AZƏRBAYCAN RESPUBLİKASI ELM VƏ TƏHSİL NAZİRLİYİ
AZƏRBAYCAN TEXNİKİ UNİVERSİTETİ
YÜKSƏK TƏHSİL İNSTİTUTU

Aqil Ağamirzəyev Şakir [I Fəsil, II Fəsil]

Sənan Manafov Zakir [Giriş, Nəticə, III Fəsil]

MİKROSERVİS ARXİTEKTURALI İNFORMASİYA SİSTEMLƏRİNDƏ ƏLÇATANLILIĞIN TƏMİN OLUNMASININ KOMPLEKS TƏDQİQİ

Mövzusunda

MAGİSTRLİK DİSSERTASİYASI

“060509” Kompüter Elmləri

“60509 ” Sistem Proqramlaşdırması

Elmi rəhbər. f.r.e.n., dos. Şahverdiyev M.Ə

BAKI – 2024

AZƏRBAYCAN TEXNİKİ UNİVERSİTETİ

YÜKSƏK TƏHSİL İNSTİTUTU

MAGİSTRANTIN ANDI

Mikroservis arxitekturalı informasiya sistemlərində əlçatanlılığın təmin olunmasının kompleks tədqiqi mövzusunda təqdim etdiyimiz magistrlik dissertasiyasını elmi əxlaq normalarına və istinad qaydalarına tam riayət etməklə və istifadə etdiyim bütün mənbələri ədəbiyyat siyahısında əks etdirməklə yazdığıma and içiriəm(ik) və magistrlik dissertasiyasının AzTU Kitabxana İnformasiya Mərkəzində saxlanılması, həmin mərkəz tərəfindən AzTU Rəqəmsal Repozitoriyasına daxil edilərək repozitoriyanın veb saytında yerləşdirilməsinə icazə veririk.

Ağamirzəyev Aqil _____

Manafov Sənan _____

Tarix

Mündəricat

GİRİŞ

Mövzunun aktuallığı. Mikroservis arxitekturalı informasiya sistemlərində əlçatanlılığın təmin olunmasının kompleks tədqiqi mövzusu, müasir texnologiya və iş mühitlərinin tələblərini qarşılamaq üçün böyük aktuallıq kəsb edir. Mikroservis arxitekturası, ənənəvi monolitik sistemlərə nisbətən daha çevik, modulyar və skalabil bir yanaşma təklif edir. Bu arxitektura, böyük və kompleks tətbiqlərin daha kiçik, müstəqil və bir-birindən asılı olmayan xidmətlərə bölünməsi əsasında işləyir. Bu yanaşma, inkişaf etdirici komandaların daha sürətli və çevik işləməsinə, yeni funksionallıqların asanlıqla əlavə olunmasına və problemlərin daha sürətli həll olunmasına imkan verir. Əlçatanlıq, hər hansı bir informasiya sisteminin əsas tələblərindən biridir. İstifadəçilər, xidmətlərin fasiləsiz və yüksək performansla işləməsini gözləyirlər. Mikroservis arxitekturası, əlçatanlıq və davamlılıq baxımından ənənəvi arxitekturalara nisbətən daha üstün imkanlar təqdim edir. Bu arxitektura, mikroservislər arasında yük paylanması, səhv təcridi və avtomatik bərpa kimi xüsusiyyətlər vasitəsilə sistemin ümumi əlçatanlığını və etibarlılığını artırır. Məsələn, bir mikroservis səhv işləsə belə, bu, digər mikroservislərin fəaliyyətinə təsir etmir və sistem ümumi olaraq işləməyə davam edir.

Müasir rəqəmsal dünyada, yüksək əlçatanlıq və performans təmin edən sistemlər qurmaq, bizneslərin rəqabət qabiliyyətini artırır. Mikroservis arxitekturası, bu tələblərə cavab verərək, xüsusilə elektron ticarət, bankçılıq, səhiyyə və digər kritik sahələrdə geniş istifadə olunur. Bu sistemlər, istifadəçilərə fasiləsiz xidmət təqdim etməklə yanaşı, biznes proseslərinin effektivliyini artırır və müştəri məmnuniyyətini təmin edir.

Tədqiqatın məqsəd və vəzifələri. Tədqiqatın əsas məqsədi mikroservis arxitekturasının xüsusiyyətlərini və bu arxitekturaya xas olan əlçatanlıq problemlərini nəzərə alaraq, bu problemlərin həlli üçün effektiv və innovativ yanaşmaların inkişaf etdirilməsi və tətbiq olunmasıdır. Bu, həm texnoloji, həm də təşkilati baxımdan əlçatanlıq səviyyəsinin artırılmasını və sistemlərin etibarlılığının yüksəldilməsini təmin edəcəkdir.

Tədqiqat metodları. Proqram təminatının hazırlanması mərhələsi üzrə həm frontend, həmdə backend texnologiyalarından istifadə olunub. Əsas istifadə olunan

texnologiyalar Java, Spring Boot, PostgreSQL, MongoDB, Redis, Spring Security, SAGA, CQRS. Serverə yerləşdirmək üçün AWS cloud istifədə olunub. Tədqiqat metodları – Proqram təminatının hazırlanması mərhələsi üzrə həm frontend, həmdə backend texnologiyalarından istifadə olunub. Əsas istifadə olunan texnologiyalar Java, Spring Boot, PostgreSQL, MongoDB, Redis, Spring Security, SAGA, CQRS. Serverə yerləşdirmək üçün AWS cloud istifədə olunub.

Elmi yeniliyin elementləri. Tədqiqatın elmi yeniliklərinə bəzi müddəalar daxil etmək olar. Bunlara:

1. Yük Balanslaması: ArticleHub platformasının müxtəlif hissələrində (məsələn, axtarış, məqalələrin çapı) yük balanslaması üçün yeni metodlar təklif etmək. Məsələn, Kubernetes və ya istənilən container orchestrator istifadə edərək, mikroservislərin avtomatik şəkildə şkala artırılması və azaldılması.

2. Keş İdarəetməsi: Populyar məhsulların məlumatlarını Redis və ya Memcached kimi keş texnologiyaları istifadə edərək daha sürətli çatdırmaq.

3. Mikroservislərin Test Edilməsi: Hər bir mikroservisin müstəqil şəkildə test edilməsi üçün yeni bir test çərçivəsi təklif etmək. Məsələn, kontrakt testləri və chaos mühəndislik istifadə edərək xidmətlərin davamlılığını və dayanıqlığını təmin etmək.

4. Real-Time Monitoring: Prometheus və Grafana kimi alətləri istifadə edərək, real-time performans monitorinqi və anomaliyanın aşkarlanması.

Praktiki həll. Tədqiqatda bəzi praktik həllər daxildir. Bunlara:

1. Yeni Arxitektura Dizaynı və Nümunələri - Məqalə İdarəetmə Mikroservisi: Məqalələrin yaradılması, redaktəsi və dərc edilməsi üçün xüsusi mikroservis dizayn edin. Bu mikroservis məqalələrin versiya idarəsini, avtomatik saxlanmasını və real-time redaktəsini təmin edə bilər.Elasticsearch və Redis kimi texnologiyaları istifadə edərək sürətli və effektiv axtarış imkanları təqdim edir.

2. İnnovativ Texnologiya və Alətlər - AI ilə Məqalə Təklifi: Maşın öyrənməsi və NLP (Natural Language Processing) texnologiyaları istifadə edərək istifadəçilərə maraqlandıqları mövzularda məqalə təklifləri verən bir mikroservis hazırlayırıq.

Məqalələrin avtomatik olaraq kateqoriyalara ayrılması və etiketlənməsi üçün AI modelləri tətbiq edirik. Blockchain ilə Məzmun Etibarlılığı: Məqalələrin orijinallığını və müəllif hüquqlarını qorumaq üçün blockchain texnologiyasından istifadə edirik. Hər bir məqalənin orijinal olduğunu və dəyişdirilmədiyini təmin edən bir sistem inkişaf etdirik.

3. Performansın Təkmilləşdirilməsi - Yük Balanslaması və Keş İdarəetməsi: Platformanın müxtəlif hissələrində yük balanslamasını optimallaşdırmaq üçün Kubernetes istifadə edirik. Sürətli məzmun çatdırılması üçün Redis və ya Memcached kimi keş texnologiyaları istifadə edirik. Məsələn, populyar məqalələrin məlumatlarını keşləyərək istifadəçilərə daha sürətli çatdırır.

4. Test və Monitoring - Avtomatlaşdırılmış Test və CI/CD: Mikroservislərin müstəqil şəkildə test edilməsi üçün kontrakt testləri tətbiq edin. Hər bir mikroservisin funksionallığını təmin etmək üçün unit və integration testlər yazırıq. CI/CD proseslərini optimallaşdırmaq üçün Jenkins və ya GitLab CI/CD kimi alətlər istifadə edirik. Avtomatlaşdırılmış test və deployment proseslərini tətbiq edirik.

5. Real-Time Monitoring və Log Analitikası - Prometheus və Grafana kimi alətləri istifadə edərək, real-time performans monitorinqi və anomaliyanın aşkarlanmasını təmin edirik. ELK stack (Elasticsearch, Logstash, Kibana) istifadə edərək logların toplanması və analitikası üçün bir sistem qururuq.

Müdafiə üçün təqdim edilən nəticələr. Sürətli Yükləmə Zamanı mikroservis arxitekturasına keçid nəticəsində səhifələrin yükləmə zamanının 60% qədər azaldılması. Redis və Elasticsearch istifadə edərək axtarış və filtrasiya proseslərinin 30% qədər sürətlənməsi. Yüksək Ölçəklənəbilənlik Kubernetes ilə mikroservislərin avtomatik şəkildə şkala artırılması və azaldılması nəticəsində platformanın daha çox istifadəçi tələbatını qarşılayabilməsi. Pik dövrlərdə xidmətin dayanıqlılığının və performansın stabil saxlanılması.

Nəticələrin aprobasiyası. Aşağıda nəticələrin aprobasiyası verilmişdir. Bunlar:

- Anketlərin İcrası: Platformanın aktiv istifadəçilərinə göndərilən anketlər vasitəsilə istifadəçi təcrübəsi haqqında məlumat toplanmışdır. İstifadəçilərin əksəriyyəti məqalə

redaktəsi və axtarışı funksiyalarının sürətlənməsi və rahatlığının artması haqqında müsbət geri dönüş vermişdir.

- İstifadəçi Geri Dönüşlərinin Analizi: Toplanan məlumatlar analiz edilərək mikroservis arxitekturasının istifadəsi nəticəsində istifadəçi məmnuniyyətinin 35% artdığı müşahidə edilmişdir.
- Load Testing: Apache JMeter istifadə edərək, platformanın müxtəlif xidmətlərinin yüksək yük altında performansı ölçülmüşdür. Məsələn, Pik saatlarda mikroservis arxitekturası ilə platformanın istifadəçi tələblərinə cavab vermək qabiliyyəti stress testlər ilə sınanmış və qənaətbəxş nəticələr əldə edilmişdir.
- Stress Testləri: Platformanın performansı maksimum yük altında test edilmiş və sistemin stabilliyi təmin olunmuşdur.

Nəşrlər. Azərbaycan Texniki Universiteti və Bakı Dövlət Universitetində aşağıdakı mövzulara aid məqalələr və tezislər təqdim etmişik. Bunlar:

1. Müasir informasiya sistemlərində n+1 performans problemi və həll üsulları (Bakı Dövlət Universiteti. 2024).
2. Müasir arxitekturalı ayrılmış bank informasiya sistemlərində tranzakiyaların idarə olunmasında saga patterinin tətbiqi (Bakı Dövlət Universiteti. 2024).
3. Müasir informasiya sistemlərində monolit və mikroservis arxitekturaların rolu və həll etdiyi problemləri (Azərbaycan Texniki Universiteti. 2024).
4. Mikroservislər arxitekturalı informasiya sistemlərdə api gateway patternin nəzəri və praktiki tətbiqi (Azərbaycan Texniki Universiteti. 2024).

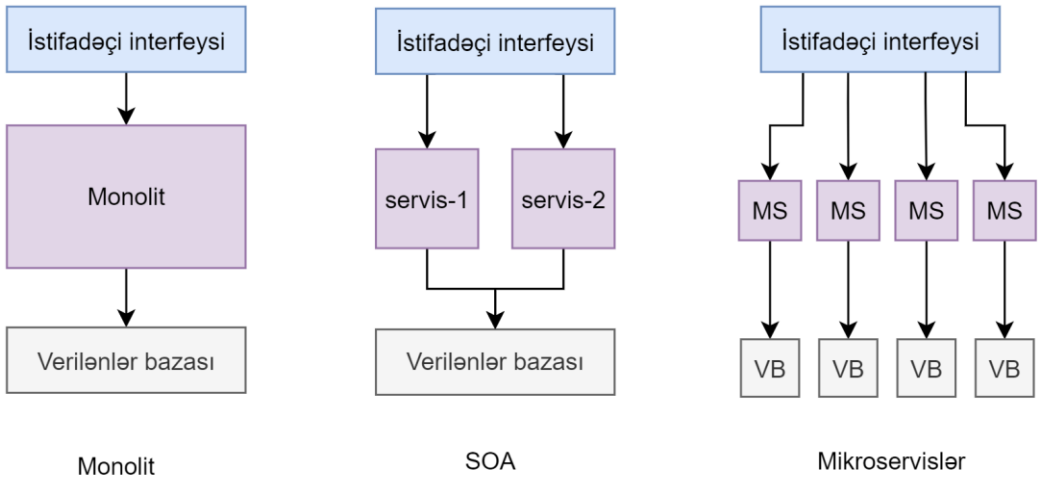
I FƏSİL. İNFORMASIYA SISTEMLƏRINDƏ ARXITEKTURA

1.1 Arxitektura nədir?

Proqram təminatının arxitekturası sistemin yüksək səviyyəli dizayını və strukturunu ifadə edir. Bir proqram təminatının müxtəlif komponentlərinin necə qarşılıqlı əlaqədə olmasını, məlumatların necə idarə olunmasını və bütövlükdə sistemin nəzərdə tutulan funksionallığına necə nail olmasını müəyyən edən qərarları və təlimatları əhatə edir. Proqram təminatının arxitekturası proqramçılar üçün bir plan rolunu oynayır və proqram təminatının qurulmasına xidmət edir [Martin, 2017].

Proqram təminatının əsasən üç fərqli arxitektura nümunəsi var (Şəkil 1.1):

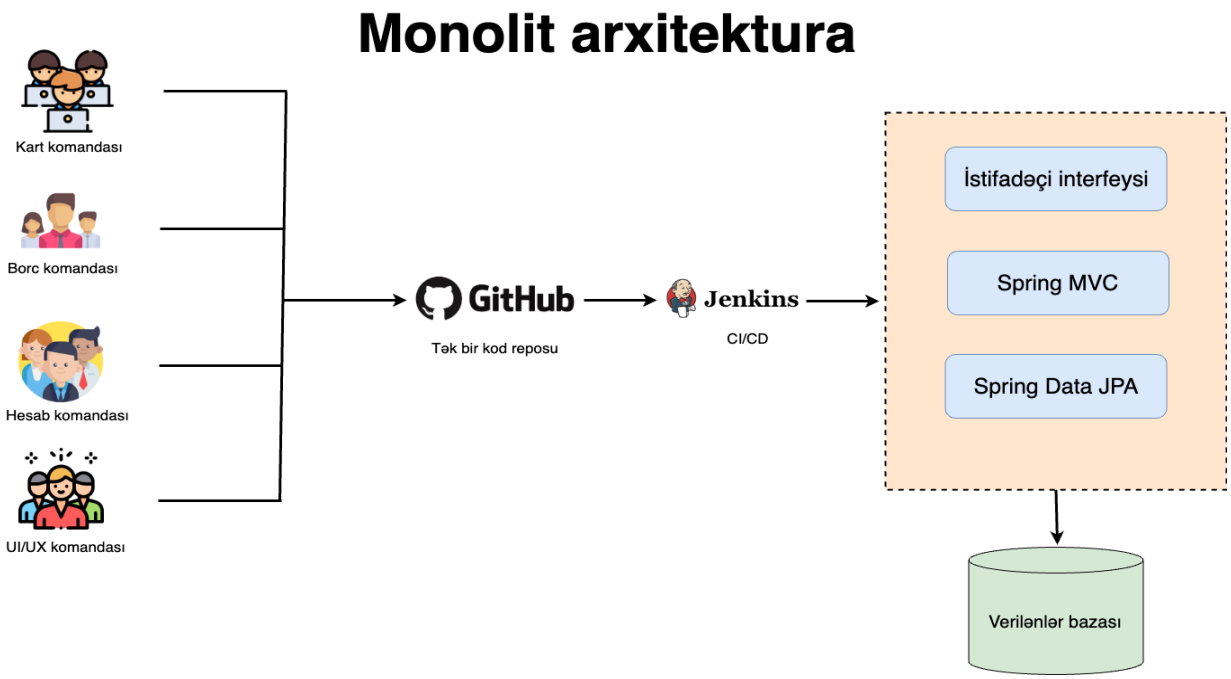
- **Monolit arxitekturası:** Bütün komponentlərin bir-birinə bağlı olduğu vahid, sıx inteqrasiya olunmuş arxitektura.
- **Xidmət yönümlü arxitektura (SOA):** Bir-biri ilə əlaqə saxlayan servislərə bölmüş və eyni verilənlər bazasını istifadə edən arxitektura.
- **Mikroservislər arxitekturası:** Bir-biri ilə əlaqə saxlayan kiçik, müstəqil servislərə bölmüş arxitektura.



Şəkil 1.1. İnformasiya sistemlərində arxitekturalar

1.2 Monolit arxitekturası

Monolit arxitektura proqram təminatının dizaynında ənənəvi yanaşmaya istinad edir, burada tətbiqin bütün komponentləri bir yedə sıx şəkildə inteqrasiya olunur və vahid kod bazasına malik olur. Monolit arxitekturada istifadəçi interfeysi, biznes məntiqi və məlumat girişi təbəqələri daxil olmaqla bütün proqram tək vahid kimi hazırlanır, serverə yerləşdirilir və miqyaslanır. Monolit proqramlar adətən bir proqramlaşdırma dili və ya texnologiya yığınından istifadə etməklə qurulur və onlar tək server və ya server klasterində işləyir (Şəkil 1.2).



Şəkil 1.2. Monolit arxitekturası

Monolit arxitekturadan nə vaxt istifadə edilməlidir?

- Startuplarda ilkin məhsulun hazırlanması zamanı
- Miqyası və mürəkkəbliyi sadə, aydın olan layihələrdə
- Komandada mikroservislər arxitekturasında təcrübəsi olan şəxlər yoxdursa monolit arxitekturasından istifadə etmək daha məsləhətlidir.

Müsbət tərəfləri:

- **Proqram təminatını yazmaq rahatdır** - Monolit yanaşma tətbiqlərin qurulmasının standart üsuludur, istənilən mühəndis komandası monolit tətbiqi inkişaf etdirmək üçün lazımi bilik və bacarıqlara malikdir.
- **Debug və test etmək daha rahatdır** - Monolit tətbiqləri sazlamaq və sınaqdan keçirmək daha asandır. Monolit tətbiqlətin tək kod bazası olduğundan, biz end-to-end testləri daha sürətli keçirə bilərik.
- **Serverə yerləşdirmək daha rahatdır** - Monolit tətbiqlərdə çoxlu fayılları serverə yükləməyə ehtiyac yoxdur, sadəcə bir .jar/.war (javada yazılan tətbiq üçün) fayılını yükləmək kifayət edir.

Mənfi tərəfləri:

- **Zaman keçdikcə mürəkkəbləşir və başa düşmək çətin olur** - Zamanla tətbiqin funksyonallığı artır və monolit kod bazası olduqca böyük və mürəkkəb olur buna görə də idarə etmək çətinləşir.
- **Dəyişiklik etmək çətinləşir** - Yüksək sıx birləşmə ilə belə böyük və mürəkkəb tətbiqdə yeni dəyişiklikləri həyata keçirmək daha çətindir. İstənilən kod dəyişikliyi bütün sistemə təsir edir.
- **Yeni texnologiyalara adaptasiya olmaq çətindir** - Yeni texnologiyanın tətbiqi son dərəcə problemlidir, çünki monolitdə olan bir-birinə bağlı asılılıqlar səbəbindən bütün tətbiq yenidən işlənib hazırlanmalıdır.
- **Miqyaslamaq çətindir** - Komponentləri müstəqil şəkildə miqyaslandırmaq olmur, yeganə seçim bütün tətbiqin miqyasını artırmaqdır bu da əlavə resurs sərf edir [Kocher, 2018].

1.3 Mikroservislər arxitekturası

Mikroservislər birlikdə işləyən kiçik, müstəqil servislərdir. Gəlin bu tərifi bir qədər daha da aydınlaşdırmağa çalışaq. Təsəvvür edək ki bizim online alış-şatış bir biznesimiz var. Bu biznes üçün proqram təminatını yazmaq lazımdır. Proqram təminatının kod hissəsi yazılmamışdan əvvəl onun arxitekturasını qurmaq lazımdır. Axrxitekturanı qurmaq üçün

isə bizə biznesin tələbləri və funksional və qeyri-funksional tələblər lazımdır. Belə bir platforma üçün təxmini aşağıdakı formada tələblər (user story) ola bilər. Bunlar:

▪ Mən bir istifadəçi kimi **məhsulları siyahıya almaq** istəyirəm.

▪ Mən bir istifadəçi kimi məhsulları **marka və kateqoriyalara görə süzgəcdən keçirmək** istəyirəm.

▪ Mən bir istifadəçi kimi **məhsulun bütün xüsusiyyətlərini** ekranında görmək istəyirəm.

▪ Bir istifadəçi kimi almaq istədiyim məhsulları **alış-veriş səbətinə qoya bilmək** istəyirəm.

▪ Bir istifadəçi kimi mən səbətimdə olan bütün məhsulların **ümumi dəyərini görmək** istəyirəm ki, **almaq imkanım olub-olmadığını öyrənim**.

▪ Bir istifadəçi olaraq alış-veriş səbətində hər bir məhsulun ümumi dəyərini görmək istəyirəm ki, **malların qiymətini yenidən yoxlaya bilim.**

▪ Mən bir istifadəçi kimi bütün **məhsulların hara göndəriləcəyini müəyyən etmək** istəyirəm.

▪ Mən bir istifadəçi kimi **çatdırılma ünvanına qeyd əlavə etmək** istəyirəm ki, **xüsusi təlimatlar** verə bilim.

▪ Bir istifadəçi kimi mən **kredit kartı məlumatımı yoxlanışı** zamanı dəqiqləşdirmək istəyirəm ki, **məhsullar üçün ödəniş edə bilim.**

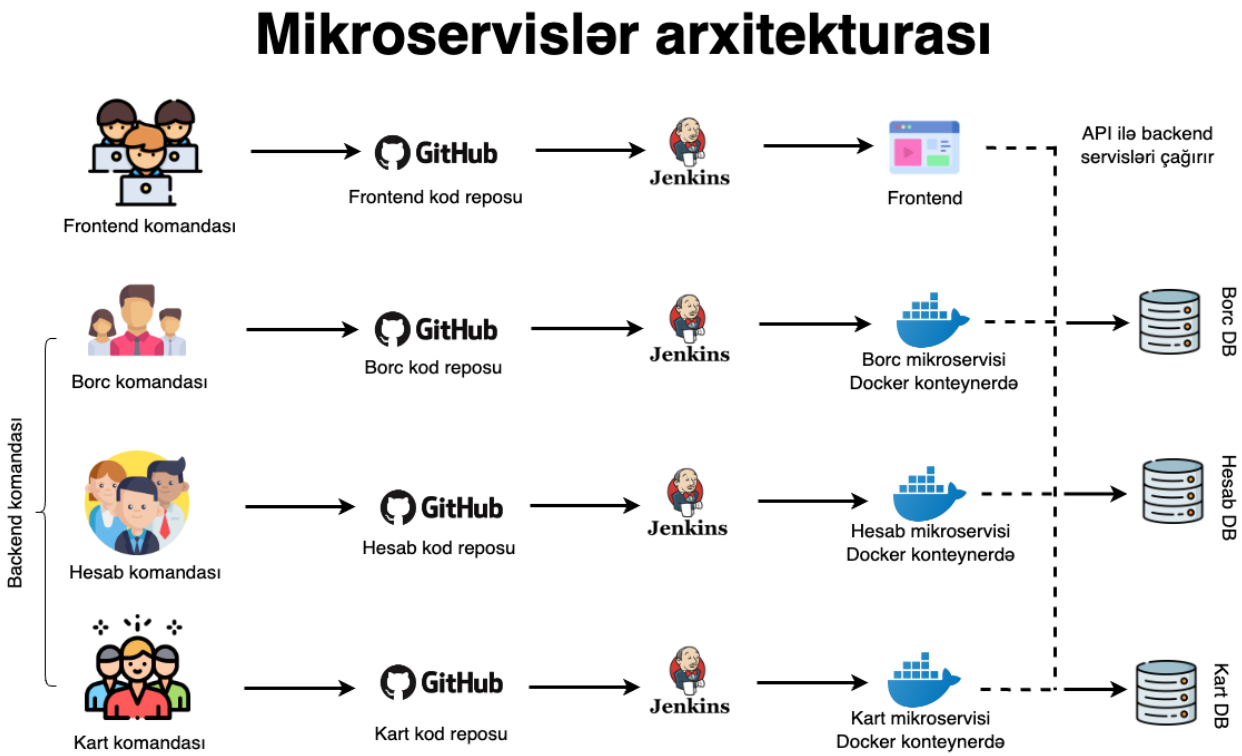
▪ Bir istifadəçi olaraq sistemin mənə **nə qədər məhsulun olduğunu** bildirməsini istəyirəm ki, nə qədər məhsul ala biləcəyimi biləcəm.

▪ Bir istifadəçi kimi mən sifariş nömrəsi ilə **sifarişin təsdiqi e-poçtu almaq istəyirəm** ki, məndə satınalma sübutu olsun.

▪ Bir istifadəçi kimi **köhnə sifarişlərimi və sifariş tarixçələrimi sadalamaq** istəyirəm.

▪ Bir istifadəçi kimi mən sistemə **istifadəçi kimi daxil olmaq** istəyirəm və sistem **alış-veriş səbətimdəki əşyaları yadda saxlamalıdır.**

Yuxarda qeyd olunan istifadəçi hekayələrinə baxsaq bu platformanın müxtəlif funksionallıqları var. Eyni zamanda proqram təminatı kifayət qədər böyükdür buna görə fərqli komandalar lazım olacaq. Zaman keçdikcə yeni funksionallıqlar gələ bilər, müəyyən dəyişikliklər ola bilər. Bu platforma üçün ən yaxşı seçimlərdən biri mikroservislər arxitekturasıdır(Şəkil 1.3).



Şəkil 1.3. Mikroservis arxitekturası

Martin Fowlerin mikroservislər haqqındakı məqaləsində: Mikroservis arxitektura üslubu hər birinin öz prosesi ilə işləyən və bir-biri ilə HTTP və ya gRPC API ilə əlaqə saxlayan kiçik servislər dəsti kimi vahid tətbiqiq hazırlanma yanaşmadır.

Mikroservislər arxitekturasının xüsusiyyətləri:

- Birlikdə işləyə bilən kiçik, müstəqil və bir-birinə bağlı olmayan servislərdir.
- Hər bir servis kiçik proqramlaşdırma qrupu tərəfindən idarə oluna bilən ayrıca kod bazalarından ibarətdir
- REST, gRPC API-lərdən istifadə etməklə bir-biri ilə əlaqə qurur.
- Bir çox müxtəlif texnologiya yığınları ilə işləyə bilər.

