UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Tatjana Kunić

RAZVOJ SISTEMA ZA OBRADU VELIKIH PODATAKA UPOTREBOM APACHE SPARK I JAVA SPRING BOOT

master rad

Beograd, 2026.

Mentor:

dr Vladimir Filipović, redovni profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Aleksandar Kartelj, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

dr Staša Vujičić Stanković, docent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

Mami, tati i baki

Naslov master rada: Razvoj sistema za obradu velikih podataka upotrebom *Apache Spark* i *Java Spring Boot*

Rezime: Podaci su postali jedan od najbitnijih faktora za donošenje poslovnih odluka i razvoja novih tehnologija širom različitih industrija. Kako se danas sve više aplikacija oslanja na podatke, principi projektovanja i razvoja softvera koji omogućavaju efikasan rad sa podacima, skladištenje i procesiranje velike količine podataka su od posebne važnosti. Društvene mreže kao jedan od primera aplikacija sa velikim podacima predstavljaju poseban izazov za implementaciju rešenja koje je otporno na greške, podržava velike količine raznovrsnih tipova podataka i koje pokazuje dobre performanse čak i kad je pod velikim radnim opterećenjem.

Cilj ovog rada je da kroz razvoj mikroservisne aplikacije predstavi najbitnije koncepte i tehnologije koje se koriste pri implementaciji sistema za obradu velikih podataka. Implementacija društvene mreže služi kao primer koliko je važna saradnja između sistema koji obrađuju korisničke zahteve i analitičkih sistema koji obrađuju velike količine podataka koja se ogleda u načinu definisanja komunikacije sistema, skladištenja podataka i infrastrukture.

Rešenje predstavljeno u radu koristi okvir *Spring Boot* u programskom jeziku *Java* za implementaciju mikroservisa koji realizuju funkcionalnosti društvene mreže. U okviru aplikacije implementirano je i analitičko rešenje za obradu tokova podataka u realnom vremenu za koje su zadužene *Apache Spark* aplikacije napisane u programskom jeziku *Scala*. Pored toga *Apache Kafka* ima važnu ulogu u povezivanju i komunikaciji u okviru sistema. Tokom razvoja, stavljen je poseban akcenat na odabir tehnologije i modela skladištenja podataka, kao najbitniji faktor za osiguravanje kvaliteta i vrednosti velikih podataka.

Ključne reči: veliki podaci, mikroservisi, baze podataka, tokovi podataka, transakciona obrada, analitička obrada, računarstvo

Sadržaj

Glava 1

Uvod

1.1 Veliki podaci

Protekle godine obeležio je nagli rast količine podataka koji se generišu, prenose i obrađuju u okviru informacionih sistema. Rast popularnosti društvenih mreža, digitalizacija poslovanja, internet stvari (engl. *Internet of Things – IoT*), mediji i zabava doveli su do eksplozije u količini i raznovrsnosti podataka koji se svakodnevno obrađuju. U tom kontekstu, termin veliki podaci (engl. *big data*) pridobio je veliki značaj. Iako ne postoji jedinstvena definicija za termin veliki podaci, opšteprihvaćene karakteristike velikih podataka čine količina (engl. *volume*), brzina kojom novi podaci pristižu u sistem (engl. *velocity*) i raznovrsnost strukture i formata podataka (engl. *variety*), uz često pominjanje dodatnih karakteristika verodostojnosti (engl. *veracity*) i vrednosti (engl. *value*) podataka [22]. Navedene karakteristike zajedno otežavaju rad tradicionalnih sistema obrade podataka, koji su većinski projektovani za rad sa strukturiranim podacima striktne sheme.

Povećanje kapaciteta hardverskih resursa sistema, poznato kao vertikalno skaliranje, često nije dovoljno za obradu velikih podataka usled fizičkih i ekonomskih ograničenja. Iz tih razloga, razvijaju se sistemi koji omogućavaju horizontalno skaliranje, što podrazumeva distribuciju radnog opterećenja na više računara, pri čemu svaki raspolaže sopstvenim hardverskim resursima [23].

Odabir tehnologije skladištenja podataka je jedan od najbitnijih aspekata u projektovanju softvera na koji primarni uticaj ima namena sistema. Ukoliko je sistem namenjen za podršku u svakodnevnim operacionim procesima organizacije, tada je reč o transakcionim softverskim sistemima ili OLTP (*OnLine Transactional Processing*) sistemima. S druge strane, analitički sistemi za potrebe donošenja poslovnih

odluka oslanjaju se na OLAP (*OnLine Analytical Processing*) skladišta podataka.

Projektovanje softverskih sistema koji generišu velike podatke, tačnije transakcionih sistema, takođe je od velikog značaja jer određuju vrstu, strukturu, kvalitet i dostupnost podataka koji se kasnije koriste u analitičkoj obradi. Definisanje i implementacija odgovarajuće arhitekture softverskog sistema je od ključne važnosti u poboljšanju efikasnosti, raspoloživosti i odzivnosti sistema.

Među savremenim aplikacijama koje rade sa velikim podacima posebno se ističu društvene mreže, kao platforme sa najvećim brojem aktivnih korisnika na dnevnom nivou. Mnogima su društvene mreže primarni način informisanja, organizovanja i komunikacije, pa samim tim i korisnici očekuju da aplikacija bude uvek dostupna, otporna na greške i da vreme odziva između zahteva i odgovora bude minimalno. Veliki broj korisnika generiše i upućuje različite vrste zahteva istovremeno, pa je zbog toga količina podataka koje bi aplikacija trebalo da obradi veoma velika, pri čemu ti podaci mogu biti u različitim formatima uključujući tekstualne sadržaje, slike i video zapise.

1.2 Mikroservisi

Mikroservisna arhitektura predstavlja obrazac projektovanja distribuiranog softvera u kom se aplikacija sastoji od komponenti, mikroservisa, gde svaka implementira jednu jasno definisanu funkcionalnost poslovnog domena i mogu se nezavisno razvijati i isporučivati [30]. Mikroservisi zajedno rade i komuniciraju kako bi činili jedan celokupan sistem. Osnovna ideja je postizanje što veće robusnosti dekompozicijom složenih sistema na manje, jednostavnije komponente. Svaki mikroservis se može nezavisno skalirati u zavisnosti od opterećenja ili, u slučaju greške, na pojedinačnim instancama mikroservisa, čime se postiže visoka dostupnost sistema. Svaki mikroservis poseduje jasno definisane granice odgovornosti, kao i sopstveni skup podataka, u vidu zasebne baze podataka, koji ne deli sa drugim mikroservisima [34]. Takav pristup olakšava razvoj, održavanje i razumljivost sistema, dok inženjerima pruža slobodu u izboru programskih jezika, tehnologija i sistema za skladištenje podataka koji su najprikladniji za potrebe konkretnog domena.

