için yalnızca 2 POD çalıştıracak şekilde konfigüre edilmiştir. Bununla birlikte Keycloak barındıran POD'lardaki ortalama sanal işlemci ve bellek kullanımlarının belirli bir eşik değerinin üstüne çıkması durumunda üzerinde Keycloak koştan toplam POD sayısı Kubernetes tarafından otomatik olarak 10 POD'a kadar çıkarılabilecek şekilde dinamik bir konfigürasyon yapılmıştır. Kubernetes cluster'ında bu eşik değeri Keycloak için 600m'lik vCPU ve 750Mi'lik bellek kullanım düzeyi olarak konfigüre edilmiştir. Dolayısıyla Keycloak'un koştığı POD'larda ortalama kullanım bu değerlerin üzerine çıktığında Kubernetes altyapısı otomatik olarak yeni bir Keycloak numunesi koştan ilave bir POD oluşturmaktadır (Maksimumda 10 POD olacak şekilde).

Diğer yandan tek bir POD özelindeki konfigürasyon göz önüne alındığında Keycloak'un kullanılabileceği maksimum ve minimum sanal işlemci ve bellek miktarı Tablo 5.4'te belirtilen aralıkta değişebilecek şekilde yapılandırılmıştır. Dolayısıyla tek bir Keycloak POD'u üzerine binen yük değiştikçe Kubernetes bu POD özelindeki sanal işlemci ve bellek tahsisini maksimum ve minimum aralığında artırıp azaltabilmektedir.

Tablo 5.4. Tek bir Keycloak POD'unun Kapasite Konfigürasyonu

Kaynak Tipi	Minimum Kapasite	Maksimum Kapasite
Sanal İşlemci (vCPU)	250m	800m
Bellek	600Mi	1Gi

5.6.2.3. Open Policy Agent (OPA) kaynak konfigürasyonu

OPA, OBA'nın API authorization bileşenidir (yani API'lere erişim yetkisi kontrolü yapan bileşendir). Kubernetes cluster'ı OPA için ilk etapta yalnızca 10 POD çalıştıracak şekilde konfigüre edilmiştir. Bununla birlikte OPA uygulamasının koştığı POD'lar ortalama sanal işlemci ve bellek kullanımlarının belirli bir eşik değerinin üstüne

çıkması durumunda toplam POD sayısı Kubernetes tarafından otomatik olarak 20 POD'a kadar çıkarılabilecek şekilde dinamik bir konfigürasyon yapılmıştır. Kubernetes cluster'ında bu eşik değer OPA için 400m'lik vCPU ve 400Mi'lik bellek kullanım düzeyi olarak konfigüre edilmiştir. Dolayısıyla OPA'nın koştığı POD'larda ortalama kullanım bu değerlerin üzerine çıktığında Kubernetes altyapısı otomatik olarak yeni bir OPA numunesi koştan ilave bir POD oluşturmaktadır (Maksimumda 20 POD olacak şekilde).

Diğer yandan tek bir POD özelindeki konfigürasyon göz önüne alındığında OPA'nın kullanılabileceği maksimum ve minimum sanal işlemci ve bellek miktarı Tablo 5.5'te belirtilen aralıkta değişebilecek şekilde yapılandırılmıştır. Dolayısıyla tek bir OPA POD'u üzerine binen yük değiştikçe Kubernetes bu POD özelindeki sanal işlemci ve bellek tahsisini maksimum ve minimum aralığında artırıp azaltabilmektedir.

Tablo 5.5. Tek bir OPA POD'unun Kapasite Konfigürasyonu

Kaynak Tipi	Minimum Kapasite	Maksimum Kapasite
Sanal İşlemci (vCPU)	250m	500m
Bellek	128Mi	512Mi

5.6.2.4. Sample-BFF için kaynak konfigürasyonu

Sample-BFF, mobil şube canlandırılmalı PoC çalışmamız kapsamında C# dilinde .NET7 runtime'ı için çalışacak şekilde geliştirdiğimiz bir BFF numunesidir. Sample-BFF, kendisine APISIX tarafından iletilen API request'leri üzerinde bir takım temel input validasyonları yaptıktan sonra OPA ile API authorization kontrolü gerçekleştirmektedir.

Kubernetes cluster'ında Sample-BFF için ilk etapta yalnızca 5 POD çalıştıracak şekilde bir konfigürasyon yapılmıştır. Bununla birlikte Sample-BFF koştan POD'lardaki ortalama sanal işlemci ve bellek kullanımlarının belirli bir eşik değerinin üstüne çıkması durumunda toplam POD sayısı Kubernetes tarafından otomatik olarak 10 POD'a kadar çıkarılabilecek şekilde dinamik bir konfigürasyon yapılmıştır. Kubernetes cluster'ında bu eşik değer Sample-BFF için 600m'lik vCPU ve 1.5Gi'lik bellek kullanım düzeyi olarak konfigüre edilmiştir. Dolayısıyla Sample-BFF'in koştığı POD'larda ortalama kullanım bu değerlerin üzerine çıktığında Kubernetes altyapısı otomatik olarak yeni bir Sample-BFF numunesi koştan ilave bir POD oluşturmaktadır (Maksimumda 10 POD olacak şekilde).

Diğer yandan tek bir POD özelindeki konfigürasyon göz önüne alındığında Sample-BFF'in kullanılabileceği maksimum ve minimum sanal işlemci ve bellek miktarı Tablo 5.6'da belirtilen aralıkta değişebilecek şekilde yapılandırılmıştır. Dolayısıyla tek bir Sample-BFF POD'u üzerine binen yük değiştikçe Kubernetes bu POD özelindeki sanal işlemci ve bellek tahsisini maksimum ve minimum aralığında artırıp azaltabilmektedir.

Tablo 5.6. Tek bir Sample-BFF POD'unun Kapasite Konfigürasyonu

Kaynak Tipi	Minimum Kapasite	Maksimum Kapasite
vCPU	200m	800m
Bellek	1Gi	2Gi

5.6.2.5. Sample-Servis için kaynak konfigürasyonu

Sample-Mikro-Servis, mobil şube canlandırılmalı PoC çalışmamız kapsamında C# dilinde .NET 7 runtime'ı için çalışacak şekilde geliştirdiğimiz bir mikro servis numunesidir. Kaynak konfigürasyon Sample-BFF ile yani Tablo 5.6 ile aynıdır.

5.6.2.6. Vault ve Consul için kaynak konfigürasyonu

Daha önce de belirtildiği gibi, Vault ve Consul, sırasıyla gizli parametreleri (ör. veri tabanı kullanıcı adı ve parolası) ve uygulamaya özgü yapılandırma parametrelerini boot (önyükleme) sırasında mikro servislere ve BFF'lere (yani, testlerimiz bağlamında burada Sample-Mikro-Servis ve Sample-BFF) besler. Bu parametreler önyükleme zamanında BFF'lere ve mikro servislere beslendiğinden, Kubernetes cluster'ında Vault ve Consul'a için sabit sayıda POD'un tahsis edilmesinin yeterli olduğu kanısına varılmıştır. Bu nedenle Kubernetes'de hem Vault hem de Consul için her POD'un 400 m vCPU ve 500 Mi bellek kapasitesine sahip olduğu 2 sabit POD'un koşaacağı bir konfigürasyon yapılmıştır.

5.6.2.7. OBA'nın test edildiği kubernetes cluster'ındaki toplam kaynak kullanımı

OBA bileşenlerinin her birinin tek bir POD özelindeki kaynak tahsislerini gösteren Tablo 5.3, 5.4, 5.5 ve 5.6'daki değerler göz önüne alındığında, OBA minvalinde gerçekleşen toplam sanal kaynak kullanımlarının özeti aşağıda Tablo 5.7'de sunulmuştur.

Bu tablo incelendiğinde, OBA bileşenleri tarafından toplam 87200m boyutunda sanal işlemci (vCPU) tüketimi gerçekleşmiştir. Burada 1000m'nin 1 tane sanal işlemciyi (vCPU) temsil ettiği düşünüldüğünde, Kubernetes test kümesinde OBA tarafından tüketilen toplam sanal işlemci sayısı yaklaşık 87 (87000m) olmaktadır. Benzer şekilde, toplam bellek kullanımı 72Gi civarındadır, bu da 72GB belleğe eşdeğerdir. Öte yandan, alt bölüm 5.5.1'e bakıldığında, monolit BOA bileşenleri tarafından yapılan toplam kaynak kullanımının ise 96 işlemci ve 96 GB'lık bellek olduğu görülmektedir.

Tablo 5.7. Kubernetes Test Cluster'ında Konumlandırılan OBA Bileşenlerinin Herbirinde Gerçekleşen Toplam Kaynak Kullanımları

OBA Bileşen Adı	Toplam POD Sayısı	Toplam Sanal İşlemci (vCPU) Sayısı	Toplam Bellek Kullanımı
APISIX	10	8000m	10Gi
Keycloak	10	8000m	10Gi
OPA	20	10000m	10240 Mi $\approx$ 10Gi
Sample-BFF	10	8000m	20Gi
Sample-Service	10	8000m	20Gi
Vault	2	800m	1Gi
Consul	2	800m	1Gi

Aşağıda Tablo 5.8'de mikro servis mimarisi esaslı OBA altyapımız ile ve monolit BOA altyapısının kaynak kullanımları özetlenmektedir. Buradan anlaşılabileceği gibi OBA gerek işlemci gerekse bellek bakımından daha az ve dolayısıyla verimli kaynak kullanımı gerçekleştirmektedir.

Tablo 5.8. OBA ve BOA'nın Toplam Kaynak Kullanımı Bakımından Karşılaştırması

Sistem Adı	Toplam İşlemci Sayısı (vCPU)	Toplam Bellek kullanımı (GB)
OBA (Mikro Servis esaslı Bankacılık Sistemi)	87	72
BOA (Monolit Bankacılık Sistemi)	96	96

5.6.3. OBA üzerinde icra edilen yük testi konfigürasyonu

Bu alt başlıkta OBA altyapısını test amaçlı oluşturulan test cluster'ı üzerinde gerçekleştirilen yük testlerinin nasıl yapılandırıldığı açıklanmaktadır. Genel yapısı Şekil 5.30'da görülen test cluster'ına karşı 6 farklı yük testi koşumu gerçekleştirilmiştir. Bu test koşumları Tablo 5.9'da belirtildiği gibi sırasıyla 100, 300, 600, 1000, 2500 ve 5000 sanal kullanıcının JMeter jeneratörlerince 5 dakika boyunca çalıştırılması yoluyla icra edilmişlerdir. Benzer test koşumları da daha önce Tablo 5.2'de belirtildiği gibi monolit BOA altyapısına uygulanmıştır.

Tablo 5.9. OBA için Uygulanan Yük Testi Koşumları

Test Koşum No	Toplam Sanal Kullanıcı (Virtual User) Adedi	Toplam Test Koşum Süresi (dakika)	Yükselme Süresi – Ramp up Time (dakika)
1	100	5	1
2	300	5	1
3	600	5	1
4	1000	5	1
5	2500	5	1
6	5000	5	1

Yukarıda Tablo 5.9'da belirtilen her bir test koşumunun başında JMeter jeneratörleri yalnızca bir kereye mahsus olarak Keycloak'tan bir access-token temin etmektedir. Bu temin Şekil 5.30'da kırmızı renkteki adımlarla vurgulanan aksiyon kollarını takip ederek gerçekleştirilmektedir. JMeter jeneratörleri Keycloak'tan token alır almaz bu token'ı oluşturdukları sanal kullanıcılara beslemektedirler. Dolayısıyla tek bir access-token aynı test koşumu sırasındaki bütün sanal kullanıcılarca ortaklaşa tekrar tekrar kullanılmaktadır. Bu durum gerçek dünyada karşımıza çıkan bir durumdur zira bir bankacılık müşterisi (yani gerçek kullanıcı) yaptığı her API çağırımı için ayrı ayrı token almamakta ve belirli bir oturum süresince aynı token'ı kullanmaktadır.

Ek olarak Tablo 5.9'da listelenen ve OBA'ya karşı uygulanan test koşumlarında daha evvel BOA test koşumlarında da uygulandığı gibi 1 dakikalık bir yükselme süresi (ramp-up period) uygulanmaktadır. Bu yükselme süresinde sanal kullanıcıların sayısı maksimuma 3 fazda ulaşmaktadır. Örneğin, 600 maksimum sanal kullanıcının koşulduğu 3 numaralı test koşumunda, testin ilk 20 saniyesinde yalnızca ilk 200 sanal kullanıcı başlatılmıştır. Ardından, sanal kullanıcı sayısı 20. ve 40. saniyeler arasında 400'e yükseltilmiş ve 40. ve 60. saniyeler arasında da maksimuma (yani 600'e) yükseltilmiştir.

Yükselme süresi ve sanal kullanıcı sayısındaki kademeli artış testlerin gerçek dünyada gerçekleşen duruma mümkün olduğunca yakın olması için kasıtlı olarak gerçekleştirilmiştir. Örneğin, bankanın mobil uygulamasını kullanan bankacılık kullanıcılarının sayısı aniden değil kademeli olarak artmaktadır. Dolayısıyla bu kademeli artış doğal bir durumdur.

5.6.4. Yük testi sonuçları

Bu bölümde, bir önceki bölümde açıklanan yük testi koşullarının sonuçlarına yer verilmektedir. Bu test koşulları bir önceki bölümde Tablo 5.2 ve Tablo 5.9'da belirtilen konfigürasyona sahiptir.

Bu bölümün ilk alt başlığında değerlendirmemiz için bir temel oluşturması amacıyla ilk olarak monolit BOA altyapısının yük testi sonuçları sunulmaktadır. Ardından, ikinci alt başlıkta MSA tabanlı açık bankacılık mimarisi (OBA) için yük testi sonuçları sunulmaktadır.

Gerek monolit BOA altyapısı olsun gerekse doktora çalışmamız kapsamında önerdiğimiz OBA altyapısı olsun yük testi sonuçlarını değerlendirirken performansı direk etkileyen üç faktör kapsamında ölçümler yapılmıştır. Bu faktörlerden birincisi '**Average Throughput (Ortalama Verim)**' denilen ölçüm olup altyapının birim zamanda gerçekleştirdiği iş olarak tasavvur edilebilir. Daha teknik bir ifadeyle tezimiz kapsamında average throughput sistemin 1 saniyede kaç tane request'i (istemi) işleyip müspet ya da menfi bir cevap döndüğünün ölçümünü ifade eder. İkinci faktör ise '**Average Response Time (Ortalama Cevap Süresi)**' olarak adlandırılan bir ölçüm olup; sistemin belirli bir zaman zarfında ortalama ne kadar sürede cevap verdiğini ifade eder ve değeri milisaniye cinsindendir. Üçüncü faktör ise '**Error Rate (Hata Oranı)**' olup sistemin belirli bir zaman zarfında yüzdelik ölçekte verdiği hatalı cevap sayısını ifade eder.

5.6.4.1. Monolit BOA altyapısına uygulanan yük testi sonuçları

Monolit BOA altyapısı üzerinde yük testleri gerçekleştirmek için Kuveyt Türk Bankası veri merkezinde halihazırda mevcut olan bir test cluster'ının kullanımı sağlanmıştır. Genel yapısı bölüm 5.6.1'de izah olunan bu test cluster'ı üzerinde 6 farklı test koşumu gerçekleştirilmiş olup bu test koşulları ile ilgili sonuçların genel özeti aşağıda Tablo 5.10'da sunulmaktadır:

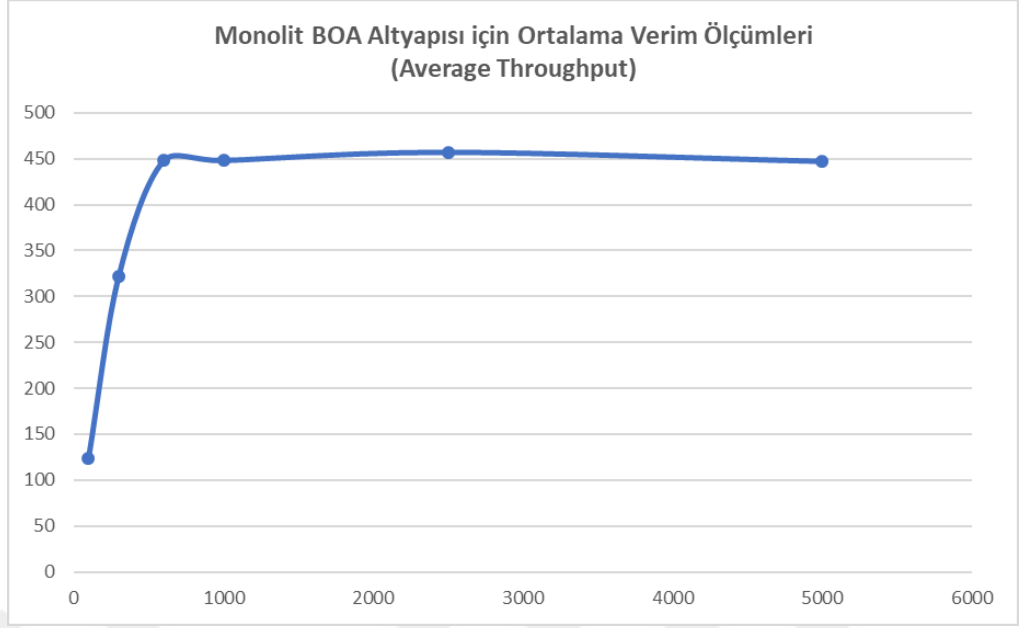
Tablo 5.10. Monolit BOA Altyapısına Tatbik Edilen Yk Testi Sonuları

Test Koşum No	Sanal Kullanıcı Adedi	5 Dakikalık Test Koşumunda İşlenen Toplam Request (istem) adedi	Ortalama Verim (Average Throughput) (hits/s)	Ortalama Cevap Süresi (Average Response Time) (milisaniye)	Hata Oranı Yüzdesi
1	100	52,462	123,66	33,41	0
2	300	121,487	321,9	125,5	0
3	600	180,037	447,84	420,67	0
4	1000	182,492	448,38	1040	0
5	2500	185,652	457,27	3290	0
6	5000	181,196	447,4	7340	0,03

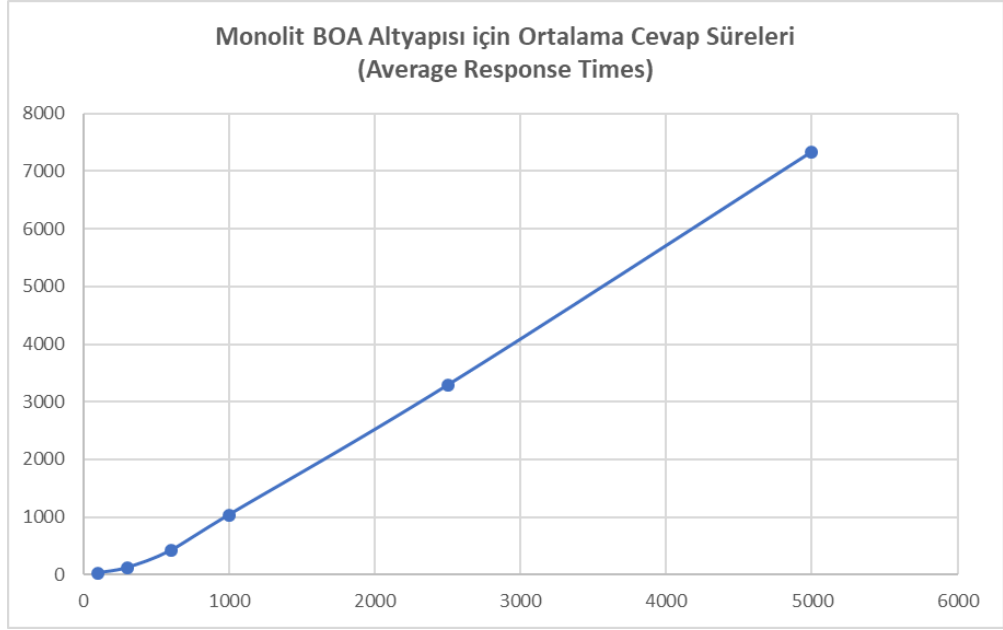
Tablo 5.10’da görlen sonular, average throughput (ortalama verim), average response time (ortalama yanıt sresi- milisaniye cinsinden) ve hata oranı yzdesi (Error Rate) olmak zere  özel faktre gre dzenlenmiřtir zira bu parametreler monolit yapının genel performans-verimliliğini yorumlamaya imkân tanımaktadır.

Bu sonulara bakıldığında, sanal kullanıcı (Virtual User- VU) sayısı arttıka BOA iin ortalama verim deėerlerinin kk adımlarla arttığı; ortalama yanıt srelerinin ise ok daha byk adımlarla arttığı gzlemlenmektedir. rneėin, sanal kullanıcı adedi 300 sanal kullanıcıdan *600 sanal kullanıcıya* ıkarken ortalama verimlilik yaklaşık *1,4 kat* artarken, *ortalama yanıt sresi* yaklaşık *20 kat artmaktadır* (yani, 46,64’ten 921,11’e ıkmaktadır).

Ayrıca, sanal kullanıcı sayısı 600’ ařtıktan sonra, ařaėıdaki řekil 5.31’de grldė gibi ortalama verim ok fazla deėiřmemiřtir. Benzer řekilde, BOA iin ortalama yanıt sreleri řekil 5.32’de grldė gibi sanal kullanıcı sayısı 600’ ařtıktan sonra srekli olarak artmaya bařlamıřtır. Bu durum, sanal kullanıcı sayısı 300’ ařtıktan sonra monolit BOA altyapısında performans darboėazları oluřmaya bařladığını gstermektedir.



Şekil 5.31. Monolit BOA Altyapısının Ortalama Verim Sonuçları³³



Şekil 5.32. Monolit BOA Altyapısının Ortalama Cevap Süreleri³⁴

³³ y eksenini ortalama verim değerlerini, x eksenini ise yük testleri sırasında paralel olarak çalıştırılan sanal kullanıcı sayısını göstermektedir

³⁴ y eksenini ortalama yanıt süresi değerlerini, x eksenini ise yük testleri sırasında paralel olarak çalıştırılan sanal kullanıcı sayısını göstermektedir.

5.6.4.2. Mikro servis mimarisi temelli OBA altyapısına uygulanan yük testi sonuçları

MSA tabanlı açık bankacılık mimarisinde (yani OBA) ortaya konulan altyapıya karşı yük testleri gerçekleştirmek için Kuveyt Türk Bankası veri merkezinde bir Kubernetes test cluster'ı tesis edilmiştir. Genel yapısı bölüm 5.6.2'de izah olunan bu test cluster'ı üzerinde 6 farklı test koşumu gerçekleştirilmiş olup bu test koşumları ile ilgili sonuçların özeti aşağıda Tablo 5.11'de sunulmaktadır:

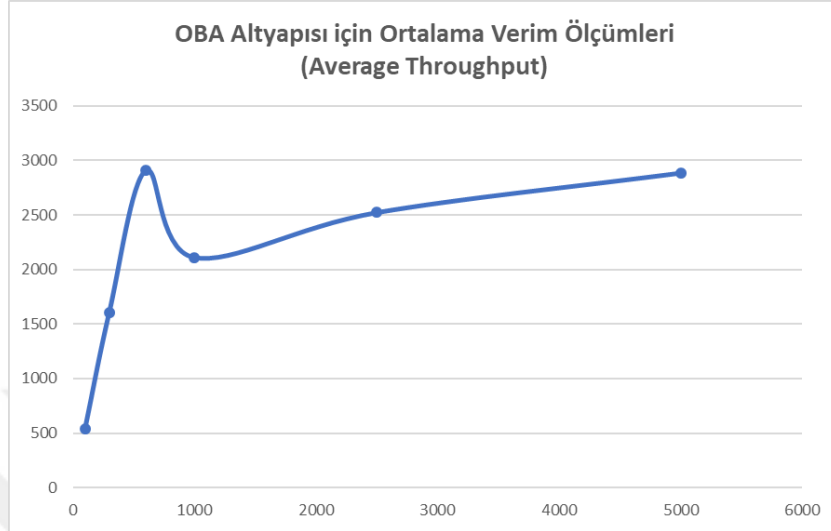
Tablo 5.11. Açık Bankacılık Mimarisi (OBA) için Yük Testi Sonuçları

Test Koşum No	Sanal Kullanıcı Adedi	5 Dakikalık Test Koşumunda İşlenen Toplam Request (istem) adedi	Ortalama Verim (Average Throughput) (İşlenen istem sayısı/saniye - hits/s)	Ortalama Cevap Süresi (Average Response Time) (milisaniye)	Hata Oranı Yüzdesi
1	100	162,875	534,55	21,85	0
2	300	481,747	1605,8	10,656	0
3	600	872,998	2909,96	20,721	0
4	1000	709,208	2112,3	51,006	0
5	2500	757,696	2524,76	135,986	0,03
6	5000	885,925	2887,92	315,804	0,03

Monolit BOA altyapısı için yapılan yük testi sonuçlarına benzer şekilde OBA ile ilgili yük testi sonuçları da ortalama verim (average throughput), ortalama yanıt süresi (average response time) ve hata oranı yüzdesi (error rate percentage) olmak üzere üç özel parametreye dayalı olarak ölçülmüştür zira bu parametreler direk olarak performans ölçücü metrikler ortaya koymaktadır. Bu kapsamda Tablo 5.10 ve 5.11'deki ortalama verim değerleri karşılaştırıldığında, MSA tabanlı mimariye (OBA'ya) ait verimliliğin monolit BOA'ya göre ortalama 5 kat daha yüksek olduğu gözlemlenmektedir. Bu bakımdan yük testlerindeki sanal kullanıcı (VU) sayısının daha fazla artırılması durumunda OBA ve BOA arasındaki ortalama verimlilik farkının daha açılarak büyüyeceği aşikardır.