- Hər bir servisin digər servislər ilə paylaşılmayan öz verilənlər bazası var

Müsbət tərəfləri:

- **Çevik, innovativ və bazara hazır**-Mikroservis arxitekturaları tətbiqlərin miqyasını artırmaq rahatdır və daha sürətli inkişaf etdirilir, innovasiyalara imkan verir və yeni funksiyalar üçün bazara çıxma müddətini sürətləndirir.
- **Rahat ölçüləndirilmə qabiliyyəti** - Mikroservislərin müstəqil şəkildə miqyasını artırmaq olar, beləliklə, siz bütün tətbiqi miqyaslandırmadan daha az resurs tələb edən kiçik servisləri genişləndirə bilərik.
- **Kiçik və fokuslanmış komandalar** - Mikroservislər tək bir komandanın rahatlıqla yaza biləcəyi, test və deploy edə biləcəyi qədər kiçik olmalıdır.
- **Kiçik və ayrı kod bazası** - Mikroservislər kodu və ya məlumat bazasını digər servislərlə paylaşmır, asılılıqları minimuma endirir və yeni funksiyalar əlavə etməyi asanlaşdırır.
- **Rahat Deployment** - Mikroservislər davamlı inteqrasiya və fasiləsiz çatdırılma (CI/CD) imkanı verir, yeni ideyaları sınamağı və bir şey işləmədikdə geri qayıtmağı asanlaşdırır.
- **Aqnostik texnologiya, iş üçün düzgün alət** - Kiçik komandalar öz mikroservislərinə uyğun texnologiyanı seçə və servislərində texnologiya yığınlarının qarışığından istifadə edə bilərlər.
- **Davamlılıq və problemin izolyasiyası** - Mikroservislər nasazlığa dözümlüdür, retry və circuit breaking patternlərini həyata keçirməklə nasazlıqları düzgün idarə edir.
- **Məlumatların izolyasiyası** - Verilənlər bazaları mikroservislərin dizaynına görə bir-biri ilə ayrılır. Sxem yeniləmələrini yerinə yetirmək daha asandır, çünki yalnız bir verilənlər bazasına təsir edir.

Mənfi tərəfləri:

- **Mürəkkəblik** - Hər bir servis sadədir, lakin bütün sistem daha mürəkkəbdir. Deployment və bir-biri ilə əlaqə yüzlərlə mikroservis üçün mürəkkəb ola bilər.

- **Şəbəkə problemləri və gecikmə** - Mikroservislərdə daxili servislər bir-biri ilə əlaqə qurur, bu zaman servislərdə zəncirli əlaqələr gecikmə problemlərini artırır və biz şəbəkə problemlərini idarə etməliyik.
- **Development və test** - Monolit arxitekturası ilə müqaisədə mikroservis arxitekturalarında E2E prosesləri develop və test etmək çətindir.
- **Məlumatların bütövlüyü** - Mikroservislər öz məlumat davamlılığına malikdir. Məlumatların ardıcıllığı (consistency) problem ola bilər. Mümkünsə, eventual consistency-yə əməl edin.
- **Deployment** - Deployment çətin olur. Çünki, çoxlu devops avtomatlaşdırma proseslərinə və alətlərinə sərmayə qoymağ lazım gəlir. Mikroservislərlərin mürəkkəbliyi insanların tərəfindən deploy edilməsi üçün hədsiz dərəcədə böyük olur buna görə avtomatlaşdırma alətlərinə ehtiyac var.
- **Logging & Monitorinq** - Paylanmış sistemlər hər şeyi bir araya gətirmək üçün mərkəzləşdirilmiş loglanma sistemini tələb edir. Problemlərin mənbələrini izləmək üçün sistemin mərkəzləşdirilmiş loglanma sistemini vacibdir.
- **Debugging** - IDE vasitəsilə debugging artıq münkün olmur, çünki onlarla və ya yüzlərlə servis bir IDE-də işləməyəcək.

1.4 Niyə bizim mikroservislər arxitekturasına ehtiyacımız var?

Mikroservislər arxitekturası sənayedə ən müasir və ən məşhur arxitektura formasından biridir. Bu arxitektura formasında bütün sistem müstəqil servislərin birləşməsindən ibarətdir. Hər bir servisin öz xüsusi funksiyonallığı var və bu servislər fərqli proqramlaşdırma dillərində yazıla bilir.

Dünyanın ən böyük və uğurlu texnologiya şirkətlərin əksəriyyəti (Google, Amazon, Netflix, Spotify, Uber və s.) və Azərbaycanda olan əsas şirkətlər (Paşa Bank, Kapital Bank, ABB, GoldenPay və s.) mikroservis arxitekturasından istifadə edir və onların texnoloji uğurlarına ən böyük töhfə verənlərdən biri bu arxitektura hesab olunur.

Əgər mikroservis arxitekturası düzgün tətbiq olunarsa, təşkilatlar layihəhər üzərində rahatlıqla işləyə bilmək üçün minlərlə hətta on minlərlə müstəqil fəaliyyət göstərən kiçik komandalar yarada bilər. Bu, öz növbəsində, həmin təşkilatlara milyardlarla istifadəçiyə çatan yüksək miqyaslı sistemlər qurmağa imkan verir ki, bu da onların əməliyyat xərclərini aşağı tutmaqla yanaşı, səmərəli və innovativ qalmasına imkan verir.

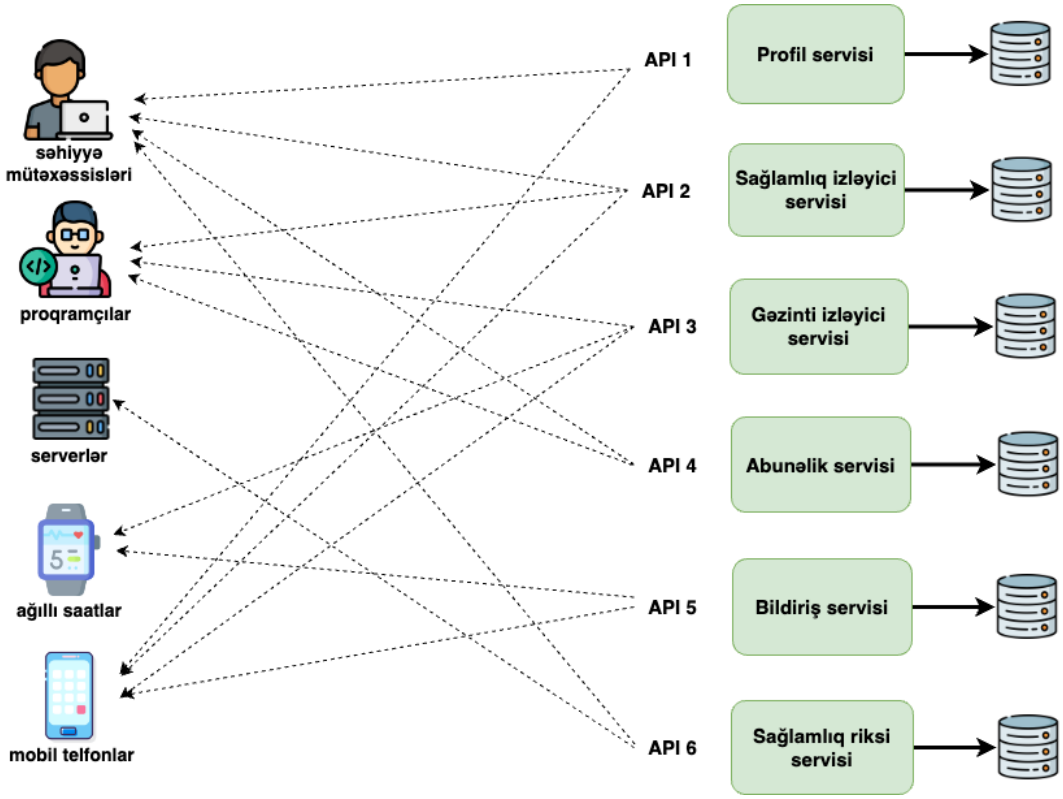
II FƏSİL. MIKROSERVISLƏR ARXITEKTURASINDA ƏLÇATANLIĞI TƏMIN ETMƏK ÜÇÜN DIZAYN PATTERNLƏR VƏ PRINSIPLƏR

2.1 API Gateway nümunəsi

Mikroservislər arxitekturasında çoxlu servislər olduğundan bəzi problemlər yaranır. Bu problemləri aradan qaldırmaq üçün hər problemə uyğun olaraq dizayn patternlər yaradılmışdır. İndi API gateway haqqında qısa məlumat və bu pattern olmayanda hansı problemlər yaranır, hansı problemləri həll etdiyinə həm nəzəri həm də praktiki olaraq baxacağıq.

API Gateway müştərilər və backend servisləri arasında əlavə şəbəkə komponentidir. Müştərilər birbaşa backendlə əlaqə saxlamaq əvəzinə sorğuları yalnız API Gateway-ə göndərirlər. Orada daxil olan sorğular ya birbaşa emal oluna bilər, ya da əsas servislərə yönləndirilə bilər. Bununla bərabər API Gateway-də routing, autorizasiya, rate limiting, load balancing, keşləmə, monitorinq, loglama, transformasiya və s. funksyonallıqları var [Scholl, Swanson and Fernandez, 2019].

Gəlin indi API Gateway patterni olmadan yazdığımız mikroservislərin yaratdığı problemlərə baxaq. Təsəvvür edək ki, biz fitness mərkəzləri, xəstəxanalar və həkimlərin ofisləri kimi müştərilərə və bizneslərə rəqəmsal xidmətlər göstərən səhiyyə və fitnes şirkətiyik. Müştəri tərəfində, platformamızla qarşılıqlı əlaqədə olan ağıllı saatlar, mobil telefonlar və digər geyinilə bilən aksesuarlar kimi müxtəlif qurğularımız var. Bundan əlavə, bizim sistemimizə daxili olamaq üçün veb brauzerlərindən istifadə edən digər şirkətlərimiz, serverlərimiz, kompüterlərimiz və səhiyyə mütəxəssislərimiz var. Və onların funksionallığına xas olan müxtəlif API-lər ilə işləyən onlarla və hətta yüzlərlə mikroservisimiz var (Şəkil 2.1).



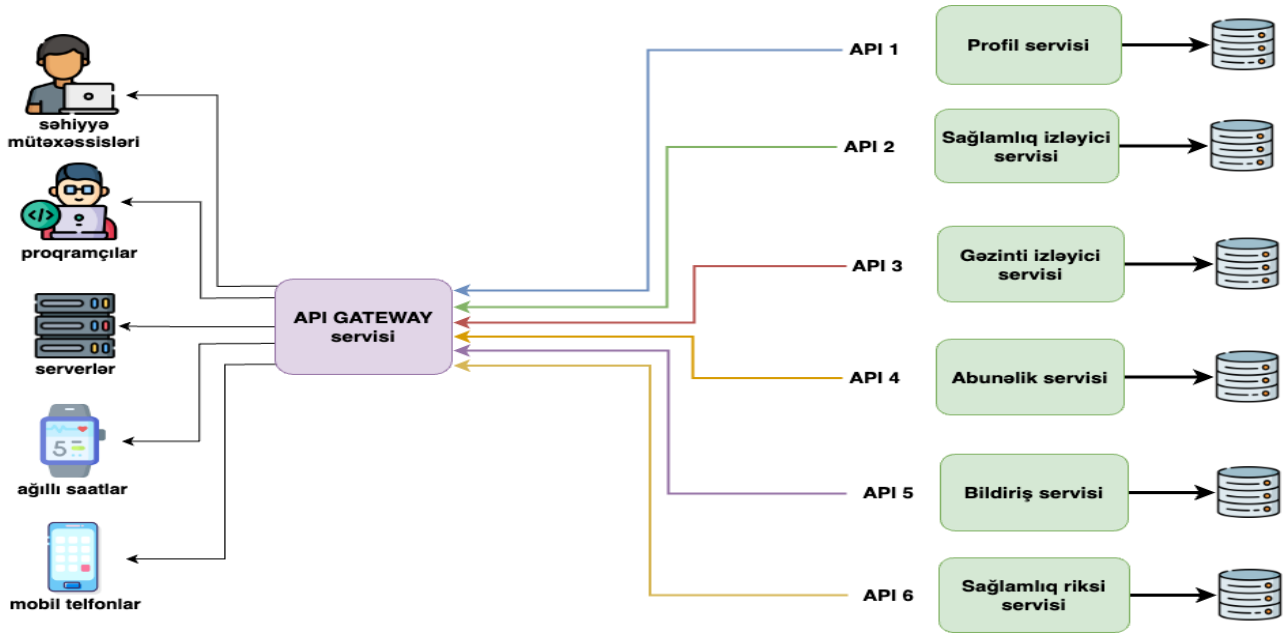
Şəkil 2.1. API Gateway olmadan yazdığımız mikroservislər

Şəkildə müştərilər və servislər arasında yaranan əlaqəni görə bilirik. Birinci problem, mikroservislər tərəfindən hazırlanmış müxtəlif API endpointləri çoxlu şəkildə hard code olunmuş formada frontendə və müştərilərin SDK-ları ilə əlaqələnib. Bu, müştəri tərəfindəki kodunu sistemimizin daxili tətbiqi ilə sıx əlaqələnib. Gələcəkdə biz sistemimizdə yeni dəyişiklik etsək bu sıx əlaqəni yenidən hər yerdən dəyişmək lazım olacaq. Gəlin bir nümunə üzərindən gedək. Məsələn, təsəvvür edək ki, nə vaxtsa biz mövcud sağlamlığı izləyən mikroservisimizi ürək döyüntülərini müəyyən edən və qan təzyiqini ölçən iki mikroservisə bölməyə qərar verdik. İndi hər iki mikroservis yalnız sağlamlığı izləyən mikroservisə aid olan API-ləri təqdim edir. Yuxarıdaki şəkildə göründüyü kimi sağlamlığı izləyən servisimiz səhiyyə mütəxəssisləri, proqramçılar və mobil telfonlar ilə əlaqələndirilib. Bu sıx birləşməyə görə, biz də yeni API-lərdən istifadə etmək üçün hər bir cihaz növündə bütün müştəri tərəfindəki kodunu yenidən nəzərdən keçirməli və düzətməli olacağıq. Bu da əlavə çətinlik və artıq işlər yaradır.

İkinci problem, hesab edək ki yazdığımız mikroservisləri üç müxtəlif şirkətə veririk. Hər şirkət isə özünün fərqli APİ-ləri qəbul etmə texnologiyası var. Birinci şirkət SOAP+XML, ikinci şirkət REST+JSON, üçüncü şirkət gRPC+ProtoBuf texnologiyası ilə işləyir. Əgər biz burda API Gateway istifadə etməsək servislərimiz üçün üç versiyada API yazmaq lazım olacaq (SOAP+XML, REST+JSON, gRPC+ProtoBuf) ki hər şirkətə uyğun formada məlumatı qaytaraq.

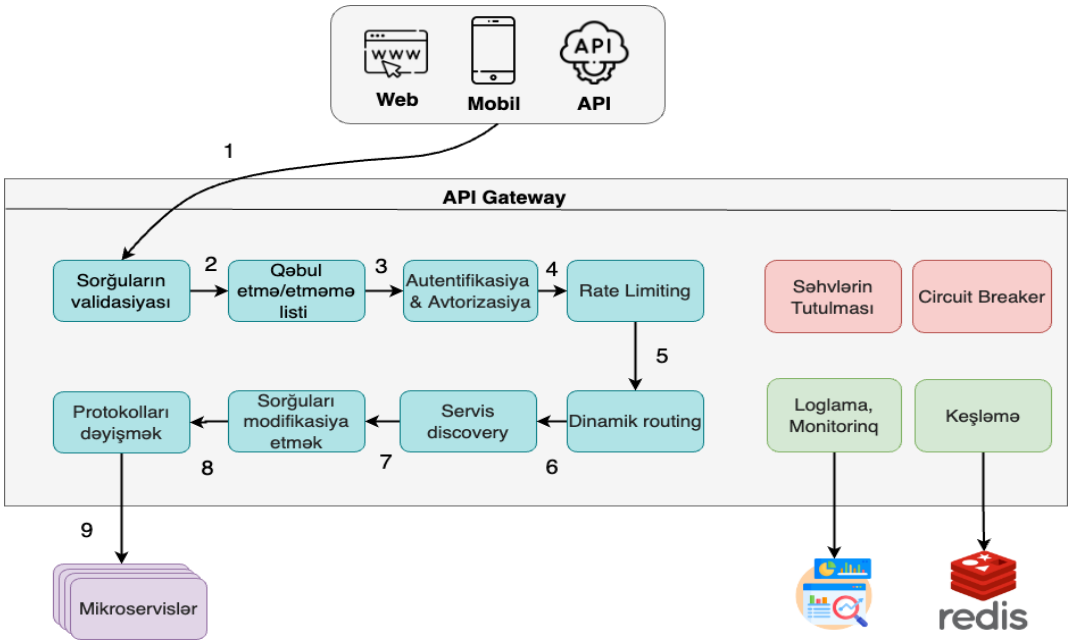
Üçüncü problem, mikroservislərimizi monitor etmək, izləmək, autorizasiya etmək, API-lərə limit qoymaqdır. Əgər bizim sistemimizdə Gateway olmasa bütün servislərdə bu məntiqləri yazmalı olacayıq. Bu həm əlavə iş olacaq ki hər komanda biznes məntiqə yox bu funksyonallıqlara baxacaq və əlavə vaxt itkisinə gətirib çıxaracaq, həm də kod dublikasiyası çox olacaq bu da proqramlaşdırmada yaxşı bir praktika deyil və əlavə olaraq biz bu funksiyonallıqlara yeni bir xüsusiyyət əlavə etsək bütün servisləri dəyişmiş olacayıq [Brian MacDonald and Holly Bauer, 2016].

Yuxarıda qeyd etdiyimiz problemləri aradan qaldırmaq, müştəri tərəfimizi daxili arxitekturadan ayırmaq və API-lərin idarəsini asanlaşdırmaq üçün API Gateway patterndən istifadə edə bilərik. API Gateway patterni xüsusilə mikroservis arxitekturası üçün faydalıdır və bir çox şirkətlər tərəfindən istifadə olunur (Şəkil 2.2).



Şəkil 2.2. API Gateway olduqda

Şəkildəki nümunədə göründüyü kimi sistemizin girişində API Gateway yerləşir. Ən sadə sennaridə bu pattern sadəcə verilmiş API endpointlərini müvafiq yerlərə yönləndirir. Lakin əksər hallarda o, yanlız yöndəndirmədən başqa daha çox şey edə bilir. API Gateway mikroservis arxitekturalarında müxtəlif məqsədlər üçün istifadə olunur. Aşağdakı şəkildə tam olaraq bu məqsədlər qeyd olunub (Şəkil 2.3).



Şəkil 2.3. API Gateway-in müxtəlif məqsədlər üçün istifadəsi

Gəlin indi bu funksyonallıqlara bir-bir baxaq:

1. Şəkildə gördüyümüz kimi, müştərilərin proqramları və mikroservislər var. Mikroservislər və müştəri proqramları arasında bizim API Gateway vasitəçi kimi çıxış edəcək. Birinci addımda mikroservislərdən gələn sorğu və cavabların gözlənilən formata və struktura uyğun olmasını yoxlamaq üçün istifadə edilə bilər. Bu, səhvlərin qarşısını almağa və mikroservislərin düzgün işləməsini təmin etməyə kömək edə bilər.

2. İkinci addımda IP ünvanının qara və ya ağ siyahıda olamasına baxır və buna uyğun olaraq müxtəlif formada davranır.

3. Üçüncü addımda bəzi autentifikasiya və avtorizasiyanı da həyata keçirə bilər. Burda artıq üçüncü bir tərəf (Keycloak) və yaxud özümüzün yazdığı servis ola bilər. Əgər bu servisdən normal cavab gəlsə mikroservislərə sorğu gedəcək, əks halda səhv olaraq geri qayıdacaq.

4. Autentifikasiya və avtorizasiyadan sonra biz bəzi rate limitini də tətbiq edə bilərik, yəni müəyyən bir müştəridən gələn sorğuların sayını və ya sorğuların sayını məhdudlaşdıra bilərik. Bu da DDOS hücumların qarşısını alır.

5. Beşinci addımda müştərilərdən sorğuları qəbul edir və onları müvafiq mikroservisə yönləndirir. Bu, müştərilərə bir giriş nöqtəsi vasitəsilə müxtəlif mikroservislərə daxil olmaq imkanı verir və ümumi sistem dizaynını sadələşdirir.

6. Altıncı addımda Servis discovery mövcud mikroservisləri və onların yerlərini aşkar etmək üçün istifadə oluna bilər ki, bu da müştərilərə xüsusi ünvanlarını bilmədən onlara daxil olmaq imkanı verir. Bu, müştərilərə təsir etmədən yeni mikroservislər əlavə etməyi və ya mövcud olanlara dəyişiklik etməyi asanlaşdıra bilər.

7. Yeddinci addımda çoxlu backend xidmətlərindən gələn cavabları müştəri üçün vahid cavabda birləşdirə bilər, müştərinin etməli olduğu sorğuların sayını azaldır və inteqrasiya prosesini sadələşdirir. Bu həmçinin aggregator da adlanır.

8. Səkkizinci addımda protokolun çevrilməsi gatewaydən gələn sorğuları backend servislərə yönləndirməzdən əvvəl bir protokoldan digərinə çevirməyi göstərib.

9. API Gateway, hətta backend xidmətləri əlçatmaz olduqda və ya gözlənilməz nəticələr qaytardıqda belə, səhvləri idarə etmək və müştərilərə səhv cavabları qaytarmaq üçün bir yol təqdim edir.

10. API Gateway-dən uğursuz mikroservisin bütün sistemi sıradan çıxarmasının qarşısını almaq üçün Circuit Breaker patternindən istifadə olunur. Circuit Breaker mikroservislərin sağlamlığına nəzarət edə bilir və zəruri hallarda avtomatik olaraq ehtiyat nüsxə cavablarını geri qaytarır.

11. API Gateway sorğular və cavablar haqqında ölçüləri və digər məlumatları toplaya bilər, mikroservislərin performansı və davranışı haqqında dəyərli fikirlər təqdim edir. Bu,

problemləri müəyyən etməyə və diaqnostika etməyə və sistemin ümumi etibarlılığını və dayanıqlığını yaxşılaşdırmağa kömək edə bilər.

12. API Gateway mikroservislərdən gələn cavabları keşləyə bilər, mikroservislərə yönləndirilməli olan sorğuların sayını azalda və sistemin ümumi performansını yaxşılaşdıra bilər [Walls 2018]

Gateway-in xidmət etdiyi bütün route-lərini təmin etmək üçün RouteLocator tipli class yaratmalıyıq. Request və responsda istənilən filtri daxil edə bilərik:

Filterlər:

1. Göndərilən sorğunu loglamaq
2. Sorğudan qayıdan cavabı loglamaq
3. Autentifikasiya filterini qoymaq

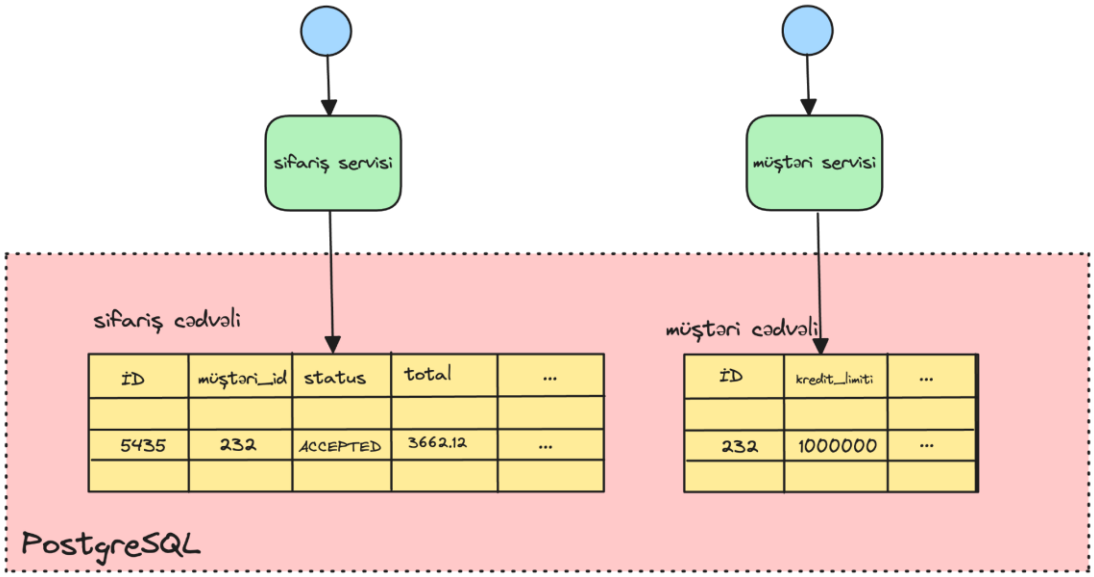
2.2 Database per service nümunəsi

Təsəvvür edək ki, siz mikroservis arxitekturası modelindən istifadə edərək onlayn mağaza proqramı hazırlayırsınız. Əksər servislər bir növ verilənlər bazasında məlumatları saxlamalıdır. Məsələn, **sifariş servisi** sifarişlər haqqında məlumatı, **müştəri servisi** isə müştərilər haqqında məlumatları saxlayır. Bu məlumatlar bir verilənlər bazasında iki ayrı cədvəl kimi saxlanır (Şəkil 2.4) [Bonér, 2016].

Problem

Mikroservislərdə verilənlər bazası arxitekturası necə olmalıdır?

- Servislər müstəqil olmalıdır ki, onlar sərbəst şəkildə inkişaf etdirilə, deploy oluna və genişləndirilə bilsin. Buna görə hər servisə uyğun öz databazası olmalıdır. Əks halda databazada bir problem baş versə bütün sistemdə problem baş verəcək.
- Bəzi biznes əməliyyatları birdən çox xidmətləri əhatə edən invariantları tətbiq etməlidir. Məsələn, Sifariş vermək nümunəsi yeni Sifarişin müştərinin kredit limitini keçməyəcəyini təsdiq etməlidir. Digər biznes əməliyyatları birdən çox servisə məxsus məlumatları yeniləməlidir.