Pošto su mikroservisi nezavisni procesi koji ne dele zajednički memorijski prostor niti zajedničko stanje, podacima se ne pristupa direktno, već se razmenjuju preko jasno definisanih aplikacijskih interfejsa, putem mreže. Komunikaciju je moguće implementirati sinhrono, pri čemu jedan servis zahteva resurs od drugog servisa i čeka

na odgovor, ili asinhrono, najčešće preko redova poruka ili emitovanjem događaja. Ovaj aspekt uvodi dodatne izazove u pogledu pouzdanosti, konzistentnosti podataka i performansi. Izbor komunikacionog modela direktno utiče na otpornost sistema na greške, kao i na njegovu skalabilnost. Neophodno je implementirati mehanizme za obradu grešaka, ponovne pokušaje, vremenska ograničenja i toleranciju na parcijalne otkaze, koji su neizbežni u distribuiranim sistemima.

Mogućnost automatskog skaliranja mikroservisa u zavisnosti od opterećenja ili grešaka u sistemu, uzrokuje dinamičko pokretanje i zaustavljanje instanci što donosi i određene komplikacije u vidu prepoznavanja adrese servisa s kojim bi klijent ili drugi servisi trebalo da komuniciraju. Ovaj problem poznat je kao problem otkrivanja servisa (engl. *service discovery*). Najčešće rešenje ovog problema je implementacija centralnog registra adresa u koji se registruju instance servisa nakon njihovog pokretanja.

Cilj ovog rada je da, kroz primer aplikacije društvene mreže, ilustruje kompleksnost razvoja softvera koji bi trebalo da podrži veliki broj konkurentnih korisnika, različite tipove podataka, kao i razvoj platforme za obradu velikih podataka za podršku analitičkim procesima. U poglavlju koje sledi su predstavljene različite tehnologije i alati čije razumevanje je od ključnog značaja kako bi moglo da se projektuje adekvatno rešenje. U trećem poglavlju predstavljena je arhitektura i implementacija aplikacije koja prikazuje praktičnu primenu prethodno opisanih alata za razvoj i isporuku. Ovo poglavlje je podeljeno na dva dela u kom je najpre predstavljena arhitektura i implementacija mikroservisa koji implementiraju funkcionalnosti društvene mreže, a u drugom delu je opisano rešenje za analitički deo aplikacije. U četvrtom poglavlju opisani su nedostaci implementiranog rešenja sa ciljem da pruži motivaciju za dalji rad i unapređivanje postojeće aplikacije. Na kraju je iznet zaključak koji sumira glavne ideje iznete u okviru rada.

Glava 2

Tehnologije i alati

2.1 Razvojni okvir Spring

Razvojni okvir *Spring* je projekat otvorenog koda koji je nastao sa ciljem da olakša razvoj poslovnih aplikacija u programskom jeziku *Java*. Razvojni okvir *Spring* je jedan od najkorišćenijih razvojnih okvira za poslovne aplikacije [38]. Centralna komponenta *Spring*-a je *IoC* (*Inversion Of Control*) kontejner zadužen za umetanje i upravljanje zavisnostima aplikacije [13]. Umesto da klasa sama kreira objekte od kojih zavisi, odgovornost za njihovo instanciranje i povezivanje preuzima razvojni okvir. Glavna prednost *Spring*-a je što kroz bogatu podršku za različite tehnologije i arhitekture pruža programerima fleksibilnost i olakšava razvoj različitih delova aplikacije - od sloja infrastrukture do sloja korisničkog interfejsa. Jedna od glavnih mana *Spring Framework*-a je složenost konfiguracije koja često oduzima dosta vremena od samog razvoja i zahteva dodatni napor. Pored toga, mnoge *Spring* aplikacije dele sličnu osnovnu konfiguraciju. Iz tih razloga je razvijen *Spring Boot* sa ciljem da olakša i ubrza postupak konfiguracije, upravljanja zavisnostima i inicijalnog postavljanja projekta. *Spring Boot* se zasniva na principu podrazumevane konfiguracije (engl. *convention over configuration*), čime se smanjuje potreba za eksplicitnim podešavanjima [36]. Proces ručnog pronalaženja i usklađivanja verzija biblioteka zamenjen je uz pomoć skupova zavisnosti koji se zovu *Spring Boot* starteri. Starteri objedinjuju kompatibilne biblioteke sa svojim podrazumevanim konfiguracijama koje se mogu po potrebi prilagoditi u okviru `application.properties` ili `application.yaml` konfiguracionih fajlova. Jedan starter služi da bi se dodala određena funkcionalnost u aplikaciju, kao npr. `spring-boot-starter-web` koji sadrži sve neophodne biblioteke za razvoj veb aplikacija, uključujući i podrazumevani ugrađeni *Apache*

Tomcat veb server putem `spring-boot-starter-tomcat` skupa zavisnosti. Moguće je zameniti *Tomcat* nekim drugim serverom isključivanjem odgovarajuće zavisnosti i uključivanjem nekog alternativnog veb servera, kao što je *Eclipse Jetty*.

Spring Boot Reaktivne Veb Aplikacije

Tradicionalni veb serveri u *Spring* aplikacijama, kao što je *Apache Tomcat*, obrađuju HTTP zahteve konkurentno tako što se svaki zahtev dodeljuje posebnoj niti iz unapred definisanog skupa niti (engl. *thread pool*). Obično je ovakva arhitektura dovoljna za aplikacije sa manjim brojem korisnika ili aplikacije koje nisu opterećene ulazno-izlaznim operacijama [31]. Međutim, u kontekstu velikih, distribuiranih, mikroservisnih sistema od kojih se očekuje da efikasno skaliraju, ovakav pristup može uzrokovati probleme posebno u vezi skalabilnosti i potrošnje resursa.

Spring iz tih razloga pruža podršku za razvoj reaktivnih veb aplikacija kroz modul *Spring WebFlux*. Reaktivni model razvoja zasniva se na neblokirajućim, asinhronim operacijama koje se izvršavaju u okviru jedne niti zvane petlja događaja (engl. *event loop*). Prednost ovog modela razvoja jeste što se prilikom čekanja na rezultat ulazno-izlazne operacije ne blokira nit na kojoj se izvršava, već petlja nastavlja sa obradom drugih zadataka i zatim „reaguje" na rezultat prethodne operacije kad postane dostupan. Time se omogućava efikasnije korišćenje resursa i rad sa manjim brojem niti, što doprinosi boljoj skalabilnosti sistema.