Öte yandan, OBA tek başına değerlendirildiğinde, sanal kullanıcı sayısı 600'ü aşıp 1000'e doğru seyrederken, ortalama verim değerlerinin yaklaşık yüzde 28 oranında (yani Tablo 5.11'in 4. sütununda görüldüğü gibi 2909,96'den 2112,3'e) düştüğü gözlemlenmiştir. Bununla birlikte sanal kullanıcı sayısı 1000'den 2500'e doğru seyrederken ortalama verimin tekrar artmaya başladığı gözlemlenmiştir. Bu durum Kubernetes'in OBA üzerindeki yükün farkına varması ve artan yükü üstlenmek için

POD'larını ölçeklendirmesi olarak yorumlanabilir. Bir başka deyişle, sanal kullanıcı sayısı 1000'i aştıktan sonra verim tekrar artmaya başlamıştır. Bu değişim aşağıda Şekil 5.33'te ifade edilmektedir.



Şekil 5.33. Mikro Servis Mimarisi Esaslı OBA Altyapısının Ortalama Verim Sonuçları³⁵

Şekil 5.33 incelendiğinde dikkat çeken bir diğer husus ise yaklaşık 2500 sanal kullanıcıdan sonra OBA'nın verim değerlerindeki artış ivmesinin sabitlenmeye başladığı ve yavaşladığıdır. Bu durum, sanal kullanıcı sayısı 5000'den yukarıya çekilse bile maksimum verim değerlerinin çok az artacağına işaret etmektedir. Bu noktada Kubernetes cluser'ının üzerinde koştuğu fiziksel bilgisayar ağının bant genişliği performans üzerinde belirleyici bir faktör haline gelmektedir. Başka bir deyişle, 2500 adet sanal kullanıcı sayısı aşıldıktan sonra Kubernetes'in ağ iletişimini yöneten istio³⁶ service mesh altyapısı bant genişliği limitlerine yaklaşmış ve performans darboğazı oluşturmaya başlamıştır. Bu nedenle OBA altyapısının ortalama verimliliğinin daha da ivmelenerek artmaya devam etmesi için, yük testleri sırasında Kubernetes's daha fazla ağ kaynağı (yani bant genişliği) eklenmesi gerekmektedir. Bu noktada ne BFF'lerin ne de diğer mikro servislerin implementasyonunda ortalama verimi artırma kapsamında yapılacak bir şey olmadığını belirtmek uygun olacaktır zira darboğaz Kubernetes cluster'ındaki bant genişliğinin yetersizliğinden kaynaklanmaktadır.

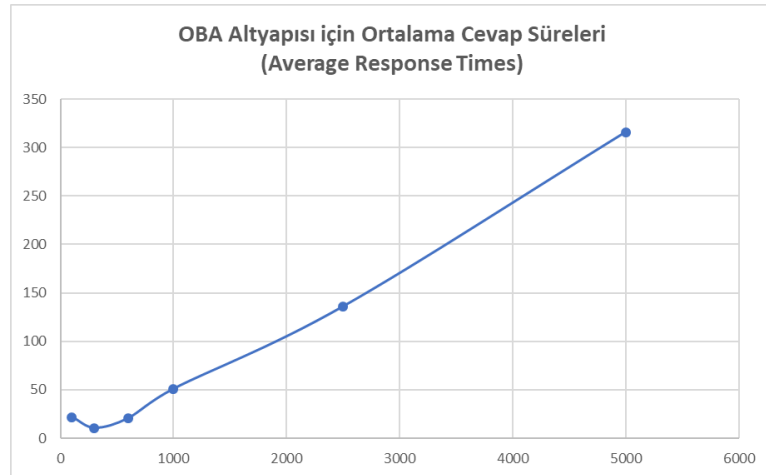
³⁵ y ekseninde ortalama verim değerlerini, x ekseninde ise yük testleri sırasında paralel olarak çalıştırılan sanal kullanıcı sayısını göstermektedir

³⁶ <https://istio.io/latest/about/service-mesh/>

Öte yandan, Tablo 5.11'deki ortalama yanıt süreleri incelendiğinde, OBA'daki ortalama yanıt süresi değerlerinin Tablo 5.10'da belirtilen monolit BOA'ya ait yanıt sürelerine göre çok daha düşük olduğu (yaklaşık 17 kat düşük olduğu) görülmektedir. Ayrıca yük testleri sırasındaki sanal kullanıcı sayısının daha da artırılması durumunda OBA ve BOA arasındaki ortalama yanıt sürelerine ilişkin bu farkın daha da açılarak artmaya devam edebileceği sonucuna varılabilir.

OBA ile ilgili için yük testi sonuçları BOA'dan bağımsız olarak tek başına değerlendirildiğinde aşağıda Şekil 5.34'te görüldüğü gibi sanal kullanıcı sayısı 600'ü aştıktan sonra OBA'nın ortalama yanıt sürelerinin (average response times) sürekli arttığı gözlemlenmektedir. Bu durum yine Kubernetes cluster'ının üzerinde koştugu fiziksel ağın bant genişliğinin sınırlarına dayanmaya başladığının ve bir performans darboğazı oluştuğunun bir göstergesidir. Bu noktada ne BFF'lerin ne de diğer mikro servislerin implementasyonunda ortalama cevap süresini artırma kapsamında yapılacak bir şey olmadığını belirtmek uygun olacaktır zira darboğaz Kubernetes cluster'ındaki bant genişliğinin yetersizliğinden kaynaklanmaktadır.

Ek olarak hem OBA'da hem de monolit BOA'da 2500 ve 5000 VU gibi yüksek sanal kullanıcı sayılarında çok az hata gözlemlendiğini vurgulamakta fayda bulunmaktadır (örneğin, yüzde 0,03). Bu az sayıdaki hata network bağlantısındaki zaman aşımıyla ilgili olup Tablo 5.10 ve 5.11'in üçüncü sütunlarında belirtilen toplam istem sayısı göz önüne alındığında ihmal edilebilir boyutlarda olduğu düşünülmektedir.



Şekil 5.34. Mikro Servis Mimarisi Esaslı OBA Altyapısının Ortalama Cevap süresi³⁷

³⁷ y eksenini ortalama yanıt süresi değerlerini, x eksenini ise yük testleri sırasında paralel olarak çalıştırılan sanal kullanıcı sayısını göstermektedir.

6. DAĞITIK SAGA İŞLEMLERİNİN (TRANSACTIONS) AKTÖR MODELİ ESASLI YÖNETİMİ

Bu bölümde doktora çalışmasının ana akademik katkılarından birini oluşturan dağıtık *Software Automation, Generation, and Administration-(SAGA)* işlem yönetimi konusundaki buluşun içeriği açıklanmaktadır. Bu buluş kapsamında ortaya konan yöntem gerek endüstride gerekse akademik literatürde türünün ilk örneği olup bu alanda yeni çalışmalara ışık tutacağına inanılmaktadır.

6.1. Buluşun Teknik İçeriği

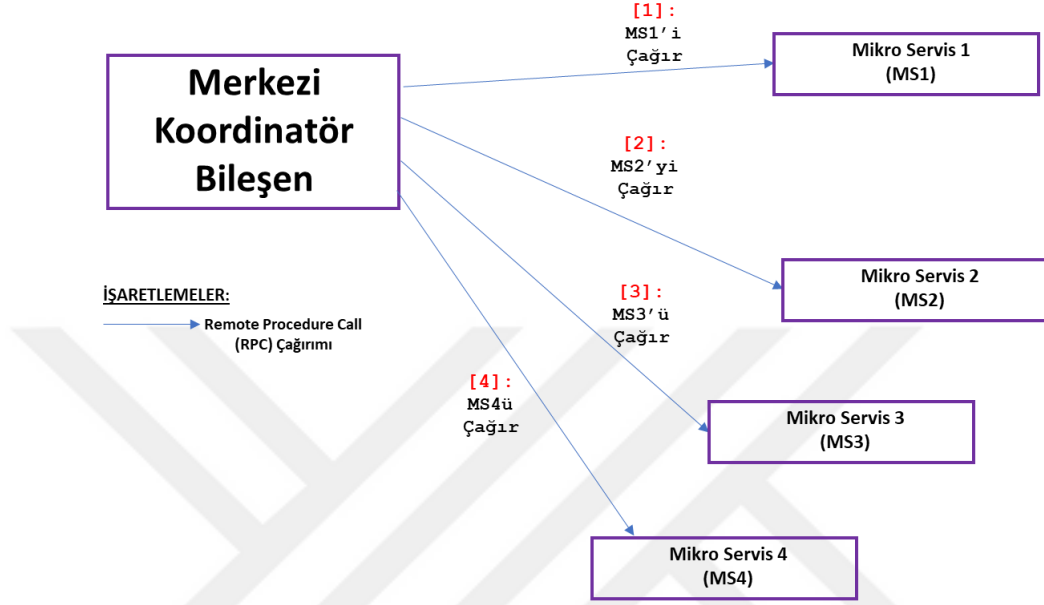
Doktora çalışması sırasında gerçekleştirdiğimiz saha tecrübesi kapsamında ortaya koymuş olduğumuz buluş, bir yazılım mimarisi olan mikro servis mimarisi minvalinde çalışan mikro servisler tarafından gerçekleştirilen dağıtık işlemlerin (distributed transactions) koordine edilmesi amacıyla kullanılagelen “*Software Automation, Generation, and Administration- (SAGA)*” tasarım deseninin aktör modeli ekseninde genişletilmesi suretiyle farklı SAGA akışlarının koordinasyonunu mümkün kılan bir tekniktir.

Mikro Servis Mimarisi esaslı olarak tasarlanmış bir yazılım sistemindeki mikro servisler belirli bir işlevi gerçekleştirmek üzere bir dizi etkileşimde bulunurlar. SAGA, mikro servisler arası etkileşimlerin belirli bir nizam çerçevesinde gerçekleştirilmesi amacıyla ortaya konmuş bir yazılım tasarım desenini (software architecture pattern) ifade eder. SAGA’nın endüstride yaygın olarak kullanılan iki varyantı mevcuttur. Bunlardan birincisi *komut-orkestrasyon esaslı SAGA* deseni olup bu SAGA deseninde her bir mikro servis üzerinden çağırılacak işlev bir koordinatör bileşen marifetiyle sıralı olarak yapılır. Şekil 6.1’de örneklenen bu yapıda görüldüğü gibi merkezi bir koordinatör bileşen, 1’den 4’e kadar numaralandırılmış mikro servisleri birbiri ardına çağırmak suretiyle belirli bir iş mantığını gerçekleştirmektedir.

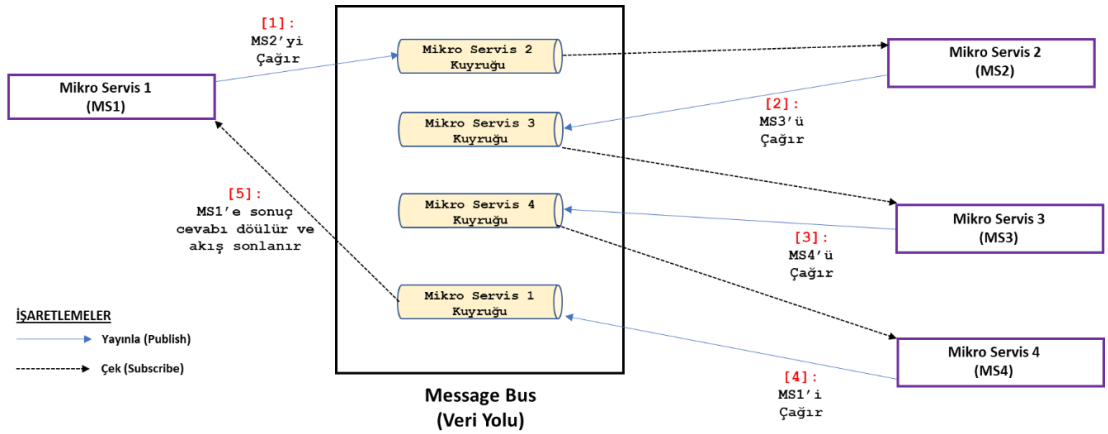
SAGA’nın ikinci varyantı olan *koreografi esaslı SAGA*’da mikro servisler kendi aralarındaki koordinasyonu Şekil 6.2’de görüldüğü gibi bir message-bus (veri yolu – Örnek: Kafka, RabbitMQ) ürünü marifetiyle asenkron olarak gerçekleştirir. Dolayısıyla merkezi bir koordinatör mevcut değildir.

Koreografi esaslı SAGA birbirinden bağımsız olarak çalışabilen ve kısıtlı sayıda mikro servisin müdahil olduğu e-ticaret sistemleri gibi gerçek-zamanlı (real-time)

çalışma gereksinimi olmayan sistemlerde idealdir. Örneğin Amazon e-ticaret sitesinde olduğu gibi kargo (shipping) mikro servisi ile ödeme mikro servisi birbirinden bağımsız çalışabilmektedir zira ödeme anlık olurken ürünün paketlenme süreci daha uzun sürebilmektedir.



Şekil 6.1. Komut-Orkestrasyon Esaslı SAGA Tasarım Deseni



Şekil 6.2. Koreografi Esaslı SAGA Tasarım Deseni

Bununla birlikte koreografi esaslı SAGA'da merkezi bir koordinatör bulunmaması bir dezavantaj oluşturmaktadır çünkü mikro servis sayısı arttıkça genel akışın koordinatör olmadan yönetimi zorlaşmakta ve hata olasılıkları artmaktadır. Bu

nedenle bankacılık ya da telekomünikasyon sektörleri gibi çok sayıda mikro servisin belirli bir nizam çerçevesinde sıralı ve gerçek zamanlı olarak (near real-time) çalışması gereken sektörlerde koreografi varyantı ideal değildir. Nitekim bu sektörlerde çok sayıda mikro servisin belirli onay mekanizmalarının işletilmesi gereken bir nizamda işletimini gerektiren iş akışları (bir başka deyişle SAGA akışları) olabilmektedir. Örneğin döviz alış gibi bir bankacılık SAGA akışını gerçekleştiren mikro servislerin işletimi sırasında bir servisin müşteriden SMS ile onay alınması sonucunda gerçekleştirebilmektedir. Bu bakımdan komut-orkestrasyon esaslı SAGA çoğu kez koreografi esaslı SAGA'ya göre yönetiminin daha kolay olması ve hata riskinin daha az olması nedeniyle daha tercih edilir bir yapı arz eder (Richardson, 2019; Torre ve ark., 2019; Bellemare, 2020).

Diğer yandan komut-orkestrasyon esaslı SAGA varyantında merkezi koordinatörün aşırı işlem yükü alması durumunda sistemin genelini etkileyen bir performans dar boğazının oluşması oldukça olasıdır (Richardson, 2019). Aktör modeli ortaya koyduğu dağıtık hesaplama (distributed computing) paradigması sayesinde bu performans dar boğazının aşılmasını sağlayabilir zira bu model merkezi koordinatörün bulut tabanlı (cloud-native) bir konteyner orkestrasyon altyapısı (Örnek: Kubernetes) üzerinde ölçeklenerek (scale edilerek) performanslı bir şekilde çalıştırılmasına olanak tanıyan bir dağıtık hesaplama modeli sunmaktadır.

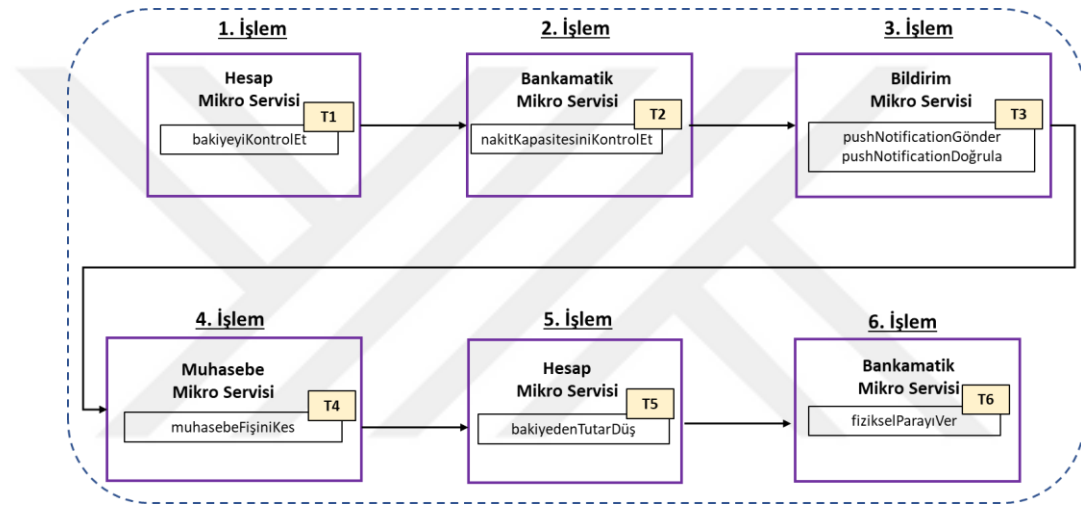
Buradan hareketle bu buluş kapsamında elde edilen **birincil kazanım** aktör modelinin kullanımından kaynaklanmaktadır. Zira bu model ortaya koyduğu dağıtık hesaplama (distributed computing) paradigması sayesinde komut-orkestrasyon esaslı SAGA desenindeki merkezi koordinatör bileşen üzerinde aşırı yük kaynaklı oluşabilecek performans darboğazını bertaraf etmektedir. Bir başka deyişle merkezi koordinatör bileşen üzerindeki iş yükü aktörlere dağıtılmaktadır. Dolayısıyla bu buluş kapsamında elde edilen **birincil kazanım** şu şekilde özetlenebilir:

Kazanım 1: “Komut-orkestrasyon esaslı SAGA deseninin aktör modeliyle genişletilmesi suretiyle merkezi koordinatör üzerinde bir performans darboğazı oluşumunun engellenmesi.”

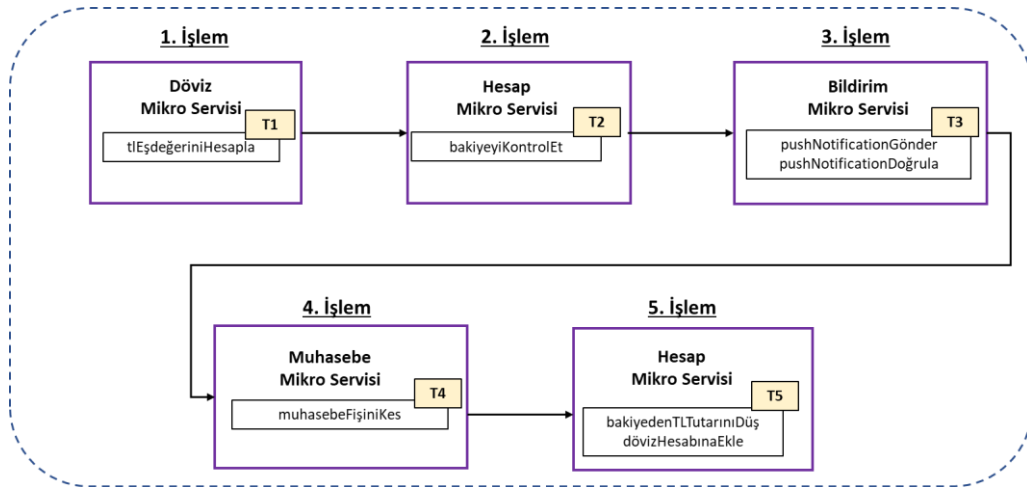
Bu buluş kapsamında elde edilen **ikincil kazanım** ise aktör modeli üzerine geliştirdiğimiz bir kümeleme algoritmasıdır. Bu algoritma sayesinde farklı SAGA akışları tarafından ortaklaşa kullanılan ve kritik veri kaynaklarına erişen mikro servislere erişim koordinasyon altına alınmaktadır. Bir başka deyişle literatürde dirty-read olarak geçen ve

kritik veri kaynakları üzerinde hatalı veri okuma ve yazma durumlarının oluşmasına neden olan sorun engellenmektedir. Kritik veri kaynaklarına örnek olarak bankacılık müşterilerinin bakiye verileri verilebilir. Bu ikincil kazanımı daha anlaşılır kılmak adına aşağıdaki paragraflarda daha detaylı bilgi sunulmaktadır.

Mikro Servis Mimarisi temelinde oluşturulmuş modern bir dağıtık yazılım sisteminde muhtelif iş mantıklarını icra eden farklı türde SAGA akışları mevcuttur. Birden çok mikro servisin belirli bir nizam çerçevesinde çağırıldığı SAGA akışlarına örnek olarak Şekil 6.3 ve 6.4'te görülen ve bankacılık sektöründe kullanılabilecek olan *nakit çekme* ve *döviz-alış* SAGA akışları verilebilir.



Şekil 6.3. Nakit Çekme SAGA Akışı



Şekil 6.4. Döviz Alış SAGA Akışı

Gerek Şekil 6.3'te örneklenen nakit çekme SAGA akışında gerekse Şekil 6.4'te örneklenen döviz alış SAGA akışında müşteri verilerinde değişiklik gerçekleştiren ve ortaklaşa kullanılan bir *hesap mikro servisi* görülmektedir (yani T5). Müşterek kullanılan banka hesaplarında (Ör: Karı-koca müşterek hesabı) farklı iki kişinin aynı anda bu iki akışı kullanım durumu söz konusu olabilir. Böyle bir durumda her iki SAGA akışında da kullanılan hesap mikro servisi tarafından yapılan *bakiyedenTLTutarınıDüş* işlemi (yani T5) sıralı bir biçimde yapılmalıdır. Yani örneğin önce nakit çekme akışı başladıysa bu akışın hesap mikro servisiyle olan işi sonlanana kadar döviz-alış SAGA akışının hesap mikro servisine erişimi bekletilmelidir. Aksi takdirde literatürde “**dirty read**” olarak bilinen hatalı veri okuma/yazma durumlarının oluşması ve dolayısıyla müşterilerin bakiyelerinde tutarsızlıklar oluşması kaçınılmaz olacaktır. Bu dirty-read durumunun nasıl oluşabileceği bankacılık sektöründe gerçekleşebilecek örnek senaryo kapsamında aşağıda adım adım açıklanmaktadır.

Örnek Dirty-Read Senaryosu (Müşterek Karı-Koca Hesabı Kullanımı):

1. Ali Bey ve Ayşe Hanım isimlerinde evli bir çiftin aynı müşteri numarası ile tanımlanan müşterek bir birikim hesabı olduğunu ve bu hesapta 3000 TL değerinde bakiye bulunduğunu varsayalım.
2. Ali Bey bankamatikten 2000 TL'lik nakit çekme girişiminde bulunur. Bu durum Şekil 6.3'te görülen nakit çekme SAGA akışının ilk işlemi olan T1'i başlatır. T1 işlemi kapsamında müşterek hesaptaki güncel bakiye hesap mikro servisinde 3000 TL olarak okunur. Ali Bey tarafından talep edilen nakit tutarı olan 2000 TL güncel bakiye tutarı olan 3000 TL'den düşük olduğu için “bakiyeKontrolEt” işlemi (T1) başarıyla sonlanır ve nakit çekme SAGA akışı T2'yi işleten bankamatik servisine geçer.
3. Ali Bey'in nakit çekme talebi kapsamında bankamatikte yeterince banknot olduğunu varsayarsak T2 de başarılı bir şekilde sonlanır ve akışın icrası bir sonraki servise geçer.
4. Bilgilendirme mikro servisindeki T3 işlemi icra edilir ve bu kapsamda Ali Bey'in cep telefonuna teyit amaçlı bir bildirim (push notification) gönderilir.
5. Tesadüfen aynı anda Ayşe Hanım, bankanın mobil uygulamasından bir döviz alış talebi yapar. Bu talep Şekil 6.4'teki döviz alış SAGA akışındaki T1 işlemini tetikler ve talep edilen dövizin TL karşılığı olan 3000TL o anki güncel kur üzerinden hesaplanır.

6. Döviz mikro servisindeki T1 işlemi bitirilince sıradaki işlem olan T2 işlemi hesap mikro servisi üzerinden çağırılarak müşterek hesaptaki 3000 TL'lik güncel bakiye değeri temin edilir. Talep edilen 3000 TL ederindeki döviz alımı için 3000 TL'lik güncel bakiye miktarının yeterli olduğu anlaşılır ve T2 işlemi başarılı şekilde sonlanır.
7. Bilgilendirme mikro servisindeki T3 işlemine geçilir ve bilgilendirme servisi Ayşe Hanım'ın cep telefonuna teyit amaçlı bir bilgilendirme mesajı gönderir. Tesadüfen Ayşe Hanım'ın Ali Bey'den daha hızlı davranarak bilgilendirme mesajını daha önce onayladığını varsayalım. Bu durumda Ali Bey hala 4. adımda ifade edilen T3 işlemi bitirmemiş yani bilgilendirme mesajını cep telefonundan onaylamamıştır. Dolayısıyla Ayşe Hanım'ın döviz alış akışı zamanlama olarak Ali Bey'in nakit çekme akışının önüne geçmiştir.
8. Ali Bey de Ayşe Hanım'ın hemen arkasından bilgilendirme mesajını (push notification) onaylar ve hem Ali Bey'in nakit çekme SAGA akışındaki; hem de Ayşe Hanım'ın döviz alış SAGA akışındaki geri kalan işlemler hemen hemen eş zamanlı olarak başarıyla bitirilir. Dolayısıyla güncel olmayan bakiye bilgisinin her iki SAGA akışında da okunma durumu nedeniyle (yani **dirty-read** oluşumu nedeniyle) müşterek hesap bakiyesi yetersiz olmasına rağmen her iki SAGA akışı da sonuna kadar gerçekleştirilmiş ve banka zarara uğratılmış olur. Bir başka deyişle normalde 3000 TL'lik bakiyesi olan müşterek hesaptan Ayşe Hanım tarafından yapılan 3000 TL'lik döviz alışına ek olarak Ali Bey'in de önden talep ettiği 2000 TL'lik nakit çekimi de gerçekleştirilmiş olur ve aslında yeterli bakiye olmamasına rağmen banka toplamda 2000 TL'lik fazla ödeme yaparak zarara uğratılmış olur.