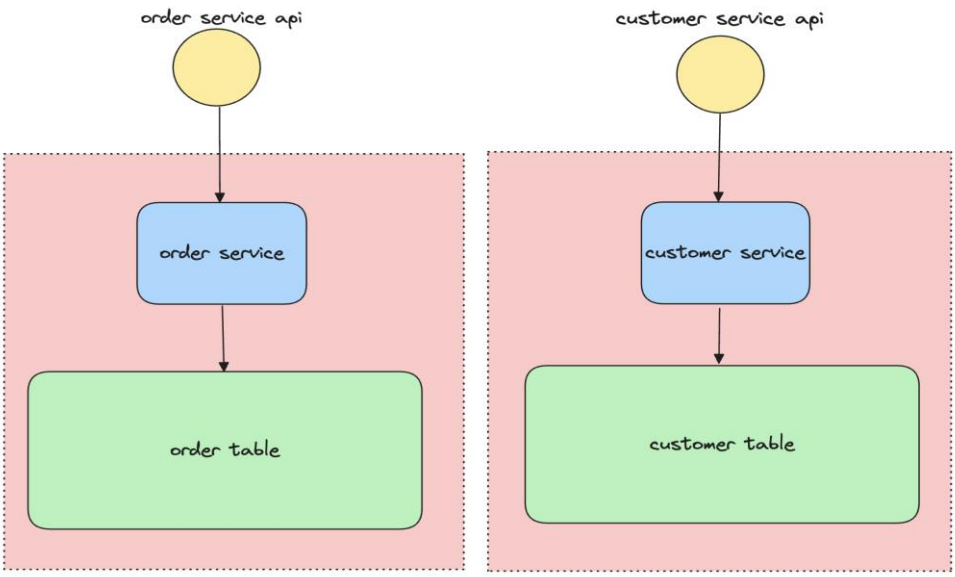


Şəkil 2.4. Bir verilənlər bazasında iki ayrı cədvəl

- Bəzi biznes əməliyyatları birdən çox servisə məxsus olan məlumatları sorğulamalıdır. Məsələn, Mövcud Kreditə Baxış sorğusu müştəridən kredit Limitini və açıq sifarişlərin ümumi məbləğini hesablamaq üçün Sifarişləri tapmaq üçün sorğu etməlidir.
- Bəzi sorğular çoxsaylı xidmətlərə məxsus olan dataya qoşulmalıdır. Məsələn, müəyyən bir bölgədə müştəriləri və onların son sifarişlərini tapmaq müştərilər və sifarişlər arasında birləşməni tələb edir.
- Məlumat bazaları bəzən miqyaslaşdırmaq üçün təkrarlanmalı və parçalanmalıdır.
- Fərqli servislərin fərqli məlumat saxlama tələbləri var. Bəzi xidmətlər üçün SQL verilənlər bazası ən yaxşı seçimdir. Digər xidmətlər üçün mürəkkəb, strukturlaşdırılmamış məlumatların saxlanması üçün yaxşı olan MongoDB kimi NoSQL verilənlər bazası və ya qrafik məlumatlarını səmərəli şəkildə saxlamaq və sorğulamaq üçün nəzərdə tutulmuş Neo4J tələb oluna bilər.

Həllər - Hər bir mikroxidmətin davamlı məlumatlarını həmin xidmətə özəl saxlayın və yalnız onun API vasitəsilə əlçatan olsun. Xidmətin əməliyyatları yalnız verilənlər bazasını əhatə edir.

Aşağıdakı diaqram bu modelin strukturunu göstərir (Şəkil 2.5).



Şəkil 2.5. Ayrı verilənlər bazasında iki ayrı cədvəl

Xidmətin məlumat bazası həmin xidmətin həyata keçirilməsinin fəal hissəsidir. Ona digər xidmətlər tərəfindən birbaşa daxil olmaq mümkün deyil.

Xidmətin davamlı məlumatlarını gizli saxlamağın bir neçə fərqli yolu var. Hər bir xidmət üçün verilənlər bazası serveri təmin etməyə ehtiyac yoxdur. Məsələn, əlaqəli verilənlər bazasından istifadə edirsinizsə, seçimlər bunlardır:

- Hər xidmətə görə şəxsi masalar - hər bir xidmətdə yalnız həmin xidmət tərəfindən istifadə edilməli olan cədvəllər dəsti var
- Hər xidmət üzrə sxem – hər bir xidmətin həmin xidmətə xas verilənlər bazası sxemi var
- Hər xidmət üçün verilənlər bazası serveri – hər bir xidmətin öz verilənlər bazası serveri var.

- Hər xidmətə görə fərdi masalar və hər xidmət sxemi ən aşağı yükə malikdir. Hər bir xidmət sxemindən istifadə cəlbedicidir, çünki bu, sahibliyi daha aydın edir. Bəzi yüksək məhsuldarlığa malik xidmətlər öz verilənlər bazası serverlərinə ehtiyac duya bilər.

Bu modulluğu gücləndirən maneələr yaratmaq yaxşı fikirdir. Məsələn, siz hər bir xidmətə fərqli verilənlər bazası istifadəçi identifikatoru təyin edə və icazələr kimi verilənlər bazasına girişə nəzarət mexanizmindən istifadə edə bilərsiniz. İnkapsulyasiyanı tətbiq etmək üçün heç bir maneə olmadıqda, tərtibatçılar həmişə xidmətin API-sini yan keçməyə və həmin xidmətin məlumatlarına birbaşa daxil olmağa meylli olacaqlar [Bruce and Paulo A. Pereira, 2018].

Mikroservislər modeli möhkəmliyi və sadəliyi ilə uzun müddətdir ki, tərtibatçılar arasında maraq mərkəzi olmuşdur. Bu, tərtibatçılara tətbiqləri tez və daha inteqrasiya olunmuş şəkildə yazmağa imkan verdi. Bir çox tərtibatçılar mikroservislərdən istifadə edərək ən müasir arxitekturalar yaratsalar da, onların yaxşıdan çox zərər verən antipatternləri də var. Tərtibatçıların etdiyi əsas dizayn səhvlərindən qaçmaq üçün müxtəlif mikroservis nümunələrini başa düşmək çox vacibdir. Bu yazıda gəlin mikroservislər paradiqması kontekstində verilənlər bazası modellərini və onların necə həyata keçirilə biləcəyini müzakirə edək. Bu nümunələrin nə olduğunu və e-ticarət sahəsində necə istifadə edildiyini nəzərdən keçirəcəyik.

Adətən monolit proqramda tətbiqin bütün ehtiyacları üçün məlumatları saxlayan tək böyük məlumat anbarı var. Mikroservislərdə biz tətbiqi müstəqil xidmətlərə bölürük. Hər bir xidmət modeli üçün verilənlər bazası ümumi məlumat anbarı əvəzinə hər bir mikroservis üçün müstəqil, genişlənə bilən və təcrid olunmuş verilənlər bazalarına malik olmağı təklif edir. Hər bir xidmət öz verilənlər bazasına müstəqil daxil olmalıdır, xidmətlər bir-birinin verilənlər bazasına birbaşa daxil ola bilməz.

Bu model xidmətlər arasında boş əlaqənin olmasını təmin edir ki, bir xidmətdəki problemlər və ya dəyişikliklər başqa xidmətlərə və ya verilənlər bazasına təsir göstərməsin. Bundan əlavə, bütün xidmətlər üçün vahid məlumat bazasından istifadə

etməyə ehtiyac yoxdur. Hər bir xidmət öz tələblərinə uyğun olaraq verilənlər bazasını seçə bilər.

Nümunə Ssenarisi-İstənilən e-ticarət platformasının ehtiyac duyduğu ən ümumi xidmətlərdən bəzilərinə misallar verək. Tipik Sifariş axını bir çox xidmətlər tərəfindən idarə olunur. Sifariş idarəetmə sisteminin qarşılıqlı əlaqədə olduğu iki xidmət məhsul xidməti və inventar xidmətidir. Element xidmətinə əşyalar, onların dəyəri, onları satan satıcılar, inventar xidmətinə isə əşyalar üçün nə qədər ehtiyat qaldığı barədə məlumat lazımdır. Hər iki məlumatın saxlanması tələbləri bir-biri ilə bağlıdır, lakin biz onları vahid verilənlər bazasında saxlasaq, o, çox böyük, mürəkkəb, idarə olunması çətinləşəcək və sorğuların (xüsusilə qoşulmaların) yerinə yetirilməsi çox vaxt aparacaq.

Bu problemi həll etmək üçün xidmət modeli üzrə verilənlər bazasından istifadə edə bilərik. Bir məhsul bazası məhsul məlumatlarını, digər inventar bazası isə inventar məlumatlarını saxlayacaq. Elementlər xidməti əşyalar bazasına, inventar xidməti isə inventar bazasına qoşulacaq. Beləliklə, məhsul xidmətində hər hansı bir problem yarandığı təqdirdə inventar xidməti fəaliyyətini davam etdirəcək.

Çoxsaylı xidmətlərdən məlumat tələb edən sorğuları necə yerinə yetirmək olar?

Burada həll edilməli olan əsas problemlərdən biri məlumatların həm elementlərdən, həm də inventar verilənlər bazalarından oxunması lazım olduqda baş verənlərdir. Məsələn, mənə məhsulun adı və o məhsulu satan bütün satıcıların inventar nömrələri lazımdırsa nə etməli? Bu halda biz adətən bir verilənlər bazası və içərisində iki cədvəl varsa birləşməni həyata keçiririk.

Proqramlar birləşməni yerinə yetirən verilənlər bazası əvəzinə birləşməni həyata keçirə bilər. Elementlər xidməti inventar xidmətindən məlumat tələb edirsə, o, sorğu verə bilər. İnventar xidmətində bu məlumatları təmin etmək üçün API-lər olacaq. Məsələn, müəyyən bir elementin inventarlaşdırılması üçün elementin adı elementlər bazasından əldə edilə bilər və sonra element ID-dən istifadə edərək, həmin element ID-si üçün inventar əldə etmək üçün inventar xidmətinə API sorğusu göndərilə bilər. Bu API tərkibi

kimi tanınır. Buna nail olmağın bir yolu aşağıda daha ətraflı izah edildiyi kimi Saga modelindən istifadə etməkdir [Rodger, 2017] .

2.3 Polyglot persistence nümunəsi

Proqram həllərinin artan mürəkkəbliyi ilə bir verilənlər bazası növü müasir tətbiqlərin müxtəlif ehtiyaclarını ödəmək üçün artıq kifayət etməyə bilər. Polyglot Persistence burada ortaya çıxır: ən yaxşı həll etdikləri problem əsasında eyni proqramda çoxlu verilənlər bazasından istifadə konsepsiyası. Hər bir xidmətin potensial olaraq öz verilənlər bazasına malik olduğu mikroservislər dünyasında Polyglot Persistence-dən istifadə etmək daha da məna kəsb edir. Bu yazıda biz Spring Microservices-də Polyglot Persistence-in necə tətbiq olunacağını araşdıracağıq [Tanasseri and Rai, 2017].

Çoxdilli Davamlılığa Giriş

Müasir rəqəmsal əsrdə proqram təminatının tətbiqi sahəsi böyüdükcə, onların emal etdiyi məlumatların müxtəlifliyi və mürəkkəbliyi də artır. Tətbiq tələblərinin müxtəlifliyi artdıqca, yalnız bir verilənlər bazası növünə etibar etmək proqramın potensialını məhdudlaşdıra bilər. Bu məhdudiyyət Multilingual Persistence kimi tanınan konsepsiyanın yaranmasına səbəb oldu.

Poliqlot Persistence nədir?

Polyglot Persistence, bir tətbiqdə müxtəlif məlumat saxlama ehtiyaclarını ödəmək üçün müxtəlif verilənlər bazası texnologiyalarından istifadə təcrübəsidir. “Çoxdilli” termini ümumiyyətlə birdən çox dildə danışan şəxsə aiddir. Eynilə, verilənlər bazası kontekstində bu, müəyyən bir tapşırığın xüsusi ehtiyaclarına uyğun olaraq, proqramın müxtəlif verilənlər bazaları ilə "danışmaq" qabiliyyətinə aiddir.

Polyglot Persistence-in mahiyyəti onun mantrasındadır: “Doğru iş üçün düzgün alətdən istifadə edin”. Tərtibatçılar bütün məlumatları bir ölçüyə uyğun həllə sığdırmağa çalışmaq əvəzinə, çoxsaylı verilənlər bazasının ən yaxşı xüsusiyyətlərindən istifadə edə bilərlər.

Ənənəvi olaraq, əlaqəli verilənlər bazaları əksər proqramlar üçün əsas həll yolu olmuşdur. Onlar məlumatların saxlanması və sorğulanması, ACID (Atomicity, Consistency, izolyasiya, davamlılıq) xassələri vasitəsilə ardıcıllıq və bütövlüyü təmin etmək üçün strukturlaşdırılmış metod təqdim edir. Bununla belə, veb-miqyaslı tətbiqlər meydana çıxmağa başladıqca, böyük həcmli, müxtəlif və sürətlə dəyişən məlumatların işlənməsi ehtiyacı meydana çıxdığından, ənənəvi relational verilənlər bazalarının məhdudiyyətləri aydın oldu.

MongoDB, Cassandra və Neo4j kimi NoSQL verilənlər bazalarının artması sənəd, açar/dəyər, sütun ailəsi və diaqram kimi müxtəlif məlumat modellərinə müraciət edən alternativlər təqdim etdi. Bu verilənlər bazaları, xüsusən də böyük verilənlər və real vaxt proqramları kontekstində əlaqəli verilənlər bazalarının tez-tez mübarizə apardığı genişlənmə və çeviklik təklif edirdi.

NoSQL verilənlər bazaları əlaqəli verilənlər bazalarının bəzi məhdudiyyətlərini həll etsə də, öz çətinliklərini də gətirir. Bəzilərində əlaqəli verilənlər bazalarının ciddi ardıcıllıq zəmanətləri yox idi, digərləri isə fərqli öyrənmə əyrilərinə və əməliyyat mürəkkəbliyinə malik idi.

Cavab relational verilənlər bazalarını NoSQL həlləri ilə tamamilə əvəz etmək deyil, hər birinin ən parlaq olduğu yerdə istifadə etmək idi. Beləliklə, Çoxdilli Davamlılıq konsepsiyası diqqəti cəlb etməyə başladı. O, tərtibatçılara strukturlaşdırılmış, ardıcıl yaddaşın tələb olunduğu relational verilənlər bazasından və çeviklik və genişlənmənin vacib olduğu NoSQL verilənlər bazasından istifadə etməyə imkan verdi.

Niyə indi həmişəkindən daha vacibdir?

Mikroservislər arxitekturasının meydana çıxması ilə, burada hər bir xidmət avtonom olmaq və müəyyən bir iş funksiyasına cavab vermək üçün nəzərdə tutulmuşdur, məlumatların xüsusi saxlanmasına ehtiyac daha aydın görünür. Tövsiyə xidməti istifadəçi seçimlərini və münasibətlərini izləmək üçün Neo4j kimi qrafik verilənlər bazasından istifadə edə bilər, giriş xidməti isə Elasticsearch kimi sənəd əsaslı NoSQL verilənlər bazasını daha uyğun tapa bilər.

Bundan əlavə, bulud əsaslı tətbiqlərin artması ilə müxtəlif idarə olunan verilənlər bazası xidmətləri ilə inteqrasiya imkanı əhəmiyyətli bir üstünlüyə çevrilir. Polyglot Persistence bu müxtəlif məlumat saxlama həllərini problemsiz şəkildə inteqrasiya etmək üçün çevikliyi təmin edərək, hər bir mikroservisin xüsusi ehtiyacları üçün ən optimallaşdırılmış verilənlər bazasına malik olmasını təmin edir.

Tətbiq tələblərinin artan müxtəlifliyi, mikroxidmətlərin artması və müxtəlif mülkiyyət bazalarının mövcudluğu birləşərək Polyglot Persistence-i sadəcə bir söz deyil, müasir tətbiqlərin səmərəliliyini, genişlənməsini və möhkəmliyini artıran əsl memarlıq strategiyasına çevrildi.

Mikroservis arxitekturaları təbii olaraq çoxdilli davamlılıq ssenarilərini uyğunlaşdırmağa meyllidir. Mikroxidmətlərin texnoloji müstəqilliyi sayəsində əslində funksiyalarından asılı olaraq müxtəlif məlumat saxlama texnologiyalarını qəbul etmək mümkündür.

Bir tərəfdən, belə bir yanaşma sistemin ümumi performansını artırmağa imkan verir. Memarlar və tərtibatçılar hər bir xidmətin spesifik performans tələblərinə ən yaxşı uyğun gələn funksionallığı təmin edən texnologiyanı seçməklə hər bir mikroservisin mümkün qədər səmərəli olması üçün sazlaya bilərlər. Digər tərəfdən, mürəkkəbliyin artması da ola bilər. Bu cür mürəkkəbliyi artıra bilən aspektlərdən biri tək mikroservislərə münasibətdə qlobal əhatə dairəsinə malik olan və beləliklə də onların sinxronizasiyasını tələb edən verilənlərlə təmsil olunur.

Poliqlot Persistence və Mikroservislər

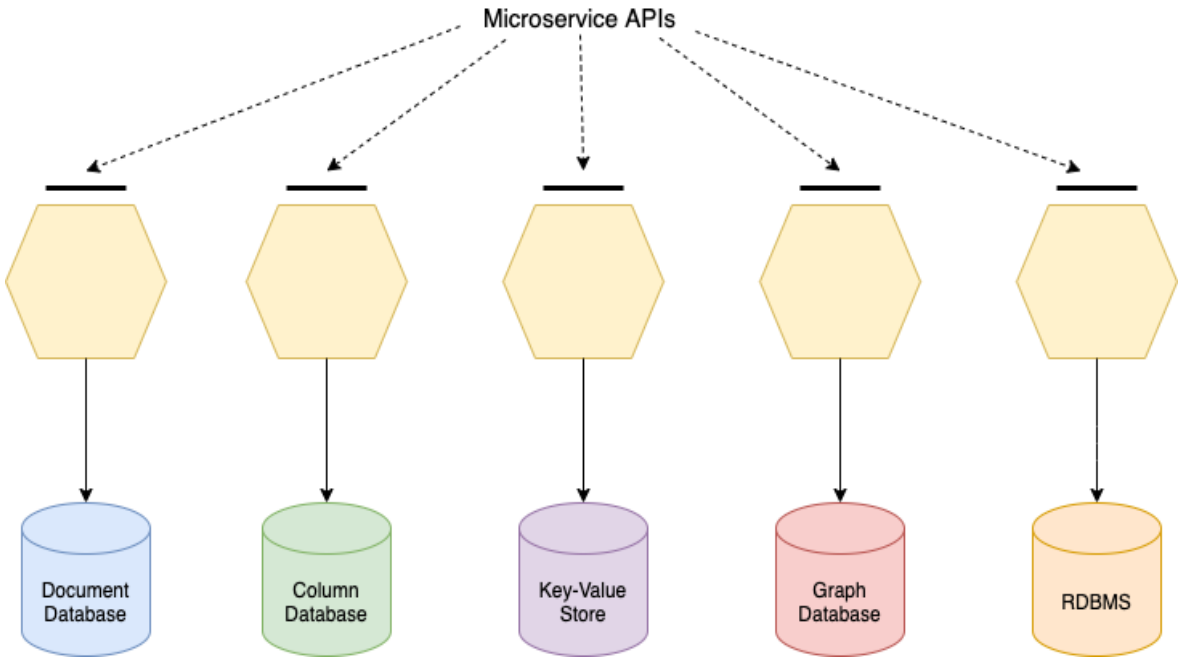
Poliqlot əzmkarlığı, hər bir texnologiyanın müxtəlif tələblərə cavab vermək üçün istifadə oluna biləcəyi eyni proqram daxilində birdən çox məlumat saxlama texnologiyalarından istifadə deməkdir. Məsələn:

- Əsas dəyər verilənlər bazaları – tez-tez sürətli oxumaq və yazmaq tələb olunduqda qəbul edilir.
- RDBMS - əməliyyatlar tamamilə zəruri olduqda və məlumat strukturları dəyişməz olduqda istifadə olunur.

- Sənədə əsaslanan verilənlər bazaları – yüksək yüklər və çevik məlumat strukturları ilə işləyərkən istifadə olunur.
- Qrafik verilənlər bazası – keçidlər arasında sürətli naviqasiya tələb olunduqda istifadə olunur.
- Sütun verilənlər bazaları – geniş miqyaslı analitika lazım olduqda istifadə olunur [Carnell, 2017].

Çoxdilli Davamlılıq Ssenarisi: One to One

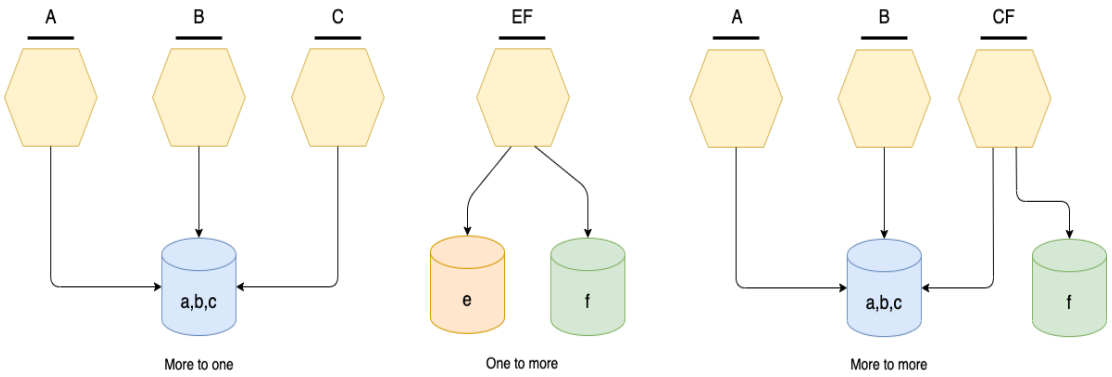
Mikroservis arxitekturasında ideal çoxdilli davamlılıq ssenarisi, hər bir məlumatın davamlılığı texnologiyası üçün unikal mikroservisin olduğu bir-bir ssenari dediyimiz şeydir (Şəkil 2.8). Bu, mikroservis yanaşmasının təbii törəmələrindən biridir, burada mikroservis sərhədləri onların idarə etdiyi texnologiya əsasında müəyyən edilə bilər və beləliklə, xarici təşəbbüskarlara əsas texnologiyanın xüsusiyyətlərindən mücərrəd olan API-lər dəsti təqdim edilir [Ibryam and Huß, 2019].



Şəkil 2.8. Çoxdilli Davamlılıq Ssenarisi

Kompleks Poliqlot Davamlı Yanaşmalar

Real həyatda ssenarilər daha mürəkkəb ola bilər. Daha çox mikroxidmət eyni məlumat mənbəyinə giriş tələb edə bilər və eyni mikroservis daxilində daha çox məlumat mənbəyi istifadə edilə bilər. Biz əvvəlki halı birdən-bir ssenari, ikinci halı isə birdən çox adlandırırıq. Nəhayət, biz qarışıq olanı daha çox adlandırırıq. Şəkil 2.9 bu ssenarilərin hər birini göstərir:



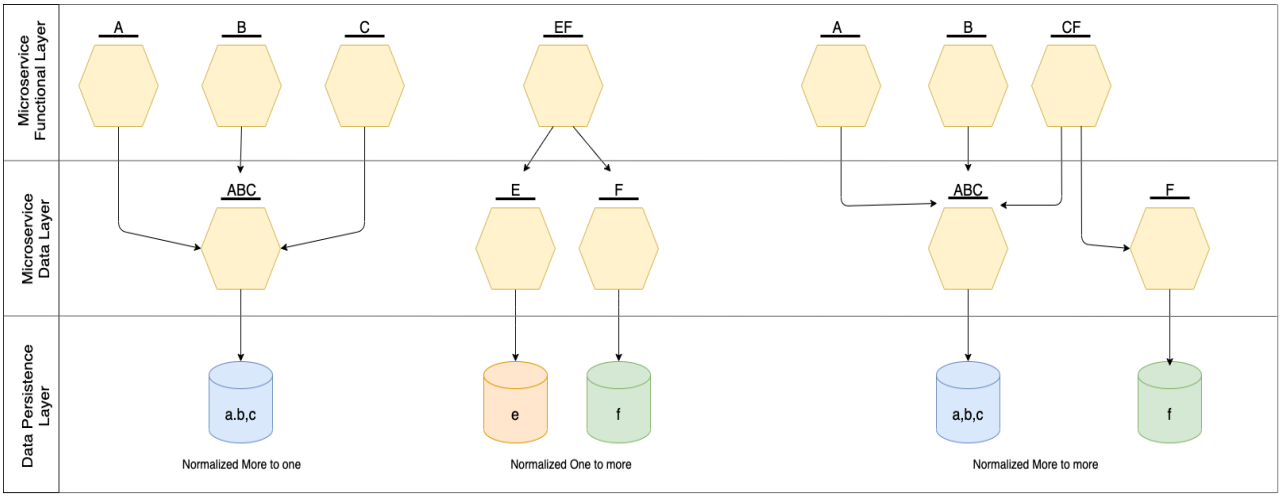
Şəkil 2.9. Kompleks Poliqlot Davamlı Yanaşmalar

Sennari	**Təsvir**
More-to-one (Çoxun birə)	Vahid məlumat mənbəyində olan məlumatlar çox böyük və müxtəlif olduğundan, müxtəlif funksional dəstləri təklif edən daha çox mikroservis tələb olunur. İntuitiv bir nümunə: müxtəlif məhsul növlərini idarə edən mikroservisləri olan müxtəlif məhsulların böyük kataloqu, çünki onlar fərqli xüsusiyyətlərə malikdirlər. Şəkil 2.9-də a, b və c mikroservis səviyyəsində API ABC ilə əlaqələndirilir. Qeyd: ABC mütləq A, B və C-nin yalın cəmi deyil; Təkmilləşdirmə və kompozisiya vasitəsilə yalnız A, B və C çıxarmaq mümkün olan əsas API dəsti ola bilər.

One-to-more (Birin çoxa)	Mikroservisin məlumat modeli məlumatların bir çox məlumat mənbəyində saxlanmasını tələb etdikdə Şəkil 2 e və f verilənlər dəstlərinin mikroservis səviyyəsində yaradıldığını və API EF-ə uyğunlaşdırıldığını göstərir.
More-to-more (Çoxun çoxa)	Bir-bir və daha çox-bir ssenarilərin birləşməsi

İdeal olaraq, bu üç ssenari bir-bir ssenariyə tamamilə normallaşdırıla bilər. Belə normallaşdırma arxitekturanı üç əsas təbəqəyə bölməyə imkan verir:

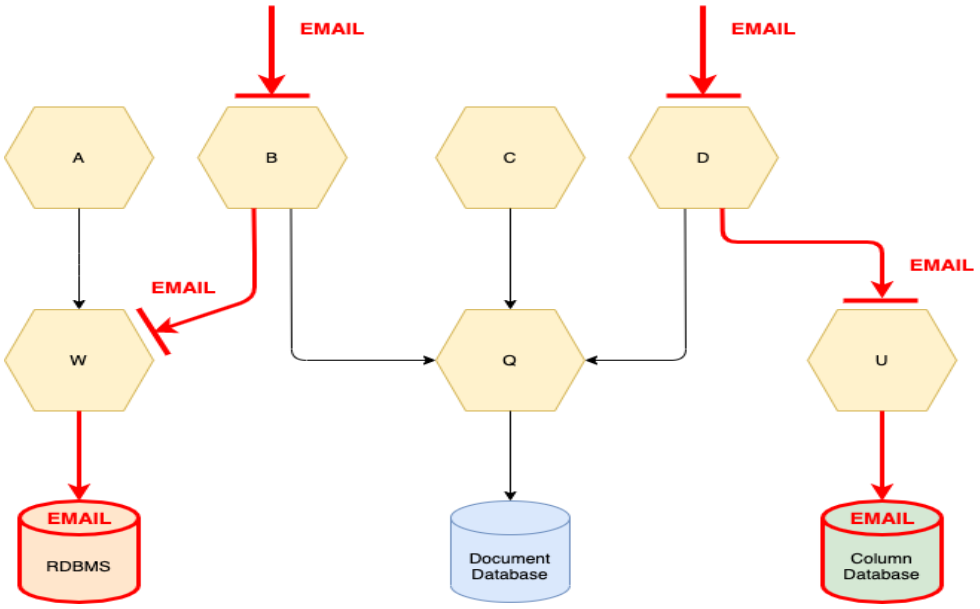
- Data davamlılıq səviyyəsi – məlumat mənbələrinin yerləşdiyi aşağı təbəqə.
- Microservice Data Layer – məlumat mənbəyini idarə edən mikroservis təbəqəsi.
- Mikroservis Funksional Layer – biznes funksiyalarını həyata keçirən mikroservislər.
- Belə bir normallaşma addımı məcburi deyil, memarlıq qərarları qəbul edərkən istinad kimi faydalı ola bilər. Şəkil 2.10 yuxarıdakı yanaşmaların üç qatlı arxitekturadan istifadə edərək bir-bir yaxınlaşmaya necə normallaşdırıla biləcəyini göstərir[Sharma, 2017] :



Şəkil 2.10. Kompleks Poliqlot Davamlı Yanaşmalar

Qlobal Data və Data Master Xidmətləri

Qlobal məlumatları müxtəlif mikroxidmətlərin imzalarında da tapmaq olar, çünki daha çox mikroxidmətləri əhatə edən əhatə dairəsinə malikdir. Nümunə olaraq müştərinin sahə elektron poçtunu götürək. O, istifadəçi profilləri ilə birlikdə məlumat mənbəyində saxlanılacaq, lakin ödənişləri emal etmək və ya hesabat yaratmaq üçün də tələb oluna bilər. Şəkil 2.11-də hər iki mikroservis B və D EMAIL qlobal məlumatının mövcud olduğu funksiyaları təmin edir. Mikroservis B E-MAIL mikroservis W-dən, mikroservis D isə mikroservis U-dan EMAIL qəbul edir. Bu halda, EMAIL-in həm RDBMS-də, həm də sütunlu verilənlər bazasında saxlandığını güman edirik.