Spring Boot starter koji obezbeđuje sve neophodne zavisnosti za rad sa *Spring WebFlux* modulom je `spring-boot-starter-webflux` koji kao podrazumevani ugrađeni server koristi *Netty* server zasnovan na modelu petlje događaja. Za konkretnu implementaciju reaktivnih aplikacija, koristi se *Project Reactor* biblioteka koja je takođe uključena u *WebFlux* starter. Počev od *Jave* 9, podrška za reaktivno programiranje uvedena je kroz *Flow API* kao deo standardne biblioteke, koji zasnovan na *Reactive Streams* specifikaciji. *Reactive Streams* (reaktivni tokovi) definišu skup interfejsa i pravila za asinhronu i neblokirajuću obradu podataka koji su modelovani kao neprekidni tokovi (engl. *streams*). Zasnovani su na obrascu izdavač-pretplatnik (engl. *publisher-subscriber*) gde izdavač emituje elemente toka kako oni postaju dostupni, a pretplatnik ih prima i obrađuje. Prilikom komunikacije između izdavača i pretplatnika može doći do greške ili do kraja toka podataka što je takođe na izdavaču da signalizira [37]. Ovakav način komunikacije se naziva *push-based*.

Project Reactor je implementacija *Reactive Streams* specifikacije i uvodi tipove *Mono* i *Flux* koji su implementacija *Publisher*-a. *Mono* modeluje tok dužine od 0 do

1 elementa, odnosno predstavlja rezultat operacije koja može vratiti najviše jedan element ili se završiti bez vrednosti. *Flux* modeluje tok dužine N, pri čemu $N \in \mathbb{N}_0$ i on predstavlja rezultat operacije koja može vratiti proizvoljan broj elemenata, uključujući i prazan rezultat.

Spring olakšava integraciju različitih izvora podataka preko modula *Spring Data*, dok *Spring Boot* dodatno olakšava integraciju sa reaktivnim modelom uz pomoć posebnih startera za potrebne baze podataka - na primer, za podršku za reaktivne operacije sa *MongoDB* bazom podataka je `spring-boot-starter-data-mongodb-reactive`, a `spring-boot-starter-data-r2dbc` za reaktivne operacije nad relacionim bazama podataka.

Uzimajući u obzir pomenute karakteristike *Spring Boot*-a, zaključuje se da ovaj okvir predstavlja odličan izbor za razvoj *Java* mikroservisnih aplikacija zahvaljujući jednostavnom podešavanju, upravljanju zavisnostima i širokoj podršci za različite izvore podataka. Modul *WebFlux* omogućava razvoj reaktivnih aplikacija koje lakše skaliraju uz bolju iskorišćenost hardverskih resursa u odnosu na blokirajući model obrade zahteva.

2.2 Komunikacija između mikroservisa

Komponente u monolitnim aplikacijama mogu međusobno da komuniciraju u okviru istog procesa, pozivanjem odgovarajućih funkcija. Usled izolovanosti podataka i nezavisnosti mikroservisa neophodno je da postoji način kojim mikroservisi mogu da razmenjuju podatke ili informacije o stanju sistema, što znači da je neophodno definisati obrasce komunikacije između servisa preko mreže. Projektovanje međuservisne komunikacije uvodi nove izazove kao što su pitanje performansi i obrade grešaka koje su neizbežne u radu sa distribuiranim sistemima. U praksi se najčešće vrši odabir između sinhronog i asinhronog obrasca komunikacije, pri čemu svaki pristup nosi specifične kompromise u pogledu složenosti implementacije i garancija koje pruža.

Sinhrona komunikacija između mikroservisa

Prilikom sinhrone komunikacije, klijent servisa upućuje zahtev drugom servisu i blokira izvršavanje dok ne primi odgovor. Ovaj model komunikacije je poznat kao zahtev-odgovor (engl. *request-response*) gde su najpoznatiji i najkorišćeniji *REST*

API i *gRPC*. *REST API* (*REpresentational State Transfer*) je *HTTP API* koji implementira pravila za razmenu podataka, nazvanim resursima, definisanim *REST* arhitekturom. Klijent zahteva resurse od servera navođenjem jedinstvenog identifikatora resursa (engl. *Uniform Resource Identifier – URI*) zajedno sa odgovarajućom *HTTP* metodom (npr. `GET`, `POST`, `DELETE`), dok server odgovara traženim resursom, *HTTP* statusnim kodom i porukom koji opisuju ishod operacije. Podaci koji se razmenjuju u komunikaciji predstavljeni su u određenom formatu, od kojih je najkorišćeniji *JSON* (*JavaScript Object Notation*) usled njegove jednostavnosti parsiranja u različitim programskim jezicima i čitljivosti [17].

RPC (***R**emote **P**ocedure **C**all*) je stil komunikacije u kom servis može da pozove metodu iz drugog servisa kao da je u pitanju lokalna metoda čime je enkapsulirana složenost komunikacije preko mreže [19]. *gRPC* je *RPC* okvir otvorenog koda koji je razvila kompanija *Google* gde se podaci između servisa serijalizuju preko protokol bafera, u binarnom formatu. Za razliku od *REST API*-ja koji se oslanja na *HTTP/1.1*, *gRPC* komunikacija se odvija preko *HTTP/2* protokola koji ima značajno bolje performanse zahvaljujući mogućnosti da se više poziva odvija preko jedne TCP konekcije i razmeni poruka u binarnom formatu [25]. Definisanje *gRPC* servisa vrši se kroz `.proto` datoteke u kojima se opisuju strukture poruka (engl. *message*) koje predstavljaju podatke koji se prenose između servisa, kao i potpisi metoda koje servis pruža na korišćenje drugim servisima. Kompajler protokol bafera, *protoc*, na kraju generiše kod na osnovu `.proto` datoteka za odabrani programski jezik.

Glavna prednost sinhrone komunikacije je u njenoj jednostavnosti i razumljivosti usled činjenice da je tok izvršavanja sličan onom kao u monolitnim sistemima. Obrada grešaka kod *REST API*-ja je dobro standardizovana zahvaljujući činjenici da *HTTP* uvek vraća kôd o uspešnosti operacije, pri čemu kodovi iz opsega `4xx` označavaju greške na strani klijenta, dok `5xx` označavaju greške na serverskoj strani. *gRPC* takođe nudi podršku za obradu grešaka kroz sopstveni skup statusnih kodova, ili je moguće programski obraditi greške prilikom definisanja metode na klijentskom servisu.