Yukardaki senaryo monolit bir sistemde de oluşabilir ancak monolit sistemin veri okuyup yazdığı veri tabanı sistemi (Örnek: MS SQL Server, PostgreSQL) haiz olduğu 2PC (2-phase-commit) prensibi gereği bu tür dirty-read durumunun oluşumunu otomatik olarak engellenmektedir. Ancak mikro servis mimarisinde farklı servisler bir veri tabanındaki kritik önemi olan bir tabloya (Örnek: Müşteri bakiyelerinin tutulduğu hesap tablosu) aynı anda erişebilirler. 2PC prensibi esaslı çalışan veri tabanı sistemleri monolit sistemlerdeki durumun aksine kendi dışında gerçekleşen ve mikro servis çağırımları sonucu oluşan uygulama kökenli işlemleri senkronize edecek şekilde tasarlanmamıştır zira bu veri tabanı sistemleri yalnızca kendi içlerindeki lokal işlemleri (local transactions)

yönetebilmektedirler. Bu nedenle bu veri tabanı sistemleri mikro servislerin dahil olduğu ve uygulama katmanında gerçekleşen SAGA akışlarını koordine edememektedirler. Dolayısıyla SAGA akışları arasındaki koordinasyonu uygulama seviyesinde (yani SAGA akışları üzerinde) gerçekleştiren bir algoritma ile bu algoritmayı sistemin genelinde işleten bir yazılım altyapısına ihtiyaç vardır. Buradan hareketle bu buluş kapsamında elde edilen **ikincil kazanım** esasen bu ihtiyacı görmektedir ve şu şekilde özetlenebilir:

Kazanım 2: “*Kritik bir takım veri kaynaklarına erişim gerçekleştiren mikro servisleri ortaklaşa kullanan SAGA akışlarının sıralı bir şekilde (yani senkronize) çalıştırılmasını sağlayan bir algoritmik tekniğin aktör modelini kullanan bir kümeleme mantığı çerçevesinde sağlanması.*”

6.2. Buluşun Amacı

Bankacılık veya telekomünikasyon gibi mikro servis mimarisi esaslı dağıtık yazılım sistemlerinin yaygın bir şekilde uygulanabileceği sektörlerde *komut-orkestrasyon esaslı SAGA tasarım deseninin* kullanımı daha uygundur zira bu sektörlerde çok sayıda mikro servisin sıralı bir nizam çerçevesinde koordineli olarak işletimi elzemdir. Bununla birlikte *komut-orkestrasyon esaslı SAGA deseni* kapsamında kullanımı gereken ve servislerin işletimini koordine etmekle görevli merkezi koordinatör bileşeni kolaylıkla bir sistem dar boğazına dönüşebilmektedir. Bu durum özellikle bankacılık ve telekomünikasyon gibi yoğun işlem trafiğinin yaşanabileceği sektörlerde ciddi bir dezavantaj oluşturacaktır zira uzun sistem kesintilerine ve dolayısıyla müşteri memnuniyetsizliğine ve bundan mütevellit mali kayıplara neden olur. Aktör modeli ile entegre çalışan bir merkezi koordinatör bu performans dar boğazından kurtarılmış olur zira bu model merkezi koordinatörün bulut tabanlı (cloud-native) bir altyapı (Örnek: Kubernetes) üzerinde ölçeklenerek (scale edilerek) performanslı bir şekilde çalışmasına olanak tanıyan bir distributed computing modeli sunmaktadır.

Diğer yandan *komut-orkestrasyon esaslı SAGA tasarım deseninin* merkezi koordinatör kaynaklı performans sorununa ek olarak ikinci bir dezavantajı daha mevcuttur. Bu desen, yalnızca tek bir SAGA akışı (Örnek: Şekil 6.3'teki nakit çekme SAGA akışı) bünyesindeki mikro servislerin kendi aralarında belirli bir sırada çalışmasını garanti etmek üzere tasarlanmıştır. Bununla birlikte bu desen, kritik öneme haiz bir takım veri tabanı tablolarına erişen mikro servisleri ortaklaşa kullanan birden çok SAGA

akışının sıralı çalışmasını garanti edecek bir nitelikte değildir. Dolayısıyla komut-orkestrasyon esaslı SAGA tasarım deseni kapsamında bölüm 6.1'deki örnek senaryo üzerinden izah edilen dirty-read durumlarının oluşumunu engelleyebilecek ilave bir algoritmik tekniğe ihtiyaç vardır.

Özet olarak bu buluş *komut-orkestrasyon esaslı SAGA tasarım deseni* kapsamında aşağıdaki iki maddede özetlenen problemleri ortadan kaldırmaktadır:

- i. Merkezi koordinatör bileşen üzerinde oluşabilecek performans dar boğazı problemi
- ii. Kritik veri kaynaklarına erişen mikro servisleri ortaklaşa kullanan farklı SAGA akışlarını koordine edememe problemi.

Buradan hareketle bu dokümanda açıklanmakta olan buluşun amacı şu şekilde ifade edilebilir:

BULUŞUN AMACI: “Komut-orkestrasyon SAGA tasarım deseni ile aktör modeli temelindeki bir kümeleme mantığı çerçevesinde geliştirilen bir algoritma marifetiyle kritik öneme haiz bir takım veri kaynaklarına erişim gerçekleştiren mikro servisleri ortaklaşa kullanan SAGA akışlarının performanslı ve sıralı bir şekilde (yani senkronize bir şekilde) çalıştırılmasının sağlanması.”

Bu amaç doğrultusunda elde edilen kazanımlar ise şunlardır:

Kazanım 1: “Komut-orkestrasyon esaslı SAGA deseninin aktör modeliyle genişletilmesi suretiyle merkezi koordinatör üzerinde bir performans darboğazı oluşumunun engellenmesi.”

Kazanım 2: “Kritik bir takım veri kaynaklarına (Örnek: Bakiye tablosu) erişim gerçekleştiren mikro servisleri ortaklaşa kullanan SAGA akışlarının sıralı bir şekilde (yani senkronize) çalıştırılmasını sağlayan bir algoritmik tekniğin aktör modelini kullanan bir kümeleme mantığı çerçevesinde sağlanması.”

Yukarıdaki paragraflarda amacı ve kazanımları ortaya konulan bu buluş mikro servis mimarisi ile ilgili saha tecrübelerimiz ışığında ortaya çıkmış olup detayları takip eden alt bölümlerde izah olunmaktadır.

6.3. Buluşun Detaylı Açıklaması

Bu bölümde buluşun içeriğini doğru ortaya koymak amacıyla öncelikle aktör modelinden bahsedilmektedir. Bunun akabinde buluşun amacı kapsamında SAGA akışlarının aktör modeli çerçevesinde nasıl senkronize edilebileceği izah edilmekte ve bu çerçevede buluşun teknik içeriği ortaya konmaktadır.

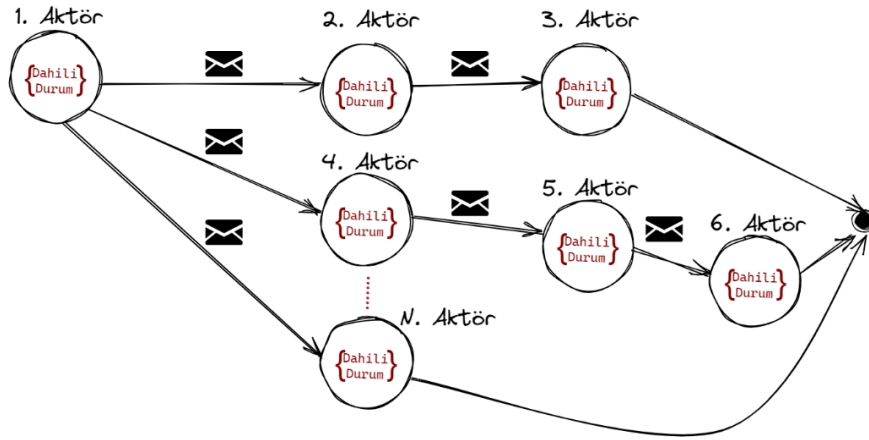
Aktör Modeli, bilgisayar bilimlerinde eş zamanlı hesaplama (concurrent computation) yapmak amacıyla geliştirilen matematiksel bir model olup temelinde “*her şey bir aktördür*” felsefesini esas alır ve bu bakımdan nesne yönelimli programlama paradigmasındaki “*her şey bir nesnedir*” felsefesini andırır. Nasıl ki bir nesnenin davranışı nesne üzerindeki metotların işletilmesiyle değişiyorsa; bir aktörün davranışı da aldığı mesaja göre değişmektedir.

Aşağıda Şekil 6.5’te ifade olunan aktör modeli şu karakteristik özelliklere sahiptir:

- i. Bir aktör diğer aktörlere kendi iş mantığı kapsamında mesaj gönderebilir.
- ii. Bir aktör başka aktörlerin oluşumunu sağlayabilir.
- iii. Bir aktörün dahili durumu (yani state’i) yalnızca diğer aktörlerden alacağı potansiyel mesajların neticesinde değişebilir. Burada mesaj alma bir nesnenin metodunu çağırmak gibi düşünülebilir.
- iv. Mesajlaşma yapısı sayesinde her bir aktör kendisine iletilen mesajları yalnızca kendisinin erişebildiği dahili bir gelen mesaj kuyruğuna alır ve bu kuyruktaki her bir mesajı sırayla işler. Başka aktörlerin bu dahili mesaj kuyruğuna erişimi hiçbir şekilde mümkün değildir. Bu nedenle aktörün gelen mesaj kuyruğuna sonradan gelen mesajlar ile önden gelen mesajlar birbirinden izole bir şekilde ve sırayla (yani senkronize edilerek, arka arkaya) ele alınıp işlenir.

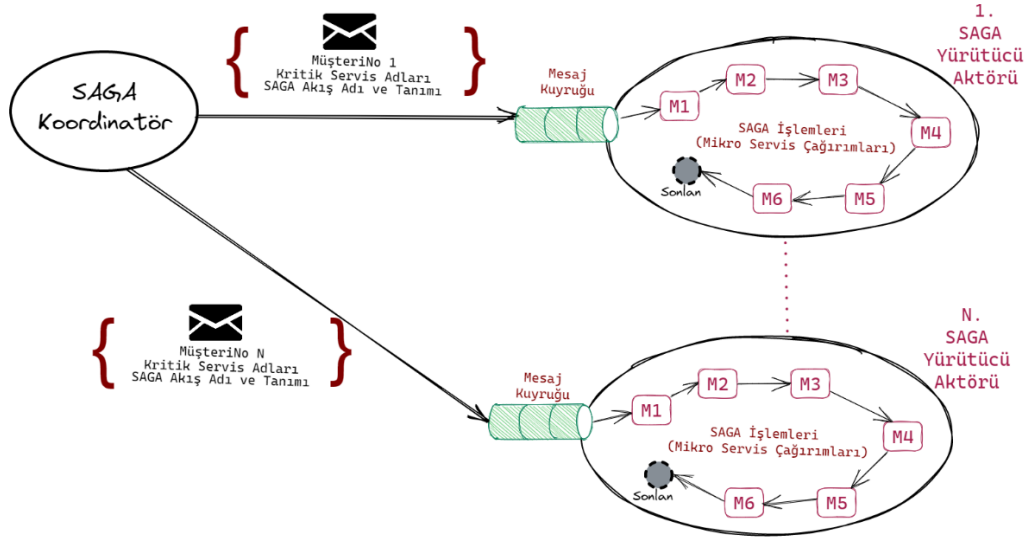
Karakteristik özellikleri yukarıdaki maddelerde belirtilen aktör modeli, her bir aktördeki dahili kuyruk yapısından kaynaklı doğal sıralama kabiliyeti nedeniyle aynı kritik mikro servis ya da servislere erişen SAGA akışlarının senkronizasyonunu (yani sıralı şekilde çalışmasını) sağlamak için kullanılabilir. Burada kritik servis ile kast edilmek istenen hesap mikro servisi örneğindeki gibi müşteri bilgilerini manipüle edebilen ve birden çok SAGA akışı tarafından müşterek olarak kullanılan bir mikro servistir. Örneğin bankacılıktan örneklediğimiz hesap mikro servisi hesap bakiyelerinden tutar düşülmesi ve eklenmesi gibi pek çok işlemi gerçekleştirmesi bakımından nakit

çekme, döviz alış/satış ve EFT gönderimi gibi bankacılık işlerini gerçekleştiren pek çok SAGA akışı kapsamında ortaklaşa kullanılabilir.



Şekil 6.5. Aktörler Arası Mesajlaşma Yapısı

Bu buluş kapsamında SAGA akışlarının aktör modeli vasıtasıyla nasıl işletilebileceği Şekil 6.6’da görüldüğü gibi modellenmektedir. Bu şekilde en solda görülen ve “SAGA koordinatör” olarak adlandırdığımız yazılım bileşeni esasen komut-orquestrasyon esaslı SAGA varyantındaki merkezi koordinatör bileşenin görevini üstlenen özelleşmiş bir mikro servis olarak tasavvur edilebilir. Bu mikro servis, SAGA istemlerini harici istemci uygulamalarından (client’lardan) alarak aktörler eliyle sıralı olarak çalıştırmaktan sorumludur. Burada çalıştırılan aktörler yüzlerce olabilir. Yoğun işlem (transaction) yükünün söz konusu olduğu bankacılık yazılım sistemlerinde bu sayı binleri dahi bulabilir. Dolayısıyla bu buluş minvalinde ortaya konulan SAGA koordinatör çok sayıda aktörü (1’den N’ye kadar) belirli bir algoritma çerçevesinde organize ederek her bir aktörün belirli bir müşteri özelindeki SAGA akışını sıralı bir nizamda (yani senkronize olarak) çalıştırmasını sağlamaktadır. Bu algoritmanın genel işleyiş adımları aşağıda bölüm 6.3.1’de açıklanmakta olup akış diyagramı Şekil 6.7’de görülmektedir.



Şekil 6.6. SAGA Akışlarının Aktörler üzerinden Çalıştırılması (M harfi Mikro Servisi temsil eder)

6.3.1. Buluş kapsamında ortaya konulan algoritmanın genel işleyiş adımları

Buluş kapsamında geliştirilen algoritma çerçevesinde her bir aktör belirli bir müşteri özelindeki SAGA akışlarını sıralı bir nizamda (yani senkronize olarak) işletebilmektedir. Söz konusu algoritmanın genel mantığı aşağıdaki adımlarda sunulmaktadır:

1. SAGA koordinatör bir SAGA akış talebini (Örnek: Nakit çekme SAGA Akışı-Şekil 3) herhangi bir istemciden (Örnek: Android uygulaması) bir network çağırımı (Remote Procedure Call -RPC) olarak alır ve bu talebin içeriğinden aşağıdaki şu üç parametreyi çeker.
 - a. **Müşteri Numarası** (Örnek: 99999)
 - b. **Kritik Servis-Metot Listesi:** Örnek 1’de görüldüğü gibi SAGA talebi kapsamındaki işletilecek kritik mikro servislerin tanımlayıcı isimleri (Örnek: Hesap, Muhasebe) ile bu servisler üzerindeki transactional işlemleri ifa eden metotların adlarından oluşan servis-metot çiftleri listesi (Java Script Object Notation – JSON formatında).

Örnek 1 :

```
[
  {
    "ServisAdı": "Servis1",
    "MetotAdı": "x"
  },
  {
    "ServisAdı": "Servis1",
    "MetotAdı": "y"
  },
  {
    "ServisAdı": "Servis2",
    "MetotAdı": "k"
  }
]
```

- c. **SAGA Akış Tanımı:** SAGA talebi kapsamında işletilecek olan SAGA akışının JSON formatında tanımı (Yani hangi mikro servislerdeki hangi metotların ne sırada çağırılacağı) EK-3'te örneklenmektedir.

2. SAGA koordinatör, kendi bünyesinde mevcut bulunan *sagaKümesiYokla* adlı metodu **1.a** ve **1.b** adımlarında temin ettiği *müşteri numarası* (Örnek: 999999) ile *kritikServisAdı-metotAdı* çiftlerinden oluşan listeyi (bkz. Örnek 1) sırasıyla *müşteriNo* ve *kritikServisMetotListesi* adlı giriş parametrelerine beslemek suretiyle çağırır. Bu çağırım kapsamında bir NOSQL veri tabanına sorgu atılır ve ilgili müşteri numarası özelinde *kritikServisMetotListesi* içindeki servis-metot çiftlerinden biriyle kesişen mevcut bir listenin veri tabanında bulunup bulunmadığı kontrol edilir.

```
sagaKümesiYokla(müşteriNo as int,
                  kritikServisMetotListesi as JSON Array):
returns SetId as UUID - Universally Unique Identifier
BEGIN
```

- NOSQL veri tabanına **müşteriNo** giriş parametresi kullanılarak bir sorgu atılır.
- Sorgudan dönen bir kayıt yoksa **sagaKümesiYokla** metodu boş (NULL)döner.
- Eğer sorgudan kayıt dönülmüşse söz konusu bu kayıt aşağıda **örnek 2'de** görüldüğü gibi JSON formatına haiz olacak olup ilgili müşteri numarasına bağlı kritik servis gruplarının içeren bir küme listesi teşkil edecektir. Bu listedeki her bir küme birbiriyle mantıksal olarak yakından ilişkili kritik servisler (Örnek: Hesap işlemleri ile alakalı kritik mikro servisler) hakkında

hemen aşağıdaki alt maddelerde açıklanan yapıları barındıracaktır:

- **SetId:** İlgili kümeyi tanımlamak amacıyla UUID formatında bir anahtardır (identifier'dır).
- **Value:** müşteriNo parametresi özelinde halihazırda NOSQL veri tabanında bulunan birbiriyle mantıksal olarak ilişkili kritik servisleri içeren bir JSON dizesi olup dizinin her bir elemanı esasen bir küme elemanıdır. Bu elemanların her birinde şu üç parametre mevcut olacaktır
 - i. **Kritik Servisin Adı** (Örnek: Hesap mikro servisi)
 - ii. Kritik Servis üzerinde tanımlı olan **kritik metodun adı** (Örnek: bakiyedenTutarDüş metodu).
 - iii. **Counter:** İlgili kritik servis üzerindeki metodu kaç tane SAGA akışına dahil olduğunun sayısı.

Örnek 2:

```
"99999":  
[  
  //Birinci Eleman  
  {  
    "SetId": "69c7b16c-d834-11ed-afa1-0242ac120002",  
    "Value":  
    [  
      {  
        "ServisAdı": "Servis1",  
        "MetotAdı": "x",  
        "Counter": 1  
      },  
      {  
        "ServisAdı": "Servis1",  
        "MetotAdı": "y",  
        "Counter": 1  
      },  
      {  
        "ServisAdı": "Servis2",  
        "MetotAdı": "k",  
        "Counter": 1  
      },  
      {  
        "ServisAdı": "Servis3",  
        "MetotAdı": "m",  
        "Counter": 1  
      }  
    ]  
  },  
  //İkinci Eleman  
  {  
    "SetId": "550e8400-e29b-41d4-a716-446655440000 ",  
    "Value":  
    [  
      {  
        "ServisAdı": "Servis4",  
        "MetotAdı": "a",  
        "Counter": 1  
      },  
      {  
        "ServisAdı": "Servis5",  
        "MetotAdı": "b",  
        "Counter": 1  
      },  
      {  
        "ServisAdı": "Servis5",  
        "MetotAdı": "c",  
        "Counter": 1  
      }  
    ],  
    {  
    }
```

- d. Örnek 2'deki görüldüğü gibi **SetId** ile tanımlanan her bir küme elemanı içerisinde bir **Value** alanı bulunmakta ve bu **Value** alanı içinde de bir JSON dizesi şeklinde ilgili müşteri numarasına bağlı kritik servisler barındırılmaktadır. Bu dizideki her bir eleman ile **kritikServisMetotListesi** adlı giriş parametresinde alınan ve Örnek 1'de görülen liste karşılaştırılır ve aralarında bir kesişim olup olmadığı kontrol edilir (Karşılaştırmada Counter alanları görmezden gelinmektedir).

- e. Eğer kesişim varsa kesişimin söz konusu olduğu kümedeki **KritikServisAdı-MetodAdı-Counter** üçlüsündeki Counter parametresinin değeri **1** artırılır. Daha önce de belirtildiği gibi buradaki counter alanı ilgili **ServisAdı-MetotAdı** çiftinin halihazırda kaç tane SAGA akışı kapsamında işletilmekte olduğunun sayısını tutmaktadır.
- f. Kesişimin bulunduğu kümeyi tanımlayan **SetId** alanında bulunan UUID değeri dönülür (Örnek: "69c7b16c-d834-11ed-afa1-0242ac120002")
- g. Eğer kesişim mevcut değilse **sagaKümesiYokla** metodu **NULL** döner.

END of method sagaKümesiYokla

3. Yukarıdaki 2. adımda gerçekleştirilen *sagaKümesiYokla* adlı metot çağırımından **NULL** dönmüş ise kaba kodu aşağıda belirtilen *sagaKümesiKaydet* adlı başka bir metot SAGA koordinatör tarafından çağırılır. Bu çağırım sırasında **1.a** adımında temin edilen müşteri numarası ile **1.b** adımında temin edilen kritik servis-metot adı çiftlerinden oluşan liste (bkz. örnek 1) sırasıyla *müşteriNo* ve *kritikServisMetotListesi* adlı giriş parametrelerine beslenir. Bu suretle gerçekleşen çağırım sonucunda *sagaKümesiKaydet* metodu *SetId* adlı tanımlayıcıyı temsil eden bir UUID'yi SAGA koordinatöre döner.

sagaKümesiKaydet(müşteriNo as int,
kritikServisMetotListesi as JSON Array):

returns SetId As UUID

BEGIN

- a. **MüşteriNo** parametresi ile gelen değer anahtar olacak şekilde; Örnek 1'de görülen yapıyı içeren **kritikServisMetotListesi** adlı giriş parametresini de değer alanında kullanacak şekilde bir JSON kaydı oluşturulur ve NOSQL veri tabanına kaydedilir. Kaydedilen veri yapısı aşağıda örnek 3'te örneklenmekte olup sıfırdan oluşturulduğu için örnek 2'de görülen kayda göre daha küçük tek bir kümeden oluşan bir listedir.

Örnek 3:

```

"99999":
[
  {
    "SetId": "032f2e56-c403-4050-980d-b91ee170baeb",
    "Value":
    [
      {
        "ServisAdı": "Servis1",
        "MetotAdı": "x",
        "Counter": 1
      },
      {
        "ServisAdı": "Servis1",
        "MetotAdı": "y",
        "Counter": 1
      },
      {
        "ServisAdı": "Servis2",
        "MetotAdı": "m",
        "Counter": 1
      }
    ]
  }
]

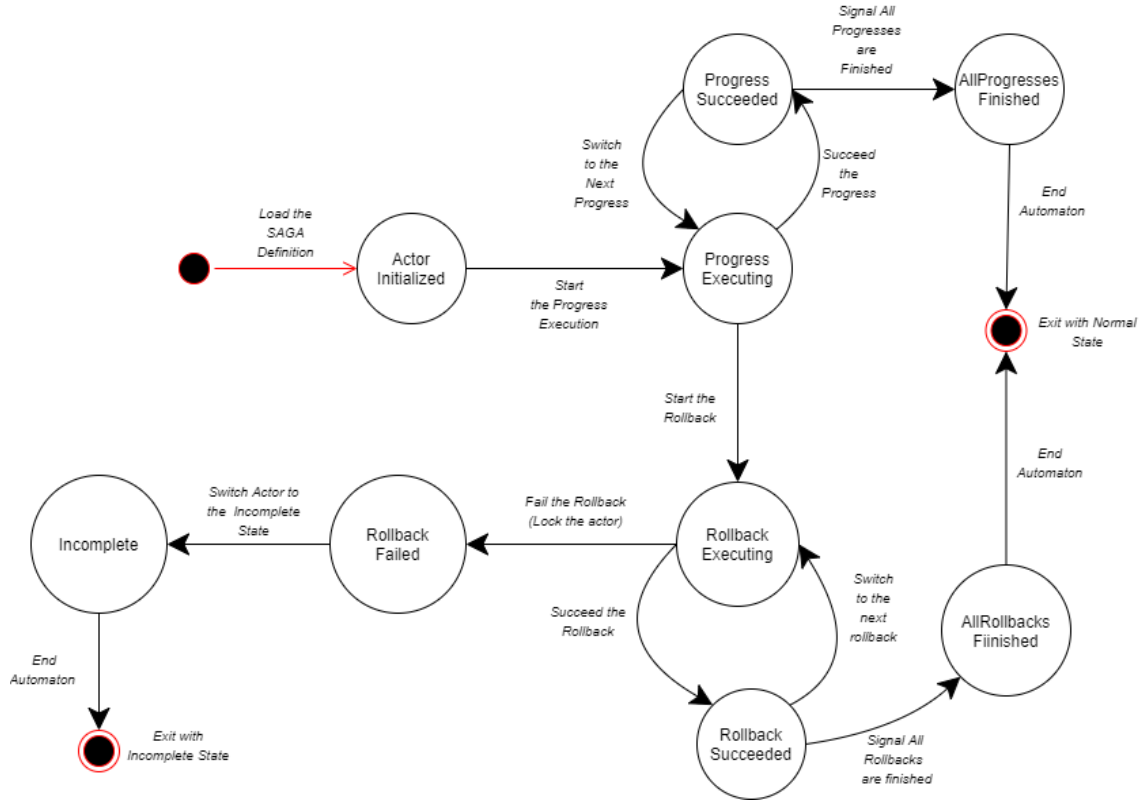
```

b. NOSQL veri tabanında atılan kayıttaki kümeyi temsil eden **SetId** değeri (Örnek: "032f2e56-c403-4050-980d-b91ee170baeb") SAGA koordinatöre dönülür.