Şəkil 2.11. Qlobal Data və Data Master Xidmətləri

Aydındır ki, göstərilən ssenaridə EMAIL-in dəyişikliyə görə fərqlənməsi mümkündür. Buna görə də, EMAIL dəyəri hər iki verilənlər bazasında sinxronlaşdırılmalıdır [Rusu, 2017].

Bunun üçün əsas məlumat mənbəyini və köməkçi mənbələri müəyyən etmək lazımdır. Məlumata edilən hər bir dəyişiklik əsas mənbədə edilməli və sonra köməkçi mənbələrə ötürülməlidir. Adətən əsas məlumat mənbəyi xüsusi qlobal məlumat növlərinə

daha çox diqqət yetirir. Əvvəlki nümunədə EMAIL əsas məlumat mənbəyində (müştəri qeydlərini saxlayan RDBMS) saxlanıla bilər, sütun verilənlər bazası isə qul kimi çıxış edə bilər.

Spring Boot Mikroservislərində Çoxdilli Davamlılığın Tətbiq edilməsi

Spring bootda mikroservis mühitində Polyglot Persistence tətbiqi məlumat ehtiyacları əsasında dizayn qərarlarının qəbul edilməsini və sonra müxtəlif verilənlər bazası texnologiyalarının qüsursuz inteqrasiyasını nəzərdə tutur. Bunu praktikada necə tətbiq edəcəyimizi araşdıraq.

Məlumat Ehtiyaclarının Qiymətləndirilməsi

Konfiqurasiyalara və koda keçməzdən əvvəl hər bir mikroservisin məlumat ehtiyaclarını qiymətləndirmək çox vacibdir:

- Məlumat Həcmi: Mikroservisin böyük həcmdə məlumatları emal etməsi gözlənilirmi?
- Sorğu Mürəkkəbliyi: Xidmət sadə CRUD əməliyyatları və ya daha mürəkkəb sorğular tələb edirmi?
- Oxuma və yazma əməliyyatları: xidmət oxumaq üçün intensivdir, yoxsa yazmaq üçün intensivdir?
- Məlumat Əlaqələri: Məlumat obyektləri arasında mürəkkəb əlaqələr varmı?

Məsələn, İstifadəçi Profili mikroxidməti çevik sxeminə görə MongoDB kimi sənəd verilənlər bazasından faydalana bilər, mürəkkəb əlaqə məlumatlarına malik Maliyyə Əməliyyatları xidməti PostgreSQL kimi əlaqəli verilənlər bazasına daha uyğun ola bilər.

Məlumat Mənbələrinin Konfiqurasiyası

Spring-in çevikliyi ona çoxlu məlumat mənbələrini konfiqurasiya etməyə imkan verir, hər bir xidmətin (hətta xidmət daxilindəki komponentlərin) müəyyən edilmiş verilənlər bazasına qoşulmasını təmin edir.

2.4 Microfrontends Arxitekturasının nümunəsi

Yaxşı frontend development çətindir. Bir çox komandanın eyni vaxtda böyük, mürəkkəb məhsul üzərində işləyə bilməsi üçün qabaqcıl inkişafı genişləndirmək daha çətindir. Bu yazıda biz ön hissənin monolitlərini daha kiçik, daha idarə edilə bilən parçalara bölmək istiqamətində son tendensiyanı və bu arxitekturanın front-end kodu üzərində işləyən komandaların effektivliyini və səmərəliliyini necə artıra biləcəyini izah edəcəyik. Müxtəlif faydalar və xərclər haqqında danışmaqdan əlavə, biz mövcud tətbiq variantlarından bəzilərini əhatə edəcəyik və texnikanı nümayiş etdirən tam nümunə proqrama daxil olacağıq.

Mikroservislər son illərdə populyarlıq qazandı və bir çox təşkilat böyük, monolit arxa tərəflərin məhdudiyyətlərindən qaçmaq üçün bu memarlıq üslubundan istifadə edir. Server tərəfi proqram təminatının qurulmasının bu üslubu haqqında çox şey yazılsa da, bir çox şirkət monolit ön kod bazaları ilə mübarizə aparmağa davam edir.

Ola bilsin ki, siz mütərəqqi və ya həssas veb tətbiqi yaratmaq istəyirsiniz, lakin bu xüsusiyyətləri mövcud koda inteqrasiya etməyə başlamaq üçün asan yer tapa bilmirsiniz. Ola bilsin ki, siz yeni JavaScript dili xüsusiyyətlərindən (və ya JavaScript-ə tərtib edilə bilən çoxsaylı dillərdən birini) istifadə etməyə başlamaq istəyirsiniz, lakin lazımi yaratma vasitələrini mövcud yaratma prosesinizə uyğunlaşdıra bilmirsiniz. Və ya bəlkə siz inkişafınızı elə genişləndirmək istərdiniz ki, birdən çox komanda eyni anda bir məhsul üzərində işləyə bilsin, lakin mövcud monolitdəki əlaqə və mürəkkəblik hər kəsin bir-birinin ayaq barmaqlarını tapması deməkdir. Bunların hamısı müştərilərinizə yüksək keyfiyyətli təcrübələr təqdim etmək qabiliyyətinizə mənfi təsir göstərə biləcək real problemlərdir [Fleming, 2018].

Son zamanlar mürəkkəb, müasir veb inkişafı üçün tələb olunan ümumi arxitektura və təşkilati strukturlara artan diqqətin şahidi oluruq. Xüsusilə, biz müştərilərə vahid inteqrasiya olunmuş məhsul kimi görünməklə yanaşı, müstəqil olaraq inkişaf etdirilə, sınaqdan keçirilə və yerləşdirilə bilən daha kiçik, daha sadə parçalara ön hissə

monolitlərini parçalamaq üçün modellərin ortaya çıxdığını görürük. Biz bu texnikanı microfrontends adlandırırıq və onu bu formada təyin edirik: "Müstəqil olaraq çatdırıla bilən ön proqramların daha böyük bir bütövlükdə qurulduğu bir memarlıq üslubu."

Mikrofrontendlərdən gördüyümüz əsas üstünlüklərdən bəziləri bunlardır:

- Daha kiçik, daha uyğun və saxlanıla bilən kod bazaları
- Xüsusi, muxtar komandaları olan daha genişlənən təşkilatlar
- Qabaq ucun hissələrini əvvəllər mümkün olduğundan daha artımlı şəkildə təkmilləşdirmək, yeniləmək və hətta yenidən yazmaq imkanı

Təsadüfi deyil ki, bu görkəmli üstünlüklər mikroservislərin təmin edə biləcəyi üstünlüklərdir.Proqram təminatının arxitekturasına gəldikdə, əlbəttə ki, pulsuz nahar yoxdur; hər şeyin qiyməti var. Bəzi mikro frontend proqramları istifadəçilərimizin yükləməli olduğu baytların sayını artıraraq asılılıqların təkrarlanmasına səbəb ola bilər. Bundan əlavə, komanda muxtariyyətinin kəskin artması komandalarınızın işində parçalanmaya səbəb ola bilər. Bununla belə, biz inanırıq ki, bu risklər idarə oluna bilər və mikro cəbhələrin faydaları çox vaxt xərclərdən üstündür.

Faydaları

Mikro cəbhələri xüsusi texniki yanaşmalar və ya həyata keçirmə təfərrüatları baxımından müəyyən etmək əvəzinə, biz əldə olunan atributları və onların təmin etdiyi faydaları vurğulayırıq.

Artan təkmilləşdirmələr

Bir çox təşkilatlar üçün bu, mikro frontend səyahətinin başlanğıcıdır. Köhnə, böyük ön hissə monolit, tam yenidən yazmağın cazibədar olduğu nöqtəyə qədər çatdırılma təzyiqi altında yazılmış texnoloji yığın və ya kod tərəfindən saxlanılır. Tamamilə yenidən yazılmanın təhlükələrindən qaçmaq üçün, monolit tərəfindən ağırlaşmadan müştərilərimizə yeni funksiyalar təqdim etməyə davam edərkən köhnə tətbiqi hissə-hissə boğmağa üstünlük veririk.

Bu, tez-tez mikro frontend arxitekturasına gətirib çıxarır. Bir komanda köhnə dünyada az dəyişikliklə istehsala bir xüsusiyyət əldə etmək təcrübəsinə sahib olduqdan

sonra, digər komandalar yeni dünyaya qoşulmaq istəyəcəklər. Mövcud kodun hələ də saxlanmasına ehtiyac var və bəzi hallarda ona yeni funksiyalar əlavə etməyə davam etmək mənasız ola bilər, lakin indi seçim etmək olar.

Nəticə budur ki, məhsulumuzun ayrı-ayrı hissələri üzrə hər bir halda qərar qəbul etmək və arxitekturamıza, asılılıqlarımıza və istifadəçi təcrübəmizə əlavə təkmilləşdirmələr etmək üçün daha çox sərbəstliyimiz var. Meynframemizdə böyük dəyişiklik olarsa, hər bir mikro cəbhə dünyanı dayandırmağa və hər şeyi bir anda təkmilləşdirməyə məcbur olmaqdansa, məntiqli olduğu zaman təkmilləşdirilə bilər. Yeni texnologiya və ya yeni qarşılıqlı əlaqə formalarını sınamaq istəyiriksə, bunu əvvəlkindən daha təcrid olunmuş şəkildə edə bilərik.

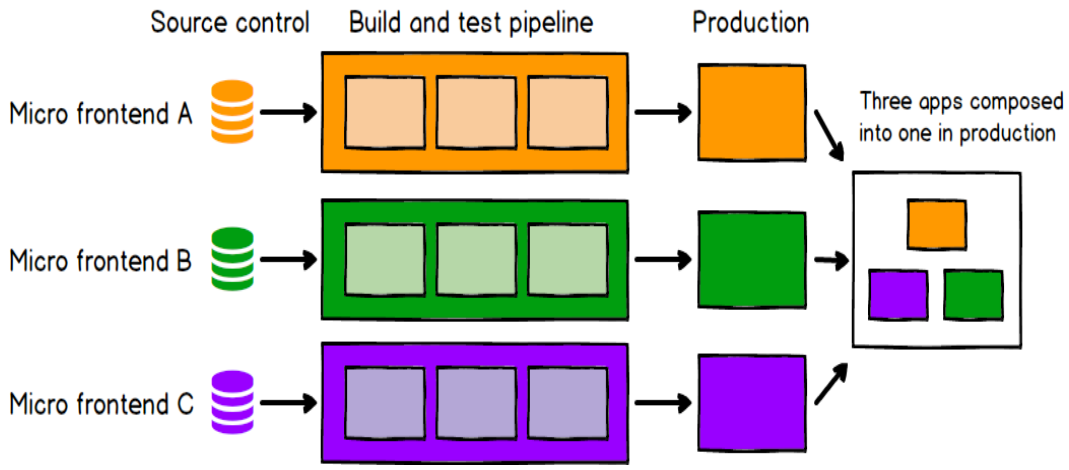
Sadə, ayrılmış kod bazaları

Hər bir mikro frontend üçün mənbə kodu, tərifinə görə, tək monolit cəbhə üçün mənbə kodundan çox kiçik olacaqdır. Bu kiçik kod bazaları tərtibatçıların işləməsi üçün daha sadə və asan olur. Xüsusilə, bir-birindən xəbərdar olmamalı olan komponentlər arasında qəsdən və qeyri-adekvat əlaqələr nəticəsində yaranan mürəkkəblikdən qaçırıq. Tətbiqin məhdud kontekstləri ətrafında daha qalın xətlər çəkməklə biz bu cür təsadüfi əlaqələrin baş verməsini çətinləşdiririk.

Əlbəttə ki, heç bir yüksək səviyyəli memarlıq qərarı (məsələn, "mikro cəbhələr edək") yaxşı köhnə təmiz kodu əvəz etməyəcək. Biz kodumuz haqqında düşünməkdən və onun keyfiyyətinə səy göstərməkdən özümüzü azad etməyə çalışmırıq. Bunun əvəzinə, pis qərarları çətin, yaxşı qərarları asan qəbul edərək uğur çuxuruna düşməyə özümüzü hazırlamağa çalışırıq. Məsələn, məhdud kontekstlərdə domen modellərini paylaşmaq çətinləşir, ona görə də tərtibatçıların bunu etmək ehtimalı azdır. Eynilə, mikro cəbhələr sizi məlumatların və hadisələrin tətbiqin müxtəlif hissələri arasında necə axdığı barədə açıq və qəsdən olmağa sövq edir, bu da bizim etməli olduğumuz şeydir [Vululleh, 2022].

Müstəqil paylanma

Mikroservislərdə olduğu kimi, mikro cəbhələrin müstəqil yerləşdirilməsi çox vacibdir. Bu, istənilən yerləşdirmənin əhatə dairəsini daraldır və bu, riski azaldır. Frontend kodunuzun necə və harada yerləşdirilməsindən asılı olmayaraq, hər bir mikro frontend onu istehsala qədər quran, sınaqdan keçirən və yerləşdirən öz davamlı yerləşdirmə boru kəmərinə malik olmalıdır. Biz digər kod bazalarının və ya boru kəmərlərinin cari vəziyyətini nəzərə almadan hər bir mikro cəbhəni yerləşdirə bilməliyik. Köhnə monolitin sabit, əl ilə, rüblük buraxılış dövründə olub-olmaması və ya yan tərəfdəki komandanın yarımçıq və ya pozulmuş funksiyanı əsas filialına itiləməsinin əhəmiyyəti olmamalıdır. Müəyyən bir mikro frontend istehsala girməyə hazırdırsa, bunu edə bilməlidir və bu qərar onu quran və saxlayan komandaya həvalə edilməlidir (Şəkil 2.12).

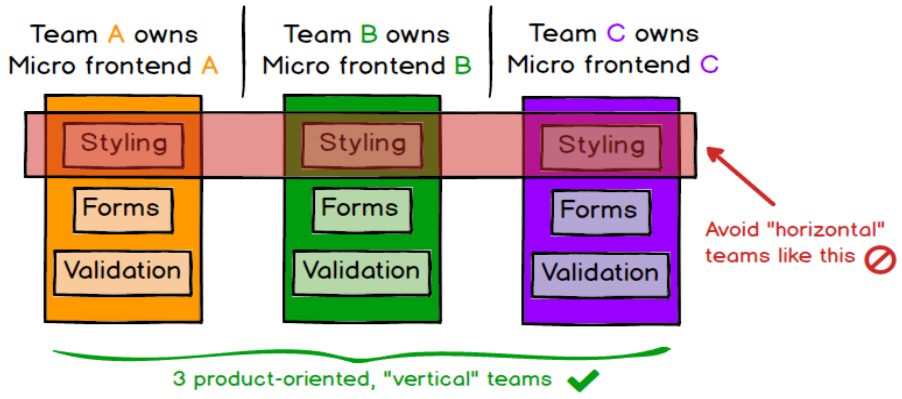


Şəkil 2.12. Müstəqil paylanma

Ayrı-ayrı komandalar

Həm kod bazalarımızı, həm də buraxılış dövrlərimizi ayırmağın daha yüksək səviyyəli faydası olaraq, ideyadan istehsala və ondan kənara qədər məhsulun bir hissəsinə sahib ola biləcək tamamilə müstəqil komandalara sahib olmaq üçün uzun bir yol qət edirik. Komandalar müştərilərə dəyər vermək üçün ehtiyac duyduqları hər şeyə tam sahib ola bilər ki, bu da onlara tez və effektiv hərəkət etməyə imkan verir. Bunun işləməsi üçün

komandalarımız texniki imkanlardan çox, biznes funksionallığının şaquli dilimləri ətrafında formalaşmalıdır. Bunun asan yolu məhsulu son istifadəçilərin görəcəkləri hissələrə bölməkdir; beləliklə, hər bir mikro frontend proqramın bir səhifəsini əhatə edir və tək komandaya başdan-başa məxsusdur. Bu, üslub, forma və ya təsdiqləmə kimi texniki və ya "üfüqi" məsələlər ətrafında komandalar yaratmaqla müqayisədə komandaların işində daha çox birlik təmin edir (Şəkil 2.13).



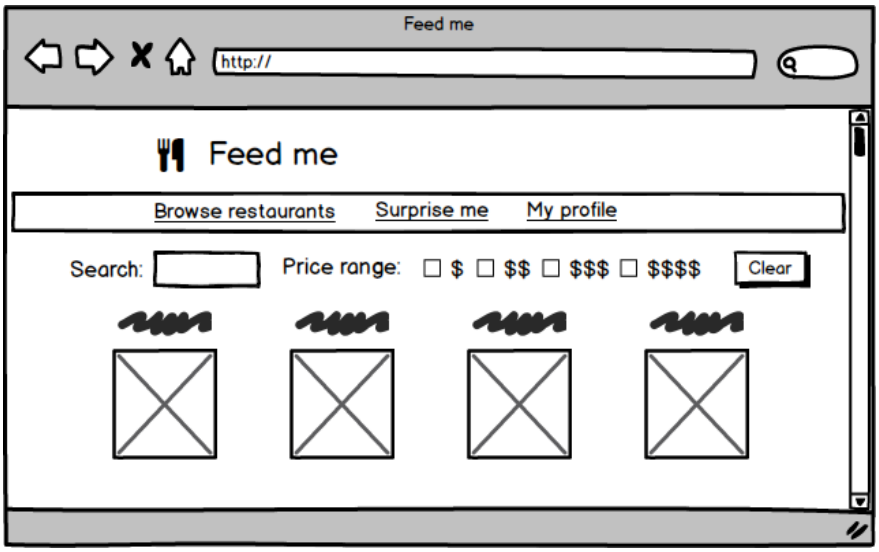
Şəkil 2.13. Ayrı-ayrı komandalar

Bir sözlə, mikro frontend böyük, qorxulu şeyləri daha kiçik, daha idarə oluna bilən hissələrə bölmək və sonra onlar arasındakı asılılıqları açıq şəkildə ifadə etməkdir. Texnoloji seçimlərimiz, kod bazalarımız, komandalarımız və buraxılış proseslərimiz həddindən artıq koordinasiya olmadan bir-birindən asılı olmayaraq işləyə və inkişaf edə bilməlidir.

Müştərilərin çatdırılması üçün yemək sifariş edə biləcəyi bir veb sayt təsəvvür edin. Səthdə bu olduqca sadə bir konsepsiyadır, lakin bunu yaxşı etmək istəyirsinizsə, təəccüblü miqdarda təfərrüat var (Şəkil 2.14):

- Müştərilərin restoranlara baxa və axtara biləcəyi açılış səhifəsi olmalıdır. Restoranlar; O, qiymət, mətbəx və ya müştərinin əvvəllər sifariş etdiyi şeylər daxil olmaqla, bir çox atributlar üzrə axtarıla və süzülməlidir.

- Hər bir restoranın menyu elementlərini əks etdirən və müştəriyə endirimlər, yemək təklifləri və xüsusi istəklərlə yemək istədiklərini seçmək imkanı verən öz səhifəsi lazımdır.
- Müştərilərin sifariş tarixçələrinə baxa, çatdırılmasını izləyə və ödəniş seçimlərini fərdiləşdirə biləcəkləri profil səhifəsi olmalıdır.



Şəkil 2.14. Yekun forma

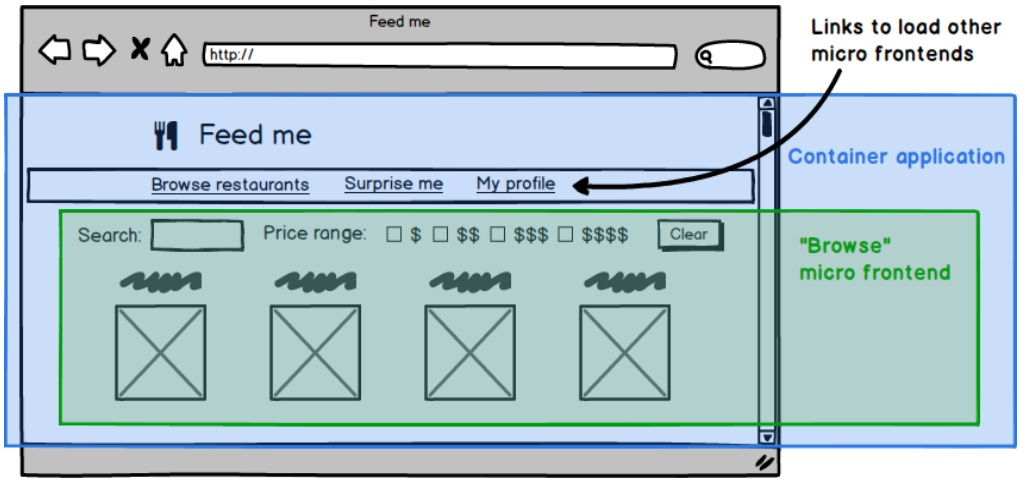
Hər səhifədə kifayət qədər mürəkkəblik var ki, biz asanlıqla hər biri üçün xüsusi komanda yaratmağa haqq qazandıra bilərik və bu komandaların hər biri bütün digər komandalardan asılı olmayaraq öz səhifələrində işləməyi bacarmalıdır. Onlar digər komandalarla münaqişə və ya koordinasiyadan narahat olmadan kodlarını inkişaf etdirməyi, sınaqdan keçirməyi, yerləşdirməyi və saxlamağı bacarmalıdırlar. Ancaq müştərilərimiz hələ də tək, qüsursuz veb sayt görməlidirlər. Bu məqalənin qalan hissəsində nümunə kod və ya ssenarilərə ehtiyacımız olan yerdə bu nümunə tətbiqdən istifadə edəcəyik.

İnteqrasiya yanaşmaları

Yuxarıdakı kifayət qədər boş tərifi nəzərə alsaq, ağlabatan olaraq mikro frontendlər adlandırıla bilən bir çox yanaşma var. Bu bölmədə bəzi nümunələr göstərəcəyik və onların

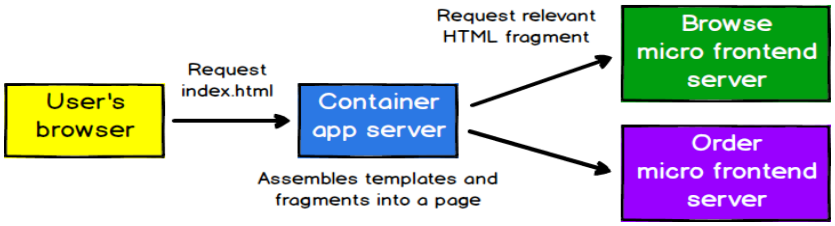
mübadilələrini müzakirə edəcəyik. Bütün yanaşmalarda ortaya çıxan çox təbii bir memarlıq var; Tətbiqdəki hər səhifə üçün adətən mikro ön hissə və tək konteyner tətbiqi var (Şəkil 2.15):

- Başlıqlar və altbilgilər kimi ümumi səhifə elementlərini təqdim edir
- İdentifikasiya və naviqasiya kimi kəsişən problemləri həll edir
- Səhifədə müxtəlif mikro frontendləri bir araya gətirir və hər bir mikro frontendin özünü nə vaxt və harada göstərəcəyini bildirir [Siriwardena and Dias, 2020].



Şəkil 2.15. Mikro ön hissə və tək konteyner

Daha böyük müstəqillik üçün hər bir mikro frontendin göstərilməsi və xidmət göstərilməsi üçün cavabdeh olan ayrıca serveri ola bilər və ön tərəfdəki server digərlərinə sorğu verə bilər. Cavabların ehtiyatlı keşləşdirilməsi ilə bu, gecikməyə təsir etmədən edilə bilər (Şəkil 2.16) .



Şəkil 2.16. Cavabların ehtiyatlı keşləşdirilməsi

Bu nümunə göstərir ki, mikro frontendlər mütləq yeni bir texnika deyil və mürəkkəb olmaq məcburiyyətində deyil. Dizayn qərarlarımızın kod bazalarımızın və

komandalarımızın muxtariyyətinə necə təsir etdiyini nəzərə alsaq, texnologiya yığınımızdan asılı olmayaraq eyni faydaların çoxunu əldə edə bilərik.

2.5 Backend For Frontend nümunəsi

Vebin yaranması və uğuru ilə istifadəçi interfeyslərinin de-fakto çatdırılma üsulu çoxlu müştəri proqramlarından veb-təslim edilən interfeyslərə keçdi; Bu tendensiya həm də ümumilikdə SAAS əsaslı həllərin böyüməsinə imkan verdi. İnternet üzərindən istifadəçi interfeysi təklif etməyin faydaları çox böyük idi; Birincisi, müştəri tərəfi quraşdırmaların dəyəri (əksər hallarda) tamamilə aradan qaldırıldığı üçün yeni funksionallığın çatdırılması dəyəri əhəmiyyətli dərəcədə azaldıldı.

Ancaq bu daha sadə dünya uzun sürmədi, mobil telefonun yaşı tezliklə gəldi. İndi bir problemimiz var idi. Bir və ya bir neçə mobil istifadəçi interfeysi vasitəsilə ifşa etmək istədiyimiz həm masaüstü veb UI, həm də server tərəfi funksionallığımız var idi. Əvvəlcə masa üstü veb interfeysi nəzərə alınmaqla hazırlanmış sistemlə biz tez-tez bu yeni UI növlərinə uyğunlaşma problemi ilə qarşılaşırdıq, çünki artıq masaüstü veb UI və dəstəklənən xidmətlərimiz arasında sıx əlaqə var idi [Kim, Debois, Willis and Humble, 2016].

Çoxsaylı UI növlərinin yerləşdirilməsinin ilk addımı adətən bir server tərəfi API təmin etmək və yeni mobil qarşılıqlı əlaqə növlərini dəstəkləmək üçün zamanla lazım olduqda daha çox funksionallıq əlavə etməkdir.