Sa druge strane, sinhroni pozivi povećavaju spregnutost među servisima jer uspešnost jedne operacije zavisi od više servisa istovremeno, a svaka promena sheme poruka u jednom servisu zahteva izmene u drugim servisima koji s njim komuniciraju. Ukoliko su pozivi ka drugim mikroservisima blokirajući to može negativno uticati na performanse čitavog sistema jer klijent servisa čeka dok ne dobije odgovor. Ovaj problem moguće je ublažiti implementacijom neblokirajućih klijenata. Na primer,

Spring WebFlux kroz *WebClient* interfejs omogućava neblokirajuće *HTTP* pozive zasnovane na reaktivnom modelu, dok *gRPC* takođe daje na raspolaganje asinhroni interfejs.

Asinhrona komunikacija između mikroservisa

Glavnu manu sinhronog obrasca komunikacije je moguće prevazići korišćenjem asinhronih stilova komunikacije. Asinhrona komunikacija između mikroservisa funkcioniše po principu razmene poruka ili događaja (engl. *event*). Za razliku od sinhrone komunikacije, događaji ne pristižu drugim servisima direktno preko mreže, već preko posrednika događaja (engl. *event broker*). Događaji su imutabilni podaci koji sadrže informacije o stanju ili promeni sistema kao i o tačnom vremenu nastanka događaja. Obrazac asinhrone komunikacije u kome su događaji centralni objekti poznat je kao komunikacija vođena događajima (engl. *event-driven communication*).

Jedna od najkorišćenijih platformi za razvoj sistema vođenim događajima je *Apache Kafka*, koja implementira sve prethodno opisane koncepte komunikacije događajima.

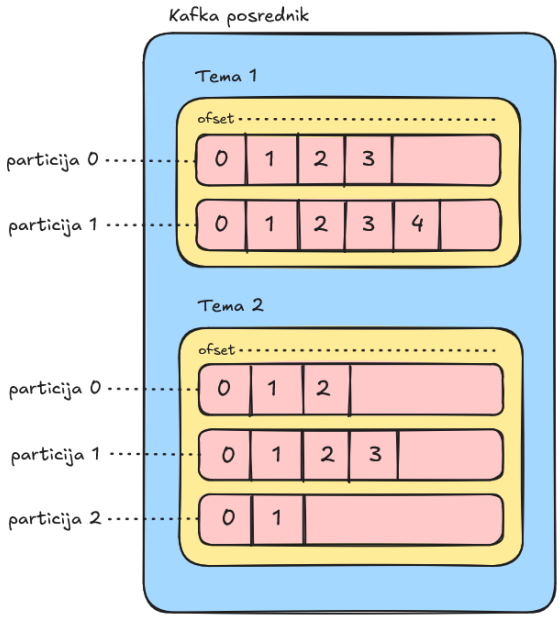
Apache Kafka

Apache Kafka je distribuirana platforma otvorenog koda koja se koristi za emitovanje događaja u sistemima koji zahtevaju visoke performanse i obradu podataka u realnom vremenu [5]. U centru *Kafka* distribuirane platforme je *Kafka* klaster koji čine jedan ili više servera poznati kao posrednici (engl. *broker*). Posrednici su zaduženi za prijem i čuvanje događaja organizovanih u logičke celine nazvane teme (engl. *topic*) čija je svrha grupisanje događaja iste vrste. Servisi koji emituju događaje nazivaju se proizvođači (engl. *producer*) - oni nakon slanja nastavljaju dalje sa radom, ne čekajući na odgovor od drugih servisa, štaviše, oni ni ne moraju da očekuju da će primiti odgovor. Servisi koji očekuju događaje, potrošači (engl. *consumer*), ne uspostavljaju direktnu konekciju sa proizvođačem, već učitavaju događaje sa posrednika kako pristižu. Na jednu temu može pisati više proizvođača, ali i jedan proizvođač može pisati na više tema. Istovremeno, svaka tema može imati više potrošača.

Događaji koje emituje proizvođač dodaju se na kraj toka događaja (engl. *event stream*) na posredniku. Kako su događaji imutabilni objekti, umesto izmene postojećeg, svaka promena se šalje kao novi događaj. Sekvenca događaja se na posredniku čuva trajno, često na disku. Struktura podataka na posredniku zadužena za čuvanje

toka događaja naziva se log događaja (engl. *event log*) i događaji se mogu upisivati samo na njen kraj (engl. *append-only*) [8]. Trajnim čuvanjem događaja u logu, potrošačima koji su u međuvremenu bili nedostupni omogućava se naknadno procesiranje propuštenih događaja, što značajno povećava otpornost sistema na greške i robusnost čitavog sistema. Pored toga, svaki potrošač kontroliše brzinu kojom čita događaje iz loga, što doprinosi elastičnosti sistema prilikom povećanog opterećenja.

U cilju poboljšanja performansi i stepena paralelizma, svaka tema može se podeliti na segmente nazvane particijama (engl. *partition*), čiji broj određuje programer prilikom kreiranja teme. Svaka novopristigla poruka označena je jedinstvenim identifikatorom - ofsetom, koji predstavlja indeks od početka particije i služi za određivanje tačne pozicije poruke u okviru jedne particije (Slika 2.1). Dodatno, particije su numerisane brojevima od 0 do $n-1$, gde je n ukupan broj particija, tako da je moguće jedinstveno odrediti poruku na osnovu trojke tema-particija-ofset [18].



Slika 2.1: Organizacija tema, particija i ofseta u okviru *Kafka* posrednika.

Događaji se šalju temama u okviru *Kafka* poruke koja, osim vrednosti događaja, sadrži ključ, oznaku tačnog vremena nastanka, kao i dodatna zaglavlja sa metapodacima. Sve poruke sa istim ključem biće poslate na istu particiju teme, a odgovornost za implementaciju algoritma kojim se određuje ključ je na proizvođaču. Cilj je kreirati algoritam kojim bi sve particije bile podjednako opterećene kako bi iskorišćenost resursa bila optimalna.

Zahvaljujući činjenici da je na log događaja moguće dodati nove događaje samo na kraj, događaji na jednoj temi su sortirani hronološki. Međutim, ako je tema particionisana na dve ili više particija, *Kafka* garantuje hronološki redosled događaja samo u okviru jedne particije, ali ne i na nivou teme. Ukoliko postoji više proizvođača koji šalju događaje i samo jedan potrošač koji čita događaje sa teme, on će najverovatnije vrlo brzo biti preopterećen. Tada je neophodno skalirati povećanjem instanci mikroservisa potrošača, pri čemu će tada sve instance mikroservisa koje čitaju sa jedne teme formirati grupu potrošača (engl. *consumer group*). Optimalno je da broj potrošača u grupi bude jednak broju particija teme tako da svaki potrošač čita sa tačno jedne particije. Ukoliko postoji više particija nego potrošača u grupi, jedan ili više potrošača mogu da čitaju sa više particija, ali tada nije garantovan hronološki redosled pročitanih događaja u okviru jednog potrošača. U obrnutom slučaju, kada postoji više potrošača u grupi nego particija, slabija je iskorišćenost resursa jer potrošači koji su u višku neće biti vezani ni za jednu particiju, pa samim tim neće učestvovati u čitanju [18].