END of method sagaKümesiKaydet

4. SAGA koordinatör, 2. veya 3. adımlar kapsamında kendisine dönülen *SetId* ile **1.a** adımıda temin edilen müşteri numarasının arasına ‘.’ karakterini bir ayırıcı olarak koymak suretiyle bir aktör tanımlayıcısı (actor identifier) oluşturur. Örneğin müşteri numarası 99999 ve *SetId* "123e4567-e89b-12d3-a456-426614174000" şeklinde bir UUID ise; aktör tanımlayıcısı (actor identifier) "99999:123e4567-e89b-12d3-a456-426614174000" şeklinde bir değere sahip olur.
5. SAGA koordinatör bu minvalde yeni oluşturduğu aktör tanımlayıcısını kullanarak yeni bir aktör oluşturur ve hemen akabinde bu aktörün mesaj kuyruğuna **1.a**, **1.b** ve **1.c** maddelerinde temin edilen parametreleri mesaj olarak geçirir. Bu durum Şekil 6’da görüldüğü gibi “**1.Saga Yürütücü Aktörü**” ile ifade olunan aktörü oluşturur.
6. Bu aktör kendi içinde **1.c** maddesi üzerinden temin edilen SAGA akışı tanımını okumak ve işletmek suretiyle gerekli mikro servis çağırımlarını arka arkaya gerçekleştirir. Bu kapsamda aktör Şekil 6.7’de ifade edilen **state-machine** diyagramında görülen durum değişimlerini icra eder. Bu değişimler aşağıda harflerle ayrılan alt adımları içermektedir:

- a. Aktöre SAGA tanımı yüklenerek "Actor initialized" durumuna geçişi sağlanır.
- b. Ardından SAGA tanımı kapsamında çağıracağı ilk mikro servis metodunu (Örnek: bakiyeKontrolEt) işleteceği "Progress Executing" durumuna geçiş yapar. Mikro servis metodunun çağırımı başarılı ise aktör "Progress Succeeded" durumuna geçer.
- c. Aktör, bütün servislerdeki bütün metot çağırımlarını başarılı bir şekilde bitinceye kadar "Progress Executing" ve "Progress Succeeded" durumları arasında geçiş yapar. Eğer bütün mikro servis metot çağırımları başarılı ise aktör sırasıyla "All Progresses Finished" durumuna geçer ve ardından state-machine sonlanır.
- d. Eğer mikro servis metot çağırımlarından herhangi birinde bir hata hasıl olursa aktör "Rollback Executing" durumuna geçer ve geriye doğru daha önce çağırdığı bütün mikro servis metotlarına tekabül eden rollback (geri sarma) metotlarını ters sırada çağırır. Bir başka deyişle aktör başarısız olan mikro servis metodu çağırımından önceki bütün mikro servislerdeki ilgili rollback metotlarını tek tek çağırır. Bu sırada aktör "Rollback Executing" ve "Rollback Succeeded" durumları (state'leri) arasında geçiş yapar. Eğer geriye doğru bütün rollback çağırımları başarılı olursa aktör "All Rollbacks Finished" durumuna geçer ve state-machine sonlanır.
- e. Eğer herhangi mikro servis üzerindeki rollback metodu çağırımı başarısız olursa aktör sırasıyla önce "Rollback Failed" ve arkasından da Incomplete duruma geçmek suretiyle state-machine'i sonlandırır.



Şekil 6.7. Bir aktörün SAGA Akışı İşletimi Sırasında Kullandığı State-Machine Diyagramı

7. Aktör SAGA akışını işlettikten sonra **1.b** adımında temin edilen kritik mikro servis-metot isimlerini içeren kümeyi kullanarak **1.a** adımında temin edilen müşteri numarası özelinde NOSQL veri tabanında bir temizlik yapar (yani bir clean-out prosedürü işletir). Bu temizlik işlemi için aktör, kaba kodu aşağıda ifade edilen `temizlikYap` metodunu işletir.

```

temizlikYap(müşteriNo as int,
              setId as UUID,
              kritikServisMetotListesi as JSON Array):void

```

BEGIN

a. **MüşteriNo** parametresi ile **setId** adlı giriş parametresini anahtar olarak kullanmak suretiyle NOSQL veri tabanına bir sorgu atılır. Bu sorgu neticesinde aşağıda örnek 4'teki gibi bir kayıt döner. Bu kayıta ilgili müşteri no ve setId özelindeki kritikServisAdı - metotAdı - Counter üçlüsünü barındıran bir liste görülmektedir. Bu liste esasen örnek 2'de görülen veri yapısındaki tek bir kümeyi ifade etmektedir.

Örnek 4:

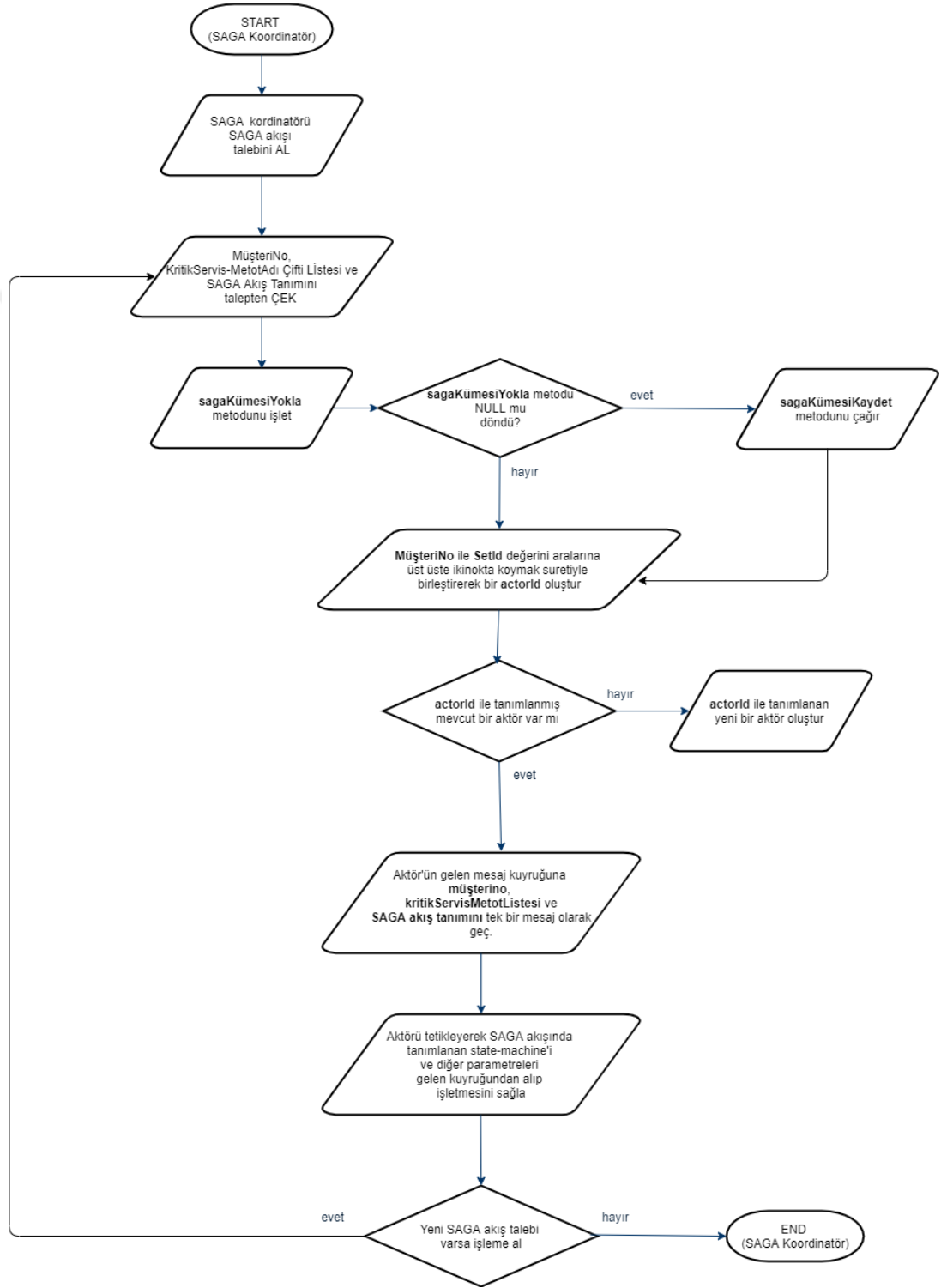
```
"Value":
[
  {
    "ServisAdı": "Servis1",
    "MetotAdı": "x",
    "Counter": 1
  },
  {
    "ServisAdı": "Servis1",
    "MetotAdı": "y",
    "Counter": 1
  },
  {
    "ServisAdı": "Servis2",
    "MetotAdı": "m",
    "Counter": 1
  }
]
```

- b. **kritikServisMetotListesi** adlı metot parametresi (içeriği örnek 1'deki gibidir) içindeki her bir **servisAdı-metotAdı** çifti ile **örnek 4'te** görülen küme'nin elemanlarındaki servis adı - metot adı çiftleri karşılaştırılır. Aralarında bir kesişim bulunması durumunda kesişimde bulunan servis adı - metot adı çiftlerinin **Counter** alanlarında görülen değerler **1** azaltılır.
- c. Değişiklikler **MüşteriNo** parametresi ile **setId** parametresini anahtar olarak kullanmak suretiyle NOSQL veri tabanına kaydedilir.
- d. Eğer c adımımda Counter alanlarında yapılan 1 azaltma sonucunda servis adı - metot adı çiftleriyle ilişkili bütün Counter alanları sıfır değerine haiz olursa bu kayıt **MüşteriNo** parametresi ile **setId** parametresini anahtar olarak kullanmak suretiyle NOSQL veri tabanından bütünüyle silinir.

END of method temizlikYap

8. Yukarıdaki 1'den 8'e kadar gerçekleştirilen bütün adımlar SAGA koordinatör tarafından aynı müşteri numarası için gelen talepler doğrultusunda binlerce kez yeniden gerçekleştirilebilir. Dolayısıyla aynı müşteri numarası için bir ya da daha fazla kritik servisi müşterek olarak kullanan binlerce farklı SAGA akışı aynı aktörün mesaj kuyruğuna sıraya konulmak suretiyle senkronize bir şekilde (yani arka arkaya) çalıştırılabilir. Bu suretle aynı müşteri için farklı SAGA akışlarının birbirinin kritik verisi üzerine yazmasının önüne geçilir ve tutarlılık (consistency) garanti edilir.

Yukarıdaki **1'den 8'a** kadar olan adımlarda izah edilen bu algoritmanın genel işleyişi aşağıda Şekil 6.8'deki akış-diyagramında özetlenmekte olup bu algoritmanın örnek bir bankacılık senaryosu üzerinden işletimi sonraki bölüm olan bölüm 6.3.2'de açıklanmaktadır.



Şekil 6.8. Buluş Kapsamındaki Algoritmik Yöntemin Akış Diyagramı

6.3.2. Örnek senaryo işletimi

Hatırlanacağı üzere bölüm 6.1’de Ali Bey ve Ayşe Hanım adlı hayali bir çift üzerinden örneklenen bir **dirty-read** senaryosu ele alınmıştı. Bu çift Şekil 6.3 ve 6.4’te görülen nakit çekme ve döviz alış adlı SAGA akışlarını gerçekleştirirken bankayı zarara uğratacak şekilde bir dirty-read senaryosunun gerçekleşmesine neden olmakta idi. Bu örnek dirty-read senaryosunun oluşumunun yukarıda bölüm 6.3.1’de açıklanan ve Şekil 6.8’de akış diyagramı verilen algoritmik yöntemin kullanımı ile nasıl bertaraf edilebileceği bir örnek oluşturması amacıyla aşağıdaki adımlarda açıklanmaktadır:

1. SAGA koordinatör, **Ali Bey** tarafından başlatılan nakit çekme SAGA akışının çalıştırılması talebini bir istemci uygulamasından bir Remote-Procedure-Call (RPC) çağırımı marifetiyle alır (Burada istemcinin bir bankamatik cihazı olduğu varsayılabilir). SAGA koordinatör, aldığı RPC talebinin (RPC request) içeriğinden aşağıdaki şu üç parametreyi çekip alır.

a. **Müşteri Numarası:** 99999

b. **Kritik Servis-Metot Listesi:** Ali Bey’in başlattığı nakit çekme talebi kapsamındaki SAGA akışında işletilecek kritik mikro servislerin tanımlayıcı isimleri (Örnek: Hesap, Muhasebe) ile bu servisler üzerinde tanımlı olan ve transactional işlemleri ifa eden metotların adlarından oluşan servis-metot çiftleri listesini (JSON formatında) temsil eder.

Örnek 5:

```
[
  {
    "ServisAdı": "HesapServisi",
    "MetotAdı": "bakiyeKontrolEt"
  },
  {
    "ServisAdı": "HesapServisi",
    "MetotAdı": "bakiyedenTutarDüş"
  },
  {
    "ServisAdı": "MuhasebeServisi",
    "MetotAdı": "muhasebeFişiniKes"
  }
]
```

c. **SAGA akışının tanımı:** JSON formatında Nakit Çekme SAGA akışı (bkz. EK- 3).

2. SAGA koordinatör, kendi bünyesinde mevcut bulunan ve bölüm 6.3.1’de açıklanan *sagaKümesiYokla* adlı metodu **1.a** ve **1.b** adımlarında temin ettiği

99999 müşteri numarası ve *kritikServisMetotListesi* adlı giriş parametresini kullanmak suretiyle işletir. Bu kapsamda *sagaKümesiYokla* metodu, bir NOSQL veri tabanına sorgu atar ve 99999 müşteri numarası özelinde *kritikServisMetotListesi* ile kesişen bir küme olup olmadığını kontrol eder.

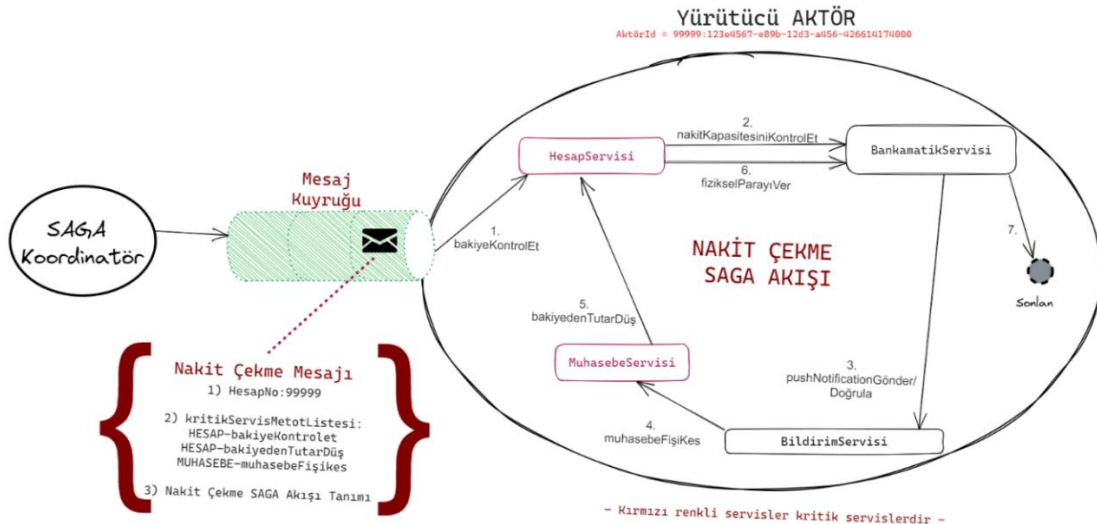
3. Yukarıdaki 2. adımda gerçekleştirilen *sagaKümesiYokla* adlı metot çağırımı neticesinde *kritikServisMetotListesi* ile kesişen bir kümenin NOSQL veri tabanında bulunmadığını buradaki örnek senaryo kapsamında varsayalım. Bu durumda *sagaKümesiYokla* metodu NULL döner.
4. 3. adımdaki *sagaKümesiYokla* adlı metot NULL döndüğü için SAGA koordinatör **1.a** adımında temin edilen müşteri numarası (yani 99999) ile **1.b** adımında temin edilen servis:metot çiftlerini (bkz. örnek 5) *sagaKümesiKaydet* metodunun sırasıyla *müşteriNo* ve *kritikServisMetotListesi* adlı giriş parametrelerine besleyerek bu metodu çalıştırır. *sagaKümesiKaydet* metodunun çalışması bitince NOSQL veri tabanında 99999 müşteri numarası özelinde aşağıda örnek 6’da görüldüğü gibi JSON formatında bir veri yapısı oluşturulur.

Örnek 6:

```
"99999":
[
  {
    "SetId": "123e4567-e89b-12d3-a456-426614174000",
    "Value":
    [
      {
        "ServisAdı": "HesapServisi",
        "MetotAdı": "bakiyeKontrolEt"
        "Counter": 1
      },
      {
        "ServisAdı": "HesapServisi",
        "MetotAdı": "bakiyedenTutarDüş"
        "Counter": 1
      },
      {
        "ServisAdı": "MuhasebeServisi",
        "MetotAdı": "muhasebeFişiniKes"
        "Counter": 1
      }
    ]
  }
]
```

5. Örnek 6’da görülen veri yapısında *SetId* alanında mevcut bulunan UUID değeri SAGA koordinatöre dönülür.

6. SAGA koordinatör, **1.a** adımında temin edilen müşteri numarası (yani 99999) ile 5. adımda kendisine dönülen "123e4567-e89b-12d3-a456-426614174000" UUID değeri arasına ':' karakterini bir ayraç olarak koymak suretiyle bir aktör tanımlayıcısı (actor identifier) oluşturur. Yani bu aktör tanımlayıcısı "99999:123e4567-e89b-12d3-a456-426614174000" şeklinde bir değere sahip olur. Bu şekilde oluşturulan aktör tanımlayıcısı bir sonraki adımda oluşturulacak olan aktör numunesini (actor instance) benzersiz (unique) bir şekilde tanımlayacaktır. Dolayısıyla bir unique identifier teşkil edecektir.
7. SAGA koordinatör 6. adımda oluşturduğu aktör tanımlayıcısını kullanarak yeni bir aktör oluşturur ve hemen akabinde bu aktörün mesaj kuyruğuna **1.a**, **1.b** ve **1.c** maddelerinde temin edilen parametreleri Şekil 6.9'da görüldüğü gibi bir "nakit çekme mesajı" olarak geçirir. Bunun akabinde nakit çekme SAGA akışını yürütecek bir aktör oluşturulur. Bu adımın pratikte yani sahada gerçekleştirilmesi için "*Distributed Application Runtime- DAPR*" ya da *Akka* gibi açık kaynak kodlu olan ve aktör modelinde uygulama geliştirmeye imkân tanıyan yazılım kütüphaneleri kullanılabilir. Bölüm 5'te bu kapsamda DAPR kütüphanesinin sahada nasıl kullanılabileceği açıklanmaktadır.



Şekil 6.9. Nakit Çekme SAGA Akışının Aktör Marifetiyle İşletilmesi

8. Bu yeni oluşan aktör kendi içinde **1.c** maddesi üzerinden temin edilen SAGA akışı tanımını okumak ve işletmek suretiyle gerekli mikro servis çağırımlarını

Şekil 6.9’da görüldüğü gibi gerçekleştirir. Bu kapsamda aktör, Şekil 6.7’de görülen state-machine’deki durum değişimlerini icra eder.

9. 8. adımda, aktör nakit çekme akışını ifa ederken bu arada SAGA koordinatör Ayşe Hanım tarafından yine 99999 numaralı müşterek hesap numarası için başlatılan yeni bir SAGA akış talebini bir RPC mesajı olarak alır. Döviz-alış SAGA akışının çalıştırılmasını gerektiren bu talep HESAP ve MUHASEBE kritik servislerine ek olarak DÖVİZ kritik servisinin de kullanımını gerektirmektedir. Burada istemcinin Ali Bey’in bankamatik kullanımından farklı olarak bir mobil cihaz uygulaması olduğu varsayılmaktadır (Bankamatikten yapılmasında esasen bir farklılık yoktur. Sadece çeşitliliğin sağlanabileceğini vurgulamak adına böyle bir varsayım yapılmıştır).

SAGA koordinatör bu adımda Ayşe Hanım adına aldığı döviz-alış talebinin içeriğinden aşağıdaki şu üç parametreyi çeker:

- a. **Müşteri Numarası:** 99999
- b. **Kritik Servis-Metot Listesi:** Ayşe Hanım’ın başlattığı döviz alış talebi kapsamındaki SAGA akışında işletilecek kritik mikro servislerin tanımlayıcı isimleri (Örnek: **Hesap, Muhasebe, Döviz**) ile bu servisler üzerindeki transactional işlemleri ifa eden metotların adlarından oluşan servis-metot çiftlerini içeren JSON formatında bir listedir (bkz. örnek 7).

Örnek 7:

```
[
  {
    "ServisAdı": "HesapServisi",
    "MetotAdı": "bakiyeKontrolEt"
  },
  {
    "ServisAdı": "HesapServisi",
    "MetotAdı": "bakiyedenTutarDüş"
  },
  {
    "ServisAdı": "MuhasebeServisi",
    "MetotAdı": "muhasabeFişiniKes"
  },
  {
    "ServisAdı": "DövizServisi",
    "MetotAdı": "tlEşdeğeriniHesapla"
  }
]
```

- c. **SAGA akışının tanımı:** JSON formatında Nakit Çekme SAGA akışı (bkz. EK 2).

10. SAGA koordinatör, ilgili müşterek hesap numarasını (99999) kullanarak NOSQL veri tabanına sorgu atar. Bu sorgudaki amaç daha evvel 99999 müşteri numarası için herhangi bir aktör tarafından bir SAGA akışının işletilip işletilmediğini ve eğer işletildiyse bu işletimin hangi kritik servisleri ve metotlarını kapsamış olduğunu tespit etmektir. Bu amaçla SAGA koordinatör, kendi bünyesinde mevcut bulunan *sagaKümesiYokla* adlı metodu **9.a** ve **9.b** adımlarında temin ettiği 99999 müşteri numarası ve kritik servis metot listesini kullanmak suretiyle çalıştırır. Bu çalıştırma sırasında *sagaKümesiYokla* metodu, NOSQL veri tabanına bir sorgu atar.

11. *sagaKümesiYokla* metodunun NOSQL veri tabanına attığı sorgu kapsamında daha evvel 4. Adımda veri tabanında oluşturulmuş olan kayıt döner. Bu kayıt 99999 müşteri numarası özelinde "123e4567-e89b-12d3-a456-426614174000" *SetId*'si ile tanımlanan ve örnek 6'da görülen kayıttır. SAGA koordinatör, bu kayıttaki *Value* alanında tutulan küme listesi ile **9.b** adımındaki kritik servis adlarından oluşan küme (bkz. örnek 7) arasında aşağıda örnek 8'de kırmızı çerçeve içinde vurgulanan kesişimin olduğunu tespit eder.

Örnek 8:

```
"99999":
[
  {
    "SetId": "123e4567-e89b-12d3-a456-426614174000",
    "Value":
    [
      {
        "ServisAdı": "HesapServisi",
        "MetotAdı": "bakiyeKontrolEt",
        "Counter": 1
      },
      {
        "ServisAdı": "HesapServisi",
        "MetotAdı": "bakiyedenTutarDüş",
        "Counter": 1
      },
      {
        "ServisAdı": "MuhasebeServisi",
        "MetotAdı": "muhasabeFişiniKes",
        "Counter": 1
      }
    ]
  }
]
```

12. SAGA koordinatör, *kritikServisMetotListesi* ile edindiği *HesapServisi-bakiyeKontrolEt*, *HesapServisi-bakiyedenTutarDüş* ve *MuhasebeServisi-muhasebeFişiniKes* adlı 3 adet servis-metot çiftinin Ali Bey'in önceki nakit çekme talebine ilaveten ikinci kez kullanımlarının söz konusu olduğunu tespit eder. Bu evvelden kaynaklı kullanım nedeniyle örnek 8'de Value alanında görülen bu üç çiftle alakalı *Counter* değerlerini 1 artırır. Bu durum aşağıda örnek 9'da görülmektedir.

SAGA koordinatör ek olarak Ayşe Hanım'ın döviz alış talebi kapsamında ilave bir kritik servis olan *DövizServisi* ile bu servisteki *tlEşdeğeriniHesapla* adlı metodu da *kritikServisMetotListesi* parametresi içinde almış olduğu için *DövizServisi-tlEşdeğeriniHesapla* çiftini de Value alanına ekleyerek NOSQL veri tabanına kaydeder. Bu amaçla bölüm 6.3.1'de 3. adımda açıklanan *sagaKümesiKaydet* adlı metodu kullanır. NOSQL veri tabanında yer alan kaydın nihai durumu aşağıda örnek 9'da görüldüğü gibi olur.