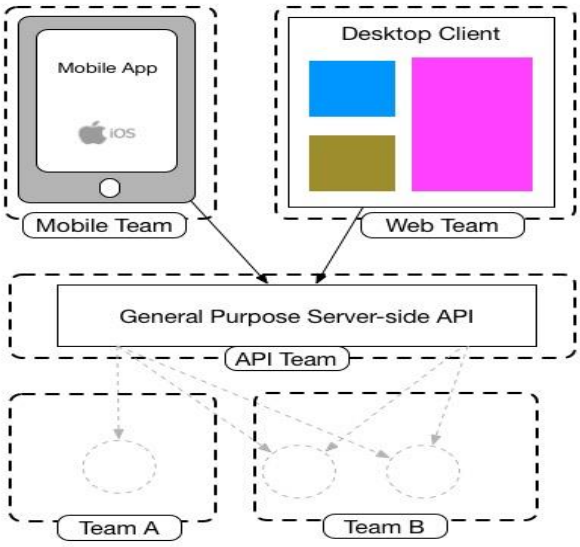
Bu müxtəlif UI-lər eyni və ya çox oxşar növ zənglər etmək istəyirlərsə, bu tip ümumi təyinatlı API-nin uğur qazanması asan ola bilər. Lakin mobil təcrübənin xarakteri tez-tez masaüstü veb təcrübəsindən kəskin şəkildə fərqlənir. Əvvəla, mobil cihazların təqdim etdiyi imkanlar çox fərqlidir. Bizdə daha az ekran sahəsi var, yəni daha az məlumat göstərə bilərik. Server tərəfindəki resurslara çoxlu bağlantıların açılması batareyanın ömrünü azalda və məlumat planlarını məhdudlaşdıra bilər. İkincisi, mobil cihazda təmin etmək istədiyimiz qarşılıqlı əlaqənin xarakteri çox fərqli ola bilər. Tipik bir kərpic və

minaatan pərakəndə satışa nəzər salın. Mən sizə satış üçün əşyaları nəzərdən keçirməyə, onlayn sifariş etməyə və ya masaüstü proqramda mağazada rezervasiya etməyə icazə verə bilərəm. Ancaq mobil telefonda qiymət müqayisəsi aparmaq və ya mağazada olarkən kontekst əsasında təkliflər təqdim etmək üçün barkodları skan etməyə icazə verə bilərəm. Getdikcə daha çox mobil proqramlar hazırladıqca, insanların onlardan çox fərqli istifadə etdiyini başa düşdük, ona görə də çatdırmaq üçün lazım olan funksionallıq da fərqli olacaq.

Beləliklə, praktikada mobil cihazlarımız masaüstü cihazlardan fərqli axtarışlar yerinə yetirmək, daha az axtarış aparmaq və fərqli (və çox güman ki, daha az) məlumatları göstərmək istəyəcəklər. Bu o deməkdir ki, mobil interfeyslərimizi dəstəkləmək üçün API backendimizə əlavə funksionallıq əlavə etməliyik.

Ümumi təyinatlı API backendləri ilə bağlı başqa bir problem, onların, tərifinə görə, birdən çox istifadəçi ilə əlaqəli tətbiqlərə funksionallıq təmin etməsidir. Bu o deməkdir ki, tək API backend yeni yerləşdirməni həyata keçirərkən darboğaza çevrilə bilər, çünki eyni yerləşdirilə bilən üçün çoxlu dəyişikliklərə cəhd edilir.

Ümumi təyinatlı API backendinin çoxsaylı məsuliyyətlər üzərinə götürməsi və buna görə də çox iş tələb etməsi tendensiyası çox vaxt bu kod bazasını idarə etmək üçün xüsusi olaraq komandanın yaradılması ilə nəticələnir. Bu, problemi daha da pisləşdirə bilər, çünki indi qabaqcıl komandalar dəyişiklik etmək üçün ayrıca komanda ilə əlaqə qurmalıdırlar; Bu komanda müxtəlif müştəri komandalarının prioritetlərini balanslaşdırmalı və bir çox alt qruplarla işləməli olacaq. Yeni API-lər mövcud olduqda istifadə edin. Bu nöqtədə iddia etmək olar ki, biz arxitekturamızda heç bir xüsusi biznes sahəsinə diqqət yetirməyən ağıllı bir ara proqram parçası yaratmışıq; Bu, bir çox insanların həssas Xidmət Yönümlü Arxitekturanın necə görünməsi ilə bağlı fikirlərinə ziddir (Şəkil 2.18).



Şəkil 2.18. Xidmət Yönümlü Arxitekturanın

2.6 SAGA nümunəsi

Tranzaksiyaların əhəmiyyəti və onların idarə olunması bizim informasiya sistemlərimizdə vacib bir hissədir. Məlumatların düzgün şəkildə inteqrasiyasını və ardıcıllığını təmin etmək üçün müxtəlif həllər mövcuddur. 2PC (2 Phase Commit) Pattern, məlumatların ardıcıllığının təmin edilməsində istifadə olunan bir şablon dur. Bu şablon, bir əməliyyatın uğurla başa çatmasından sonra növbəti əməliyyatın işə salınması ilə əhəmiyyətli nəticələr əldə edir.

Məsələn, e-ticarət saytında bir sifariş yaratma prosesini düşünək. Bu proses əvvəlcə müştərinin məlumatlarını qəbul edərək başlayır. Müştəri məlumatları daxil edildikdən sonra, sistem sifarişi yaratma prosesinə başlayır. Bu proses iki faza bölünür: 1) Müştəri məlumatlarının qəbul edilməsi və 2) Sifarişin yaradılması. İlk faza başlanıldıqda, məlumatlar verilənlər bazasında uğurla yadda saxlanılır. Yalnız bu faza uğurla başa çatdıqdan sonra, ikinci faza keçə bilərik - sifarişin yaradılmasına. Bu, 2PC Pattern ilə əməliyyatların ardıcıllığını təmin edən bir prinsipdir.

Lakin, bizim modern informasiya sistemlərimiz çoxalmaqla, həm də müxtəlif xidmətlərin ayrı-ayrı olması ilə, 2PC Pattern-ə yaxşı uyğun gəlməyə bilər. Bu zaman Saga Pattern kimi həllərə yönlənə bilərik. Saga Pattern, birbiri ilə bağlı əməliyyatların ardıcıllığını və inteqrasiyasını təmin etmək üçün effektiv bir şəkildə istifadə olunur. Bu şəkildə, hər bir addımın əvvəlki addımın uğurla tamamlanmasından sonra işə salınması təmin edilir. Bu, tranzaksiyanın geri qaytarmağa və ya uğursuzluq hallarında düzəldici tədbirlər görməyə imkan verir.

Bu iki pattern, informasiya sistemlərinin effektiv və düzgün şəkildə işləməsi üçün mövcud olan əhəmiyyətli həllərdir. Hər birinin özünəməxsus üstünlükləri və çətinlikləri var, bu səbəbdən dəyərləndirərək seçilməlidir.

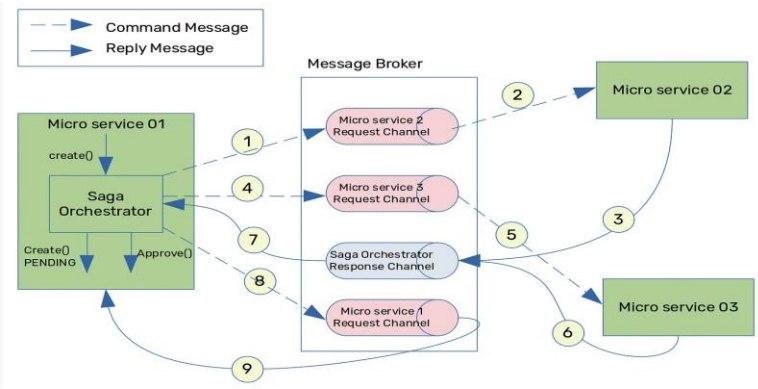
Saga Pattern nədir?

Saga Pattern ilə yaradılmış sistemlərdə hər bir sonrakı addım daxil olan sorğu ilə əvvəlki addımın uğurla tamamlanmasından sonra işə salınır. Tranzaksiyanı geri qaytarmağa və ya hər hansı uğursuzluq halında düzəldici tədbir görməyə imkan verən nümunədir.

SAGA patternin iki forması var:

1. Orkestrə əsaslanan SAGA

Bu yanaşmada, bütün əməliyyatları idarə edən bir Saga orkestratoru var. Bu orkestrator, bütün əməliyyatların ardıcıl olaraq yerinə yetirilməsini təmin edir və digər istehlakçılara (servislərə) nə zaman və nə etməli olduğunu bildirir. Elektron ticarət məsələsində, orkestrator, sifarişi qəbul etmə, ödənişi yoxlama, məhsulu göndərmə kimi əməliyyatları ardıcıllıqla yerinə yetirir. Bu şəkildə, hər bir əməliyyatın uğurla tamamlanmasından sonra növbəti əməliyyat işə salınır. Orkestrator, uğursuzluq hallarında da tədbirlər görmək üçün tənzimləmələr edə bilər.



Şəkil 2.19. Orkestrə əsaslanan SAGA-nın diaqramı

Şəkil 2.19-də hər bir saga iştirakçısı (burada o, mikroservisdir) hadisələri mübadilə edərək bir-biri ilə əlaqə qurur. Bu diaqramda:

- Event Topic 1 Microservice 2 və 3 təqib edir
- Event Topic 2 Microservice 3 təqib edir
- Event Topic 3 Microservice 1 and 2 təqib edir

Hər bir saga iştirakçısı yerli məlumat bazasını yeniləyir və növbəti iştirakçını tetikleyen bir hadisəni yayımlayır. Məsələn, Şəkil 2.19-da mikroservis 01 yerli verilənlər bazası yeniləməsini tamamladıqdan sonra mesaj brokerində Hadisə Mövzu 1-ə bir hadisəni yayımlayır. Tədbir Mövzu 1-ə abunə olan saga iştirakçıları indi mikroservis daxilində digər hərəkətləri yerinə yetirməyə və digər saga iştirakçıları ilə əlaqə yaratmaq üçün digər hadisələri (əgər varsa) dərc etməyə tətiklənirlər. Eynilə, axın saga tamamilə tamamlanana qədər davam edə bilər.

Xoreoqrafiya tipli sagaların sadəlik və boş birləşmə kimi əsas üstünlükləri var. Lakin bu tip sagaların bəzi əhəmiyyətli çatışmazlıqları da var. Bu çatışmazlıqların başlıcaları aşağıdakılardır:

- **Axını anlamaqda çətinlik:** Bu tip sagaların icrasını müəyyən etmək üçün mərkəzi bir yer yoxdur, bu da axını anlamaqda çətinliklər yaradır.
- **Dairəvi asılılıqlar:** Saga iştirakçıları arasında dairəvi asılılıqlar ola bilər, məsələn, bir iştirakçının digərini tətikləməsi və sonra onun əvəzində yenidən birinci iştirakçıya əmr göndərməsi kimi.

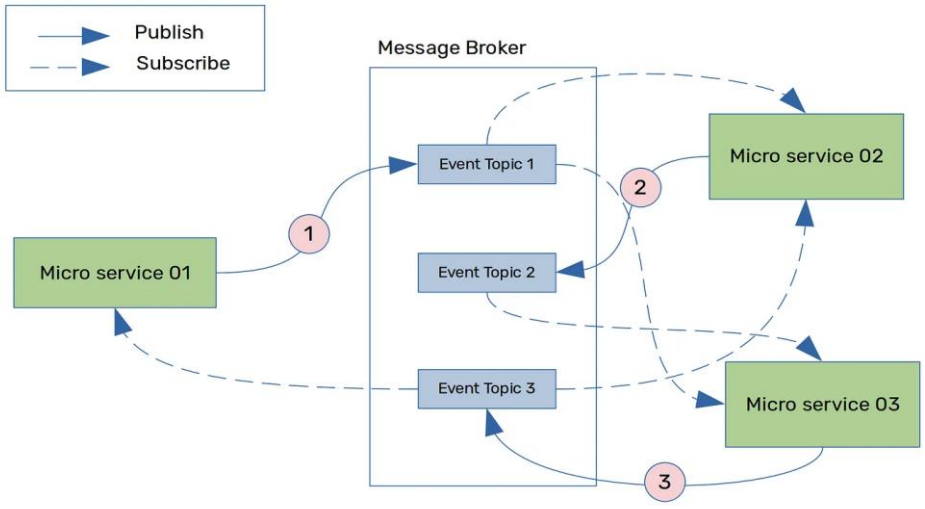
- **Sıx birləşmə riski:** Hər bir hadisəyə abunə olmaq, xidmətlər arasında sıx əlaqə ilə nəticələnə bilər, bu da sistemi mürəkkəbləşdirir.

Xoreoqrafiya tipli sagaların çatışmazlıqlarını qarşılamaq üçün, az xidmətlərlə isə mürəkkəb xidmət quruluşu ilə yaxşı olduğunu qeyd etmək məsləhətdir. Bu, çatışmazlıqların azalmasına və sistemin daha effektiv olmasına kömək edir [Indrasiri and Siriwardena, 2018].

2. Xoreoqrafiyaya əsaslanan SAGA

Xoreoqrafiyaya əsaslanan SAGA, bir sistemdə mərkəzi idarəçi olmadan əməliyyatların ardıcılığını təmin etmək üçün istifadə olunan bir yanaşmadır. Bu yanaşmada hər bir xidmət öz əməliyyatını tamamlayır və bu tamamlanmış əməliyyatı bir hadisə kimi atır. Növbəti xidmət bu hadisəni istehlak edir və öz prosesini davam etdirir. Bu şəkildə, hər bir xidmət digər xidmətlərlə birbaşa əlaqə yaratmaq və əməliyyatların ardıcılığını təmin etmək üçün öz prosesini nəzarət edir.

Xoreoqrafiyaya əsaslanan SAGA yanaşması, sistemdə mərkəzi idarəçi olmadan dinamik və sərbəst bir əməliyyat düsturu təmin edir. Bu, sistemin genişlənməsi və məhdudiyyətlərin azaldılması üçün böyük bir potensial yaradır. Hər bir xidmət öz prosesini nəzarət edərək, sistemin daha qabaqcıl və müstəqil funksionallıqlarını icra etmək üçün güclü bir zəmin yaradır.



Şəkil 2.20. Xoreoqrafiyaya əsaslanan SAGA-nın diaqramı

Şəkil 2.20-də hər bir saga iştirakçısı (burada o, mikroservisdir) hadisələri mübadilə edərək bir-biri ilə əlaqə qurur. Bu diaqramda:

- Fəaliyyət Mövzu 1 Mikroservis 2 və 3-ə abunə olur
- Fəaliyyət Mövzu 2 Microservice 3-ə abunə oldu
- Fəaliyyət Mövzu 3 1 və 2 Mikroservislərə abunə olur

Hər bir saga iştirakçısı yerli məlumat bazasını yeniləyir və növbəti iştirakçını tetikleyen bir hadisəni yayımlayır. Məsələn, Şəkil 20-də mikroservis 01 yerli verilənlər bazası yeniləməsini tamamladıqdan sonra mesaj brokerində Hadisə Mövzu 1-ə bir hadisəni yayımlayır. Tədbir Mövzu 1-ə abunə olan saga iştirakçıları indi mikroservis daxilində digər hərəkətləri yerinə yetirməyə və digər saga iştirakçıları ilə əlaqə yaratmaq üçün digər hadisələri (əgər varsa) dərc etməyə tətiklənirlər. Eynilə, axın saga tamamilə tamamlanana qədər davam edə bilər.

Sadəlik və boş birləşmə xoreoqrafiyalı sagalardan istifadənin əsas üstünlüklərindən bəziləridir. Bununla belə, xoreoqrafik sagaların bəzi əhəmiyyətli çatışmazlıqları da var. Əsas çatışmazlıqlardan bəziləri bunlardır:

Axını anlamaqda çətinlik - Çox vaxt bu tip sagaın icrasını xidmətlər arasında paylayır. Bu üsulda sagaın cərəyanını müəyyən etmək üçün mərkəzi yer yoxdur. Xidmətlər arasında dairəvi asılılıqlar — Saga iştirakçılarının mikroservis 01 -> mikroservis 02 -> mikroservis 01 kimi dairəvi asılılıqları ola bilər. Sıx birləşmə riski -> Onlara təsir edən bütün hadisələrə abunə olmaq, xidmətlər arasında sıx əlaqə ilə nəticələnə bilər.

Yuxarıda göstərilən çatışmazlıqlara və arxitekturanın necə qurulduğuna görə, xoreoqrafik sagaların az xidmətlərlə, lakin mürəkkəb xidmət quruluşu ilə yaxşı olduğunu söyləmək məsləhətdir.

Nəticədə, SAGA və iki fazalı öhdəliyin (2PC) müqayisəsi paylanmış sistemlərdə tranzaksiyaların idarə edilməsinə dair mühüm anlayışları ortaya qoyur. Saga, kompensasiya edən əməliyyatları və qeyri-mərkəzləşdirilmiş yanaşması ilə çeviklik və

miqyaslılıq təklif edir və onu mürəkkəb iş axınları və uzunmüddətli əməliyyatlar üçün uyğun edir. Digər tərəfdən, 2PC güclü ardıcıllıq təmin edir, lakin xüsusilə geniş miqyaslı paylanmış mühitlərdə bloklanma və genişlənmə problemlərindən əziyyət çəkir.

Bu təhlildən əldə edilən əsas məqamlardan biri paylanmış sistemlərin layihələndirilməsi zamanı ardıcıllıq, əlçatanlıq və bölmələrə dözümlülük (CAP teoremi) arasındakı mübadilələrin başa düşülməsinin vacibliyidir. Saga-nın son ardıcıllıq modeli CAP prinsipləri ilə yaxşı uyğunlaşır, sistemlərin mövcud və bölünməyə dözümlü qalmasına imkan verir, eyni zamanda müvəqqəti olaraq güclü ardıcıllığı qurban verir.

Bundan əlavə, əməliyyatların idarə edilməsi üsullarının təkamülü paylanmış sistem arxitekturasının dinamik xarakterini vurğulayır. Texnologiyalar irəlilədikcə və yeni problemlər ortaya çıxdıqca, memarlar və tərtibatçılar dayanıqlığı, miqyaslılığı və performansı təmin etmək üçün öz yanaşmalarını davamlı olaraq qiymətləndirməli və uyğunlaşdırmalıdırlar.

Praktik baxımdan saga və 2PC arasında seçim etmək sistemin xüsusi tələblərindən və məhdudiyyətlərindən asılıdır. Yüksək əlçatanlığın və genişlənmənin əsas olduğu ssenarilər üçün saga cəlbedici seçim kimi ortaya çıxır. Əksinə, ciddi ardıcıllıq tələb edən proqramlar məhdudiyyətlərinə baxmayaraq 2PC-yə meyl edə bilər.

Ümumilikdə, SAGA və 2PC vasitəsilə səyahət paylanmış sistemlərdə ardıcıllıq, genişlənmə və nasazlığa dözümlülük arasındakı mürəkkəb inteqrasiyasını göstərir. Hər bir yanaşmanın güclü tərəflərindən ağıllı şəkildə istifadə etməklə tərtibatçılar müasir paylanmış arxitekturaların mürəkkəbliklərini effektiv şəkildə idarə edə, davamlı və həssas tətbiqləri istifadəçilərə çatdıra bilərlər [Kleppmann, 2017].

Anormallıqlar

Tipik bir sagada üç növ anomaliya var. Bunlar:

1. İtirilmiş Yeniləmələr — Bir epik başqa bir epik tərəfindən edilən yeniləmənin üzərinə yazır.

2. Dirty Reads — Saga başqa bir saga tərəfindən yenilənmə prosesində olan məlumatları oxuyur.

3. Qeyri-səlis/təkrar olunmayan oxumalar — Bir sagaın iki fərqli dəsti eyni məlumatları oxuyur və başqa bir saga yeniləndiyi üçün fərqli nəticələr əldə edir.

Bu üç ssenaridən itirilmiş yeniləmə və çirkli oxunma ssenariləri ən çox yayılmışdır. Anomaliyaları düzəltmək üçün dizaynlarınızda əks tədbirlər həyata keçirilməlidir. Ədəbiyyatda bir çox əks-tədbir yanaşmaları var və mühüm olanlardan bəziləri aşağıdakılardır.

1. Semantik Kilid — Bu, sağa-ın kompensasiya edilə bilən əməliyyatlarının yaratdığı və ya yenilədiyi hər hansı qeyddə bayraq təyin etdiyi proqram səviyyəsində kiliddir (məsələn, Sifarişin yaradılması APPROVAL_PENDING, REVISION_PENDING və s. kimi bayraq statusuna malik ola bilər). Bu bayraq qeydin saxlanmadığını və dəyişmə potensialına malik olduğunu göstərir. Bu, təkrar cəhd və ya kompensasiya hərəkəti ilə həll edilə bilər.

2. Kommutativ Yeniləmələr - Sistemin gələcək yeniləmə əməliyyatlarının kommutativ olmasını təmin etmək üçün layihələndirilməsi (mütəmadi olaraq yenilənmələr). Bu, əsasən itkin yeniləmələri aradan qaldıra bilər.

3. Pessimist Görünüş - Çirkli oxunuşların təsirini minimuma endirmək üçün saga iştirakçılarının/xidmətlərin yenidən sıralanması.

4. Dəyərləri Yenidən Oxuyun - Bu əks tədbir dəyərlərin proses zamanı dəyişmədiyini yenidən yoxlamaq üçün yeniləmədən əvvəl dəyərləri yenidən oxuyur. Bu, itirilmiş yeniləmələri minimuma endirəcəkdir.

5. Dəyərlə - Bu strategiya biznes riskinə əsaslanan paralel mexanizmləri seçəcəkdir. Bu sagalardan istifadə edərək aşağı riskli sorğuları və paylanmış əməliyyatlardan istifadə edərək yüksək riskli sorğuları yerinə yetirməyə kömək edə bilər.

2.7 CQRS nümunəsi

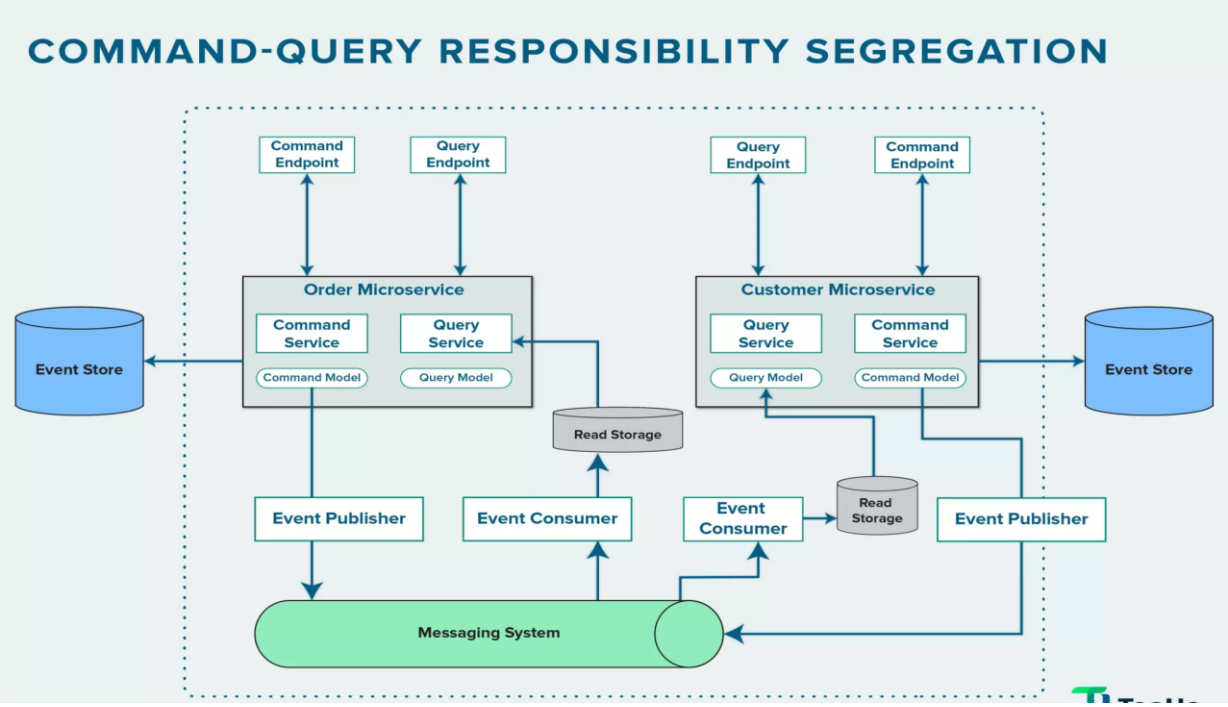
CQRS (Command Query Responsibility Separation), Greg Young tərəfindən təsvir edilmiş bir nümunədir. Bu nümunə, məlumatı oxumaq və yeniləmək üçün fərqli modellərdən istifadə etməyin mümkün olduğunu bildirir. Yəni, məlumatı oxumaq üçün istifadə etdiyiniz modeldən fərqli bir modeldən məlumatı yeniləmək üçün istifadə edə bilərsiniz. Bu yanaşma bəzi hallarda çox faydalı ola bilər, çünki oxuma və yazma əməliyyatlarının fərqli tələblərini nəzərə alaraq sistemin performansını və miqyaslanmasını artırır. Lakin nəzərə almaq lazımdır ki, əksər sistemlər üçün CQRS əlavə mürəkkəblik və risklər gətirir.

İnsanların informasiya sistemi ilə qarşılıqlı əlaqədə istifadə etdikləri əsas yanaşma, onu CRUD (Create, Read, Update, Delete) məlumat anbarı kimi qəbul etməkdir. Yəni, biz yeni qeydlər yarada, mövcud qeydləri oxuya, yeniləyə və silə biləcəyimiz bir qeyd strukturuna sahibik. Ən sadə halda, qarşılıqlı əlaqəmiz bu qeydlərin saxlanması və əldə edilməsi ilə bağlıdır [Woods and Rozanski, 2020].

Ehtiyaclarımız daha mürəkkəbləşdikcə, biz bu modeldən uzaqlaşırıq. Məlumatlara fərqli şəkildə baxmaq istəyə bilərik; bir qeyddə birdən çox qeyd toplaya, müxtəlif yerlərdən məlumatları birləşdirərək virtual qeydlər yarada bilərik. Yeniləmə tərəfində isə, yalnız müəyyən məlumat kombinasiyalarının saxlanmasına imkan verən doğrulama qaydaları tətbiq edə bilərik və ya hətta təqdim etdiyimizdən fərqli olan məlumatların saxlanacağını qərarlaşdıra bilərik.

Bu baş verdikdə, məlumatın çoxsaylı təmsillərini görməyə başlayırıq. İstifadəçilər məlumatla qarşılıqlı əlaqədə olduqda, bu məlumatın müxtəlif təqdimatlarından istifadə edirlər. Tərtibatçılar adətən modelin əsas elementlərini manipulyasiya etmək üçün öz konseptual modellərini qururlar. Domen Modelindən istifadə edirsinizsə, bu, adətən domenin konseptual təmsilidir. Həmçinin, davamlı yaddaşı mümkün qədər konseptual modelə yaxın etməyə çalışırsınız.

Çoxsaylı təmsilçiliyin bu strukturu mürəkkəbləşə bilər, lakin insanlar bunu etdikdə, onu bütün təqdimatlar arasında konseptual inteqrasiya nöqtəsi kimi çıxış edən vahid konseptual təmsilə qədər həll edirlər (Şəkil 2.21).



Şəkil 2.21. CQRS patterni

CQRS-in təqdim etdiyi dəyişiklik, həmin konseptual modeli yeniləmə (Command) və nümayiş etdirmək (Query) üçün ayrıca modellərə bölməkdir. Əsas odur ki, bir çox problemlər üçün, xüsusən də daha mürəkkəb domenlərdə, əmrlər və sorğular üçün eyni konseptual modelə malik olmaq daha mürəkkəb və səmərəsiz modellərə gətirib çıxarır.

Ayrı-ayrı modellər dedikdə, adətən, müxtəlif məntiqi proseslərdə və ola bilsin ki, ayrı-ayrı aparatda işləyən fərqli obyekt modelləri nəzərdə tutulur. Veb nümunəsində, sorğu modelindən istifadə edərək yaradılmış veb səhifəyə baxan istifadəçini görərik. Dəyişiklik baş verəndə isə bu dəyişiklik emal üçün ayrıca komanda modelinə yönləndirilir, nəticədə yenilənmiş vəziyyət sorğu modelinə ötürülür.