Na osnovu navedenih karakteristika *Apache Kafka* je veoma moćna platforma za integraciju u mikroservisnim aplikacijama. Otporna je na greške zahvaljujući trajnom čuvanju loga događaja i omogućava skaliranje servisa zahvaljujući mogućnosti particionisanja tema. *Kafka* posrednici su takođe skalabilni - moguće je konfigurisati željeni broj posrednika, pri čemu će sve teme i particije biti replicirane na onoliko instanci posrednika koliko je definisano faktorom replikacije (engl. *replication factor*).

Sinhroni i asinhroni stilovi komunikacije, kao i konkretne tehnologije zasnovane na njima imaju svoje prednosti i mane. Sa jedne strane je jednostavnost i razumljivost sinhronih tehnologija, koja može biti prikladna u situacijama u kojima izvršavanje operacije direktno zavisi od rezultata drugog servisa. Asinhrona komunikacija, sa druge strane, pruža veću fleksibilnost, bolju skalabilnost i slabiju spregnutost među servisima. U praksi se često koriste oba obrasca komunikacije u zavisnosti od problema koji se rešava.

2.3 Sistemi skladištenja podataka

U modernim aplikacijama, sistemi za upravljanje podacima su od ključnog značaja kako za korisnike, tako i za poslovne potrebe. Za korisničke aplikacije, baze podataka omogućavaju funkcionalnosti kao što su kreiranje personalizovanog sa-

držaja, praćenje dostupnosti proizvoda, beleženje finansijskih transakcija i mnoge druge. S druge strane, poslovni korisnici iz postojećih podataka mogu kreirati izveštaje na osnovu kojih mogu donositi poslovne odluke. Načini pristupa podacima između ove dve grupe se razlikuju po tome što aplikativni korisnici kroz interakciju sa aplikacijom indirektno dohvataju, dodaju, menjaju i brišu podatke, dok poslovni korisnici imaju direktan pristup podacima koje koriste isključivo za čitanje.

Upravo iz ovih različitih potreba, razvila su se dva sistema za pristup podacima - transakciona obrada (OLTP - *OnLine Transactional Processing*) i analitička obrada (OLAP - *OnLine Analytical Processing*). Upiti u OLTP sistemima su uglavnom jednostavni, unapred definisani i pozivaju se preko aplikacijskog interfejsa kako bi se korisnici sprečili da pristupaju podacima kojima nemaju prava pristupa ili da kreiraju složene upite koji mogu značajno da utiču na celokupne performanse baze podataka. Rezultujući skup podataka je relativno mali, a vreme izvršavanja upita se meri u milisekundama. Sa druge strane, OLAP sistemi rade sa velikim skupovima podataka koji mogu da budu mereni čak i u petabajtima. Upiti se izvršavaju direktno nad OLAP bazom podataka, korisnici definišu upite *ad hoc* i često se vrši spajanje i agregiranje velikog broja tabela. Usled velike složenosti upita i količine podataka, vreme njihovog izvršavanja je dosta sporije u odnosu na OLTP sisteme [23].

Prilikom implementacije monolitnih sistema, inženjeri moraju da se odluče za korišćenje jedne tehnologije baza podataka, što znači da bi trebalo uzeti u obzir sve kompromise rada u određenoj tehnologiji. Međutim, mikroservisne aplikacije podržavaju korišćenje različitih tehnologija za skladištenje podataka zahvaljujući autonomiji servisa i izolovanosti podataka. Umesto biranja kompromisnog rešenja, bira se ona tehnologija koja je najpogodnija za implementaciju funkcionalnosti mikroservisa. U nastavku sledi opis različitih modela baza podataka, analitičkih sistema, kao i njihovi slučajevi upotrebe.

Relacioni modeli

Koncepti relacionog modela prvi put predstavljeni u radu Edgara Koda 1970. godine doveli su do razvoja danas najpoznatijih sistema za upravljanje bazama podataka. Podaci u relacionim bazama podataka organizovani su u relacije (tabele), svaki podatak predstavljen je kao torka (red u tabeli) i karakterisan atributima (kolonama u tabeli) [23]. Interakcija sa bazom podataka najčešće se ostvaruje putem deklarativnog jezika *SQL*, koji na intuitivan način omogućava uvid u podatke i manipulaciju njima. Struktura podataka unapred je definisana shemom, pri čemu su

podaci najčešće normalizovani. Normalizacijom se smanjuje redundantnost podataka time što se složene relacije razdvajaju na više manjih, poput relacija koje opisuju entitete i njihov međusobni odnos, čime se omogućava efikasnije upravljanje i pristup podacima. Svaki red u tabeli jedinstveno je identifikovan atributom nazvanim primarni ključ koji pomaže pri izvršavanju upita da se ne vrši pretraga cele tabele. Stranim ključevima referiše se na primarni ključ druge tabele čime se opisuje odnos između tabela i omogućava njihovo spajanje radi boljeg razumevanja veza između podataka.

Operacije nad relacionim bazama podataka izvršavaju se u vidu transakcija koje imaju ACID (*Atomicity, Consistency, Isolation, Durability*) svojstva. U praksi to znači da će svaka transakcija biti izvršena u celini ili neće biti izvršena uopšte, pri čemu konkurentne transakcije ne utiču jedna na drugu te da rezultat izvršavanja čuva integritet podataka i ostaje trajno sačuvan. Zbog ovih karakteristika relacione baze su prikladan izbor za sisteme u kojima je korektnost i pouzdanost podataka od presudnog značaja. Usled široke upotrebe razvijena su mnoga rešenja otvorenog koda kao i komercijalna rešenja, od kojih se posebno ističu *PostgreSQL*, *MySQL*, *Oracle Database*, *Microsoft SQL Server*.

Iako su relacione baze podataka decenijama predstavljale dominantno rešenje za upravljanje podacima, razvoj distribuiranih sistema i potreba za obradom velikih količina podataka ukazali su na ograničenja relacionog modela. Strogo definisana shema otežava razvoj u situacijama kada se zahtevi, a samim tim i struktura podataka, menjaju često. Takođe, operacije spajanja nad velikim skupovima podataka mogu imati značajan uticaj na performanse sistema. Horizontalno skaliranje uz očuvanje ACID garancija je jedan od glavni problema kod relacionih sistema. Usled distribucije podataka na više čvorova, postoji mogućnost mrežnih particija, odnosno privremenog gubitka komunikacije između čvorova. Ovo ponašanje formalno je opisano CAP teoremom, koja tvrdi da u prisustvu particionisanja mreže sistem mora da napravi kompromis između konzistentnosti i dostupnosti podataka.