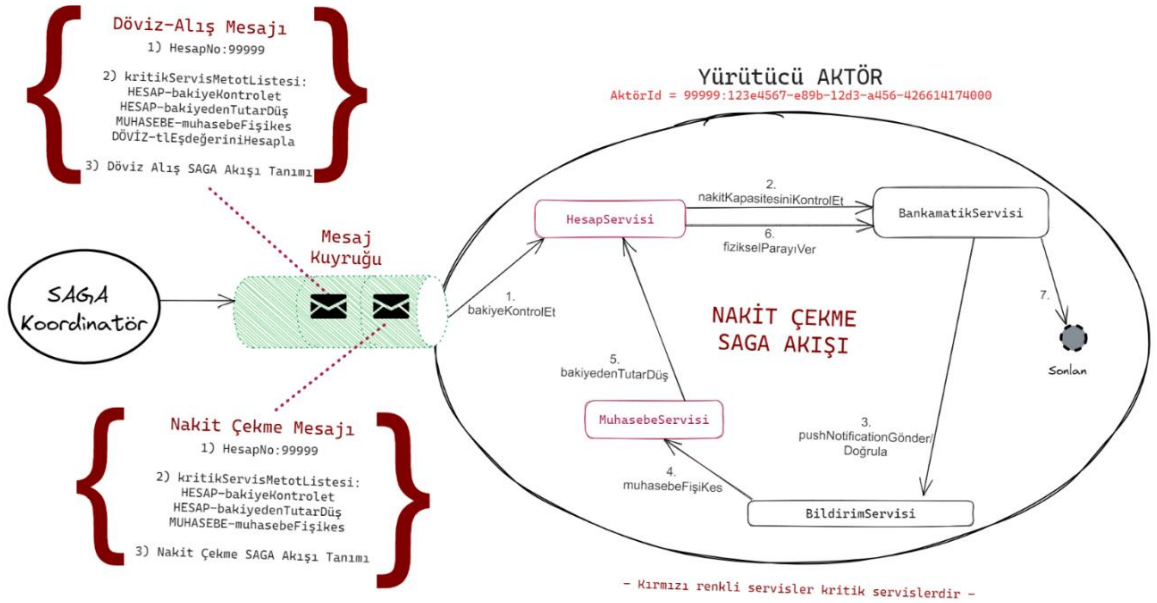
Örnek 9:

```
"99999":
[
  {
    "SetId": "123e4567-e89b-12d3-a456-426614174000",
    "Value":
    [
      {
        "ServisAdı": "HesapServisi",
        "MetotAdı": "bakiyeKontrolEt"
        "Counter": 2
      },
      {
        "ServisAdı": "HesapServisi",
        "MetotAdı": "bakiyedenTutarDüş"
        "Counter": 2
      },
      {
        "ServisAdı": "MuhasebeServisi",
        "MetotAdı": "muhasebeFişiniKes"
        "Counter": 2
      },
      {
        "ServisAdı": "DövizServisi",
        "MetotAdı": "tlEşdeğeriniHesapla"
        "Counter": 1
      }
    ]
  }
]
```

13. SAGA koordinatör, 9.a adımında temin edilen müşteri numarası (yani 99999) ile 11. adımda veri tabanından dönülen kayıttaki *SetId* (yani "123e4567-

e89b-12d3-a456-426614174000”) arasına ‘:’ karakterini bir ayıraç olarak koyar ve bu iki unsurun birleşimi olan bir aktör tanımlayıcısı (actorId) oluşturur. Yani bu aktör tanımlayıcısı “99999:123e4567-e89b-12d3-a456-426614174000” şeklinde bir değere sahip unique (benzersiz) bir identifier olur.

14. Dikkat edilirse önceki adımda oluşturulan aktör tanımlayıcısı (actorId) daha evvel 6. adımda Ali Bey’in nakit çekme talebi kapsamında oluşturulan aktör tanımlayıcısı ile birebir aynıdır. Bu nedenle SAGA koordinatör, başka bir aktör oluşturmaya lüzum görmeden halihazırda Ali Bey’in nakit çekme akışını işleten aktörün mesaj kuyruğuna Ayşe Hanım’ın döviz alış talebi için 9.a, 9.b ve 9.c maddelerinde temin edilen parametreleri Şekil 6.10’da görüldüğü gibi “Döviz-Alış Mesajı” başlıklı tek bir mesaj içerisine koyarak geçirir.



Şekil 6.10. Döviz Alış SAGA Akışı Talebinin Aktörün Mesaj Kuyruğuna Atılması

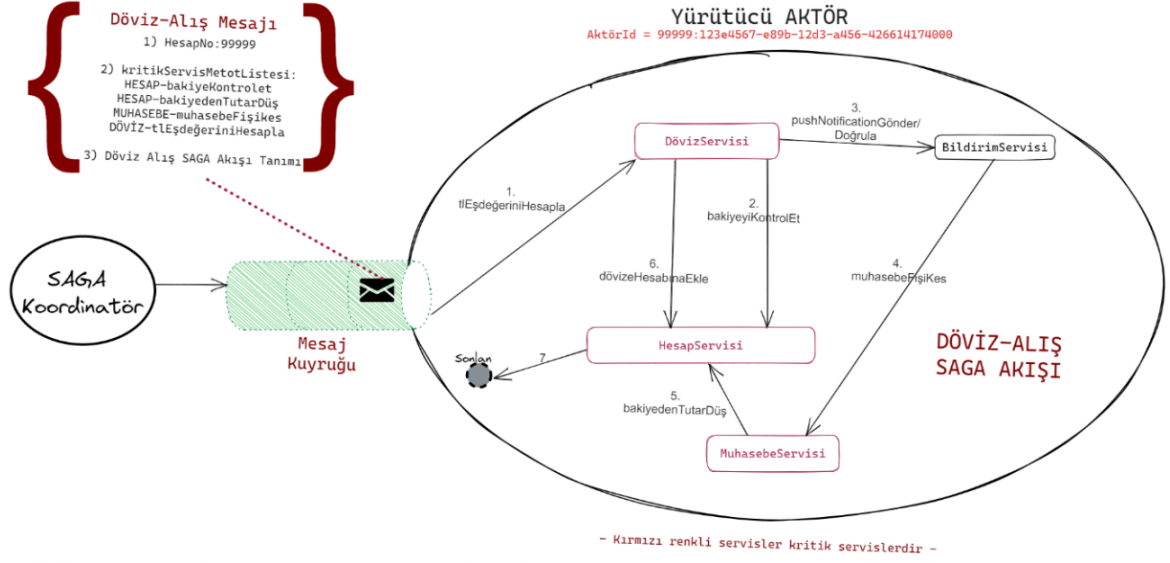
15. Aktör, halihazırda elinde çalıştırmakta olduğu Ali Bey’in başlattığı nakit çekme SAGA akışını tamamlayıncaya kadar Ayşe Hanım tarafından başlatılan döviz-alış SAGA akışının işlenmesini gelen mesaj kuyruğunda bekletir (Bu bekleme normal şartlarda oldukça kısadır). Aktör, elindeki nakit çekme SAGA akışını bitirir bitirmez *HesapServisi-bakiyeKontrolEt*, *HesapServisi-bakiyedenTutarDüş* ve *MuhasebeServisi-muhasebeFişiniKes* adlı 3 adet servis-metot çiftinin Counter değerlerini 1 azaltır ve *sagaKümesiKaydet*

metodu ile NOSQL veri tabanına kaydeder. Bunun sonucunda ilgili kaydın veri tabanındaki son hali aşağıda örnek 10’da belirtildiği gibi olur:

Örnek 10:

```
"99999":  
[  
  {  
    "SetId": "123e4567-e89b-12d3-a456-426614174000",  
    "Value":  
    [  
      {  
        "ServisAdı": "HesapServisi",  
        "MetotAdı": "bakiyeKontrolEt",  
        "Counter": 1  
      },  
      {  
        "ServisAdı": "HesapServisi",  
        "MetotAdı": "bakiyedenTutarDüş",  
        "Counter": 1  
      },  
      {  
        "ServisAdı": "MuhasebeServisi",  
        "MetotAdı": "muhasabeFişiniKes",  
        "Counter": 1  
      },  
      {  
        "ServisAdı": "DövizServisi",  
        "MetotAdı": "tlEşdeğeriniHesapla",  
        "Counter": 1  
      }  
    ]  
  }  
]
```

16. Aktör, gelen mesaj kuyruğundan sonraki yapacağı iş olan Ayşe Hanım’ın döviz alış talebi ile alakalı mesajı alır. Ardından bu mesajda belirtilen döviz alış SAGA tanımını okur ve Şekil 6.11’de görüldüğü şekilde işletmeye başlar (yani Döviz-alış SAGA akışındaki mikro servis çağırımlarını başlatır).



Şekil 6.11. Döviz Alış SAGA Akışı Talebinin Aktör Tarafından İşletilmesi

17. Aktör, Ayşe Hanım'ın döviz alış talebiyle ilgili SAGA akışının ikinci işlemi olan (*HesapServisi-bakiyeKontrolEt*) işletirken bakiyenin Ayşe Hanım'ın istediği döviz miktarının alınması için yetersiz olduğunu tespit eder çünkü Ali Bey tarafından daha önce talep edilen nakit çekme SAGA akışı müşterek hesap bakiyesinden Ali Bey'in talep ettiği nakit miktarını çoktan düşmüştür. Dolayısıyla aktör, döviz alış SAGA akışının kalan işlemlerini devam ettirmeden akışı sonlandırır. Bu nedenle bir **dirty-read** durumu Ali Bey ve Ayşe Hanım tarafından müşterek kullanılan 99999 numaralı müşterek hesap üzerinde söz konusu olmaz. Yani iki farklı SAGA akışı (nakit çekme ve döviz alış SAGA akışları) eş zamanlı değil; aksine arka arkaya (senkronize) çalıştırılmış olur. Bu bakımdan buraya kadar anlatılan aktör modeli esaslı yöntem sayesinde bakiye üzerinde tutarlılık sağlanmakta ve bankanın maddi bir zarara uğraması engellenmiş olmaktadır.

18. Aktör, nakit çekme ve döviz-alış SAGA akışlarını tamamladıktan sonra 1.a adımında temin edilen müşteri numarası ile 1.b ve 9.b adımlarında temin edilen kritik mikro servis isimlerini içeren liste özelinde ilgili NOSQL veri tabanında bir temizlik çalışması yapar. Bu amaçla, aktör bölüm 6.3.1'in 7. adımında açıklanan *temizlikYap* adlı metodu işletir. Bu kapsamda "123e4567-e89b-12d3-a456-426614174000" nolu *SetId* ile ilişkili *Value* alanındaki bütün servis-metot çiftleri ile alakalı *Counter*'ları bir azaltır ve NOSQL veri

tabanına kaydeder. Bu kapsamda eğer *Counter'ların* tamamı sıfıra inmiş ise örnek 10'da görülen ve 99999 numaralı müşteri özelindeki JSON kaydını NOSQL veri tabanından bütünüyle siler.

6.5. Bölüm Özeti

Bu bölümde, doktora çalışmamızın en can alıcı konusu ve akademik katkısı olan dağıtık SAGA işlem (transaction) yönetimi konusundaki buluşumuz ile ilgili detaylar açıklanmıştır. Bu buluş kapsamında, komut-orkestrasyon SAGA tasarım desenindeki merkezi koordinatör bileşeninde oluşabilecek performans darboğazını aktör modeli marifetiyle bertaraf eden bir teknik ortaya konulmuştur. Bu teknik sayesinde, hassas veri içeren veri kaynaklarına erişim yapan mikro servisleri müşterek olarak kullanan SAGA iş akışlarının (Örnek: Nakit çekme ve döviz alış SAGA iş akışları) sıralı (yani senkron) olarak işletilmesi mümkün kılınmış ve bu suretle hassas veri kaynakları üzerinde (Örnek: bakiye) hatalı güncelleme yapılmasının önüne geçilmiştir. Bu bakımdan buluş iki temel kazanım sunmaktadır:

- i. Komut-orkestrasyon esaslı SAGA deseninin aktör modeliyle genişletilmesi suretiyle merkezi koordinatör üzerinde bir performans darboğazı oluşumunun engellenmesi.
- ii. Hassas/kritik veri kaynaklarına (Örn.: Bankacılık müşterilerinin hesap bakiyelerini tutan veri tabanı tablosu) erişim gerçekleştiren mikro servisleri müşterek kullanan SAGA akışlarının sıralı bir şekilde (yani senkronize) çalıştırılmasını sağlayan bir algoritmik tekniğin aktör modeliyle birlikte çalışan bir kümeleme mantığı marifetiyle sağlanması.

Detayları bu bölümde izah olunan buluşumuzun sahada gerçekleştirilen endüstriyel uygulama şekli sonuçlar bölümünde (bölüm 7.3) sunulmaktadır.

7. SONUÇLAR

Bu bölümde doktora sürecinde gerçekleştirdiğimiz çalışmalar genel olarak özetlendikten sonra ortaya konulan mimari yapının bölüm 5’te 5.6.4 nolu alt başlıkta ortaya konulan yük testi sonuçlarının değerlendirilmesi yapılmaktadır. Bunun ardından Bölüm 6’da sunulan ve dağıtık SAGA işlemlerinin aktör modeli esaslı yönetimini sağlayan buluşun sahada uygulanışı açıklanmaktadır. Son olarak mikro servis mimarisinin sahada kullanımı konusunda edindiğimiz tecrübeler ışığında genel bir değerlendirme sunulmaktadır.

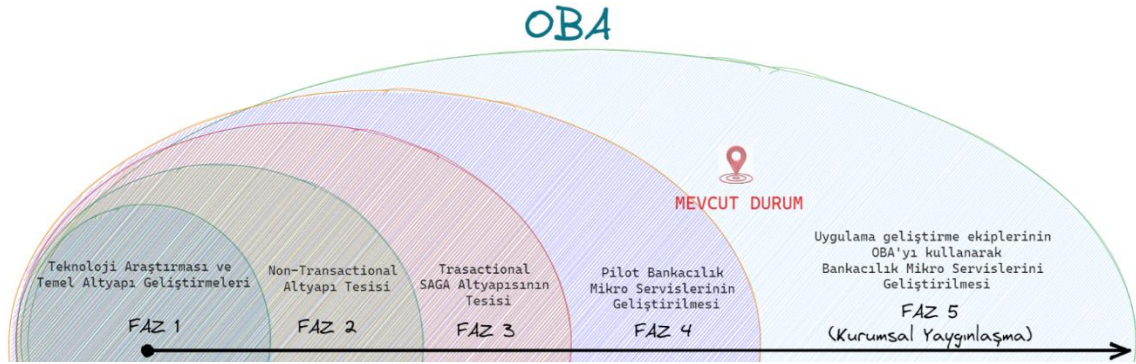
7.1. Tez Çalışmasının Genel Özeti

Tez kapsamında bir bankanın sahip olduğu monolit yapıdaki temel bankacılık sisteminin açık bankacılık standartlarına uyumlu bir bankacılık platformuna mikro servis mimarisi temelinde nasıl dönüştürülebileceği ele alınmıştır. Bu doğrultuda temelinde mikro servis mimarisini esas alan bir yazılım sisteminin geliştirilmesi için gerekli teknikler/modeller/tasarım kalıpları araştırılıp uygulanabilirliği tespit edilmiş ve bu doğrultuda mimari bir model geliştirilmiştir. Geliştirilen bu modelin sahadaki doğrulaması Kuveyt Türk Katılım Bankası bünyesinde gerçekleştirilen bir TÜBİTAK 1505 üniversite-sanayi iş birliği projesi kapsamında endüstriyel bir vaka çalışması kapsamında uygulamalı olarak yapılmıştır. Bu vaka çalışması kapsamında Kuveyt Türk Bankası bünyesinde halihazırda kullanılmakta olan monolit yapıdaki **“Business Oriented Architecture (BOA)”** adlı temel bankacılık yazılım altyapısı **mikro servis mimarisi** esaslı bir açık bankacılık yazılım altyapısına evirilmiştir. Bu doğrultuda uygulamalı olarak ortaya koyduğumuz mikro servis mimarisi tabanlı yazılım platformu **Açık Bankacılık Mimarisi (Open Banking Architecture- OBA)** olarak adlandırılmıştır. Buradan hareketle doktora çalışmamız kapsamında ortaya konulan çalışmalar şu şekilde özetlenebilir:

- 1) OBA tarafından dış kullanıma açılacak olan bankacılık servislerinin açık bankacılık standartlarında bankacılık işlev sınıflarına ayrıştırılmak suretiyle tespit edilmesi ve servisler bünyesinde sağlanacak Application Programming Interface’ler (API’ler) tespit edilmiştir.
- 2) OBA’ya evirilecek olan monolit BOA mimarisinin incelenmiş ve bu mimarideki sorunlar ortaya konmuştur.

- 3) OBA'nın mikro servis mimarisi çerçevesinde genel mimari unsurları ve temel altyapı bileşenleri (yani OBA bileşenlerinin) tespit edilip bu bileşenlerin sorumlulukları ve kendi aralarındaki etkileşimleri tasarlanmış ve vaka çalışması kapsamında uygulanmıştır.
- 4) OBA'da mikro servisler arası dağıtık SAGA işlemlerinin (Distributed SAGA Transactions) yönetimini sağlayan bir buluş ortaya konmuştur.

Yukarıda 1'den 4'e kadarki maddelerde ifade olunan işler doktora çalışmamızın ana akademik katkılarını oluşturmaktadır. Bu katkılar, gerçekleştirdiğimiz TÜBİTAK 1505 projesi kapsamında birebirde sahada uygulanma imkânı bulmuş ve bu tezin yazımı sırasında Kuveyt Türk Bankası veri merkezinde fiilen canlıya alınmıştır. Dolayısıyla bu tez çalışması kapsamında ortaya konan mimari model (OBA) bütünüyle yerli imkanlarla tasarlanıp geliştirilen mikro servis mimarisi temelli bir bankacılık yazılım altyapısı ürününe dönüşmüştür. Bu bakımdan doktora çalışmamız kapsamında gerçekleştirilenler kâğıt üzerinde kalmayıp ülkemizin yerli ARGE kapasitesine gerçek manada bir endüstriyel katkı ortaya koymuştur. Bu endüstriyel katkı Kuveyt Türk Bankası bünyesinde fazlandırılmış bir yazılım projesi olarak ele alınmış olup Şekil 7.1'de görülen ilerleme sürecine sahne olmuştur.



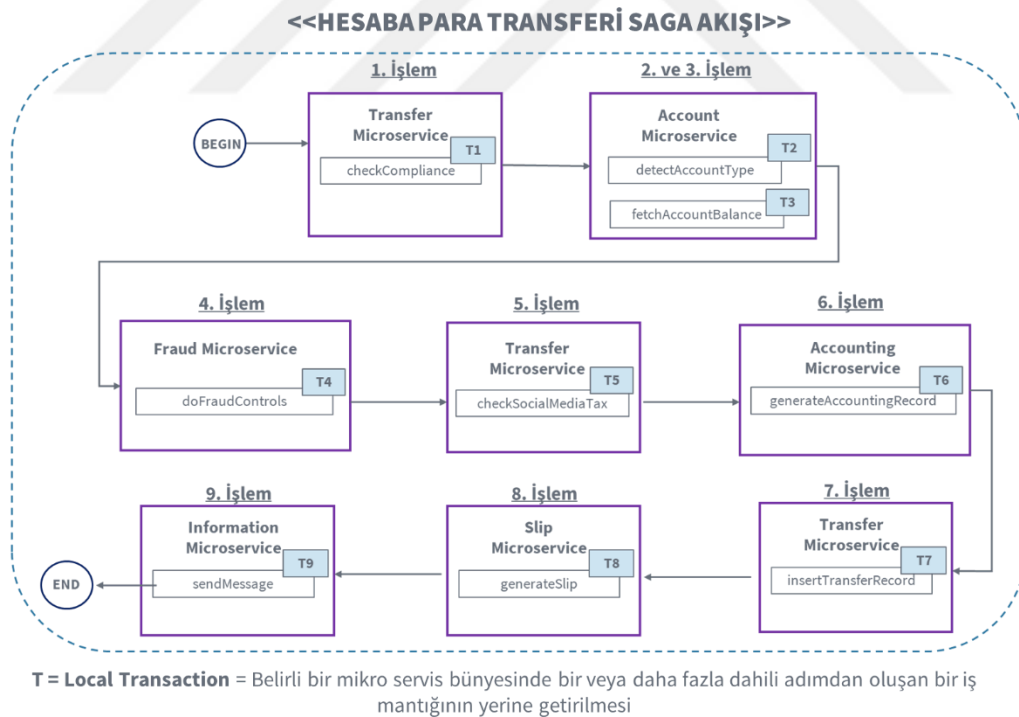
Şekil 7.1. Mikro Servis Temelli OBA Altyapısının Sahada Uygulanış Fazları

Şekil 7.1'de görülen ilk 3 faz doktora çalışmamız kapsamında icra ettiğimiz TÜBİTAK projesi çerçevesinde yürütülmüştür. Bu fazlar kapsamında bölüm 5'te 5.4 numaralı alt başlıkta açıklanan ve EK-1'de sunulan mimari model uygulanarak mikro servis mimarisi temelinde bir bankacılık yazılım geliştirme alt yapısı tesis edilmiştir. Ayrıca yazılım geliştiricilerin bu altyapıda BFF ve mikro servis geliştirmelerini

kolaylaştırmak amacıyla veri tabanı erişimi, hata yönetimi ve network iletişimi gibi konularda C# dilinde (.NET 7) kütüphaneler yazılmıştır.

Faz 4’te iki pilot mikro servis tasarlanıp geliştirilmiştir. Bu pilot servislere örnek olarak *döviz kuru mikro servisi (Foreign Exchange Streaming Service)* ve *içerik mikro servisi (Content Service)* verilebilir. Bunlardan ilki güncel döviz kurlarındaki değişimleri Kuveyt Türk istemcilerine (Örnek: mobil şube uygulaması) anlık olarak iletmek amacıyla geliştirilmiştir. İkincisi ise mobil şube uygulaması üzerinde görülen görsel imajların performanslı bir şekilde çekilmesini sağlayan bir içerik sağlama servisi.

Tezin savunma aşaması sırasında 5. faz henüz başlamış ve uygulama geliştirme (UG) ekiplerine eğitimler verilmiştir. Dolayısıyla UG ekiplerindeki (Örnek: Krediler ekipleri) yazılımcılar yakın zamanda OBA altyapısını kullanarak bankacılık servislerinin yazımına başlayacaktır. Bu servisler hususiyetle ilk olarak para transferleri ve genel müdürlük teftiş faaliyetleri kapsamında ihtiyaç duyulacak mikro servislerdir. Örneğin Şekil 7.2’de faz 5’te para transferi kapsamında geliştirilecek olan mikro servislerin etkileşimini barındıran SAGA akış tasarımı görülmektedir.



Şekil 7.2. Hesaba Para Transferi SAGA akışı

Şekil 7.2’de görülen SAGA akışı Kuveyt Türk’te hesaplar arası para transferi iş mantığı çerçevesinde geliştirilecek mikro servislerin hangi işlemleri (T: Transaction)

gerçekleştireceğini ve bu servislerin bir aktör tarafından ne sırada işletileceğini göstermektedir. Bu minvalde şu adımlar işletilecektir:

1. Transfer mikro servisi yapılacak para transferi kapsamında gönderici hesap ve alıcı hesap özelinde bazı uyum kontrolleri gerçekleştirecektir (`checkCompliance` işlemi - T1).
2. Hesap mikro servisi (Account Microservice) kapsamında para çekilecek ve gönderilecek müşteri hesabının tipi tespit edilecektir (`detectAccountType` işlemi - T2).
3. Hesap mikro servisi kapsamında para çekilecek müşteri hesabının bakiye durumu kontrol edilecektir (`fetchAccountBalance` işlemi - T3).
4. Sahtekarlık Kontrolü mikro servisi (Fraud Microservice) kapsamında transfer işleminin bir sahtekârlık teşebbüsü olup olmadığı kontrol edilecektir (`doFraudControls` işlemi - T4).
5. Transfer mikro servisi kapsamında para çekilecek müşterinin sosyal medya ile alakalı vergi kontrolleri sağlanacaktır (`checkSocialMediaTax` işlemi - T5).
6. Muhasebe mikro servisi (Accounting Microservice) kapsamında muhasebe fişi kesilecektir (`generateAccountingRecord` işlemi - T6).
7. Transfer mikro servisi tarafından veri tabanı üzerinde para transferi işlemi fiilen kayıt altına alınacaktır (`insertTransferRecord` işlemi - T7).
8. Dekont mikro servisi (Slip Microservice) kapsamında yapılan transfer işlemi ile ilgili dekont kaydı veri tabanında oluşturulacaktır (`generateSlip` işlemi - T8).
9. Bilgilendirme mikro servisi (Information Microservice) kapsamında transfere muhatap olan gönderici ve alıcı müşteriye bilgilendirici SMS mesajı gönderilecektir (`sendMessage` işlemi - T9).

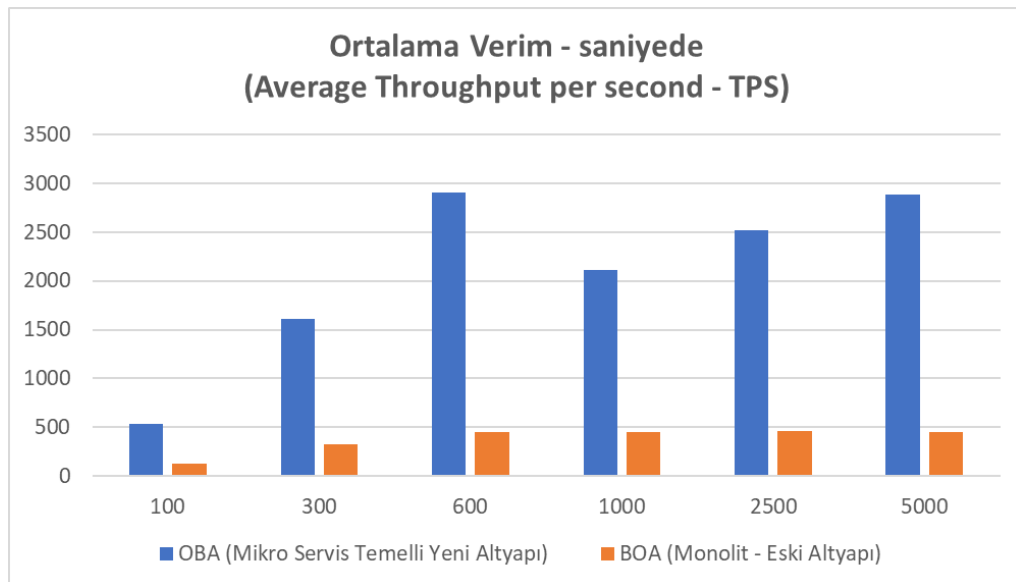
7.2. Yük Testi Değerlendirmesi

Bu alt bölümde mikro servis mimarisi temelinde ortaya koyduğumuz OBA altyapısının monolit BOA altyapısına karşı verimliliğini doğrulamak amacıyla gerçekleştirdiğimiz yük testleri kapsamında edindiğimiz sonuçların bir değerlendirmesi sunulmaktadır. Bu amaçla Şekil 7.3 ve 7.4'te iki adet histogram grafiği sunulmaktadır. Bu grafikler, Bölüm 5.6'da detayları açıklanan 5 dakikalık yük testi koşumları

kapsamında OBA ve BOA üzerinde gözlemlenen ortalama verim ve ortalama cevap sürelerindeki değişimi göstermektedir.