Burada mühüm fərqlər üçün yer var. Yaddaşdaxili modellər eyni verilənlər bazasını paylaşa bilər; bu halda verilənlər bazası iki model arasında əlaqə rolunu oynayır. Bununla

belə, onlar sorğu tərəfinin verilənlər bazasını real vaxt rejimində ReportingDatabase-ə çevirərək ayrıca verilənlər bazalarından da istifadə edə bilərlər. Bu halda iki model və ya verilənlər bazası arasında əlaqə mexanizmi olmalıdır.

İki model ayrı-ayrı obyekt modelləri olmaya bilər, əksinə, eyni obyektlər əlaqəli verilənlər bazasındakı görünüşlərə bənzər əmr tərəfi və sorğu tərəfi üçün fərqli interfeyslərə malik ola bilərlər. Lakin adətən CQRS-i eşidəndə onların açıq-aydın ayrı modellər olduğunu görürük.

CQRS təbii olaraq bəzi digər memarlıq modelləri ilə uyğun gəlir:

- **CRUD Əlaqəsi:** CRUD vasitəsilə qarşılıqlı əlaqədə olduğumuz tək nümayəndəlikdən uzaqlaşdıqca, asanlıqla tapşırıq əsaslı istifadəçi interfeysinə keçə bilərik.
- **Hadisələrə Əsaslanan Proqramlaşdırma:** CQRS hadisələrə əsaslanan proqramlaşdırma modelləri ilə yaxşı uyğunlaşır. CQRS sisteminin Event Collaboration ilə əlaqə saxlayan ayrı-ayrı xidmətlərə bölündüyünü görmək adi haldır. Bu, bu xidmətlərə Hadisə Mənbəsindən asanlıqla yararlanmağa imkan verir.
- **Son Ardıcıllıq:** Ayrı-ayrı modellərə malik olmaq bu modelləri ardıcıl saxlamağın nə qədər çətin olduğuna dair suallar doğurur ki, bu da son ardıcıllığın istifadə olunma ehtimalını artırır.
- **EagerReadDerivation:** Bir çox domenlər üçün yeniləmə zamanı çoxlu məntiq tələb olunur, ona görə də sorğu tərəfi modellərinizi sadələşdirmək üçün EagerReadDerivation-dan istifadə etməyin mənası ola bilər.
- **EventPosters:** Yazma modeli bütün yeniləmələr üçün hadisələr yaradırsa, siz oxunmuş modelləri EventPosters kimi konfiqurasiya edə, onların MemoryImages olmasına icazə verə və beləliklə, verilənlər bazası ilə çoxlu qarşılıqlı əlaqədən qaça bilərsiniz.
- **Domain Driven Design (DDD):** CQRS, Domain Driven Design-dan da faydalanan kompleks domenlərə uyğun gəlir. Bu, xüsusən mürəkkəb biznes məntiqinin və genişlənən sistemlərin idarə olunması üçün əhəmiyyətlidir.

Nə vaxt istifadə edilməlidir?

CQRS, informasiya sistemi üzrə çox əhəmiyyətli bir addımdır, ancaq hər bir mühit və tələblər üçün uyğun deyil. Çoğu zaman, CRUD əsaslı informasiya sistemləri üçün CQRS tətbiq etmək əhəmiyyətli deyil, ona görə ki, bu sistemlər çox sadədir və CRUD modelinə daha uyğun gəlir. CQRS-in qazanclığı qarşısında, onun mürəkkəblikdə əlavə mürəkkəblik yaratma ehtimalını göz önündə tutmalı və yalnız təcili bir ehtiyacın varsa, ona müraciət etməlidirsiniz.

CQRS, əgər uyğun şərait varsa, məhsuldarlığı artırmaqda böyük rol oynayır. Məsələn, yüksək performanslı tətbiqlər üçün CQRS yararlıdır, çünki yükü oxumaq və yazmaqdan ayrılmağa və hər birini müstəqil şəkildə ölçməyə imkan verir. Lakin unutmamalısınız ki, domeniniz uyğun deyilsə, CQRS tətbiq etmək mürəkkəblikləri artıra bilər və sistemi risklərə məruz qalda bilər.

CQRS-i məsuliyyətli şəkildə istifadə etmək üçün, onun uyğun olduğu məsələləri aydın şəkildə müəyyən etməlisiniz. CQRS, yalnız sistemin müəyyən hissələrində, məsələn, DDD dilində BoundedContext kimi məhdud sahələrdə tətbiq edilməlidir. Bu, hər bir məhdud kontekstin öz qərarlarını qəbul etməsi üçün tələb olunan zehni istinad nöqtəsini təmin edir.

Bu addımı atmaq istədiyinizdə, uyğunluq və tələblərini yaxşı qiymətləndirin. Əgər informasiya sisteminiz oxumaq və yazmaq arasında böyük fərq görürsə, CQRS sizin üçün faydalı ola bilər. Lakin əgər domeniniz bu yanaşmaya uyğun deyilsə, digər yolları düşünmək daha məqsədəuyğundur.

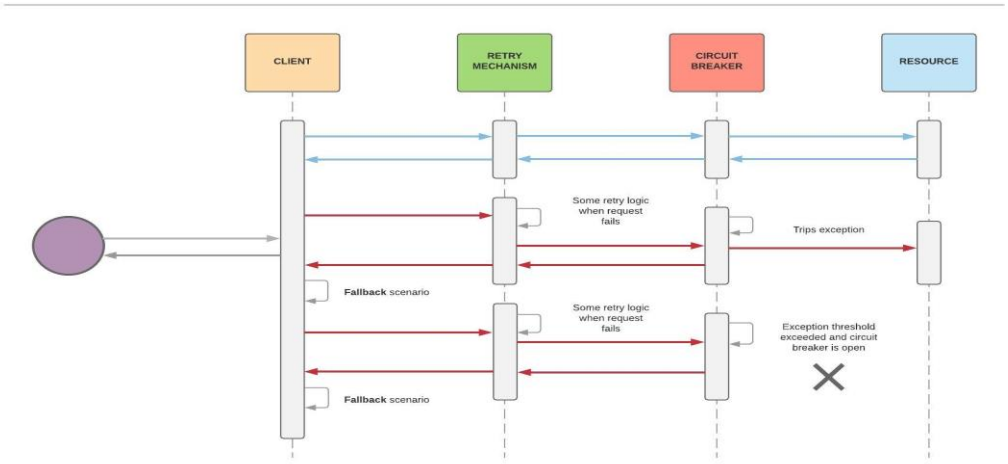
Nəzərə alın ki, CQRS-i dərhal tətbiq etməzdən əvvəl dəqiqliklə qiymətləndirmək əhəmiyyətlidir. Əgər sisteminizin tələbləri və məhdudiyyətləri CQRS ilə uyğun gəlirsə, bu istiqaməti izləmək məqsədəuyğun ola bilər. Ancaq, uyğunluq olmadıqda və ya dəqiq qiymətləndirilmədikdə, CQRS məhsuldarlığı azaldaraq və riskləri artıraraq problemlər yarada bilər.

III FƏSİL. MIKROSERVISLƏRDƏ DAVAMLILIQ

3.1 Mikroservislər Arxitekturasında Dayanıqlılığa Giriş

Müasir proqram təminatının inkişafı dinamik mənzərəsində mikroservislər arxitekturası, güclü, miqyaslana bilən və çevik tətbiqlər yaratmaq üçün güclü bir paradigma kimi ortaya çıxıb. Monolitik arxitekturaların əksinə, burada bir səhv bütün sistemi çökdürə bilər, mikroservislər tətbiqi zəif əlaqəli, müstəqil şəkildə yerləşdirilə bilən xidmətlərə parçalayır. Hər bir xidmət spesifik bir funksiyanı yerinə yetirir və yaxşı təyin olunmuş API-lər vasitəsilə digər xidmətlərlə əlaqə qurur. Bu yanaşma bir çox üstünlüklər təklif etsə də, xüsusi olaraq dayanıqlılıq baxımından unikal problemlər də ortaya çıxarır [Davis, 2019].

Mikroservislərdə dayanıqlılıq, sistemin nasazlıqlara qarşı dayanıqlı qalma və bərpa olunma qabiliyyətinə aiddir. Mikroservislərin dağıdılmış təbiətini nəzərə alsaq, burada çoxsaylı xidmətlər bir-birinə asılı olduğu halda, dayanıqlılığı təmin etmək vacibdir. Dayanıqlı mikroservislər arxitekturası, nasazlıqları zərif şəkildə idarə etmək üçün dizayn olunur, beləliklə, bəzi komponentlər işə yaramadıqda belə sistem işlək vəziyyətdə qalır. Bu giriş, mikroservislərdə dayanıqlılığı artıran əsas nümunələri və təcrübələri araşdırır (Şəkil 3.1).



Şəkil 3.1. Dayanıqlıq patternləri

Dayanıqlılıq Üçün Nümunələr

Circuit Breaker (Dövrə Kəsici) Nümunəsi:

Dövrə Kəsici nümunəsi, elektrik dövrə kəsicilərindən ilham alır və nasazlıqları aşkarlamaq və onların təkrarlanmasının qarşısını almaq üçün istifadə olunur. Bir xidmət digər xidmətə çağırış etdikdə, dövrə kəsici cavabları monitorinq edir. Əgər nasazlıqların sayı müəyyən bir həddi aşarsa, dövrə kəsici açılır və uğursuz olan xidmətə edilən sonrakı çağırışlar kəsici tərəfindən qarşılanır, adətən əvvəlcədən təyin olunmuş ehtiyat cavab və ya xəta mesajı qaytarılır. Bu, nasazlıqların yayılmasının qarşısını alır və uğursuz xidmətə bərpa olunmaq üçün vaxt verir.

Retry (Yenidən Cəhd) Nümunəsi: Yenidən Cəhd nümunəsi, uğursuz olan əməliyyatı müəyyən edilmiş sayda təkrarlamaqdan ibarətdir. Bu nümunə, xüsusilə müvəqqəti nasazlıqlar, məsələn, müvəqqəti şəbəkə problemləri və ya anlıq xidmət əlçatmazlığı üçün faydalıdır. Eksponensial geriləmə tətbiq etməklə, yəni təkrarlar arasında zamanın eksponensial şəkildə artması ilə, sistemlər uğursuz xidmətlərə yükü azaldaraq bərpa olunma şansını artırır.

Bulkhead (Dalğaqıran) Nümunəsi: Dalğaqıranlar, bir sistemin ayrı-ayrı bölmələrə bölünməsini təmin edir ki, bir hissədəki nasazlıq digər hissələri təsir etməsin. Mikroservislər kontekstində, bu, xidmətlərin və ya resursların izolyasiya olunması deməkdir ki, əgər bir xidmət və ya resurs hovuzu uğursuz olarsa, bu nasazlıq sistemin digər hissələrinə yayılmasın. Bu nümunə, nasazlıqların təsirini məhdudlaşdırmaqla dayanıqlılığı artırır.

Fallback (Ehtiyat) Nümunəsi: Ehtiyat nümunəsi, xidmət çağırışı uğursuz olduqda alternativ cavab və ya tədbir təklif edir. Bu, keşlənmiş məlumatın qaytarılması, standart bir dəyərdən istifadə edilməsi və ya sorğunun alternativ xidmətə yönəldilməsi ilə həyata keçirilə bilər. Ehtiyat mexanizmləri, bəzi xidmətlər əlçatan olmadıqda belə sistemin minimum funksionallığını təmin edir.

Dayanıqlılığı Artıran Təcrübələr

- **Sağlamlıq Monitorinqi və Öz-Özünə Sağalma:** Xidmətlərin davamlı sağlamlıq monitorinqi, problemlərin vaxtında aşkarlanması üçün vacibdir. Prometheus və Grafana kimi alətlər real vaxtda monitorinq və xəbərdarlıq üçün istifadə oluna bilər. Bundan əlavə, avtomatik yenidən başlatma və miqyaslandırma kimi öz-özünə sağalma mexanizmlərinin tətbiqi, nasazlıqlardan sürətlə bərpa olunmağa kömək edir.
- **Mərkəzləşdirilməmiş Məlumat İdarəçiliyi:** Hər bir mikrosistemin öz məlumatlarını idarə etməsi (poliglot persistensiya) bir nasazlıq nöqtəsinin riskini azaldır. Bu mərkəzləşdirilməmiş yanaşma, bir xidmətin verilənlər bazasının nasazlığının bütün sistemi çökməsinin qarşısını alır.
- **Hadisə Mərkəzli Arxitektura:** Mesaj brokerlər (məsələn, Kafka və ya RabbitMQ) istifadə edərək mikroservislər arasında hadisə mərkəzli əlaqə, xidmətlərin əlaqələndirilməsini artırır. Asinxron mesajlaşma, xidmətlərin zəif bağlı qalmasına və tələb artımını idarə etməyə imkan verir.
- **Xaos Mühəndisliyi:** Xaos mühəndisliyi, sistemin dayanıqlığını test etmək üçün qəsdən nasazlıqların yaradılmasını nəzərdə tutur. Chaos Monkey kimi alətlər müxtəlif nasazlıq ssenarilərini simulyasiya etməyə kömək edir, beləliklə, inkişafçılar real iş şəraitində ortaya çıxmadan sistemdəki zəif nöqtələri aşkar edə və aradan qaldıra bilərlər [Ford, Richard, Sadalage and Dehghani, 2022].

Nəticədə, dayanıqlılıq mikroservislər arxitekturasının əsas daşlarından biridir.

Dövrə Kəsici, Yenidən Cəhd, Dalğakıran və Ehtiyat kimi nümunələri tətbiq etməklə və sağlamlıq monitorinqi, mərkəzləşdirilməmiş məlumat idarəçiliyi, hadisə mərkəzli arxitektura və xaos mühəndisliyi kimi ən yaxşı təcrübələri mənimsəməklə, inkişafçılar nasazlıqlara qarşı dayanıqlı və onlardan zərif şəkildə bərpa oluna bilən sistemlər qura bilərlər. Mikroservislərdə dayanıqlılığı təmin etmək, yalnız nasazlıqların qarşısını almaq deyil, həm də sistemlərin zərbəyə davamlı, uyğunlaşa bilən və əlverişsiz şəraitlərdə funksionallığını qoruyan bir şəkildə dizayn edilməsi deməkdir.

3.2 Mikroservislər Arxitekturasında Sədd Keçirici (Circuit Breaker) Nümunəsi

Müasir proqram təminatı inkişafında mikroservislər arxitekturası sistemi davamlı və etibarlı etmək üçün əsas nümunələrdən biridir. Bu nümunə elektrik şəbəkələrindəki sədd keçiricilərdən ilham alınıb və qarşılıqlı əlaqədə olan xidmətlər arasında yaranan nasazlıqların qabağını almağa kömək edir.

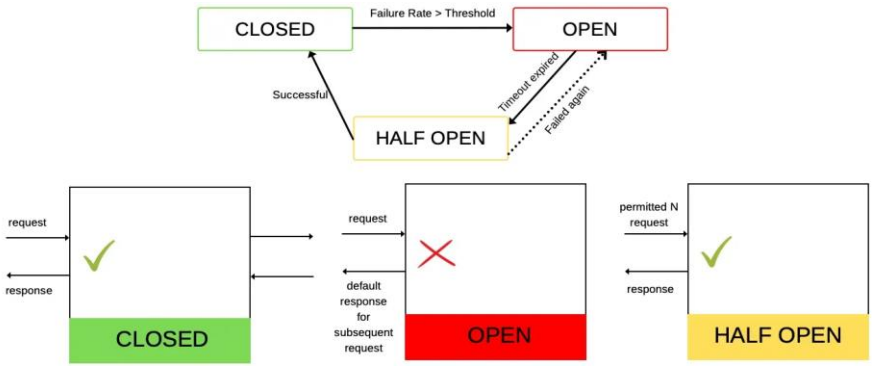
Sədd Keçirici (Circuit Breaker) Nümunəsinin Anlayışı

Sədd Keçirici nümunəsi mikroservislər arasında qarşılıqlı əlaqəni qorumaq üçün işləyir. Bu nümunə xidmətlər arasındakı kommunikasiya zamanı nasazlıqları aşkarlayır və xidmətin sıradan çıxması ehtimalını müəyyən edir. Sədd keçirici üç vəziyyətdə ola bilər (Şəkil 3.2):

1. **Qapalı (Closed):** Bu vəziyyətdə sədd keçirici bütün sorğuların xidmətə keçməsinə icazə verir. Əgər sorğular uğurlu olarsa, sədd keçirici qapalı qalır. Lakin sorğular uğursuz olmağa başlayanda, sədd keçirici bu nasazlıqları izləyir.

2. **Açıq (Open):** Uğursuzluq sayı müəyyən edilmiş həddi keçdikdə, sədd keçirici açıq vəziyyətə keçir. Bu vəziyyətdə bütün sorğular dərhal uğursuz kimi qəbul edilir və xidmətə göndərilmir. Bu, nasaz xidmətin əlavə yüklənməsinin qarşısını alır və xidmətə bərpa olmaq üçün vaxt verir.

3. **Yarı Açıq (Half-Open):** Müəyyən bir müddət sonra, sədd keçirici yarı açıq vəziyyətə keçir və məhdud sayda test sorğusuna icazə verir. Əgər bu sorğular uğurlu olarsa, sədd keçirici yenidən qapalı vəziyyətə keçir. Əgər uğursuz olarsa, sədd keçirici yenidən açıq vəziyyətə keçir.



Şəkil 3.2. Circuit Breaker

Bu vəziyyət idarəçiliyi sistemin nasazlıqları daha asan idarə etməsinə və geniş yayılmasının qarşısını almağa kömək edir [Bellemare, 2020].

Sədd Keçirici Nümunəsinin Tətbiqi

Sədd Keçirici nümunəsini tətbiq etmək bir neçə əsas addımı əhatə edir:

- **Monitorinq və Həddin Müəyyənləşdirilməsi:** Sədd keçirici uğurlu və uğursuz sorğuları izləməlidir. Bu, müəyyən bir zaman pəncərəsində icazə verilən uğursuzluq sayını təyin etməyi tələb edir. Hədlər xidmətin tarixi performansına və qəbul edilən səhv dərəcələrinə əsasən təyin olunmalıdır.
- **Vəziyyətin İdarə Edilməsi:** Sədd keçirici öz cari vəziyyətini (qapalı, açıq və ya yarı açıq) saxlamalı və bu vəziyyətlər arasında uğurlu və ya uğursuz sorğulara əsasən keçid etməlidir. Bu, uğurlu və uğursuz sorğular üçün sayğaclar və açıq və yarı açıq vəziyyətlər üçün taymerlər saxlamağı tələb edir.
- **Ehtiyat Mexanizmi:** Sədd keçirici açıq vəziyyətdə olduqda, ehtiyat mexanizmi işə düşməlidir. Bu, önbelleklenmiş cavabı qaytarmaq, standart bir cavab təqdim etmək və ya sorğunu alternativ bir xidmətə yönləndirmək ola bilər. Ehtiyat mexanizmi sistemin minimal funksionallığını davam etdirməsini təmin edir.
- **Monitorinq və Qeydiyyat:** Sədd keçiricinin fəaliyyətlərinin şəffaf və izah edilə bilən olması üçün bu fəaliyyətlər qeydiyyatdan keçməlidir. Monitorinq vasitələri vəziyyət keçidlərinə və nasazlıqların tezliyinə real vaxtda baxış imkanı verir, bu da proaktiv problemlərin həllinə imkan yaradır.

Sədd Keçirici Nümunəsinin Faydaları

Sədd Keçirici nümunəsi mikroservislər arxitekturasında bir neçə fayda təqdim edir:

- **Yüksək Davamlılıq:** Sədd Keçirici nümunəsi geniş yayılmaların qarşısını almaqla sistemin ümumi davamlılığını artırır. Bu, bir xidmətin nasazlığının bütün sistemi sarsıtmamasını təmin edir.
- **Daha Sürətli Bərpa:** Sədd Keçirici nümunəsi nasaz xidmətə sorğuları müvəqqəti olaraq dayandırmaqla, xidmətin bərpa olunması üçün vaxt verir. Bu, daha sürətli bərpa vaxtlarına və daha stabil xidmət performansına gətirib çıxarır.
- **Resursların Qorunması:** Sədd Keçirici nümunəsi sistem resurslarını qorumağa kömək edir. Bir xidmət nasaz olduqda, sorğuların davamlı olaraq göndərilməsi xidmətin daha çox yük altında qalmasına səbəb ola bilər. Sədd keçirici bu yükün qarşısını alır.
- **Yüksək İstifadəçi Təcrübəsi:** Ehtiyat mexanizmləri təmin etməklə, Sədd Keçirici nümunəsi istifadəçilərin minimal pozulmalar yaşamasını təmin edir. Xidmət nasaz olsa belə, sistem alternativ cavablar təqdim edə bilir [Hohpe, 2020].

Real Dünyada Tətbiqi

Real dünyada, Sədd Keçirici nümunəsi xarici sistemlərdən asılı olan xidmətlərdə geniş istifadə olunur, məsələn:

- **E-ticarət Platformaları:** E-ticarət platformalarında, ödəniş xidməti problemlər yaşasa, sədd keçirici ödəniş prosesinin tamamilə uğursuz olmasının qarşısını almaq üçün alternativ ödəniş üsulları təklif edə bilər.
- **Media Axın Xidmətləri:** Media axın xidmətlərində, əgər tövsiyə xidməti sıradan çıxsa, sədd keçirici fərdiləşdirilmiş tövsiyələr əvəzinə populyar və ya trend olan məzmunu göstərə bilər.
- **Maliyyə Sistemləri:** Maliyyə sistemlərində, əgər valyuta çevrilməsi xidməti sıradan çıxsa, sədd keçirici əməliyyatların hələ də işlənməsi üçün önbelleklenmiş çevrilmə dərəcələrini təqdim edə bilər.

Nəticə olaraq sədd Keçirici nümunəsi davamlı mikroservislər arxitekturasının əsas komponentidir. Xidmət qarşılıqlı əlaqələrini izləməklə və nasazlıqları zərərsiz idarə

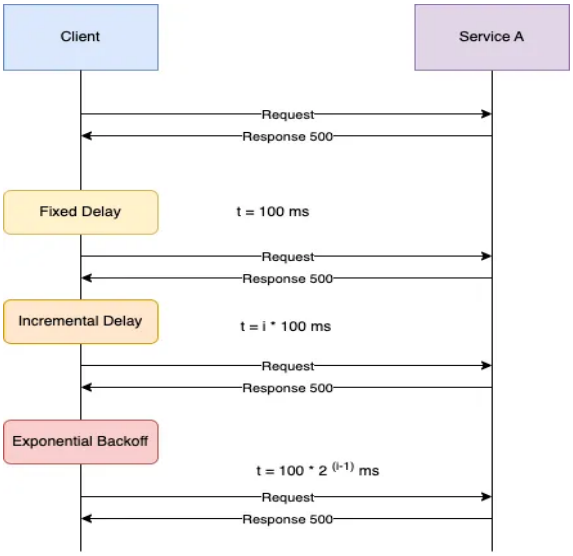
etməklə, bu nümunə geniş yayılmaların qarşısını alır və sistemin dayanıqlığını artırır. Sədd Keçirici nümunəsinin tətbiqi uyğun hədlərin təyin edilməsini, vəziyyət keçidlərinin idarə olunmasını və ehtiyat mexanizmlərinin təmin edilməsini tələb edir. Bu nümunənin üstünlükləri, o cümlədən yüksək davamlılıq, daha sürətli bərpa, resursların qorunması və yüksək istifadəçi təcrübəsi, hər hansı bir möhkəm mikroservislər tətbiqi üçün həyati əhəmiyyət kəsb edir. Mikroservislər arxitekturası inkişaf etdikcə, Sədd Keçirici nümunəsi etibarlı və davamlı sistemlərin təmin edilməsi üçün əsas vasitə olaraq qalacaq [Newman, 2015].

3.3 Mikroservislər Arxitekturasında Retry (Təkrar) Nümunəsi

Mikroservislər arxitekturası, böyük və mürəkkəb tətbiqləri kiçik, müstəqil xidmətlərə bölərək onları daha asan idarə edilə və miqyaslana bilən hala gətirir. Lakin bu xidmətlər arasında qarşılıqlı əlaqə zamanı nasazlıqlar meydana gələ bilər. Bu nasazlıqların öhdəsindən gəlmək və sistemin davamlılığını təmin etmək üçün müxtəlif dizayn nümunələri istifadə olunur. Bu nümunələrdən biri Retry (Təkrar) nümunəsidir. Retry nümunəsi, xüsusilə müvəqqəti xətalar və keçici nasazlıqlar zamanı xidmətlərin daha etibarlı və dayanıqlı olmasına kömək edir.

Retry Nümunəsinin Anlayışı

Retry nümunəsi, müvəqqəti nasazlıqların öhdəsindən gəlmək üçün müəyyən bir sayda uğursuz cəhdin ardınca təkrar cəhdlər edilməsini təmin edir. Bu nümunə, şəbəkə problemləri, qısa müddətli xidmət dayanmaları və ya başqa müvəqqəti problemlər səbəbilə meydana gələn uğursuzluqları aradan qaldırmaq üçün nəzərdə tutulub(Şəkil 3.3).



Şəkil 3.3. Retry mexanizması

Retry nümunəsinin əsas prinsipi, uğursuz sorğuların müəyyən bir sayda təkrar edilməsi və hər təkrardan sonra artan gözləmə vaxtlarının (eksponensial backoff) tətbiq edilməsidir. Bu yanaşma, xidmətlər arasında yükü tarazlayır və xidmətin bərpa olunması üçün kifayət qədər vaxt təmin edir [Newman, 2019].

Retry Nümunəsinin Tətbiqi - Retry nümunəsini tətbiq etmək üçün bir neçə əsas addım var:

- **Uğursuzluqların Aşkarlanması:** Uğursuz sorğuların aşkarlanması üçün xidmətlər arasında göndərilən sorğuların cavabları izlənməlidir. Bu cavablar, sorğunun uğurlu və ya uğursuz olduğunu göstərən kodlar və ya mesajlar olmalıdır.
- **Təkrar Sayının və Gözləmə Vaxtının Müəyyən Edilməsi:** Təkrar cəhdlərinin sayı və hər təkrardan sonra gözləmə vaxtı müəyyən edilməlidir. Gözləmə vaxtı adətən eksponensial backoff əsasında təyin olunur, yəni hər uğursuz təkrardan sonra gözləmə vaxtı eksponensial olaraq artır. Məsələn, ilk təkrar cəhdi üçün 1 saniyə, ikinci üçün 2 saniyə, üçüncü üçün 4 saniyə və s.
- **Təkrar Mexanizminin İdarə Edilməsi:** Təkrar mexanizmi uğursuz sorğuları yenidən göndərməlidir. Əgər müəyyən edilmiş sayda təkrar cəhdindən sonra sorğu hələ

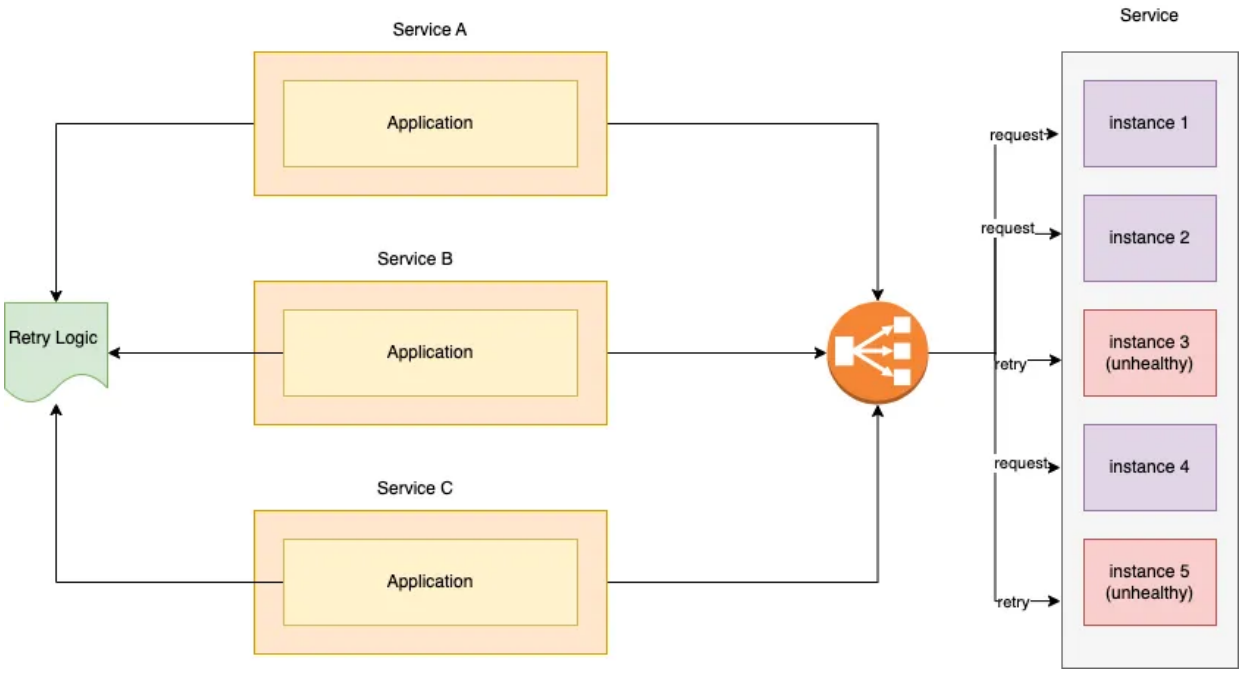
də uğursuz olursa, sorğu tamamilə uğursuz hesab edilməli və uyğun ehtiyat mexanizmi işə düşməlidir.