Kao odgovor na ove izazove, razvijene su nerelacione baze podataka koje pružaju mogućnost definisanja fleksibilnijih shema podataka, omogućavaju lakše skaliranje i efikasnije funkcionisanje u distribuiranim sistemima kao što je mikroservisna arhitektura.

NoSQL modeli

Nerelacione baze podataka, poznate kao *NoSQL* baze podataka, obuhvataju skup tehnologija koje definišu različite modele podataka sa fleksibilnijim shemama, omogućavaju lakše skaliranje, visoku raspoloživost i efikasno skladištenje podataka. Neke od najpopularnijih *NoSQL* baza podataka su skladišta dokumenata, parovi ključeva i vrednosti, grafovske baze podataka i sistemi za pretragu.

Baze dokumenata - *MongoDB*

Podaci u bazama dokumenata organizovani su u vidu struktura parova svojstava i vrednosti koje nazivamo dokumentima, najčešće predstavljeni u *JSON* ili *XML* formatu. Najpoznatiji predstavnik ove grupe je *MongoDB* koji podatke čuva u *BSON* formatu - binarnoj reprezentaciji *JSON* dokumenta. *BSON* format osim parova svojstava i vrednosti podatka čuva i informacije o tipu i veličini polja što čini pretragu efikasnijom nego kod *JSON* formata [27]. Česta je praksa da se sva povezana svojstva čuvaju u okviru jednog dokumenta, što obično znači da su podaci denormalizovani. S obzirom na to da operacije spajanja nisu neophodne za čitanje podataka, operacije čitanja su veoma efikasne, dok su operacije ažuriranja podataka manje efikasne usled redundantnosti.

Dokumenti su organizovani u kolekcije, što bi bio ekvivalent tabelama u relacionom modelu, pri čemu dokumenti jedne kolekcije ne moraju imati istu shemu. Shema ne mora biti striktno definisana pri operacijama pisanja, već će tačna shema biti poznata samo nakon čitanja podataka. Ovakav tip sheme naziva se shema pri čitanju (engl. *schema-on-read*), u poređenju sa shemom pri pisanju (engl. *schema-on-write*) karakteristična za relacione baze podataka [23]. *MongoDB* koristi *MQL* (*MongoDB Query Language*) za definisanje upita, koji se u praksi najčešće koriste kroz klijentske biblioteke dostupne u različitim programskim jezicima.

Baze parova ključeva i vrednosti - *Redis*

Baze parova ključeva i vrednosti su najjednostavnije nerelacione baze podataka u kojima je podacima moguće pristupiti isključivo preko ključa ili raspona vrednosti ključa. Jedna od najpoznatijih baza parova ključeva i vrednosti je projekat otvorenog koda *Redis* koji podatke primarno čuva u radnoj memoriji. *Redis* takođe implementira mehanizme otpornosti na greške kojima podatke trajno čuva na disku u vidu snimaka stanja (engl. *snapshot*) ili upisivanjem svih prethodno izvršenih

komandi u posebnu datoteku (*AOF - append-only file*) [32]. Budući da se podaci primarno čuvaju u radnoj memoriji *Redis* omogućava izuzetno efikasan pristup podacima, što ga čini pogodnim izborom za keširanje rezultata upita drugih baza podataka i skupih operacija.

Grafovske baze podataka - *Neo4J*

Odnosi više-više se u relacionom modelu realizuju uvođenjem vezne tabele koje, osim stranih ključeva ka tabelama koji učestvuju u odnosu, mogu takođe imati i sopstvene atribute koji opisuju sam odnos. Vezne tabele su dovoljne za standardne operacije pretrage i filtriranja podataka. Međutim kada je iz odnosa među entitetima neophodno izvesti zaključke o podacima ili pratiti odnose do određene dubine, relacione baze se pokazuju kao neefikasne usled skupih operacija spajanja čija složenost raste sa dubinom pretrage.

U navedenim slučajevima, grafovski model podataka predstavlja adekvatniji izbor, dok grafovske baze podataka omogućavaju skladištenje entiteta kao čvorove, a odnose među njima kao grane grafa. Najpoznatiji predstavnik koji se danas smatra vodećom grafovskom bazom podataka je *Neo4J* projekat otvorenog koda implementiran u programskim jezicima *Java* i *Scala*. Podaci *Neo4J* baze podataka strukturirani su kao grafovi svojstava (engl. *property graph*). Čvorovi grafa predstavljaju entitete i mogu imati svoja svojstva u obliku parova ključeva i vrednosti, kao i jednu ili više labela koje označavaju tip entiteta. Grane grafa predstavljaju odnose između čvorova. Svaka grana je usmerena i definiše početni i krajnji čvor veze, takođe imaju svoja svojstva i tačno jednu labelu koja označava tip veze [28]. Slično kao i kod prethodno opisanih *NoSQL* baza podataka, podaci u *Neo4J* nemaju striktno definisanu shemu čime je otvorena mogućnost dodavanja novih čvorova i grana bez remećenja postojećih.

Deklarativni jezik *Cypher* daje mogućnost pretrage grafova i prepoznavanja obrazaca (engl. *pattern matching*). Upiti se definišu u formatu `(čvor) - [:ODNOS] -> (drugiČvor{svojstvo: vrednost})`, pri čemu se u oblim zagradama navode čvorovi, u uglastim naziv odnosa između čvorova, a u vitičastim zagradama po potrebi se navodi vrednost svojstva koji čvor ili grana moraju da zadovoljavaju. Ovakav vizuelan način definisanja upita omogućava da se na čitljiv i koncizan način opišu složene operacije nad grafovima [29]. Primer upita napisan u *SQL* i *Cypher* jasno pokazuje da je *Cypher* daleko moćniji jezik za izražavanje kompleksnih upita vezanih za pretragu veza između podataka. Upiti iz Listinga 1 i 2 za korisnika sa identifi-

katorom `userId` pronalaze preporuku novog korisnika za praćenje koji najveći broj njegovih već praćenih korisnika takođe prati.

```
MATCH (u1: User {id: $userId}) - [:FOLLOWING] ->
      (u2: User) - [:FOLLOWING] -> (u3: User)
WHERE u1 <> u3
    AND NOT (u1) - [:FOLLOWING] -> (u3)
RETURN u3, count(u2) as numCommonFollowers
ORDER BY numCommonFollowers DESC
```