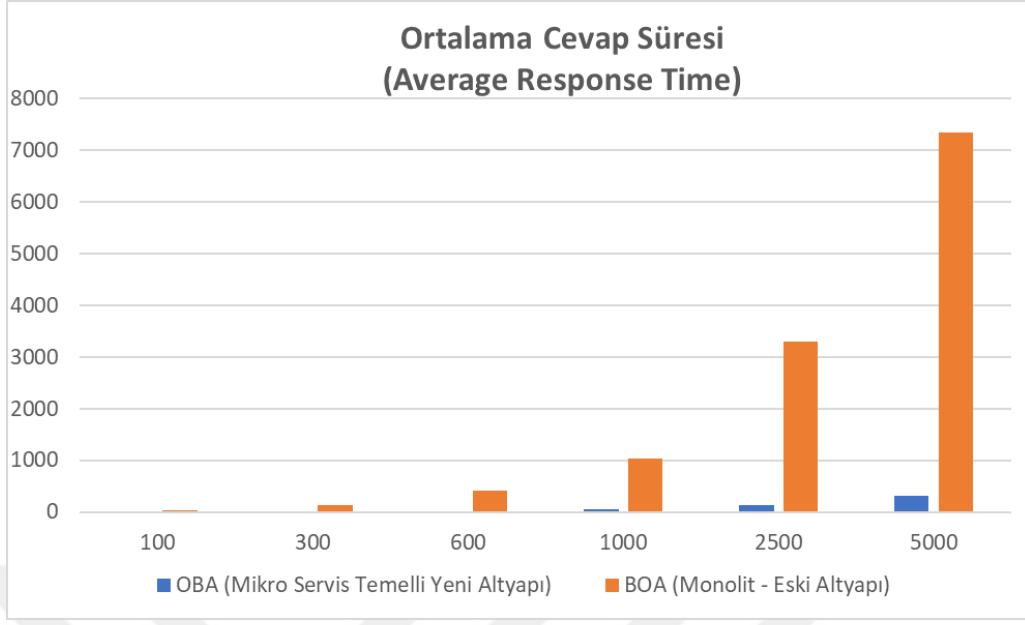
Bu grafiklerden Şekil 7.3'te görülen grafiğin düşey eksen 5 dakikalık toplam test koşumu süresinde saniyede ölçülen ortalama verim (average throughput) değerlerindeki değişimi gösterirken; Şekil 7.4'te görülen grafiğin düşey eksen ise yine 5 dakikalık toplam test koşumu süresinde ölçülen ortalama cevap sürelerini (average response times) göstermektedir. Diğer yandan her iki grafikte de yatay eksenler 100'den 5000'e kadar değişen aralıkta JMeter³⁸ tarafından koşturulan sanal kullanıcı sayısını (Virtual User-thread) temsil etmektedir. Burada 5000 sanal kullanıcı, 5 dakikalık toplam test koşumu süresince dakikada yaklaşık 5 milyon bankacılık request'ini (istemi) temsil eden bir canlandırma (yani simülasyon) olarak tasavvur edilebilir zira burada her sanal kullanıcı yüzlerce request'i (istemi) eş zamanlı olarak gerçekleştirebilmektedir.

Şekil 7.3 incelendiğinde OBA'nın ortalama veriminin sanal kullanıcı sayısı arttıkça istikrarlı bir şekilde ivmelenerek artış gösterdiği; buna karşın BOA'nın ortalama verim değerlerinin oldukça düşük olduğu gözlemlenmektedir. Öte yandan, Şekil 7.4 incelendiğinde ise OBA için ortalama cevap sürelerinin BOA'ya kıyasla çok az olduğu görülmektedir. Bu durum, bu tez çalışması kapsamında ortaya konulan mikro servis mimarisi tabanlı mimari modelin (yani OBA) sahada monolit muadiline kıyasla çok daha performanslı sonuçlar ortaya koyduğunu göstermektedir.



Şekil 7.3. Mikro Servis Temelli OBA ile Monolit BOA'nın Ortalama Verim Karşılaştırması

³⁸ <https://jmeter.apache.org/>



Şekil 7.4. Mikro Servis Temelli OBA ile Monolit BOA’nın Ortalama Cevap Sürelerinin Karşılaştırması

Diğer yandan donanım kaynaklarının kullanımı bakımından bir değerlendirme yapılmak istendiğinde Bölüm 5’teki 5.6.2.7 nolu başlık altındaki Tablo 5.7’ye bakıldığında Kubernetes test cluster’ındaki OBA bileşenleri tarafından toplamda yaklaşık 87 işlemci çekirdeği (Core) ve 72 GB bellek kullanımının yapıldığı görülmektedir. Buna karşın 5.5.1 nolu başlık altındaki Tablo 5.1 incelenerek bir hesaplama yapıldığında eski monolit (yani BOA) bileşenleri tarafından yapılan toplam kaynak kullanımının 96 işlemci çekirdeği ve 96 GB bellek olduğu sonucuna varılmaktadır. Bu durum, önerilen mikro servis tabanlı OBA altyapısının performans üstünlüğünün yanı sıra aynı zamanda kaynak kullanımı bakımından da BOA’ya göre daha verimli ve dolayısıyla üstün olduğunu göstermektedir.

7.3. Aktör Modeli Esaslı Dağıtık İşlem (Transaction) Yönetiminin Sahada Uygulanışı

SAGA akışlarının koordinasyonunu sağlamak amacıyla doktora çalışmamız kapsamında ortaya konulan ve detayları Bölüm 6’da sunulan buluş mikro servis mimarisi temelinde geliştirilecek dağıtık yazılım istemlerinin tamamında uygulanabilir bir niteliktedir. Bununla birlikte yoğun dağıtık işlemlerin (distributed transactions) gerçekleştiği bankacılık ve telekomünikasyon gibi sektörlerde daha fazla kullanım alanı

bulması muhtemeldir (Aydemir ve Başçiftçi, 2022 (1, 2)). Bu nedenle buluşumuzun sahada uygulanışı yoğun transactional işlemlerin söz konusu olması münasebetiyle Kuveyt Türk Bankası tarafından sağlanan kaynaklar kullanılarak bankacılık sektöründe tatbik edilmiştir. Bu kapsamda Şekil 7.2’de görülen para transferi SAGA akışını buluşumuz kapsamında izah olunan algoritmayı kullanarak icra eden bir mimari altyapı tasarlanıp geliştirilmiş tir. Detayları Şekil 7.5’te görülen bu altyapıda büyük harflerle işaret edilen şu temel sistem bileşenleri bulunmaktadır:

A. İstemci Uygulaması (Client App): Cep telefonu gibi mobil cihazlarda çalışan bir yazılım uygulamasıdır. Örnek olarak cep telefonlarında çalışan e-ticaret uygulamaları (Örnek: Trend yol) ile bankacılık uygulamaları (Örnek: Kuveyt Türk Mobil şube) verilebilir.

B. Keycloak: Keycloak, Open Authentication Güvenlik Protokolü (OAuth2.1) çerçevesinde istemci uygulamalarına yönelik kimlik doğrulama hizmeti sağlayan açık kaynak kodlu ve ücretsiz bir yazılım uygulamasıdır. Mikro servis altyapısına erişmek isteyen istemci uygulamalarının ve onu kullanan bireylerin kimlik kontrolünü (Authentication) gerçekleştirir. Kimlik kontrolünden geçen bir istemciye bir erişim anahtarı (access token) sağlar. İstemci bu anahtarı kullanarak API Geçidine network çağırımı gerçekleştirir. Bu ürünün genel dokümantasyonuna şu linkten erişilebilir: <https://www.keycloak.org/>

C. API Gateway (API Geçidi): İstemci uygulamalarının network üzerinden yaptıkları Uygulama Programlama Ara Yüzü (Application Programming Interface – API) çağırımlarını alarak işleyen ve akabinde bu çağırımları mikro servis altyapısındaki ilgili mikro servislere sevk eden bir yazılım uygulamasıdır. API Geçidi, ilgili mikro servislerin ürettiği cevapları çağırımı başlatan istemci uygulamaya geri döner. Şekil 10’da ortaya konan mimari kapsamında yine açık kaynak kodlu olarak ücretsiz bir kullanımı olan APISIX adlı API geçidinin kullanımı benimsenmiştir. Bu ürünün genel dokümantasyonuna şu linkten ulaşılabilir: <https://apisix.apache.org/>

D. SAGA Koordinatör: Bölüm 6.3.1’de ortaya konulan algoritmik yöntem kapsamında izah olunan özelmış bir mikro servistir. Aktörlerin oluşturulup bu aktörlerin sahip olduğu gelen kutusuna mesaj gönderiminden sorumludur.

E. mongoDB: Bir NOSQL veri tabanı sistemi olup ücretsiz bir şekilde edilecek olan ve ticari olmayan versiyonu (community version) kullanılmıştır. SAGA

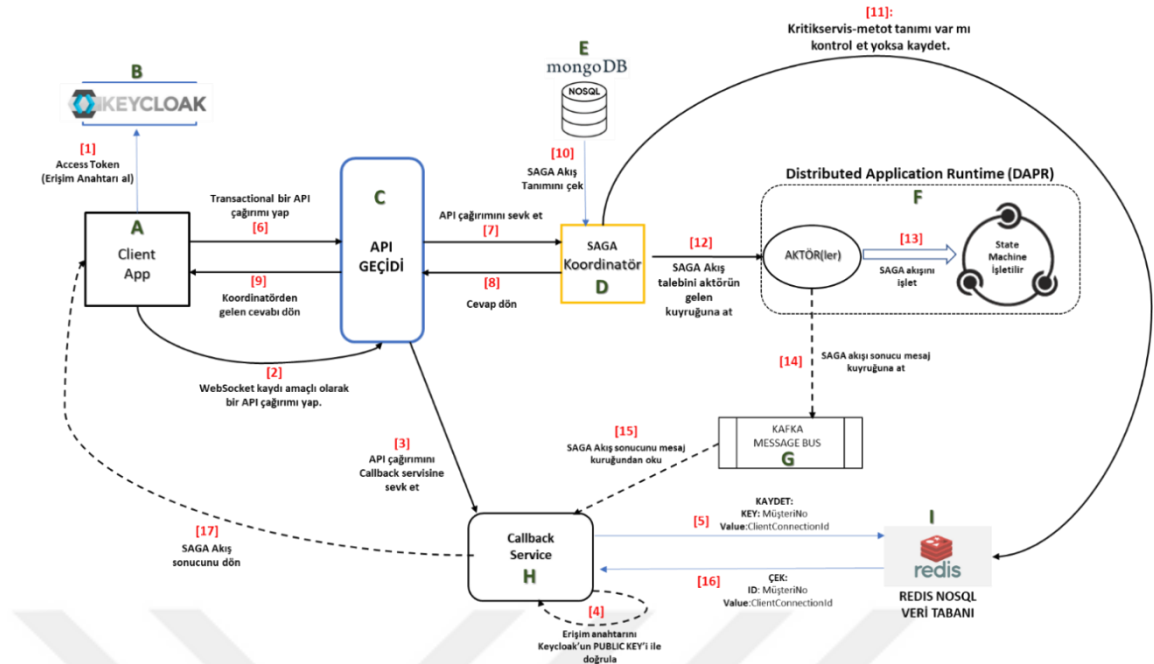
akışlarının kayıtlarını JSON formatında tutmaktadır. Bu ürünün genel dokümantasyonuna şu linkten erişilebilir: <https://www.mongodb.com/docs/>

F. Distributed Application Runtime (DAPR): Microsoft firmasının öncülüğünde, bulut tabanlı ve sunucusuz bilgi işlemi desteklemek için tasarlanmış ücretsiz ve açık kaynaklı bir dağıtık çalışma zamanı sistemidir (<https://dapr.io>). Bu sistem, aktörlerin oluşturulmasını, gelen kutularının ve yaşam döngülerinin yönetimini otomatik olarak sağlaması bakımından aktör modeli esaslı dağıtık sistem tasarımını kolaylaştırmaktadır. Bu nedenle bu buluşun pratik manada uygulanması kapsamında aktif olarak kullanılmıştır.

G. Kafka Veri Yolu (Kafka Message Bus): Kafka, açık kaynak kodlu ve ücretsiz bir mesajlaşma sistemidir. Mesajlar onları üreten bir yazılım uygulaması tarafından Kafka mesaj kuyruğuna beslenir. Bu kuyruğa abone olan başka yazılım uygulamalarına mesajlar otomatik olarak iletilir. Bu buluş kapsamında Kafka hususiyetle aktörlerin ilettiği SAGA akışlarının sonuçları hakkında istemci uygulamalarına bilgilendirici mesaj göndermek amacıyla kullanılmaktadır.

H. CallbackService: Transactional API çağırımları neticesinde bir aktör tarafından işletilen SAGA akışının sonucunu istemci uygulamasına (ClientApp'a) websocket protokolü üzerinden asenkron olarak iletmek amacıyla kullanılan özelleşmiş bir mikro servistir.

İ. Redis: Bir NOSQL veri tabanı sistemi olup ücretsiz bir şekilde edilecek olan ve ticari olmayan versiyonu (community version) kullanılmıştır. CallbackService tarafından kullanılır. ClientApp'larla websocket protokolü tabanlı geri bildirim iletişimi için gerekli `connectionId` değerleri tutulur. Ayrıca kritik mikro servis-metot çiftlerini barındıran JSON veri yapıları da redis'te tutulmaktadır. Bu ürünün genel dokümantasyonuna şu linkten ulaşılabilir: <https://docs.redis.com/latest/index.html>



Şekil 7.5. Buluşun Sahada Uygulanışını Yansıtan Sistem Mimarisi
(Daha büyük bir görünüm için EK-4'e bakınız)

Şekil 7.5'te görülen sistem mimarisindeki bileşenler üzerinde aşağıdaki şu adımlar çerçevesinde örneklenen bir işleyiş gerçekleşmektedir:

1. Client App, Keycloak'a bir authentication (kimlik doğrulama) talebinde bulunur ve cevabında Keycloak'tan bir access token (erişim anahtarı) alır.
2. Client App, kendini kullanan müşteriye ait müşteri numarası ile birlikte access token'ı içeren bir kayıt (registration) talebini bir API çağırımı yapmak suretiyle API Geçiði'ne iletir.
3. API geçiði aldığı kayıt talebini CallbackService olarak adlandırılan özelleşmiş bir mikro servise sevk eder (route eder).
4. CallbackService, kendisine daha önce keycloak tarafından sağlanan ve bünyesinde tuttuğu PUBLIC KEY vasıtasıyla access token'ı doğrular (validate eder).
5. CallbackService, genel dokümantasyonu **RFC6455** kapsamında yapılan **websocket** protokolüne göre clientApp ve müşteri numarası (Örnek: 99999) özelinde bir connectionId oluşturur ve bu connectionId'yi müşteri numarasını anahtar (key) olarak kullanacak şekilde redis veri tabanına kaydeder. Bu adım senkron bir Remote Procedure Call (RPC) çağırımı olarak gerçekleştiği için CallbackService clientApp'a hemen cevap dönerek (yani 200 OK dönerek) kayıt işleminin başarılı olduğunu bildirir.

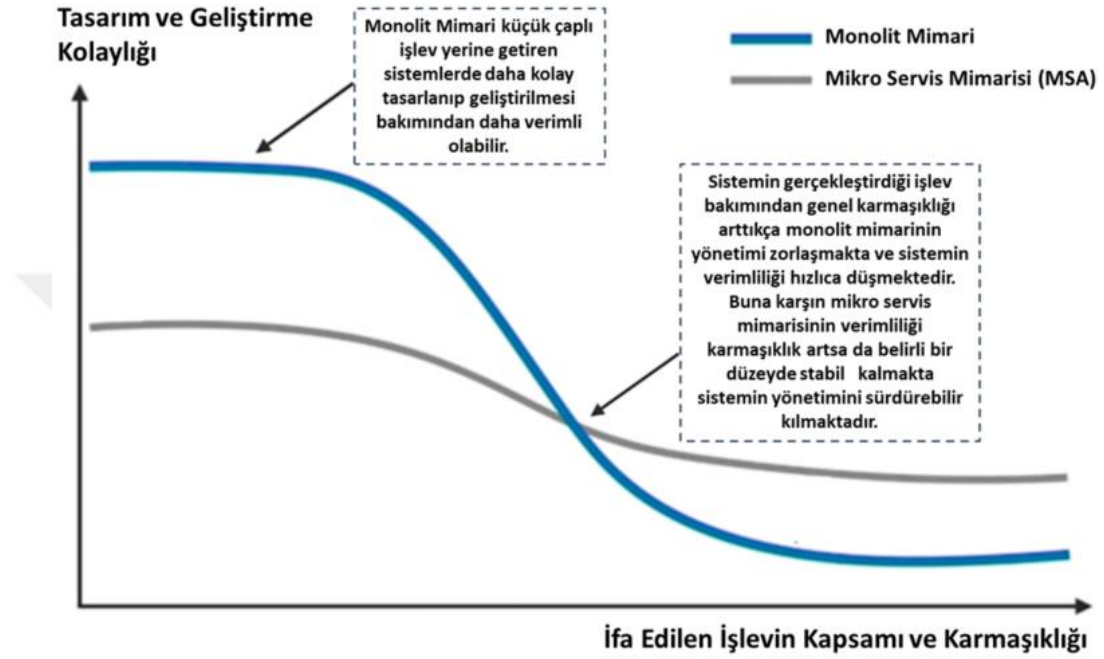
6. ClientApp, API geçidine transactional bir API çağırımı yapar (Örnek: nakit çekme talebi).
7. Transactional API çağırımı SAGA koordinatör adlı özelleşmiş mikro servise iletilir.
8. SAGA Koordinatör non-blocking tasarımı gereği hemen bir UUID oluşturur ve bunu API geçidine döner. Bu UUID bundan sonra CorrelationId olarak adlandırılacaktır.
9. API geçidi, SAGA koordinatörden dönen cevabı CorrelationId değeriyle birlikte ClientApp'a döner.
10. SAGA koordinatör, eş zamanlı olarak mongoDB adlı NOSQL veri tabanından transactional API çağırımı ile alakalı akış tanımını çeker (Örnek: Nakit çekme SAGA akışı tanımı – bkz. EK-3).
11. SAGA koordinatör, bu transactional API çağırımı kapsamında halihazırda bir kritik mikro servis-metot kaydının Redis veri tabanında olup olmadığını kontrol eder. Varsa SetId'sini alır ve actorId oluşturur. Yoksa da Redis'te yeni bir SetId'li yeni bir kayıt oluşturur ve müşteriNo ile setId'yi birleştirerek bir **actorId** oluşturur (Örnek: actorId = "99999:123e4567-e89b-12d3-a456-426614174000").
12. SAGA koordinatör, **müşteriNo**, **kritikServisMetot listesi** ve **SAGA akış tanımını** bir mesaj olarak paketleyip **actorId** eşliğinde DAPR runtime'ına havale eder. Bu amaçla DAPR'ın C#, Go, Java gibi programlama dillerindeki kütüphaneler (Software Development Kit- SDK) kullanılabilir (Biz hususen C#'ı tercih ettik). DAPR, bunun akabinde SAGA koordinatörden bağımsız olarak hareket ederek bir aktör oluşturur ve mesajı bu aktörün kuyruğuna atar.
NOT 1: SAGA koordinatör tarafından oluşturulan **actorId** halihazırda DAPR sisteminde mevcutsa DAPR sistemi yeni bir aktör oluşturmaz ve mevcut aktörün kuyruğuna ilgili mesajı sıraya koyarak atar.
NOT 2: DAPR ile alakalı daha geniş kapsamlı bilgiye şu adresten erişilebilir: <https://dapr.io/>
13. Aktör, Şekil 7.2'de görülen hesaba para transferi SAGA akışı kapsamında tanımlı bütün mikro servis çağırımlarını daha önce Bölüm 6.3.1'de Şekil 6.7'de görülen state-machine'i işletmek suretiyle gerçekleştirir.

- 14.** Aktör, SAGA akışının sonucunu (müspet ya da menfi olsun fark etmeden) Kafka Mesaj kuyruğuna bir mesaj olarak atar. Bu mesajda sırasıyla şu veriler barındırılır:
- a. SAGA akışının (yani hesaba para transferinin) hangi müşteri numarası için gerçekleştirildiği
 - b. Transactional API response'ı olarak 9. adımda dönülen CorrelationId
 - c. SAGA akışının işletilmesi sonucunda oluşan JSON formatındaki response objesi
- 15.** CallbackService, Kafka mesaj kuyruğuna abone durumda (subscribed) olduğu için 14. adımda üretilen mesajı alır.
- 16.** CallbackService, aldığı mesajdan müşteri numarasını çıkarır ve bunun Redis veri tabanına bir sorgu atarak daha evvel 4. adımda kaydedilmiş olan websocket connectionId değerini çeker.
- 17.** CallbackService, websocket connectionId'sini kullanarak kafkadan aldığı mesaj içeriğinde JSON formatında bulunan SAGA akışı sonucunu ClientApp'a asenkron olarak iletir.
- 18.** 2'den 17'ye kadar olan adımların herhangi birinde altyapı yeni farklı bir clietApp'tan farklı bir müşteri adına yeni bir transactional API çağırımı alabilir. Bu durumda 2-17 arasındaki adımlar bir şekilde paralel olarak işletilir zira SAGA koordinatör non-blocking tasarımı gereği kendisine gelen API çağırımlarına bekletmeden bir correlationId üreterek hemen cevap verir ve akabinde talebi DAPR sistemindeki bir aktöre havale eder.

7.4. Mikro Servis Mimarisinin Endüstride Genel Kullanımı Hakkındaki Değerlendirme

Doktora çalışmamız kapsamında gerçekleştirdiğimiz vaka çalışmasıyla elde edilen saha deneyimi bize mikro servis mimarisi temelli bir sistemin tasarımının ve uygulamasının dağıtık bir monolit sisteme kıyasla daha kompleks olduğunu ve farklı bir uzmanlık ve bakış açısı gerektirdiğini göstermektedir. Bu bakımdan Şekil 7.6'da ifade edildiği gibi gerçekleştirdiği işlev bakımından küçük boyutlu olan ve karmaşıklığı düşük olan yazılım sistemlerinin (Örn.: Muhasebe uygulaması, öğrenci kayıt sistemi) tasarımında ve geliştirilmesinde monolit mimarinin kullanımı daha kolay bir seçenek olarak karşımıza çıkmaktadır zira bu küçüklükteki bir sistemin tasarımı ve geliştirmesi

monolit mimari de daha kolaydır. Buna karşın yükleneceği işlev bakımından boyutu ve genel karmaşıklığı büyük olan sistemlerde (Örn.: Temel bankacılık sistemleri, telekomünikasyon sistemleri) genel karmaşıklığı daha verimli bir şekilde yönetmesi bakımından mikro servis mimarisi çok daha stabil ve kolay yönetilir bir yapı ortaya koymaktadır.



Şekil 7.6. Mikro Servis Mimarisi ile Monolit Mimarinin Karşılaştırması

Bu durumun altında yatan nedenlerden biri olarak monolit bir yapıya nazaran daha dağıtık (merkezi olmayan- distributed) bir yapı arz eden mikro servis mimarisinin arkasında bulunan konteyner, Kubernetes ve OAuth2 gibi açık kaynak kodlu teknolojilerin geliştiriciler için kavranıp uygulanmasının nispeten daha meşakkatli olması söylenebilir. Bu açıdan bakıldığında mikro servis mimarisi monolit mimariye kıyasla daha uzun ve kapsamlı bir öğrenme ve deneyimleme süreci gerektirmektedir. Bu nedenle, tasarımı çok dikkatli bir şekilde fazlı olarak yapılmalı ve her faz sonrası yük testleri tekrarlanarak sistem bileşenlerinin kalibrasyonu (Örn: POD sayısı) yapılmalıdır zira sistemin herhangi bir noktasında bir darboğaz oluşmadığından emin olmak gerekir. Bu nedenle bankacılık sistemleri gibi büyük ölçekli ve kompleks dağıtık sistemler için mikro servis mimarisi idealken; nispeten küçük boyutlu sistemler için daha basit olması nedeniyle monolit mimarinin uygulanması daha idealdir. Bununla birlikte karmaşık transactional işlemler gerektiren mikro servis mimarisinin, sistemsel değişikliklerin

gerçekleştirilmesini ve yönetimini kolaylaştırması, bulut tabanlı ölçeklenebilir teknolojilerin kullanımına olanak tanınması ve kısa aralıklarla yaygınlaştırmaya ve dolayısıyla çevik (agile) yazılım geliştirme yöntemlerinin kullanımına olanak tanınması bakımından geleceğin baskın mimarisi olacağı düşünülmektedir.