- **Loqlaşdırma və Monitorinq:** Təkrar cəhdlərinin fəaliyyətləri və uğursuzluqları loqlaşdırılmalıdır. Bu, sistemin davamlılığını izləmək və potensial problemləri müəyyən etmək üçün vacibdir [Richards and Ford, 2020].

Retry Nümunəsinin Faydaları

Retry nümunəsi mikroservislər arxitekturasında bir neçə əsas fayda təqdim edir (Şəkil 3.4):

- **Müvəqqəti Xətaların Aradan Qaldırılması:** Retry nümunəsi, müvəqqəti şəbəkə problemləri və ya qısa müddətli xidmət dayanmaları kimi müvəqqəti xətaları aradan qaldırmağa kömək edir. Bu, xidmətlərin davamlı işləməsini təmin edir və istifadəçilərin minimal pozulmalar yaşamasına imkan yaradır.
- **Yük Tarazlığı:** Eksponensial backoff ilə təkrar cəhdləri tətbiq etməklə, Retry nümunəsi xidmətlər arasındakı yükü tarazlayır. Bu, xidmətlərin bərpa olunmasına imkan verir və nasazlıqların daha da pisləşməsinin qarşısını alır.



Şəkil 3.4 Retry nümunəsi

Real Dünyada Tətbiqi

Retry nümunəsi, real dünyada müxtəlif sahələrdə geniş istifadə olunur. Məsələn:

- **E-ticarət Platformaları:** E-ticarət platformalarında, ödəniş xidmətləri və ya inventar idarəetmə sistemləri kimi xidmətlərdə müvəqqəti nasazlıqlar yaranarsa, Retry nümunəsi bu sorğuların təkrar cəhd edilməsini təmin edir. Bu, müvəqqəti problemlər səbəbilə satışların itirilməsinin qarşısını alır.
- **Media Axın Xidmətləri:** Media axın xidmətlərində, kontent çatdırılma şəbəkələrində (CDN) və ya məlumat bazası xidmətlərində müvəqqəti nasazlıqlar yaranarsa, Retry nümunəsi bu sorğuların təkrar cəhd edilməsini təmin edir. Bu, istifadəçilərin axını dayanmadan izləməsinə imkan verir.
- **Bank və Maliyyə Xidmətləri:** Bank və maliyyə xidmətlərində, ödənişlərin işlənməsi və ya valyuta çevrilməsi kimi əməliyyatlarda müvəqqəti nasazlıqlar yaranarsa, Retry nümunəsi bu əməliyyatların təkrar cəhd edilməsini təmin edir. Bu, müştərilərin maliyyə əməliyyatlarının problemsiz yerinə yetirilməsinə kömək edir.

Retry Nümunəsinin Məhdudiyyətləri

Retry nümunəsi, müvəqqəti xətaların aradan qaldırılmasında effektiv olsa da, müəyyən məhdudiyyətlərə malikdir:

- **Sonsuz Təkrar Döngələri:** Təkrar cəhdlərin düzgün idarə edilməməsi sonsuz təkrar döngələrinə səbəb ola bilər. Bu, xidmətlər arasında əlavə yük yarada bilər və sistemin performansını pisləşdirə bilər. Bu problemin qarşısını almaq üçün təkrar cəhdlərinin maksimum sayı və hər təkrar cəhdi üçün gözləmə vaxtı düzgün təyin edilməlidir.
- **Resursların İstifadəsi:** Hər bir təkrar cəhdi əlavə resurs tələb edir. Bu, xüsusilə böyük həcmdə sorğular zamanı sistem resurslarının effektiv istifadəsini təmin etməyi çətinləşdirə bilər. Resursların optimal istifadəsi üçün təkrar cəhdlərinin sayı və gözləmə vaxtı diqqətlə planlaşdırılmalıdır.
- **Qeyri-müvəqqəti Xətalar:** Retry nümunəsi, müvəqqəti xətaları aradan qaldırmaq üçün effektivdir, lakin qeyri-müvəqqəti xətalar və ya əsas problemlər qarşısında təsirli

deyil. Bu hallarda, sistemin ümumi arxitekturası və xidmətlərin dayanıqlılığı nəzərdən keçirilməlidir.

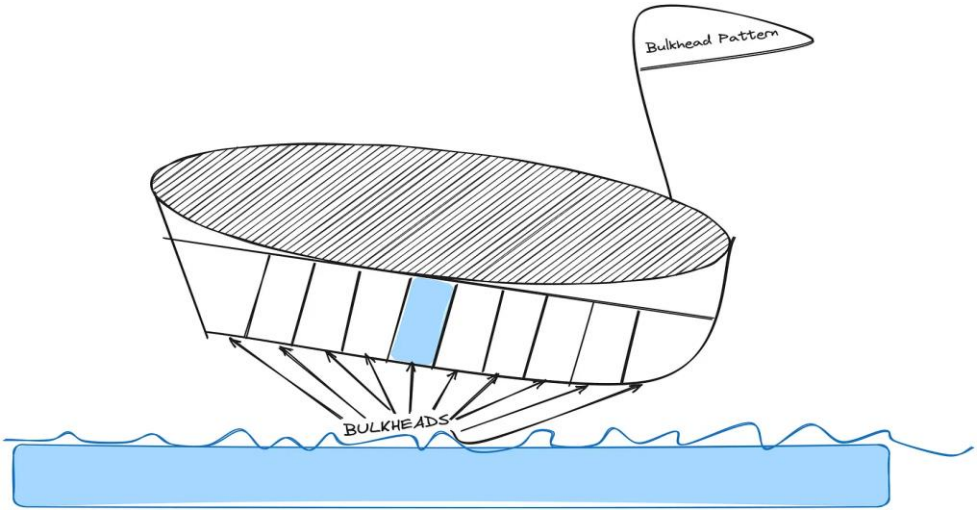
Nəticə olaraq Retry nümunəsi, mikroservislər arxitekturasında müvəqqəti xətaların aradan qaldırılması və xidmətlərin davamlılığının təmin edilməsi üçün mühüm bir dizayn nümunəsidir. Bu nümunə, müvəqqəti problemlərin öhdəsindən gəlmək və xidmətlərin daha etibarlı olmasını təmin etmək üçün istifadə olunur. Retry nümunəsinin düzgün tətbiqi, müvəqqəti nasazlıqların təsirini minimuma endirir və sistemin ümumi dayanıqlığını artırır. Hər hansı bir mikroservislər arxitekturasının etibarlılığı və davamlılığı üçün Retry nümunəsi vacib bir komponentdir. Bu nümunənin effektiv istifadəsi, müvəqqəti problemlərin öhdəsindən gəlmək və istifadəçilərin daha yaxşı bir təcrübə yaşamasını təmin etmək üçün əsas vasitələrdən biridir.[Fowler, 2017]

3.4 Mikroservislər Arxitekturasında Bulkhead Nümunəsi

Mikroservislər arxitekturası, mürəkkəb tətbiqlərin daha kiçik, müstəqil xidmətlərə bölünməsini təmin edir. Bu yanaşma, tətbiqlərin miqyaslanmasını və idarə edilməsini asanlaşdırır, lakin xidmətlər arasında qarşılıqlı asılılıq və nasazlıqların yayılması kimi problemləri də özündə gətirir. Bu problemlərin öhdəsindən gəlmək üçün müxtəlif dizayn nümunələri tətbiq olunur. Bulkhead nümunəsi, xidmətlərin bir-birindən izolyasiya olunmasını təmin etməklə, nasazlıqların təsirinin məhdudlaşdırılması və sistemin davamlılığının artırılması üçün istifadə olunur.

Bulkhead Nümunəsinin Anlayışı

Bulkhead nümunəsi, gəmi dizaynında istifadə olunan alyans divarlarından ilhamlanıb. Gəmilərdə alyans divarları, suyun bir bölmədən digərinə keçməsinin qarşısını alaraq gəminin batmasını əngəlləyir. Eyni prinsip mikroservislər arxitekturasında da tətbiq olunur: sistemin bir hissəsində yaranan nasazlığın digər hissələrə yayılmasının qarşısını almaq üçün xidmətlər izolyasiya edilir. Bu yanaşma, xidmətlərin müstəqil şəkildə işləməsini və nasazlıqların təsirinin məhdudlaşdırılmasını təmin edir (Şəkil 3.5).



Şəkil 3.5. Bulkhead nümunəsi

Bulkhead Nümunəsinin Tətbiqi

Bulkhead nümunəsini tətbiq etmək üçün bir neçə əsas addım var:

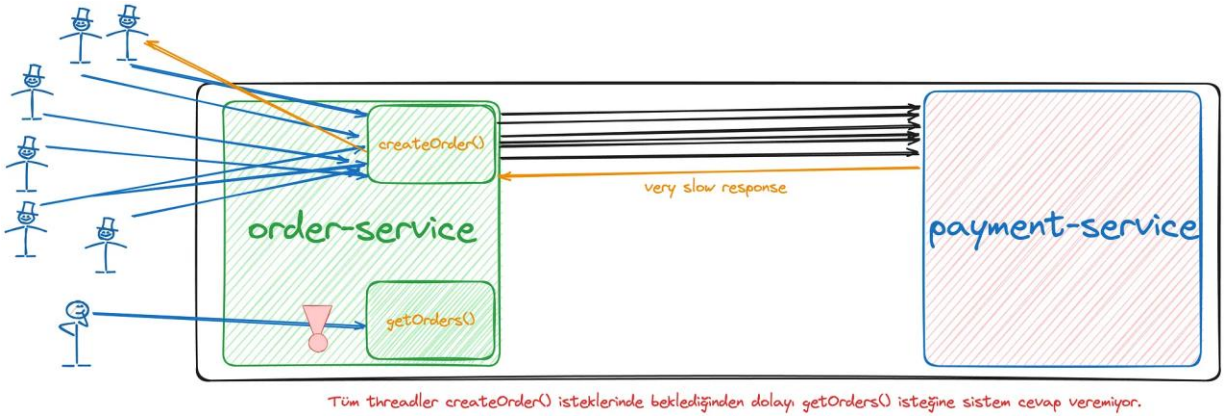
- **Xidmətlərin Bölünməsi:** İlk addım xidmətlərin müstəqil bölmələrə (alyanslara) bölünməsidir. Hər bir alyans, müəyyən bir xidmət və ya xidmətlər qrupunu təmsil edir və digər alyanslardan izolyasiya olunur. Bu bölmə, xidmətlərin funksional və ya resurs əsaslı olaraq qruplaşdırılmasına əsaslanır.
- **Resursların Təcrid Edilməsi:** Hər bir alyans öz resurslarına (CPU, yaddaş, şəbəkə bağlantısı və s.) malik olmalıdır. Bu resurslar digər alyanslardan izolyasiya edilməlidir ki, bir alyansdakı resurs çatışmazlığı digər alyanslara təsir etməsin. Resursların təcrid edilməsi, xidmətlərin bir-birindən asılı olmadan işləməsini təmin edir və sistemin ümumi davamlılığını artırır.
- **Yük Balanslaşdırma:** Hər bir alyansda yük balanslaşdırma mexanizmləri tətbiq edilməlidir. Bu mexanizmlər, hər bir alyansın yüklənməsini idarə etməyi və resursların effektiv istifadəsini təmin edir. Yük balanslaşdırma, xidmətlərin performansını və mövcudluğunu artırır.
- **Nasazlıq İdarəetməsi:** Hər bir alyansda nasazlıq idarəetmə mexanizmləri tətbiq edilməlidir. Bu mexanizmlər, nasazlıqların aşkarlanmasını və təsirinin

məhdudlaşdırılmasını təmin edir. Nasazlıq idarəetməsi, xidmətlərin davamlılığını və istifadəçi təcrübəsini qorumaq üçün vacibdir [Abbott and Fisher, 2015].

Bulkhead Nümunəsinin Faydaları

Bulkhead nümunəsi mikroservislər arxitekturasında bir neçə əsas fayda təqdim edir (Şəkil 3.6):

- **Nasazlıqların Təsirinin Məhdudlaşdırılması:** Bulkhead nümunəsi, sistemin bir hissəsində yaranan nasazlığın digər hissələrə yayılmasının qarşısını alır. Bu, xidmətlərin daha stabil və etibarlı işləməsini təmin edir və nasazlıqların təsirini minimuma endirir.
- **Yüksək Davamlılıq:** Xidmətlərin izolyasiya olunması, sistemin ümumi davamlılığını artırır. Nasazlıqların bir alyansda məhdudlaşdırılması, digər alyansların təsirlənmədən işləməsinə imkan verir və sistemin ümumi iş qabiliyyətini qoruyur.
- **Effektiv Resurs İstifadəsi:** Bulkhead nümunəsi, resursların izolyasiya olunmasını və hər bir alyansın öz resurslarına malik olmasını təmin edir. Bu, resursların effektiv istifadəsini və xidmətlərin performansını artırır.
- **Yaxşılaşdırılmış İstifadəçi Təcrübəsi:** Nasazlıqların təsirinin məhdudlaşdırılması və xidmətlərin davamlılığının artırılması, istifadəçilərin daha yaxşı bir təcrübə yaşamasına imkan verir. Bulkhead nümunəsi, istifadəçilərin minimal pozulmalar yaşamasını təmin edir.



Şəkil 3.6. Bulkhead nümunəsi

Real Dünyada Tətbiqi

Bulkhead nümunəsi, real dünyada müxtəlif sahələrdə geniş istifadə olunur. Məsələn:

- **E-ticarət Platformaları:** Məsələn, deyək ki, bizdə e-ticarət platforması var və bu platformada sifariş-xidmət və ödəniş xidməti adlı 2 mikroservis var. Sifariş xidmətinin vəzifəsi keçmiş sifarişləri sadalamaq və yeni sifarişlər yaratmaqdan, ödəniş xidmətinin vəzifəsi isə müvafiq məlumatlarla ödəniş əməliyyatını tamamlamaqdan ibarətdir. Sifariş gəldikdə, sifariş xidməti sifariş ödənişini tamamlamaq üçün ödəniş xidmətinə sorğu göndərir. Amma tutaq ki, ödəniş xidməti daxilindəki bəzi problemlər və ləngliksəbəbindən işini çox gec başa çatdırır. Bu halda, sifariş xidmətinə gələn yeni sifariş sorğuları faktiki olaraq ödəniş xidmətinə qoşulduqları üçün ləngiyəcək və bir müddət sonra bu ləngliyə görə mövcud threadların sayı xeyli azalacaq və gözləniləcək. Bu halda, "keçmiş sifarişlərim" sorğusunu göndərən istifadəçilər təsirlənəcək, çünki uyğun boş threadımız yoxdur. Nəticədə ödəniş xidmətindəki ləngliknəinki “sifariş yarat” sorğusuna təsir etdi, hətta “keçmiş sifarişlərim” sorğularını da bloklayır. Biz createOrder() metoduna bulkhead modelini tətbiq etdikdə, o, bölgü modelinin konfiqurasiyasında göstərilən eyni vaxtda sorğuların sayının bu metoda çatmasına imkan verir. Ona görə də createOrder() əməliyyatları gözləməli olacaq, çünki onlar thread pooldakı bütün threadlardan istifadə edə bilmirlər və onlar bütün threadları tutmadığından sistemin qalan hissələrinə edilən sorğular yerinə yetirilə bilər.
- **Media Axın Xidmətləri:** Media axın xidmətlərində, müxtəlif kontent tədarükçüləri və ya məzmun tipləri ayrı-ayrı alyanslarda yerləşdirilir. Bu, bir tədarükçünün və ya məzmun tipinin nasazlığı digər məzmunların mövcudluğuna təsir etmədən işləməsinə imkan verir.
- **Maliyyə Sistemləri:** Bank və maliyyə xidmətlərində, müxtəlif əməliyyat tipləri (məsələn, ödənişlər, valyuta çevrilməsi, kredit tədqiqatları) ayrı-ayrı alyanslarda yerləşdirilir. Bu, bir əməliyyatın nasazlığı digər əməliyyatların iş qabiliyyətinə təsir etmədən işləməsinə imkan verir.

Bulkhead Nümunəsinin Məhdudiyyətləri

Bulkhead nümunəsi, xidmətlərin izolyasiya olunmasını təmin etməklə bir çox fayda təqdim etsə də, müəyyən məhdudiyyətlərə də malikdir:

- **Komplekslik və İdarəetmə Xərcləri:** Bulkhead nümunəsinin tətbiqi sistemin kompleksliyini artırır. Xidmətlərin izolyasiya olunması və resursların təcrid edilməsi əlavə idarəetmə xərcləri tələb edir. Bu, xidmətlərin inkişafı və idarəedilməsi proseslərini çətinləşdirə bilər.
- **Resursların Ayrılması:** Hər bir alyansın öz resurslarına malik olması resursların effektiv istifadəsini təmin etsə də, bəzi hallarda resursların bölüşdürülməsi qeyri-effektiv ola bilər. Resursların izolyasiya olunması bəzi alyansların resurs çatışmazlığı yaşamasına, digərlərinin isə resurs bolluğundan istifadə etməsinə səbəb ola bilər.
- **İzolyasiya Səviyyəsi:** Bulkhead nümunəsinin effektivliyi izolyasiya səviyyəsindən asılıdır. Yüksək səviyyədə izolyasiya daha çox davamlılıq təmin etsə də, xidmətlər arasında qarşılıqlı əlaqələrin çətinləşməsinə səbəb ola bilər. İzolyasiya səviyyəsi düzgün təyin edilməlidir ki, həm xidmətlərin davamlılığı, həm də qarşılıqlı əlaqələr arasında balans qorunsun.

Nəticə olaraq Bulkhead nümunəsi, mikroservislər arxitekturasında xidmətlərin bir-birindən izolyasiya olunmasını təmin etməklə nasazlıqların təsirinin məhdudlaşdırılması və sistemin davamlılığının artırılması üçün mühüm bir dizayn nümunəsidir. Bu nümunə, xidmətlərin müstəqil şəkildə işləməsini və nasazlıqların təsirinin minimuma endirilməsini təmin edir. Bulkhead nümunəsinin düzgün tətbiqi, nasazlıqların təsirini məhdudlaşdırır, resursların effektiv istifadəsini təmin edir və xidmətlərin performansını artırır. Mikroservislər arxitekturasının davamlılığı və etibarlılığı üçün Bulkhead nümunəsi vacib bir komponentdir. Bu nümunənin effektiv istifadəsi, sistemin ümumi dayanıqlığını və istifadəçilərin təcrübəsini yaxşılaşdırmaq üçün əsas vasitələrdən biridir.

3.5 Mikroservislər Arxitekturasında Fallback (Ehtiyat) Nümunəsi

Mikroservislər arxitekturası, mürəkkəb və böyük miqyaslı tətbiqləri kiçik, müstəqil xidmətlərə bölərək onların idarə edilməsini və miqyaslanmasını asanlaşdırır. Lakin bu xidmətlər arasında qarşılıqlı asılılıq və nasazlıqların yayılması kimi problemlər də meydana çıxa bilər. Bu problemlərin öhdəsindən gəlmək və sistemin davamlılığını təmin etmək üçün müxtəlif dizayn nümunələri tətbiq olunur. Fallback (Ehtiyat) nümunəsi, xidmətlərin əsas funksionalitetlərinin müvəqqəti olaraq işləməməsi halında alternativ həll yolları təqdim etməklə xidmətin davamlılığını təmin etmək üçün istifadə olunur.

Fallback Nümunəsinin Anlayışı

Fallback nümunəsi, əsas xidmətin işləmədiyi və ya cavab vermədiyi hallarda ehtiyat (fallback) funksionaliteyin tətbiq olunmasını nəzərdə tutur. Bu ehtiyat funksionalitey, əsas xidmətin funksiyalarının məhdudlaşdırılmış və ya alternativ variantını təqdim edir. Beləliklə, istifadəçilər və ya digər xidmətlər əsas xidmətin nasazlığından təsirlənmədən fəaliyyətlərini davam etdirə bilərlər [Richardson, 2018].

Fallback nümunəsinin əsas məqsədi, müvəqqəti xətaların və nasazlıqların istifadəçilərə və ya digər xidmətlərə təsirini minimuma endirməkdir. Bu yanaşma, xidmətlərin daha dayanıqlı və etibarlı olmasını təmin edir.

Fallback Nümunəsinin Tətbiqi

Fallback nümunəsini tətbiq etmək üçün bir neçə əsas addım var:

- **Nasazlıqların Aşkarlanması:** İlk addım, əsas xidmətin nasazlıqlarını və ya uğursuz cəhdlərini aşkarlamaqdır. Bu, uğursuz sorğuların cavab kodları, zaman aşımı xətaları və ya digər nasazlıq göstəriciləri əsasında edilə bilər.
- **Ehtiyat Funksionaliteyin Təmin Edilməsi:** Əsas xidmətin işləmədiyi hallarda istifadə ediləcək ehtiyat funksionaliteyin müəyyən edilməsi və tətbiqi vacibdir. Bu ehtiyat funksionalite, əsas xidmətin əsas funksiyalarının məhdudlaşdırılmış və ya alternativ variantını təmin etməlidir.

- **Geri Dönüş Mexanizmlərinin Qurulması:** Geri dönüş mexanizmləri, əsas xidmətin nasazlığı halında ehtiyat funksionaliteyin işə düşməsini təmin edir. Bu mexanizmlər, əsas xidmətin nasazlığını aşkarladıqdan sonra avtomatik olaraq ehtiyat funksionaliteyi işə salmalıdır.
- **Loqlaşdırma və Monitorinq:** Ehtiyat funksionaliteyin istifadəsi və nasazlıqların izlənməsi vacibdir. Loqlaşdırma və monitorinq mexanizmləri, nasazlıqların tez bir zamanda aşkarlanması və aradan qaldırılması üçün vacib məlumatları təmin edir.

Fallback Nümunəsinin Faydaları

Fallback nümunəsi mikroservislər arxitekturasında bir neçə əsas fayda təqdim edir:

- **Xidmət Davamlılığı:** Fallback nümunəsi, əsas xidmətin nasazlığı halında xidmətin davamlılığını təmin edir. Ehtiyat funksionaliteyin işə düşməsi, istifadəçilərin və digər xidmətlərin minimal pozulmalar yaşamasına imkan verir.
- **İstifadəçi Təcrübəsinin Yaxşılaşdırılması:** Ehtiyat funksionaliteyin təmin edilməsi, istifadəçilərin müvəqqəti xətalar səbəbindən xidmətin işləməməsindən təsirlənməsinin qarşısını alır. Bu, istifadəçilərin təcrübəsini yaxşılaşdırır və onların məmnunluğunu artırır.
- **Nasazlıqların Təsirinin Azaldılması:** Fallback nümunəsi, əsas xidmətin nasazlığı halında nasazlıqların təsirini azaldır. Ehtiyat funksionalite, nasazlıqların yayılmasının və sistemin digər hissələrinə təsirinin qarşısını alır.
- **İdarəetmə Xərclərinin Azaldılması:** Fallback nümunəsi, nasazlıqların öhdəsindən gəlmək üçün daha az vaxt və resurs tələb edir. Ehtiyat funksionaliteyin təmin edilməsi, nasazlıqların tez bir zamanda aradan qaldırılmasını və idarəetmə xərclərinin azaldılmasını təmin edir.

Real Dünyada Tətbiqi

Fallback nümunəsi, real dünyada müxtəlif sahələrdə geniş istifadə olunur. Məsələn:

- **E-ticarət Platformaları:** E-ticarət platformalarında, ödəniş sistemləri və ya inventar idarəetmə xidmətləri kimi əsas xidmətlərin nasazlığı halında ehtiyat funksionaliteyin təmin edilməsi vacibdir. Məsələn, əsas ödəniş sistemi işləmədiyi halda

alternativ ödəniş metodlarının təmin edilməsi istifadəçilərin alış-veriş prosesini davam etdirməsinə imkan verir.

- **Media Axın Xidmətləri:** Media axın xidmətlərində, kontent çatdırılma şəbəkələri (CDN) və ya media serverləri kimi əsas xidmətlərin nasazlığı halında ehtiyat funksionaliteyin təmin edilməsi vacibdir. Məsələn, əsas media serverinin işləmədiyi halda alternativ serverlərin istifadəsi istifadəçilərin axını dayanmadan izləməsinə imkan verir.
- **Maliyyə Sistemləri:** Bank və maliyyə xidmətlərində, ödənişlərin işlənməsi və ya valyuta çevrilməsi kimi əsas əməliyyatların nasazlığı halında ehtiyat funksionaliteyin təmin edilməsi vacibdir. Məsələn, əsas ödəniş xidmətinin işləmədiyi halda alternativ ödəniş üsullarının təmin edilməsi müştərilərin maliyyə əməliyyatlarını problemsiz yerinə yetirməsinə imkan verir.

Fallback Nümunəsinin Məhdudiyyətləri

Fallback nümunəsi, xidmətlərin davamlılığını təmin etmək üçün effektiv bir yanaşma olsa da, müəyyən məhdudiyyətlərə malikdir:

- **Komplekslik:** Fallback nümunəsinin tətbiqi, sistemin kompleksliyini artırır. Ehtiyat funksionaliteyin müəyyən edilməsi və idarə olunması əlavə vaxt və resurs tələb edir. Bu, inkişaf və idarəetmə proseslərini çətinləşdirə bilər.
- **Resursların Səmərəsiz İstifadəsi:** Ehtiyat funksionalite əsas xidmətlə eyni resursları istifadə edə bilər. Bu, bəzi hallarda resursların səmərəsiz istifadəsinə və ya resurs çatışmazlığına səbəb ola bilər. Resursların optimal idarə edilməsi üçün ehtiyat funksionalite düzgün planlaşdırılmalıdır.
- **İzolyasiya Problemləri:** Fallback nümunəsi, ehtiyat funksionaliteyin əsas xidmətlə eyni infrastrukturda yerləşdirilməsi halında tam izolyasiya təmin edə bilməz. Bu, əsas xidmətin nasazlığı halında ehtiyat funksionaliteyin də təsirlənməsinə səbəb ola bilər.