Listing 1: Pronalaženje preporuka za praćenje preko zajedničkih praćenja - *Cypher*

```
SELECT u.*, COUNT(f2.follower) as num_common_followers
FROM Follow f1 JOIN Follow f2
        ON f1.followee = f2.follower JOIN User u
                ON u.id = f2.followee
WHERE f1.follower = :userId
AND f2.followee != :userId
AND NOT EXISTS ( SELECT 1
                 FROM Follow f3
                 WHERE f3.follower = :userId
                 AND f3.followee = f2.followee
               )
GROUP BY u.id, f2.followee
ORDER BY num_common_followers DESC
```

Listing 2: Pronalaženje preporuka za praćenje preko zajedničkih praćenja - *SQL*

Sistemi pretrage - *Elasticsearch*

Sistemi pretrage su specijalizovani sistemi prvenstveno namenjeni za efikasnu pretragu i rangiranje tekstualnih podataka. Osnovna ideja sistema za pretragu je kreiranje strukture koja liči na indekse u knjigama - indeks sadrži spisak reči koje se pojavljuju u knjizi zajedno sa listom stranica na kojima se mogu pronaći te reči [26]. Biblioteka *Apache Lucene* implementira ovu ideju tako što najpre izvršava tokenizaciju i normalizaciju teksta iz različitih dokumenata, a zatim formira strukturu podataka koja se naziva invertovani indeks (engl. *inverted index*) koji preslikava tokene u listu dokumenata koji ih sadrže.

Elasticsearch je distribuirani sistem za pretragu i analitiku koji interno koristi *Lucene* za funkcionalnost pretrage, definiše *REST* interfejs za interakciju sa sistemom, dolazi sa skalabilnom arhitekturom i mogućnostima za skladištenje podataka. Podaci se čuvaju kao dokumenti u *JSON* formatu u kolekcijama koje se nazivaju indeksi (engl. *index*). Za pretragu, filtriranje i agregiranje podataka *Elasticsearch* pruža mogućnost korišćenja različitih jezika za upite, među kojima je najizražajniji i najfleksibilniji *Query DSL* koji se deklariše u *JSON* formatu.

Objektna skladišta - *MinIO*

Podatke poput slika, video i audio snimaka, podataka sa senzora je često veoma teško predstaviti u strukturiranom formatu, pa se oni čuvaju u binarnom ili izvornom formatu. Pošto je reč o polustrukturiranim ili nestrukturiranim podacima koji nastaju velikom brzinom od strane aplikacija, njihovo čuvanje u relacionim i nerelacionim bazama podataka, iako izvodljivo, nije najefikasniji način. Objektna skladišta predstavljaju model arhitekture skladištenja nestrukturiranih velikih podataka, gde se podaci čuvaju kao objekti - imutabilne celine sa jedinstvenim identifikatorom i metapodacima koji služe za lociranje objekata prilikom čitanja i opisivanje sadržaja objekta [20]. Ukoliko su objekti serijalizovani u binarnom formatu, objektna skladišta se još nazivaju i blob (***B**inary **L**arge **Ob**ject*) skladišta.

Popularnost objektnih skladišta rasla je sa potrebama obrade velikih podataka i razvoja aplikacija u oblaku. Glavna prekretnica bila je pojava *Amazon S3* objektnog skladišta u oblaku kojim je definisan standard aplikacijskih interfejsa za objektna skladišta. Pored Amazona, i drugi provajderi okruženja u oblaku nude rešenja za objektno skladištenje kao što su *Azure Blob Storage* i *Google Cloud Storage* [33]. Takođe, postoje i brojna rešenja otvorenog koda kao što su *MinIO*, *OpenStack Swift* i *Ceph*.

MinIO distribuirano objektno skladište je projekat otvorenog koda koji je u potpunosti kompatibilan sa *S3* aplikacijskim interfejsom i nezavisan je od okruženja u oblaku. Kompatibilnost sa *S3* aplikacijskim interfejsom podrazumeva da sve klijentske biblioteke namenjene za rad sa *S3* skladištem takođe mogu da se koriste i za rad sa *MinIO* skladištem, kao i da je eventualna zamena *MinIO* skladišta *S3* skladištem jednostavna. Nezavisnost od okruženja u oblaku omogućava isporuku *MinIO* skladišta u svim okruženjima u oblaku, pruža mogućnost lokalnog razvoja aplikacija i olakšava razvoj jer programeri nisu u obavezi da poznaju interfejse specifične za skladišta konkretnih okruženja u oblaku. Objekti u *MinIO* organizo-

vani su u logičke celine nazvane kofe (engl. *bucket*). U okviru jedne kofe objekti moraju da imaju jedinstvene identifikatore, dok nazivi kofa moraju biti jedinstveni na nivou celog skladišta. Pretraga objekata u skladištu vrši se pomoću ključa čija se semantika poklapa sa putanjama do datoteke u fajl sistemima [33], na primer `s3://moja-kofa/datoteka.json`. *MinIO* pruža mogućnost jednostavnog skaliranja i replikacije podataka za aplikacije velikog opterećenja.

Sistemi skladištenja podataka za analitičku obradu

Razlika između transakcionih i analitičkih sistema ogleda se pre svega u strukturi skladišta, nameni i izvoru podataka. U operacionim sistemima podaci potiču uglavnom od strane korisnika aplikacije, dok se u analitičkim podacima radi sa izvedenim podacima iz operacionih sistema. Ovo znači da su podaci u analitičkim sistemima redundantni jer dupliraju postojeće informacije, ali je njihovo razdvajanje od transakcionih sistema i dalje neophodno kako bi se smanjilo opterećenje produkcionih sistema i poboljšale performanse analitičkih sistema. U nastavku su opisana najbitnija skladišta za analitičku obradu.

Skladišta podataka - ClickHouse

Proces kojim se podaci iz OLTP sistema izvlače, transformišu za analitičke potrebe i na kraju učitavaju u analitičke sisteme naziva se *ETL* (engl. *Extract Transform Load*). Podaci se prečišćavaju i transformišu u oblik namenjen za efikasno pretraživanje i analiziranje [33]. Tako jako strukturirani podaci se na kraju čuvaju u centralizovanim sistemima nazvanim skladišta podataka. Skladište podataka (engl. *data warehouse*) je sistem za skladištenje i organizaciju strukturiranih podataka prikupljenih iz različitih izvora radi njihove analitičke obrade. Podaci se često prikupljaju periodično, u paketima velikog obima radi istorijske analize (na primer, kreiranje izveštaja o prodaji na dnevnom, mesečnom ili godišnjem nivou). Ova vrsta prikupljanja i obrade podataka naziva se paketna obrada (engl. *batch processing*). Postoji veliki broj rešenja za skladištenje podataka od kojih se izdvajaju projekti otvorenog koda *Apache Hive* i *DuckDB*, a od skladišta podataka u oblaku *Snowflake*, *Amazon Redshift*, *Google BigQuery*, *Azure Synapse Analytics*.