KAYNAKLAR

- Abdullah, M., Iqbal, W., ve Erradi, A., 2019, Unsupervised learning approach for Web application auto-decomposition into micro services. *Journal of Systems and Software*, 151, 243–257.
- Abernethy, R., 2019, *Programmer’s Guide to Apache Thrift*. Manning Publications Co.
- Afonso, L., Cerqueira, R., ve Souza, C. D., 2012, Evaluating application programming interfaces as communication artefacts, *PPIG*.
- Akalın, A., 2018, GraphQL – Veri Erişim Modellerine Farklı Bir Yaklaşım [online] <https://bilisim.io/2018/01/13/graphql-veri-erisim-modellerine-farkli-bir-yaklasim/> [Ziyaret Tarihi: 27 Kasım 2021].
- Aksoy B. ve Bilgel, D., 2019, Finansal Teknoloji Şirketleri ve Geleceğin Bankacılığı: Açık Bankacılık, *Avrupa Bilim ve Teknoloji Dergisi*, (17), 1097-1105.
- Al-Masri, E., 2019), Enhancing the Micro Service Architecture for the Internet of Things. In *Proceedings- 2018 IEEE International Conference on Big Data, Big Data 2018*.
- Alagöz, S., 2018, Entity Framework Ders5- EF Yaklaşımları [online], <https://snanalagoz.wordpress.com/2018/12/09/entity-framework-ders5-ef-yaklasimlari/> [Ziyaret Tarihi: 8 Aralık 2021].
- Aslan, R. M., 2019, Entity Framework Nedir? [Online], <https://medium.com/t%C3%BCrkiye/entity-framework-nedir-bff3943bec72> [Ziyaret Tarihi: 8 Aralık 2021].
- Anonim, 2015, Payment services (PSD 2) - Directive (EU) 2015/2366 [Online], European Commission, https://ec.europa.eu/info/law/payment-services-psd-2-directive-eu-2015-2366_en [Ziyaret Tarihi: 4 Kasım 2021].
- Anonim, 2017, Apache Kafka: An open-source distributed event streaming platform [Online], <https://kafka.apache.org/> [Ziyaret Tarihi: 6 Haziran 2021].
- Anonim, 2019 (1), Dünyada ve Türkiye’de Açık Bankacılık [online], <https://bkm.com.tr/wp-content/uploads/2015/06/Dunyadaveturkiyedeacikbankacilik.pdf> [Ziyaret Tarihi: 5 Haziran 2021].
- Anonim, 2019 (2), Ey.com [online], https://assets.ey.com/content/dam/ey-sites/ey-com/en_gl/topics/financial-services/ey-global-fintech-adoption-index-2019.pdf
- Anonim, 2019 (3), Türkiye’de ve Dünyada FinTek- Digital Banking Report. [Online] Digital Banking Report, Fintech İstanbul, [online], https://bkm.com.tr/wp-content/uploads/2015/06/Turkiyede_ve_Dunyada_Fintech.pdf [Ziyaret Tarihi: 5 Haziran 2021].

- Anonim, 2021 (1), .NET 5 Software Development Kit (SDK) and Runtime Environment [online]. <https://dotnet.microsoft.com/download/dotnet/5.0> [Ziyaret Tarihi: 5 Haziran 2021].
- Anonim, 2021 (2), Visual Studio 2019: Tmleřik yazılım geliřtirme ortamı (Integrated Development Environment – IDE) [online]. <https://visualstudio.microsoft.com/tr/vs/> [Ziyaret Tarihi: 5 Haziran 2021]
- Anonim, 2021 (3), Entity Framework (EF) Core: A lightweight, extensible, open source and cross-platform version of the popular Entity Framework data access technology [online]. <https://docs.microsoft.com/en-us/ef/core/> [Ziyaret Tarihi: 5 Haziran 2021].
- Anonim, 2021 (4), PostgreSQL: The World's Most Advanced Open Source Relational Database [online]. <https://www.postgresql.org/> [Ziyaret Tarihi: 5 Haziran 2021].
- Anonim, 2021 (5), Redis: An open source, in-memory data structure store, used as a database, cache, and message broker [online]. <https://redis.io> [Ziyaret Tarihi: 5 Haziran 2021].
- Anonim, 2021 (6), ASP.Net Web APIs: Building secure REST APIs with C# [online]. <https://dotnet.microsoft.com/apps/aspnet/apis> [Ziyaret Tarihi: 5 Haziran 2021].
- Anonim, 2021 (7), Docker: The de facto standard to build and share containerized apps-from desktop, to the cloud [online]. <https://www.docker.com/> [Ziyaret Tarihi: 5 Haziran 2021].
- Anonim, 2021 (8), JUnit: Testing framework for Java and the JVM [online]. <https://junit.org/junit5/> [Ziyaret Tarihi: 5 Haziran 2021].
- Anonim, 2021 (9), SoundCloud - evrimii ses daėıtım platformu ve mzik paylařım sitesi [online]. <https://tr.wikipedia.org/wiki/SoundCloud> [Ziyaret Tarihi: 5 Haziran 2021].
- Anonim, 2021 (10), Ruby on Rails: A server-side web application framework written in Ruby [online]. https://en.wikipedia.org/wiki/Ruby_on_Rails [Ziyaret Tarihi: 5 Haziran 2021].
- Anonim, 2021 (11), Kuveyt Trk's technology company Architect is the fastest-growing information company. <https://www.kuveytturk.com.tr/en/about-us/about-kuveyt-turk/news/kuveyt-turk-speeds-up-its-investments-in-technology-and-transformation-through-architect> [Ziyaret Tarihi: 5 Haziran 2021].
- Anonim, 2021 (12), Definition of FinTech by Oxford Dictionary [online], <https://en.oxforddictionaries.com/definition/fintech> [Ziyaret Tarihi: 18 Mayıs 2021].
- Anonim, 2021 (13), FinTech Investment H1 2021 – Innovate Finance [online], <https://www.innovatefinance.com/capital-and-investment/fintech-investment-2021-h1/> [Ziyaret Tarihi: 26 Aėustos 2021].

- Anonim, 2021 (14), Saga Distributed Transactions Pattern [online], <https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga> [Ziyaret Tarihi: 28 Kasım 2021].
- Anonim, 2023 (1), INNOPAY Open Banking Monitor <https://www.innopay.com/en/publications/sneak-preview-innopay-open-banking-monitor-q1-2023-old-guard-are-embracing-new> [Ziyaret Tarihi: 16 Ocak 2024].
- Anonim, 2023 (2), BirAPI Open Banking Türkiye Monitor https://www.linkedin.com/posts/birapi_openbanking-openbankingturkey-fintech-activity-7091696936046538753-nvSF/?utm_source=li_share&utm_content=feedcontent&utm_medium=g_dt_web&utm_campaign=copy [Ziyaret Tarihi: 16 Ocak 2024].
- Anonim, 2023 (3), The Berlin Group NextGenPSD2 Downloads <https://www.berlin-group.org/nextgenpsd2-downloads> [Ziyaret Tarihi: 16 Ocak 2024].
- Araluze, G. B., 2020, Adoption of Open Banking in Spain Do Spanish customers need Open Banking? Confident future November, 2020.
- Arner, D. W., Barberis, J. ve Buckley, R.P., 2015, The Evolution of Fintech: A New PostCrisis Paradigm? University of Hong Kong Faculty of Law. 1-14.
- Aydemir, F., Başçiftçi, F., 2022 (1), Building a Performance Efficient Core Banking System Based on the Microservices Architecture. J Grid Computing 20, 37 (2022). <https://doi.org/10.1007/s10723-022-09624-z>
- Aydemir, F., Basciftci F., 2022 (2), Strategies for Request-Response Logging in Microservices Architecture, 2022 *IEEE 20th Jubilee International Symposium on Intelligent Systems and Informatics (SISY)*, Subotica, Serbia, 2022, pp. 000121-000126, doi: 10.1109/SISY56759.2022.10036308.
- Aydemir, F., Basciftci F., 2022 (3), Application of HATEOAS Principle in RESTful API Design, 2022 *IEEE 22nd International Symposium on Computational Intelligence and Informatics and 8th IEEE International Conference on Recent Achievements in Mechatronics, Automation, Computer Science and Robotics (CINTI-MACRo)*, Budapest, Hungary, 2022, pp. 000051-000056, doi: 10.1109/CINTI-MACRo57952.2022.10029427
- Baidhowi, B., 2018, Sharia Banking Opportunities and Challenges in the Digital Era. *Advances in Social Science, Education and Humanities Research*, 192, 157-161.
- Bakshi, K., 2017, Microservices-based software architecture and approaches, 2017 IEEE Aerospace Conference, doi:10.1109/aero.2017.7943959
- Balalaie, A., Heydarnoori, A., ve Jamshidi, P., 2016, Mikroservice Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software*, 33(3), 42–52.

- Barbasura D., 2021, Why open banking APIs are so different, <https://www.linkedin.com/pulse/why-open-banking-apis-so-different-dmitrii-barbasura> [Ziyaret Tarihi: 24.10.2021].
- Beck, H., 2001, NETNOMICS: Economic Research and Electronic Networking 3: 7.
- Bejeck, W., P., 2018, Kafka Streams in Action, Manning Publishing.
- Bellemare, A., 2020, Building Event-Driven Microservices: Leveraging Organizational Data at Scale, O'Reilly Media.
- Bentaleb, O., Belloum, A.S.Z., Sebaa, A. et al. Containerization technologies: taxonomies, applications and challenges. J Supercomput 78, 1144–1181 (2022). <https://doi.org/10.1007/s11227-021-03914-1>
- Biehl, M., 2015, API Architecture- The Big Picture for Building APIs, API-University Press.
- Birrell, A. D., ve Nelson, B. J., 1984, Implementing Remote Procedure Calls. ACM Trans. Comput. Syst., 2(1), 39–59.
- Borsoft, 2016, Dapper Nedir? [Online], <https://www.borakasmer.com/dapper-nedir/> [Ziyaret Tarihi: 8 Aralık 2021].
- Bucchiarone, A., Dragoni, N., Dustdar, S., Larsen, S. T., ve Mazzara, M. (2018). From Monolithic to Microservices: An Experience Report from the Banking Domain. IEEE Software, 35(3), 50–55.
- Burns, B., 2018, Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services, O'Reilly Media.
- Çağıl G., Candemir G., 2020, A'WOT Analysis of Turkish Banking Sector Against FinTechs, Maliye ve Finans Yazıları, (113), 207 – 238.
- Calçada, P., 2014, Building Products at SoundCloud —Part I: Dealing with the Monolith [online], <https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith> [Ziyaret Tarihi: 16 Mayıs 2021].
- Carlson, J., 2013, Redis in Action, Manning Publishing.
- Carney, S., 2020, Why We Are Moving off REST and Implementing GraphQL [online] <https://www.courier.com/blog/why-we-are-moving-off-rest-and-implementing-graphql> [Ziyaret Tarihi: 4 Kasım 2021].
- Chen, R., Li, S., ve Li, Z., 2017, From Monolith to Microservices: A Dataflow-Driven Approach, 24th Asia-Pacific Software Engineering Conference (APSEC), 466–475.
- Colley, D., STANIER, C., ASADUZZAMAN, M., 2018, *The Impact of Object-Relational Mapping Frameworks on Relational Query Performance*, 2018

- Constantinides, P., Henfridsson, O., ve Parker, G. G., 2018, Introduction: Platforms and Infrastructures in the Digital Age, *Information Systems Research*, 29(2), 381–400.
- Conway, M., 1968, Conway's Law [online], http://www.melconway.com/Home/Conways_Law.html [Ziyaret Tarihi: 5 Haziran 2021].
- Cortet, M., Rijks, T., ve Nijland, S., 2016, PSD2: The digital transformation accelerator for banks, *Journal of Payments Strategy & Systems*, 10(1), 13-27(15).
- Cotton, I. W. ve Grestorex, F. S., 1968, Data structures and techniques for remote computer graphics, *AFIPS 1968 Fall Joint Computer Conference*. I. San Francisco, California: Association for Computing Machinery. pp. 533–544.
- Currie, T., C., 2017, Airbnb's 10 Takeaways from Moving to Microservices [online] <https://thenewstack.io/airbnbs-10-takeaways-moving-microservices/> [Ziyaret Tarihi: 5 Haziran 2021].
- Daigle, K., 2016, The GitHub GraphQL API [online] <https://github.blog/2016-09-14-the-github-graphql-api/> [Ziyaret Tarihi: 4 Kasım 2021].
- Daraghmi E., Zhang C-P., ve Yuan S-M. (2022). Enhancing Saga Pattern for Distributed Transactions within a Microservices Architecture. *Applied Sciences*. 2022; 12(12):6242.
- Date, C. J. ve E. F. Codd, 1974, *Relational Theory: A Detailed Review and Analysis of Codd's Major Database Writings*. p. 135.
- Deloitte, 2017, How to Flourish in an Uncertain Future: Open Banking and PSD2. <https://www2.deloitte.com/content/dam/Deloitte/uk/Documents/financial-services/deloitte-ukopen-banking-how-to-flourishin-an-uncertain-future.pdf>, [Ziyaret Tarihi: 10 Temmuz 2021].
- Demirdöğen, Y., 2019, FinTek Ekosistemi için Gerekli Düzenlemeler (REGTEK). *Vizyoner Dergisi*, 10(24), 311-321.
- Devrim, F., Özpençe, A., 4. KOBİ'ler ve Verimlilik Kongresi - openaccess.iku.edu.tr
- Dragoni, N., Dustdar, S., Larsen, S., ve Mazzara, M., 2017, Microservices: Migration of a Mission Critical System. *IEEE Transactions on Services Computing*, PP.
- Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., ve Safina, L., 2017, Microservices: Yesterday, Today, and Tomorrow, *Present and Ulterior Software Engineering*, 195–216.
- Du, S. G., Lee, J. W., ve Kim, K., 2018, Proposal of GRPC as a New Northbound API for Application Layer Communication Efficiency in SDN. *Proceedings of the 12th*

International Conference on Ubiquitous Information Management and Communication.

Edpresso Team, 2021, What is Object Relational Mapping? [Online], <https://www.educative.io/edpresso/what-is-object-relational-mapping> [Ziyaret Tarihi: 8 Aralık 2021].

Eroglu H., 2017, Comparing the Berlin Group and Open Banking UK API Standards for PSD2, <https://www.finextra.com/blogposting/14834/comparing-the-berlin-group-and-open-banking-uk-api-standards-for-psd2> [Ziyaret Tarihi: 24.10.2021].

Escobar, D., Diana, C., Amarillo, R., Castro, E., Garc, K., Parra, C., ve Casallas, R., 2016, Towards the Understanding and Evolution of Monolithic Applications as Microservices. XLII Latin American Computing Conference (CLEI).

European Banking Federation, 2016, Guidance for Implementation Of the Revised Payment Services Directive PSD2 Guidance. https://www.betalvereniging.nl/wpcontent/uploads/EBF_PSD2_guidance_september2016.pdf. [Ziyaret Tarihi: 10 Temmuz 2021].

Fan, W., Han, Z., Zhang, Y., ve Wang, R., 2018, Method of Maintaining Data Consistency in Microservice Architecture. In Proceedings- 4th IEEE International Conference on Big Data Security on Cloud, Bigdata Security 2018, 4th IEEE International Conference on High Performance and Smart Computing, HPSC 2018 and 3rd IEEE International Conference on Intelligent Data and Security.

Fernandes, J. L., Lopes, I. M. C., Rodrigues, J., ve Ullah, S., 2013, Performance evaluation of RESTful web services and AMQP protocol. 2013 Fifth International Conference on Ubiquitous and Future Networks (ICUFN), 810–815.

Fielding, R. T., 2000, Architectural Styles and the Design of Network-based Software Architectures, Doktora Tezi, *Kaliforniya Üniversitesi*, Irvine, Kaliforniya, A.B.D

Fowler, 2006, Continuous Integration [online], <https://martinfowler.com/articles/continuousIntegration.html> [Ziyaret Tarihi: 5 Haziran 2021].

Fowler, M., ve Lewis, J., 2014, Microservices- A definition of this new architectural term [online]. <https://martinfowler.com/articles/microservices.html> [Ziyaret Tarihi: 5 Haziran 2021].

Fritzsche, J., Bogner, J., Zimmermann, A., ve Wagner, S., 2019, From Monolith to Microservices: A Classification of Refactoring Approaches. Lecture Notes in Computer Science, 128–141.

Furda, A., Fidge, C., Zimmermann, O., Kelly, W., ve Barros, A., 2018, Migrating Enterprise Legacy Source Code to Microservices: On Multitenancy, Statefulness, and Data Consistency, IEEE Software, 35(3), 63–72.

- Gai, K., Qiu, M., ve Sun, X., 2018, A survey on FinTech, *Journal of Network and Computer Applications*, 103, 262–273.
- Gan, Y., ve Delimitrou, C., 2018, The architectural implications of cloud mikroservices. *IEEE Computer Architecture Letters*.
- Gartner, Inc. (2011); Magic Quadrant for International Retail Core Banking, Don Free
- Giesen, H., 2017, Thoughts and Observations about Banking Platforms [online], <http://www.next-finance-blog.de/banking-platforms/> [Ziyaret Tarihi: 5 Haziran 2021].
- Gouigoux, J.-P., ve Tamzalit, D., 2017, From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture, 2017 IEEE International Conference on Software Architecture Workshops (ICSAW).
- Gozman, D., Hedman, J., ve Olsen, K. S., 2018, Open Banking: Emergent Roles, Risks ve Opportunities, *Research Papers*, 183.
- Gün, U., 2019, Bankacılık Hukukunda Yeni Kavram: Açık Bankacılık, *Finans Hukuku Gündemi*, 2, 35-59.
- Guo, D., Wang, W., Zeng, G., ve Wei, Z., 2016, Mikroservices architecture based cloudware deployment platform for service computing. In *Proceedings- 2016 IEEE Symposium on Service-Oriented System Engineering, SOSE 2016*.
- Gyara, F., 2019, Account Access Consents [online], <https://openbanking.atlassian.net/wiki/spaces/DZ/pages/1077805332/Account+Access+Consents+v3.1.2> [Ziyaret Tarihi: 24 Ocak 2024].
- Harley, J., 2011, The Advantages and Disadvantages of Using ORM [online], <https://stackoverflow.com/questions/4667906/the-advantages-and-disadvantages-of-using-orm> [Ziyaret Tarihi: 8 Aralık 2021].
- Haralayya B. , 2021, Core Banking Technology and Its Top 6 Implementation Challenges. *J Adv Res Oper Markt Mgmt*; 4(1): 25-27
- Heinrich, R., van Hoorn, A., Knoche, H., Li, F., Lwakatare, L. E., Pahl, C. Wettinger, J., 2017, Performance Engineering for Mikroservices, 223–226.
- Hirsch, F., Kemp, J., ve Ilkka, J., 2007, *Mobile Web Services: Architecture and Implementation*. John Wiley & Sons
- Howarth, J., 2023, 57+ Incredible Fintech Stats (2023-2025) [online] <https://explodingtopics.com/blog/fintech-stats> [Ziyaret Tarihi: 28 Ocak 2024].
- Hsieh, K., 2019, Open Banking: A Case Study for Open API Economy. <https://nccur.lib.nccu.edu.tw/bitstream/140.119/124377/1> [Ziyaret Tarihi: 5 Eylül 2021].

- Jarwar, M. A., Kibria, M. G., Ali, S., ve Chong, I., 2018, Mikroservices in Web objects enabled IoT environment for enhancing reusability. *Sensors* (Switzerland).
- Jeremy H., 2020, Microservices Examples: Amazon, Netflix, Uber, and Etsy [online], <https://blog.dreamfactory.com/microservices-examples/> [Ziyaret Tarihi: 5 Haziran 2021].
- Johansson, L. ve Dossot, D., 2020, *RabbitMQ Essentials*, Packt Publishing.
- Kabakova, O., Plaksenkov, E., ve Korovkin, V., 2016, Strategizing for Financial Technology Platforms: Findings from Four Russian Case Studies, *Psychology & Marketing*, 33(12), 1106–1111.
- Kalske, M., Mäkitalo, N., ve Mikkonen, T., 2018, Challenges When Moving from Monolith to Microservice Architecture, *Lecture Notes in Computer Science*, 32–47.
- Kane, B., 2020, Coursera's journey to GraphQL [online], <https://www.apollographql.com/blog/community/courseras-journey-to-graphql-a5ad3b77f39a/> [Ziyaret Tarihi: 22 Kasım 2021].
- Kanber S., C., 2021, SAGA Pattern Nedir? [online], <https://sefikcankanber.medium.com/saga-pattern-nedir-e4a447bef361> [Ziyaret Tarihi: 28 Kasım 2021].
- Kappagantula, S., 2018, Microservice Architecture- Learn, Build, and Deploy Applications [online], <https://dzone.com/articles/microservice-architecture-learn-build-and-deploy-a> [Ziyaret Tarihi: 22 Mayıs 2021].
- Kartal, M. T. Ve Zengin, S., 2019, Yeni Nesil Bankacılık: API Uygulamaları ve Bankacılık Sistemine Muhtemel Etkileri Üzerine Bir İnceleme, Conference: 2. Uluslararası Eğitim ve Sosyal Bilimlerde Yeni Ufuklar (ICES- 2019), İstanbul.
- Kazanavicius, J., ve Mazeika, D., 2019, Migrating Legacy Software to Mikroservices Architecture. 2019 Open Conference of Electrical, Electronic and Information Sciences (EStream), 1–5.
- Kindson, M. ve Martinek, P. (2020). Enhancing Performance of Distributed Transactions in Microservices via Buffered Serialization. *Journal of Web Engineering*. 10.13052/jwe1540-9589.19564.
- Klynveld Peat Marwick Goerdeler (KPMG), 2017, Yeni Ödeme Hizmetleri Yönergesi. <https://assets.kpmg/content/dam/kpmg/tr/pdf/2018/05/yeni-odeme-hizmetleripsd2.pdf> [Ziyaret Tarihi: 10 Temmuz 2021].
- Knoche, H., ve Hasselbring, W., 2019, Drivers and Barriers for Mikroservice Adoption- A Survey among Professionals in Germany. *Enterprise Modelling and Information Systems Architectures (EMISAJ) - International Journal of Conceptual Modeling*, 14(1), 1–35.

- Kolobova, Y. I., Mokhnitskaya, D. S., Sidorova, V. E. ve Skorokhod, A. A. (2018). Risks and Threats of Using Open Banking in Russia and in the World. *KnE Social Sciences*, 3(2),415-426.
- Koutanov, E., 2020, *Effective Kafka: A Hands-On Guide to Building Robust and Scalable Event-Driven Applications with Code Examples in Java*, Lean Publishing.
- Kristianto, H., ve Zahra, A. (2021). Performance Analysis of Choreography and Orchestration in Microservices Architecture. *Journal of Theoretical and Applied Information Technology*. 2021; 99:18.
- Krylovskiy, A., Jahn, M., ve Patti, E. (2015). Designing a Smart City Internet of Things Platform with Microservice Architecture. In *Proceedings- 2015 International Conference on Future Internet of Things and Cloud, FiCloud 2015 and 2015 International Conference on Open and Big Data, OBD 2015*.
- Laigner, R., Zhou, Y., ve Salles, M. A. V. (2021). A distributed database system for event-based microservices. *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems*. doi:10.1145/3465480.3466919
- Lane, K., 2019, Introduction to APIs: History of APIs [online], <https://blog.postman.com/intro-to-apis-history-of-apis/> [Ziyaret Tarihi: 22 Mayıs 2021].
- Laplante, P., ve Kshetri, N. (2021). Open Banking Definition and Description. *IEEE Computer Magazine*. DOI: 10.1109/MC.2021.3055909
- Lee, I., Shin, Y. J., 2018, Fintech: Ecosystem, business models, investment decisions, and challenges. *Business Horizons*, 61(1), 35-46.
- Lewis, James; Fowler, M., 2014, *Microservice- A definition of this new architectural term*.
- Li, S., Zhang, H., Jia, Z., Li, Z., Zhang, C., Li, J., ... Shan, Z., 2019, A data flow driven approach to identifying microservices from monolithic applications. *Journal of Systems and Software*, 157.
- Lindman, J., Horkoff, J., Hammouda, I., ve Knauss, E., 2020, Emerging Perspectives of Application Programming Interface Strategy: A Framework to Respond to Business Concerns. *IEEE Software*, 37(2), 52–59.
- Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L., ve Pallickara, S., 2018, Serverless computing: An investigation of factors influencing microservice performance. In *Proceedings- 2018 IEEE International Conference on Cloud Engineering, IC2E 2018*.
- Loesch, S., 2018, *A Guide to Financial Regulation for Fintech Entrepreneurs*, 10.1002/9781119436775.

- Lu, D., Huang, D., Walenstein, A., ve Medhi, D., 2017, A Secure Microservice Framework for IoT. Proceedings- 11th IEEE International Symposium on Service-Oriented System Engineering, SOSE 2017, 9–18.
- Malyuga, K., Perl, O., Slapoguzov, A., ve Perl, I. (2020). Fault Tolerant Central Saga Orchestrator in RESTful Architecture. 2020 26th Conference of Open Innovations Association (FRUCT). doi:10.23919/fruct48808.2020.9087
- Marginingsih, R., 2019, Analysis of SWOT, Technology Financial (FinTech), Terhadap Industri Perbankan. Jurnal Humaniora dan Sosial, 19 (1), 55-60.
- Martinez, R., 2019, BBVA uses GraphQL and Hasura to build an open source security product [online], <https://hasura.io/blog/bbva-used-graphql-api-hasura/> [Ziyaret Tarihi: 22 Kasım 2021].
- Megargel A., Shankararaman V., Walker D.K., 2020, Migrating from Monoliths to Cloud-Based Microservices: A Banking Industry Example. In: Ramachandran M., Mahmood Z. (eds) Software Engineering in the Era of Cloud Computing, Computer Communications and Networks, Springer.
- Menikgama, D., 2020, Why Banks Should Consider Becoming Third Party Providers [online] <https://wso2.com/library/articles/why-banks-should-consider-becoming-third-party-providers/> [Ziyaret Tarihi: 24 Ocak 2024].
- Michael, J., 2017 (1), Account and Transaction API Specification - v1.1.0 [online] <https://openbanking.atlassian.net/wiki/spaces/DZ/pages/5785171/Account+and+Transaction+API+Specification+-+v1.1.0> [Ziyaret Tarihi: 28 Ocak 2024].
- Michael, J., 2017 (2), Payment Initiation API Specification - v1.1.0 [online] <https://openbanking.atlassian.net/wiki/spaces/DZ/pages/5786479/Payment+Initiation+API+Specification+-+v1.1.0> [Ziyaret Tarihi: 28 Ocak 2024].
- Montesi, F., ve Weber, J., 2016, Circuit Breakers, Discovery, and API Gateways in Microservices. <https://arxiv.org/abs/1609.05830>
- Newman, S., 2021, Building Microservices, O'Reilly Media.
- Niu, Y., Liu, F., ve Li, Z., 2018, Load Balancing Across Microservices, In Proceedings- IEEE INFOCOM.
- Noctor, M., 2018, PSD2: Is The Banking Industry Prepared? Computer Fraud & Security, 2018(6), 9-11.
- Noor, M. B. K., 2008, Case Study: A Strategic Research Methodology, American Journal of Applied Sciences 5(11): 1602-1604.
- Omarini, A. E., 2018, Banks and Fintechs: How to Develop a Digital Open Banking Approach for the Bank's Future. International Business Research, 11(9), 23-36.