Nəticə olaraq Fallback nümunəsi, mikroservislər arxitekturasında xidmətlərin əsas funksionaliteyin nasazlığı halında davamlılığını təmin etmək üçün mühüm bir dizayn nümunəsidir. Bu nümunə, ehtiyat funksionaliteyin təmin edilməsi və nasazlıqların təsirinin minimuma endirilməsi ilə xidmətlərin daha dayanıqlı və etibarlı olmasını təmin

edir. Fallback nümunəsinin düzgün tətbiqi, nasazlıqların təsirini azaldır, istifadəçilərin təcrübəsini yaxşılaşdırır və idarəetmə xərclərini azaldır.

Fallback nümunəsi, müasir mikroservislər arxitekturasının davamlılığını və etibarlılığını təmin etmək üçün vacib bir komponentdir. Bu nümunənin effektiv istifadəsi, xidmətlərin müvəqqəti xətalar və nasazlıqlar səbəbindən dayanmadan işləməsini təmin edir və istifadəçilərin minimal pozulmalar yaşamasına imkan verir. Fallback nümunəsi, xidmətlərin performansını və mövcudluğunu artırmaq üçün əsas vasitələrdən biridir və mikroservislər arxitekturasının uğurlu tətbiqi üçün vacibdir.

NƏTİCƏ

Bu dissertasiya, informasiya sistemlərində arxitektura mövzusunda dərin bir təhlil aparmaqla başlayır və monolit arxitekturasının çatışmazlıqlarını nəzərə alaraq, mikroservislər arxitekturasının üstünlüklərini və tətbiq sahələrini əhatə edir. Müasir proqram təminatında mikroservislər arxitekturasının əhəmiyyəti getdikcə artır və bu iş, həmin arxitekturanın əlçatanlıq və dayanıqlığını təmin edən dizayn patternləri və prinsipləri üzərində fokuslanır.

Mikroservislər Arxitekturasının Əhəmiyyəti - mikroservislər arxitekturası, böyük və kompleks sistemləri kiçik, müstəqil xidmətlərə bölməklə onları daha idarəolunan və çevik hala gətirir. Bu yanaşma, sistemin genişlənməsini asanlaşdırır və hər bir xidmətin müstəqil olaraq inkişaf etdirilməsi, yenilənməsi və miqyaslanması imkanını verir. Monolit arxitekturasının tək bir kod bazasında bütün funksionallıqları toplaması nəticəsində meydana gələn performans və texniki borc problemlərini aradan qaldırmaq üçün mikroservislər ideal bir həll kimi çıxış edir.

Dizayn Patternləri və Prinsipləri - mikroservislər arxitekturasında əlçatanlığı və dayanıqlığı təmin etmək üçün bir sıra dizayn patternləri və prinsiplər tətbiq olunur. API Gateway patterni, xidmətlər arasındakı trafikin mərkəzi bir nöqtədən idarə olunmasını təmin edir, bu da təhlükəsizlik və yük balanslaması kimi məsələlərin həllində vacib rol oynayır. Database per service və polyglot persistence patternləri, hər bir xidmətin öz müstəqil verilənlər bazasına sahib olması və müxtəlif verilənlər bazası texnologiyalarının istifadəsi ilə sistemin daha elastik və performanslı olmasına kömək edir.

Micro-frontends architecture patterni, istifadəçi interfeyslərinin də mikroservis prinsiplərinə uyğun olaraq parçalanmasını təmin edir. Backend for frontend patterni isə fərqli müştəri tətbiqləri üçün xüsusi backend xidmətlərinin yaradılmasını mümkün edir ki, bu da müştəri təcrübəsinin yaxşılaşdırılmasına yönəlmiş bir yanaşmadır. SAGA, CQRS və Event Sourcing patternləri, mürəkkəb biznes proseslərinin və məlumat idarəçiliyinin effektiv və etibarlı şəkildə həyata keçirilməsini təmin edir.

Mikroservislərdə Dayanıqlıq - Mikroservislər arxitekturasında dayanıqlığın təmin edilməsi, sistemin ümumi fəaliyyətinin yüksək səviyyədə saxlanılmasına kömək edir. Circuit Breaker patterni, xidmətlər arasındakı nasazlıqları izolyasiya edərək sistemin daha dayanıqlı olmasını təmin edir. Retry patterni, müvəqqəti uğursuzluqların öhdəsindən gəlmək üçün tək-tək təkrarlanan sorğular vasitəsilə etibarlılığı artırır. Bulkhead patterni, fərqli xidmətlərin müstəqil şəkildə işləməsini təmin edərək bir xidmətin problemi digər xidmətləri təsir etməsinin qarşısını alır. FallBack patterni isə uğursuzluqlar zamanı alternativ həllərin tətbiq olunmasını təmin edir.

Nəticələrin Aprobasiyası - "Article-Hub" platformasında mikroservis arxitekturasının tətbiqi nəticəsində performans, istifadəçi təcrübəsi, təhlükəsizlik və etibarlılıq baxımından əhəmiyyətli irəliləyişlər əldə edilmişdir. Performans testləri, yükləmə zamanının azaldığını və yüksək yük şərtləri altında platformanın dayanıqlılığının artdığını göstərir. İstifadəçi təcrübəsi testləri, real-time redaktə və şəxsi məqalə təklifləri kimi funksiyaların istifadəçilər tərəfindən yüksək qiymətləndirildiyini təsdiqləmişdir. Təhlükəsizlik və etibarlılıq testləri, blockchain texnologiyası ilə məzmun etibarlılığının təmin olunduğunu və mikroservislər arasında təhlükəsiz əlaqələrin qurulduğunu göstərmişdir.

Bu dissertasiya, mikroservislər arxitekturasının müasir informasiya sistemlərində necə tətbiq oluna biləcəyini və bu tətbiqin necə optimallaşdırıla biləcəyini göstərir. Əldə edilən nəticələr, mikroservis yanaşmasının effektivliyini və onun müxtəlif sahələrdəki tətbiq potensialını bir daha vurğulayır. Bu yanaşma, böyük və mürəkkəb sistemlərin idarə edilməsində daha çox üstünlüklər təqdim edir və gələcəkdə daha da geniş yayılması gözlənilir.

"Article-Hub" platforması üzərində aparılan bu iş, mikroservis arxitekturasının faydalarını və bu arxitekturanın tətbiqi ilə əldə edilən nəticələri göstərməklə, mövzunun elmi və praktiki əhəmiyyətini ortaya qoyur. Mikroservislər arxitekturasının tətbiqi, kompleks sistemlərin idarə edilməsini asanlaşdırır, performansı artırır və istifadəçi təcrübəsini yaxşılaşdırır. Bu yanaşma, müasir proqram təminatının inkişafında mühüm bir

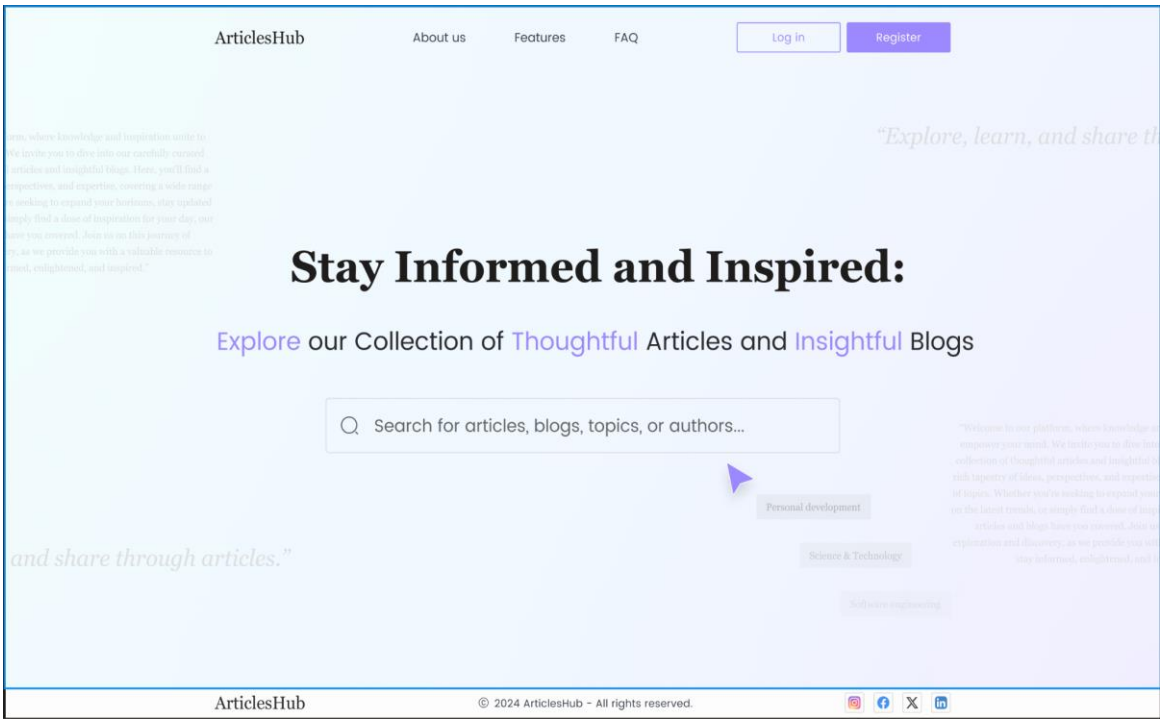
mərhələ olaraq qəbul edilə bilər və gələcəkdə daha geniş bir tətbiq sahəsi tapacağına inanılır.

İSTİFADƏ EDİLMİŞ ƏDƏBİYYAT

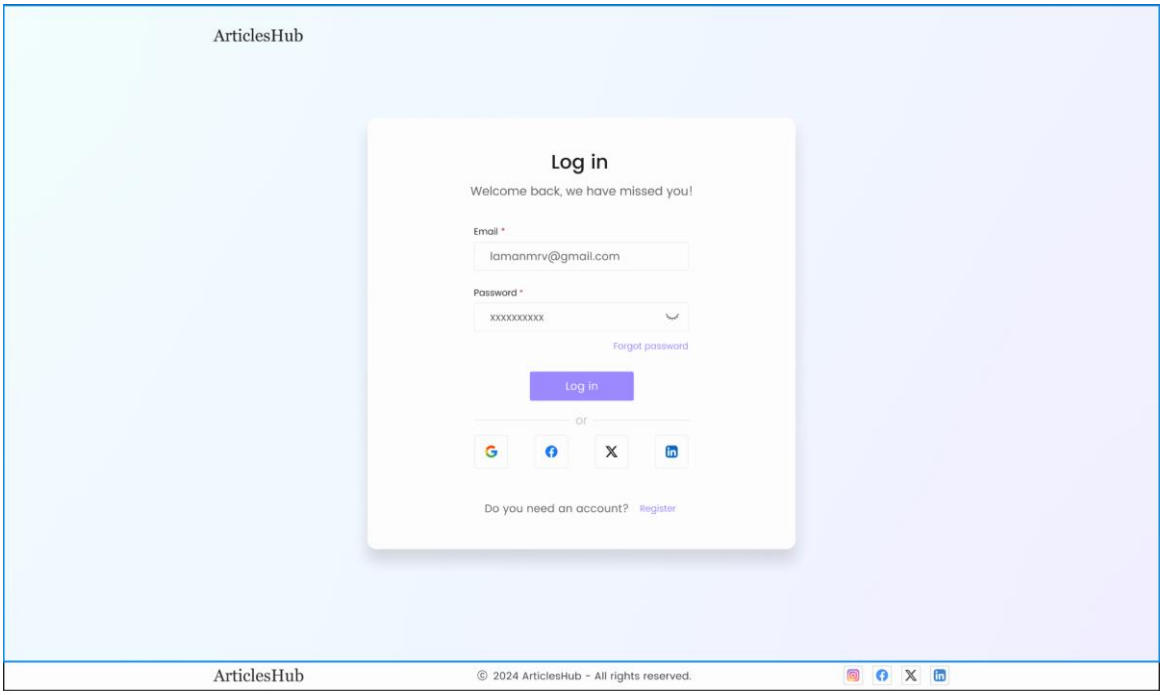
1. Adam, B. (2020). Building Event-Driven Microservices: Leveraging Organizational Data at Scale.
2. Brian, M., & Holly, B. (2016). Microservices Architecture. O'Reilly Media.
3. Boris, S., Trent, S., Peter, J.(2019). Microservices with Docker on Microsoft Azure: Includes Content Update Program.
4. Bilgin, I., Roland, H.(2019). Kubernetes Patterns: Reusable Elements for Designing Cloud-Native Applications.
5. Craig, W.(2018).Spring in Action.
6. Cornelia, D. (2019). Cloud Native Patterns: Designing change-tolerant software Cornelia Davis.
7. Eoin, W., Nick, R. (2020) Evolving Software Architectures: A Critical Look at Practice and Research.
8. Gene, K., Patrick D., John, W., Jez, H. (2016). The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations.
9. Gregor, H. (2020). The Software Architect Elevator: Redefining the Architect's Role in the Digital Enterprise.
10. Jonas, B.(2016). Reactive Microservices Architecture: Design Principles for Distributed Systems.
11. John, C. (2017). Spring Microservices in Action.
12. Kasun, I., Prabath, S.(2018). Microservices for the Enterprise: Designing, Developing, and Deploying.
13. Martin, A., Michael, F (2015). Art of Scalability, The: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise.
14. Martin, K. (2017). Designing Data-Intensive Applications.
15. Morgan, B., Paulo A. Pereira (2018). Microservices in Action.
16. Mark, R., Neal F. (2020). Fundamentals of Software Architecture.

17. Newman, S. (2015). Building Microservices.
18. Namit, T., Rahul, R. (2017). Microservices with Azure: Building, Deploying, and Monitoring.
19. Nerman, S (2019). Monolith to Microservices.Evolutionary Patterns to Transform Your Monolith.
20. Neal, F., Mark R., Pramod S., Zhamak D. (2022). Software Architecture: The Hard Parts.
21. Parminder, S. , K.(2018). Microservices and Containers
22. Rusu, L. (2017). Information Technology Governance in Public Organizations: Theory and Practice
23. Robert, M. (2017). Clean Architecture: A Craftsman's Guide to Software Structure and Design.
24. Richard, R. (2017). The Tao of Microservices.
25. Richardson, C. (2018). Microservices Patterns: With examples in Java. Manning Publications.
26. Sourabh, S(2017). Mastering Microservices with Java.
27. Susan, J. Fowler (2017). Production-Ready Microservices: Building Standardized Systems Across an Engineering Organization.
28. Stephen F. (2018). Continuous Delivery Handbook: Non Programmer's Guide to DevOps, Microservices and Kubernetes.
29. Siriwardena, P.(2020). Microservices Security in Action.
30. Vululleh, P. (2022). Cyber CareersThe Basics of Information Technology and Deciding on a Career Path.

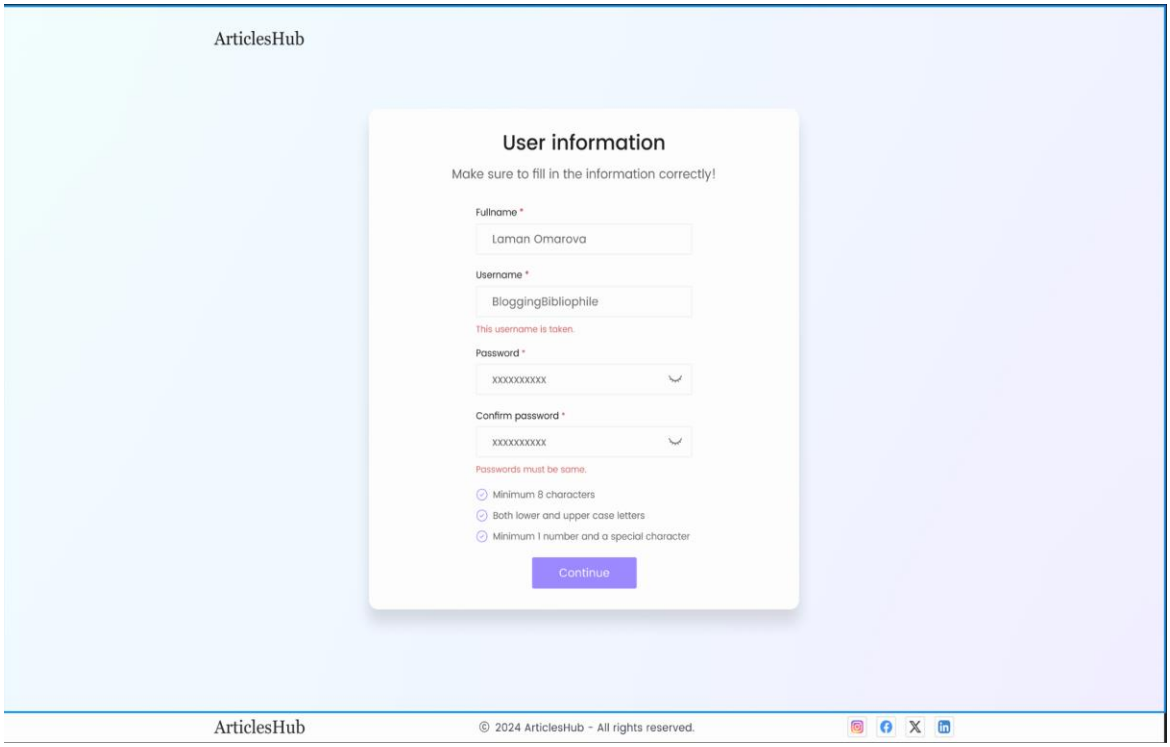
ƏLAVƏLƏR



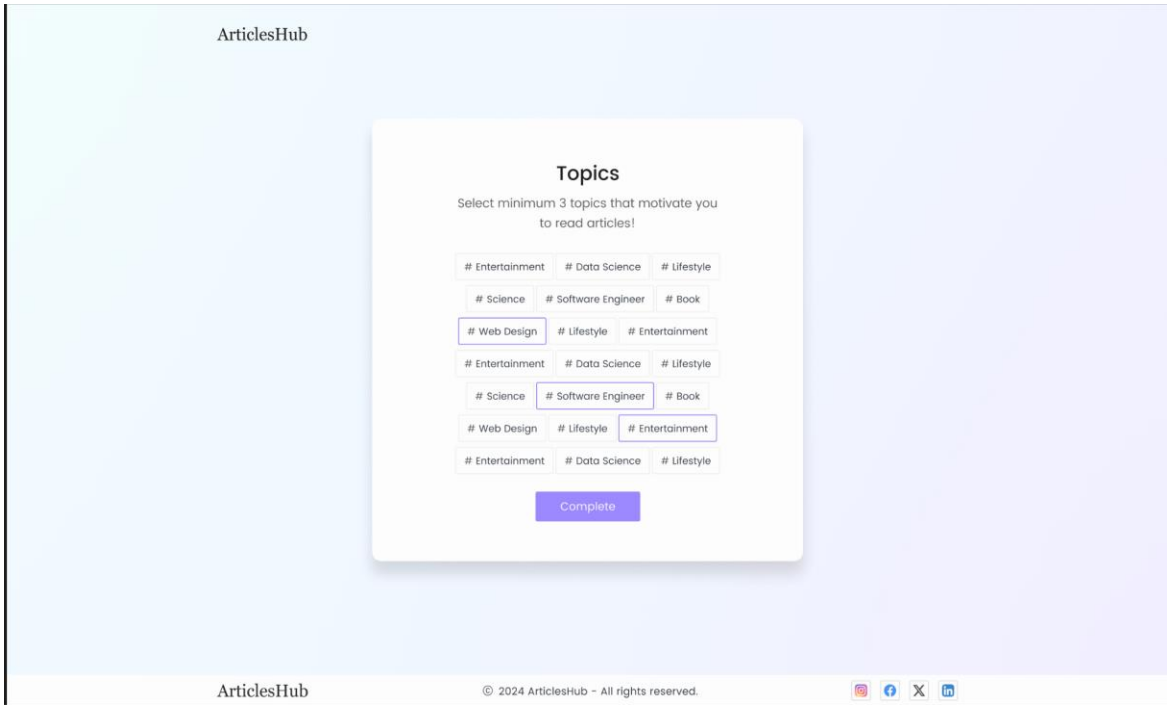
Əlavə 1



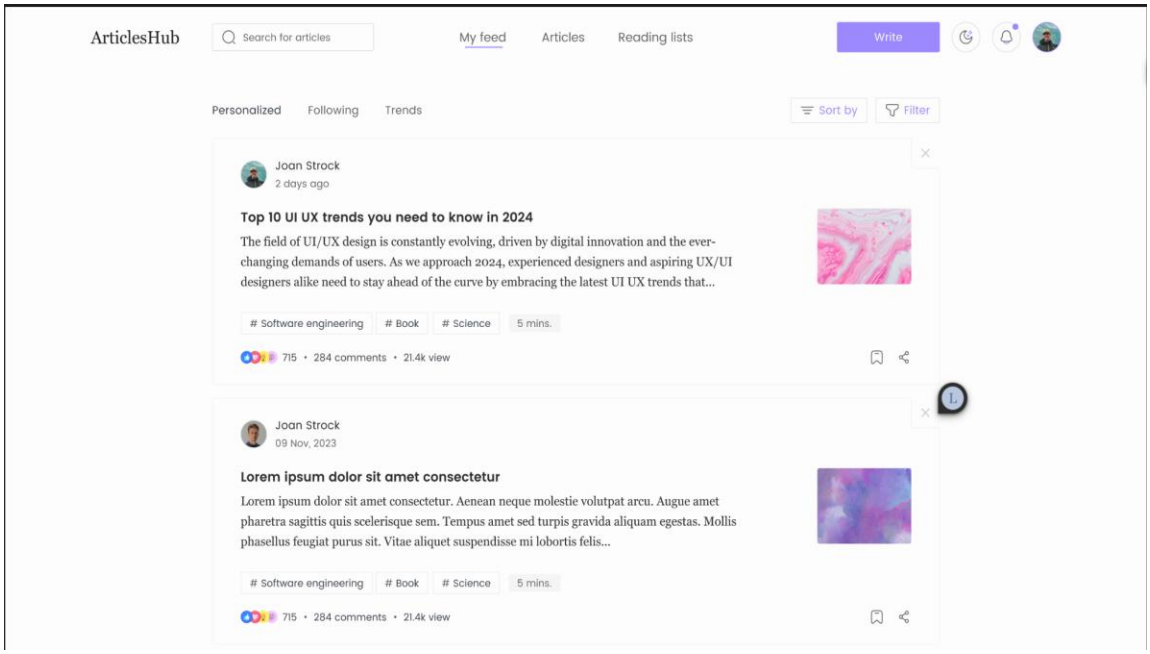
Əlavə 2



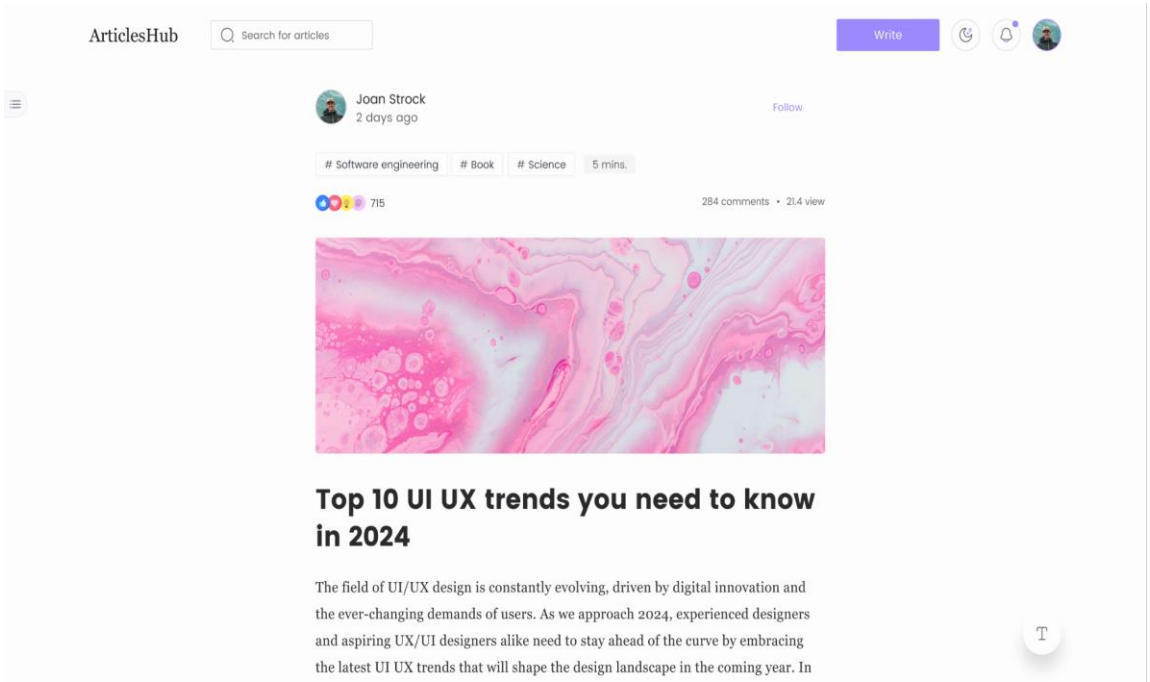
Əlavə 3



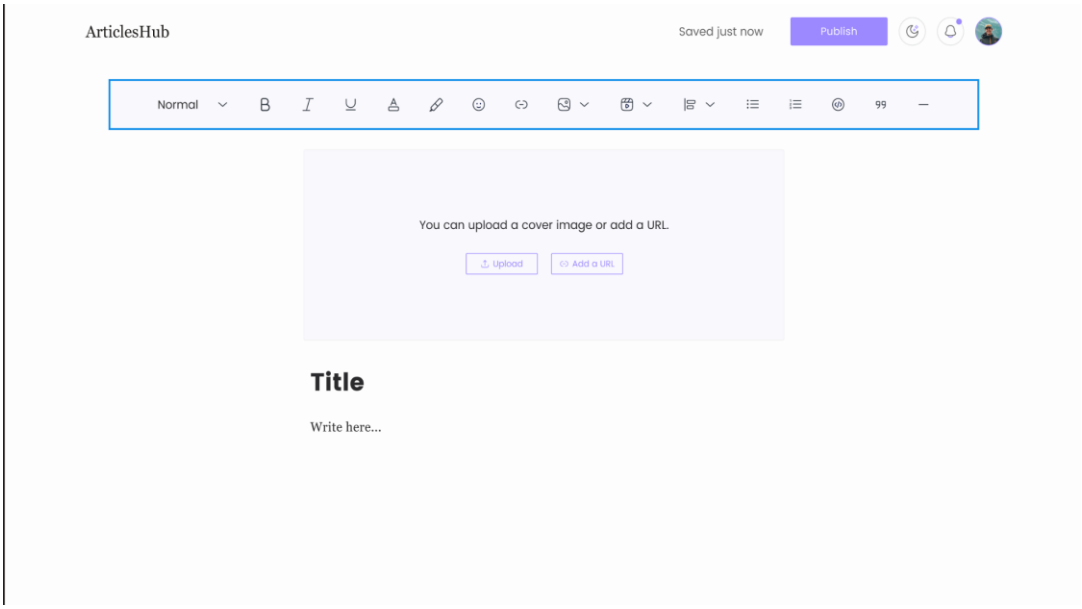
Əlavə 4



Əlavə 5



Əlavə 6


ArticlesHub
Saved just now
Publish
Normal
You can upload a cover image or add a URL.
Upload
Add a URL
Title
Write here...

Əlavə 7