- Parker, B. D., 2019, PSD2 Open Banking: The Challenges of Third-Party Provider Identity and Regulatory Checking. <https://sci-hub.se/https://onlinelibrary.wiley.com/doi/10.1002/9781119551973.ch19>
- Perng, B. B. J., 2019, Align Open Banking and Future-Proof RegTech for Regulators and Third-Party Providers to Deliver the Optimal Consumer Convenience and Protection.
- Piccioni, M., Furia, C. A., ve Meyer, B., 2013, An Empirical Study of API Usability. 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, 5–14.
- Petrovic, M., 2020, PSD2 Influence on Digital Banking Transformation – Banks’s Perspective. 2020 Journal of Process Management, 8(14).
- Pollari, I., 2016, The rise of Fintech opportunities and challenges. JASSA Financial Services Institute of Australasia, 3, 15-21.
- Premchand, A., ve Choudhry, A., 2018, Open Banking APIs for Transformation in Banking, International Conference on Communication, Computing and Internet of Things (IC3IoT), 25–29.
- Puschmann, T., 2017, Digitization of the Financial Services Industry. Business & Information Systems Engineering, 59(1), 69-76.
- Queue, A., 2006, A Conversation with Werner Vogels [online], <https://queue.acm.org/detail.cfm?id=1142065> [Ziyaret Tarihi: 5 Haziran 2021].
- Rajagopalan, S. ve Jamjoom, H., 2015, Autonomous Healing for Microservice-based Apps, USENIX Conference on Hot Topics in Cloud Computing.
- Rastogi, S., Sharma, A. ve Panse, C., 2020, Open Banking and Inclusive Growth in India, Indian Journal of Ecology (2020) 47 Special Issue (9): 75-79.
- Rauf, I., Troubitsyna, E., ve Porres, I., 2019, A systematic mapping study of API usability evaluation methods. Computer Science Review, 33, 49–68.
- Reddy, M., 2011, API Design for C++, Morgan Kauffman Publishing.
- Reinhold, E., 2016, Uber Darwin [online], <https://www.infoq.com/presentations/uber-darwin/> [Ziyaret Tarihi: 22 Mayıs 2021].
- Richardson, C., 2016, Choosing a Microservices Deployment Strategy. <https://www.nginx.com/blog/deploying-microservices/> [Ziyaret Tarihi: Haziran 2023].
- Richardson, C., 2019, Microservices patterns: with examples in Java, Manning Publications.
- Roy, G., M., 2017, RabbitMQ in Depth, Manning Publishing.

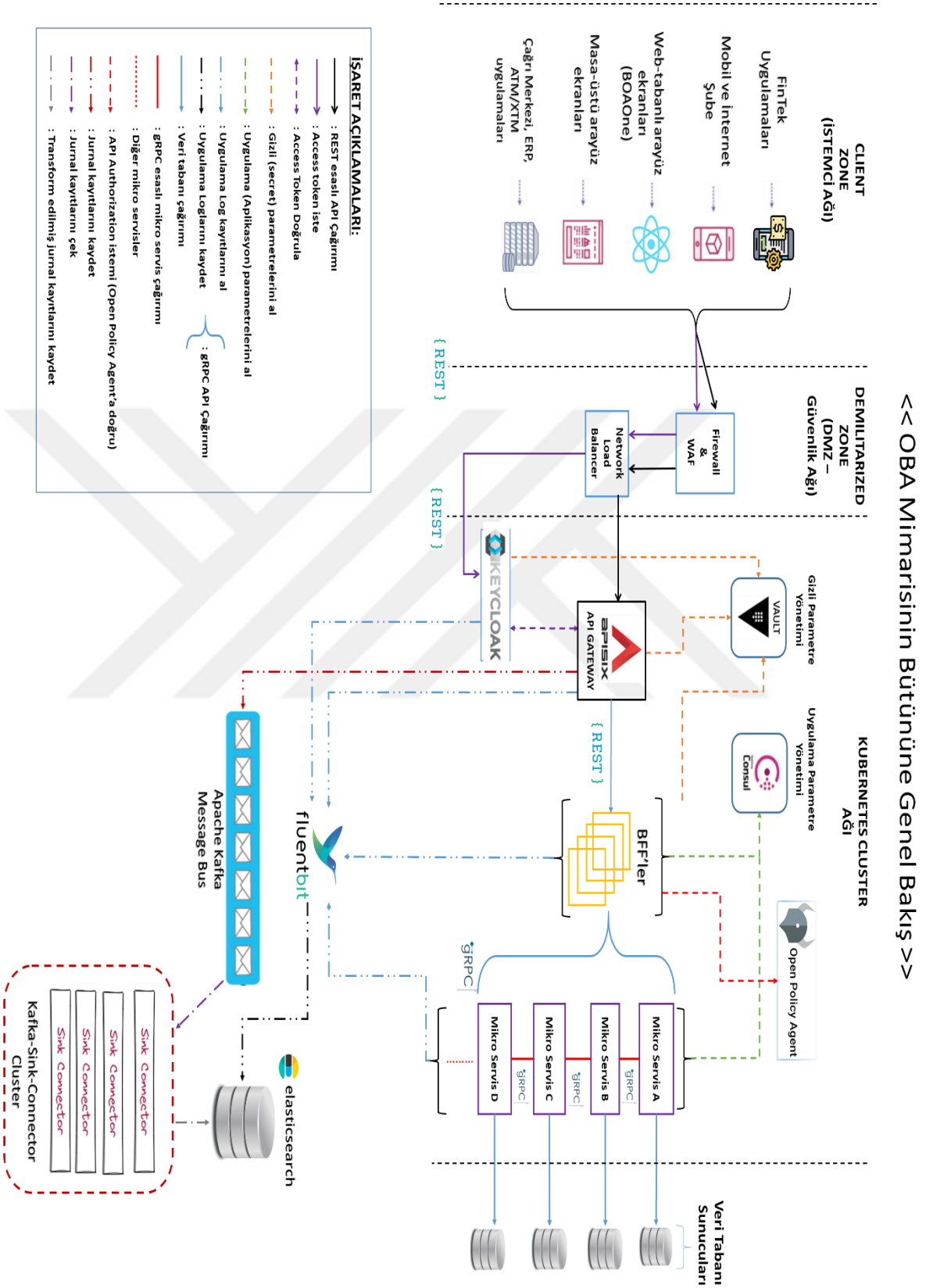
- Rudrabhatla, C.K., (2018). Comparison of Event Choreography and Orchestration Techniques in Microservice Architecture. *International Journal of Advanced Computer Science and Applications*. 9. 10.14569/IJACSA.2018.090804.
- Schueffel, P., 2016, Taming the Beast: A Scientific Definition of Fintech. *Journal of Innovation Management*, Vol:4, Iss:4, 32-54.
- Sezer, Y., 2019, The Advantages and Disadvantages of Using ORM [online], <https://stackoverflow.com/questions/4667906/the-advantages-and-disadvantages-of-using-orm> [Ziyaret Tarihi: 8 Aralık 2021].
- Shafabakhsh, B., Lagerström, R., ve Hacks, S., 2020, Evaluating the impact of inter-process communication in microservice architectures, *CEUR Workshop Proceedings*, 2767(December), 55–63.
- Shewale, R., 2024, 43 Fintech Statistics For 2024 (Startups, Financials & Trends) [online], <https://www.demandsage.com/fintech-statistics/> [Ziyaret Tarihi: 28 Ocak 2024].
- Singh, V., ve Peddoju, S. K., 2017, Container-based micro service architecture for cloud applications, *IEEE International Conference on Computing, Communication and Automation, ICCCA 2017*.
- Sivathanu, B., 2019, An Empirical Study on the Intention to Use Open Banking in India, *Information Resources Management Journal (IRMJ)*, 32(3), 27-47.
- Stylos, J., Jeffrey, Clarke, S., Steven, Myers, B., ve Brad, B, 2006, Comparing API Design Choices with Usability Studies: A Case Study and Future Directions.
- Taibi, D., Lenarduzzi, V., ve Pahl, C., 2018, Architectural Patterns for Microservices: A Systematic Mapping Study, *CLOSER*, pp. 221-232.
- Taştan F.G. ve Saruhan U., 2020, Açık Bankacılık: Kişisel Verilerin Korunmasına Bir Tehdit mi? *Academia.edu*, Yayın Tarihi: 4 Kasım 2020.
- Taştan, S., 2019, Küresel Finansal Teknoloji Sektöründe Ortaya Çıkan Yeni Girişimlerin Ekonomik ve Teknolojik Belirleyicileri, *Haliç Üniversitesi Sosyal Bilimleri Dergisi* 2/1:41-69.
- Tavares, L., H. ve Silva, M., D., 2015, *Redis Essentials: Harness the power of Redis to integrate and manage your projects efficiently*, Packt Publishing.
- Taylor, S., 2017, React, Relay and GraphQL: Under the Hood of The Times Website Redesign [online], <https://open.nytimes.com/react-relay-and-graphql-under-the-hood-of-the-times-website-redesign-22fb62ea9764> [Ziyaret Tarihi: 22 Kasım 2021].
- T.C. Cumhurbaşkanlığı Finans Ofisi., 2023, Türkiye Fintek Rehberi [online], <https://www.cbfo.gov.tr/sites/default/files/docs/2023-03/turkiye-fintek-rehberi.pdf> [Ziyaret Tarihi: 28 Ocak 2024].

- Torre, C., Wagner, B., Rousos, M., 2021, .NET Microservices: Architecture for containerized .NET applications, Microsoft.
- Ueda, T., Nakaike, T., ve Ohara, M., 2016, Workload characterization for microservices. 2016 IEEE International Symposium on Workload Characterization (IISWC), 1–10.
- Villamizar, M., Garces, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., ve Gil, S., 2016, Evaluating the monolithic and the micro services architecture patterns to deploy Web applications in the cloud, 10th Colombian Computing Conference, 10CCC.
- Wang, H., Ma, S., Dai, H.N., Imram, M., Wang, T., 2020, Blockchain-Based Data Privacy Management with Nudge Theory in Open Banking, Future Generation Computer Systems, 110 (2020) 812-823.
- Xu, R., Jin, W., ve Kim, D., 2019, Microservice Security Agent Based On API Gateway in Edge Computing. Sensors, 19(22).
- Yağcı, M., 2018, Yükselen Finansal Teknolojilerin Ekonomi Politikası: Fintek ve Bitcoin Örnekleri. İktisat ve Toplum, 17-24.
- Yarygina, T., ve Bagge, A. H., 2018, Overcoming Security Challenges in Microservices Architectures, 12th IEEE International Symposium on Service Oriented System Engineering, SOSE 2018 and 9th International Workshop on Joint Cloud Computing, JCC 2018.
- Yin, R., 1993, Application of Case Study Research, Sage Publication California, pp: 33-55.
- Yu, D., Jin, Y., Zhang, Y., ve Zheng, X., 2017, A survey on security issues in services communication of mikco services-enabled fog applications, Concurrency Computation.
- Zachariadis, M. ve Özcan, P., 2017, The API Economy and Digital Transformation in Financial Services: The Case of Open Banking (June 15, 2017), SWIFT Institute Working Paper No. 2016-001.
- Zhang, G., Ren, K., Ahn, J.-S., ve Ben-Romdhane, S. (2019). GRIT: Consistent Distributed Transactions Across Polyglot Microservices with Multiple Databases. 2019 IEEE 35th International Conference on Data Engineering (ICDE). doi:10.1109/icde.2019.00230
- Zhao, J. T., Jing, S. Y., ve Jiang, L. Z., 2018, Management of API Gateway Based on Micro-service Architecture, Journal of Physics: Conference Series, 1087, 32032.
- Zheng, L., ve Wei, B., 2018, Application of micro services architecture in cloud environment project development. MATEC Web of Conferences. <https://doi.org/10.1051/mateconf/201818903023>

Zuckmantel, T., Zhou, Y., Döder, B., ve Hildebrandt, T., 2022. Event-based data-centric semantics for consistent data management in microservices. In Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems (DEBS '22). Association for Computing Machinery, New York, NY, USA, 97–102. <https://doi.org/10.1145/3524860.3539807>



EK-1: OBA Mimarisine Genel Bakış



EK-2: Docker-compose.yml

```
version: "3.8"

services:
#####
# MSSQL: The database used by KeyCloak
#####
keycloak-database:
  container_name: keycloak-database
  image: mcr.io/mssql/server:2017-latest

#####
# KeyCloak database Creation
#####
keycloak-migration:
  container_name: keycloak-migration
  image: mcr.io/mssql-tools

#####
# KeyCloak: The Identity Provider
#####
keycloak:
  container_name: keycloak
  #image: quay.io/keycloak/keycloak:latest
  #image: oba/kt-keycloak:15.0.2.1
  image: oba/kt-keycloak:15.0.2.3

#####
# Postgres: The database used by Kong
#####
kong-database:
  container_name: kong-database
  image: docker.io/library/postgres:11-alpine

#####
# Kong database migration
#####
kong-migration:
  container_name: kong-migration
  image: docker.io/library/kong:2.5.0-alpine

#####
# Kong: The API Gateway
#####
kong:
  container_name: kong
  image: dev/ktkong:2.5.0-alpine

#####
# Konga database prepare
#####
konga-prepare:
  container_name: konga-prepare
  image: docker.io/pantssel/konga:0.14.9

#####
# Konga: Kong GUI
#####
konga:
  container_name: konga
  image: docker.io/pantssel/konga:0.14.9

#####
# KeyCloak Identity-API
#####
identity-api:
```

```

    container_name: identity-api
    image: oba/keycloak-identity-api:latest
    #image: local-keycloak-identity-api:latest
    build:
      context: .
      dockerfile: src/Identity/Services/IdentityIntegrationApi/Dockerfile

#####
# OPA
#####
opa:
  image: docker.io/openpolicyagent/opa:0.34.0-rootless
  container_name: opa

#####
# OPA Bundle Server
#####
opa-bundle-server:
  image: docker.io/library/nginx:1.20.0-alpine
  container_name: opa-bundle-server

#####
# Fluentd Service
#####
fluentd:
  container_name: fluentd
  image: oba/kt-fluentd:1.12.1

#####
# Elasticsearch Service
#####
elasticsearch:
  container_name: elasticsearch
  image: docker.io/library/elasticsearch:7.13.1

#####
# Kibana Service
#####
kibana:
  container_name: kibana
  image: docker.io/library/kibana:7.13.1

#####
# Httpd Service
#####
#httpdservice:
#  container_name: httpdservice
#  image: docker.io/library/httpd

```

Docker-compose.override.yml

```

version: "3.8"

services:

#####
# MSSQL: The database used by KeyCloak
#####
keycloak-database:
  restart: unless-stopped
  networks:
    - oba-net
  environment:

```

```

- ACCEPT_EULA=Y
- SA_PASSWORD=Password!23
- MSSQL_PID=Developer
ports:
- "5433:1433"
volumes:
- keycloak-datastore:/var/opt/mssql

#####
# KeyCloak database Creation
#####
keycloak-migration:
  networks:
    - oba-net
  depends_on:
    - keycloak-database
  command: /bin/bash -c 'until /opt/mssql-tools/bin/sqlcmd -S keycloak-
database -U sa -P "Password!23" -Q "create database Keycloak"; do sleep 5;
done'

#####
# KeyCloak: The Identity Provider
#####
keycloak:
  networks:
    - oba-net
  depends_on:
    - keycloak-database
    - keycloak-migration
  ports:
    - "8080:8080"
  environment:
    - KEYCLOAK_USER=admin
    - KEYCLOAK_PASSWORD=admin
    - DB_VENDOR=mssql
    - DB_USER=sa
    - DB_PASSWORD=Password!23
    - DB_ADDR=keycloak-database
    - DB_DATABASE=Keycloak
    - KEYCLOAK_LOGLEVEL=DEBUG
    #- KEYCLOAK_IMPORT=-
Dkeycloak.migration.file=/opt/jboss/keycloak/imports/keycloak_export_whole.js
n

#####
# Postgres: The database used by Kong
#####
kong-database:
  restart: unless-stopped
  networks:
    - oba-net
  volumes:
    - kong-datastore:/var/lib/postgresql/data
  environment:
    POSTGRES_DB: api-gw
    POSTGRES_USER: kong
    POSTGRES_PASSWORD: password
  ports:
    - "5432:5432"
  healthcheck:
    test: ["CMD", "pg_isready", "-U", "kong", "-d", "api-gw"]
    interval: 30s
    timeout: 3s
    retries: 5

#####
# Kong database migration
#####

```

```

kong-migration:
  command: "kong migrations bootstrap"
  networks:
    - oba-net
  restart: on-failure
  environment:
    KONG_DATABASE: postgres
    KONG_PG_HOST: kong-database
    KONG_PG_DATABASE: api-gw
    KONG_PG_USER: kong
    KONG_PG_PASSWORD: password
  links:
    - kong-database
  depends_on:
    - kong-database

#####
# Kong: The API Gateway
#####
kong:
  restart: unless-stopped
  networks:
    oba-net:
      ipv4_address: 172.1.1.40
  environment:
    KONG_DATABASE: postgres
    KONG_PG_HOST: kong-database
    KONG_PG_DATABASE: api-gw
    KONG_PG_USER: kong
    KONG_PG_PASSWORD: password
    KONG_ADMIN_LISTEN: "0.0.0.0:8001, 0.0.0.0:8444 ssl"
    KONG_PROXY_LISTEN: "0.0.0.0:8000, 0.0.0.0:8443 ssl"
    KONG_DB_UPDATE_FREQUENCY: 1m
    KONG_PROXY_ACCESS_LOG: /dev/stdout
    KONG_ADMIN_ACCESS_LOG: /dev/stdout
    KONG_PROXY_ERROR_LOG: /dev/stderr
    KONG_ADMIN_ERROR_LOG: /dev/stderr
    KONG_PLUGINS: bundled,oidc
    KONG_LOG_LEVEL: debug
  depends_on:
    - kong-migration
  ports:
    - "8001:8001"
    - "8444:8444"
    - "8000:8000"
    - "8443:8443"
  extra_hosts:
    - "host.internal.ipaddr: 172.20.165.61"

#####
# Konga database prepare
#####
konga-prepare:
  command: "-c prepare -a postgres -u postgresql://kong:password@kong-
database:5432/konga_db"
  networks:
    - oba-net
  restart: on-failure
  links:
    - kong-database
  depends_on:
    - kong-database

#####
# Konga: Kong GUI
#####
konga:

```

```

restart: unless-stopped
networks:
  - oba-net
environment:
  DB_ADAPTER: postgres
  DB_HOST: kong-database
  DB_USER: kong
  DB_PASSWORD: password
  DB_DATABASE: konga_db
  # any string will OK for token
  TOKEN_SECRET: iu7YDcPLiZkozQXzZ9kka3Ee1Vid5ZgQ
  NODE_ENV: development
depends_on:
  - konga-prepare
ports:
  - "1337:1337"

#####
# KeyCloak Identity-API
#####
identity-api:
  restart: unless-stopped
  networks:
    - oba-net
  depends_on:
    - keycloak
  environment:
    - ASPNETCORE_ENVIRONMENT=Development
    - ASPNETCORE_URLS=http://0.0.0.0:5105
    #- Logging__LogLevel__Default=Trace
  links:
    - fluentd
  logging:
    driver: "fluentd"
    options:
      fluentd-address: host.docker.internal:24224
      tag: oba.identityapi
  ports:
    - "5105:5105"

#####
# OPA
#####
opa:
  networks:
    - oba-net
  ports:
    - 8181:8181
  # WARNING: OPA is NOT running with an authorization policy configured.
  # This
  # means that clients can read and write policies in OPA. If you are
  # deploying OPA in an insecure environment, be sure to configure
  # authentication and authorization on the daemon. See the Security page
  # for
  # details: https://www.openpolicyagent.org/docs/security.html.
  command:
    - "run"
    - "--server"
    - "--log-format=json-pretty"
    - "--set=decision_logs.console=true"
    - "--set=services.nginx.url=http://opa-bundle-server"
    - "--set=bundles.nginx.service=nginx"
    - "--set=bundles.nginx.resource=bundles/bundle.tar.gz"
  depends_on:
    - opa-bundle-server

#####

```

```

# OPA Bundle Server
#####
opa-bundle-server:
  networks:
    - oba-net
  ports:
    - 8888:80
  volumes:
    - ./opa/bundles:/usr/share/nginx/html/bundles

#####
# Fluentd Service
#####
fluentd:
  networks:
    - oba-net
  volumes:
    - ./src/Logging/fluentd/conf:/fluentd/etc
  links:
    - "elasticsearch"
  ports:
    - "24224:24224"
    - "24224:24224/udp"

#####
# Elasticsearch Service
#####
elasticsearch:
  networks:
    - oba-net
  environment:
    - "discovery.type=single-node"
    - bootstrap.memory_lock=true # along with the memlock settings below,
disables swapping
    - "ES_JAVA_OPTS=-Xms512m -Xmx512m" # minimum and maximum Java heap size,
recommend setting both to 50% of system RAM
  ulimits:
    memlock:
      soft: -1
      hard: -1
    nofile:
      soft: 262144 # maximum number of open files for the Elasticsearch
user, set to at least 65536 on modern systems
      hard: 262144
  expose:
    - "9200"
  ports:
    - "9200:9200"

#####
# Kibana Service
#####
kibana:
  networks:
    - oba-net
  links:
    - "elasticsearch"
  ports:
    - "5601:5601"

#####
# Httpd Service
#####
#httpdservice:
#  ports:
#    - "5209:80"
#  depends_on:
#    - fluentd

```

```
# links:
#   - fluentd
# logging:
#   driver: "fluentd"
#   options:
#     fluentd-address: localhost:24224
#     tag: httpd.access

networks:
  oba-net:
    name: oba-net
    driver: bridge
    ipam:
      config:
        - subnet: 172.1.1.0/24

volumes:
  kong-datastore:
    driver: local
  keycloak-datastore:
    driver: local
```



EK-3: Nakit Çekme SAGA Akışı Tanımı:

EK 3 – Nakit Çekme SAGA Akışı Tanımı:

Şekil 6.7'de görülen state-machine yapısını işletebilecek şekilde tanımlanmış bir JSON dokümanıdır.

JSON dokümanın oluşturulmasında şu terminolojik kavramlar kullanılmıştır:

- i. Id: Akışın seri numarası
- ii. Name: Akışın adı (CashWithdrawalSagaFlow)
- iii. Phase: Bir mikro servis çağırımı (1 veya daha fazla olabilir ancak her mikro servis çağırımı için bir tanedir.)
 - o Progress: Phase kapsamındaki mikro servisin pozitif çağırım senaryosunda üzerinde hangi metodun hangi request objesiyle ve hangi parametrelerle çağırılacağını tanımlar.
 - o Rollback: Phase kapsamındaki mikro servis çağırımının hata vermesi durumunda hangi telafi (rollback) metodunun hangi request objesiyle ve hangi parametreleriyle çağırılması gerektiğini tanımlar.

JSON Dokümanı İçeriği

```
{
  "_id": "646e606b366093685bb71145",
  "Name": "CashWithdrawalSagaFlow",
  "Phases": [
    {
      "_id": "0",
      "Seq": 1,
      "Name": "AccountServiceForBalanceCheck",
      "Progress": {
        "DaprAppId": "sample-saga-account-service",
        "ServiceMethodName": "CheckAccountBalance",
        "ProtoRequestType": "OpenBanking.Sample.Saga.Account.Service.Api.Generated.AccountBalanceCheckRequestGrpcModel",
        "ProtoResponseType": "OpenBanking.Sample.Saga.Account.Service.Api.Generated.AccountBalanceCheckResponseGrpcModel",
        "RequestPropertyMap": {
          "EndUserId": "CustomerId",
          "Amount": "AmountToCheck"
        }
      },
      "Rollback": null
    },
    {
      "_id": "0",
      "Seq": 2,
      "Name": "LimitService",
      "Progress": {
        "DaprAppId": "sample-saga-limit-service",
        "ServiceMethodName": "CheckLimit",
        "ProtoRequestType": "OpenBanking.Sample.Saga.Limit.Service.Api.Generated.LimitCheckRequestGrpcModel",
        "ProtoResponseType": "OpenBanking.Sample.Saga.Limit.Service.Api.Generated.LimitCheckResponseGrpcModel",
      }
    }
  ]
}
```

```

        "RequestPropertyMap":{
            "CustomerId":"CustomerId",
            "CheckedAmount":"Amount"
        },
        "Rollback":null
    },
    {
        "_id":"0",
        "Seq":3,
        "Name":"SendNotificationService",
        "Progress":{
            "DaprAppId":"sample-saga-notification-service",
            "ServiceMethodName":"SendApprovalNotification",

"ProtoRequestType":"OpenBanking.Sample.Saga.Notification.Service.Api.Generated.NotificationSendRequestGrpcModel",

"ProtoResponseType":"OpenBanking.Sample.Saga.Notification.Service.Api.Generated.NotificationSendResponseGrpcModel",
            "RequestPropertyMap":{
                "CustomerId":"CustomerId",
                "Amount":"AmountToNotify"
            },
            "Rollback":null
        },
        {
            "_id":"0",
            "Seq":4,
            "Name":"CheckNotificationService",
            "Progress":{
                "DaprAppId":"sample-saga-notification-service",
                "ServiceMethodName":"CheckApprovalNotification",

"ProtoRequestType":"OpenBanking.Sample.Saga.Notification.Service.Api.Generated.NotificationCheckRequestGrpcModel",

"ProtoResponseType":"OpenBanking.Sample.Saga.Notification.Service.Api.Generated.NotificationCheckResponseGrpcModel",
            "RequestPropertyMap":{
                "CustomerId":"CustomerId",
                "NotifiedAmount":"AmountToCheck"
            },
            "Rollback":null
        },
    },

```

EK-4: Dağıtık SAGA Akışlarının Yönetiminin Sahada Uygulanışı