Paketna obrada nije prilagođena modernim sistemima od kojih se očekuje da serviraju analitike ili pokreću učenje modela mašinskog učenja sa sekundama ili milisekundama zakašnjenja u odnosu na kreiranje podataka. U navedenim slučaje-

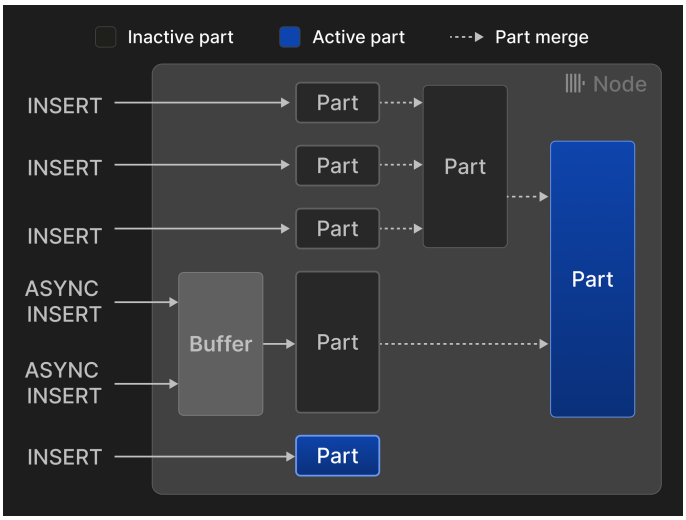
vima, pogodnije je podatke posmatrati kao jedan neprekidni tok i analizu obavljati u realnom vremenu. Prikupljanje i obrada podataka u realnom vremenu naziva se obrada tokova (engl. *stream processing*), dok postoje skladišta podataka koja su prilagođena čuvanju i izvršavanju upita nad brzo pristižućim podacima - OLAP u realnom vremenu (engl. *real-time OLAP*). Primeri takvih skladišta su *ClickHouse*, *Apache Druid* i *Apache Pinot* [23].

ClickHouse je sistem za upravljanje bazama podataka za OLAP u realnom vremenu. Na logičkom nivou nije uočljiva razlika između relacionih OLTP baza podataka i *ClickHouse* jer oba sistema organizuju podatke u tabele i upiti se definišu kao *SQL* upiti. Međutim, na fizičkom nivou se razlikuje način organizacije podataka - tabele se čuvaju kao kolekcija kolona umesto redova. Razlog ovakve organizacije podataka jeste u činjenici da se u analitičkim upitima retko čitaju vrednosti iz celih redova, već se najčešće vrše operacije nad kolonama kao što su agregacija i filtriranje. Model skladištenja podataka u kojem su podaci iz istih kolona organizovani zajedno naziva se kolonski orijentisani model. Takođe, za razliku od relacionih baza podataka, definisanje primarnog ključa u *ClickHouse* nije obavezno, već će podrazumevani primarni ključ biti kolona (ili lista kolona) koja se definiše u `ORDER BY` naredbi i naziva se ključ sortiranja (engl. *sorting key*). Pored drugačije organizacije podataka, visokim performansama u *ClickHouse* doprinosi specijalizovana mašina za skladištenje velike količine podataka velikom brzinom pristizanja nazvana `MergeTree`. Svaka tabela podeljena je na imutabilne delove sortirane u odnosu na ključ sortiranja nad kojima se izvršavaju upiti. Prilikom dodavanja novih podataka u tabelu, kreira se novi deo. Kako bi se sprečilo preterano fragmentisanje tabele, u pozadini se vrši operacija spajanja, prikazana na Slici 2.2 kojim se formiraju veći delovi od manjih [11].

Druge mašine za skladištenje iz `MergeTree` familije tokom operacije spajanja rade obično zahtevne operacije agregacije i transformacije podataka koje su česte za analitičke baze podataka. Na primer, `ReplacingMergeTree` mašina uklanja podatke sa istim vrednostima ključa sortiranja, dok `SummingMergeTree` sumira numeričke podatke prilikom operacije spajanja. Unapred izvršena agregacija tokom operacije spajanja značajno unapređuje performanse izvršavanja upita.

Jezera podataka

Analitički sistemi koji koriste skladišta podataka susreću se sa istim problemom kao i transakcioni sistemi sa relacionim bazama podataka - problem rigidne struk-



Slika 2.2: Prikaz spajanja delova tabele u *ClickHouse* mašini `MergeTree` [11]

ture podataka. Umesto formiranja striktne sheme podataka koja nije uvek pogodna za potrebe mašinskog učenja ili analize tekstualnih podataka, podatke je pogodnije čuvati u originalnom formatu u skladištu koje može da podrži podatke različitih formata kao što je objektno skladište. Arhitektura skladištenja strukturiranih, nestrukturiranih ili polustrukturiranih podataka na jednom zajedničkom mestu naziva se jezero podataka (engl. *data lake*).

Prva jezera podataka oslanjala su se na *Hadoop* distribuirani fajl sistem (*HDFS*) iz grupe softverskih projekata *Apache Hadoop* za analitiku nad velikim podacima. Usled kompleksnosti podešavanja prilagođenih *Hadoop* klastera, ali i porasta popularnosti okruženja u oblaku i objektnih skladišta, jezera podataka implementirana su u okruženjima u oblaku. Učitavanje u jezero podataka može da se posmatra kao međukorak u analitičkoj obradi, nakon kojeg može da sledi učitavanje u skladište podataka *ETL* procesom ili učitavanje relevantnih podataka u sistem za mašinsko učenje. Koncept *ETL* obrade je sada obuhvaćen širim pojmom pod nazivom protok podataka (engl. *data pipeline*) tako da obuhvata arhitekturu, sisteme i procese koji prenose i transformišu podatke u analitičkim sistemima. Jezera podataka omogućila su da se razdvoje koncepti skladištenja i obrade podataka koji su bili čvrsto vezani kod skladišta podataka. Posledica ovog razdvajanja je pojava tehnologija za distribuiranu obradu velikih podataka od kojih poseban značaj imaju *MapReduce*, *Apache Spark*, *Apache Flink*.

Data Lakehouse

Šira upotreba jezera podataka dovela je do drastičnog rasta količine učitanih podataka. Ubrzo je postalo teško ispratiti koji podaci su učitani, kako doći do određenih podataka i kako su strukturirani podaci. Performanse analitičkih upita su takođe značajno opale kao posledica nedostatka strukturiranja i optimizacije podataka, što ranije nije bio problem kod skladišta podataka [33]. *Data lakehouse* arhitektura osmišljena je kako bi se kombinovale prednosti skladišta podataka u vidu poboljšanih performansi za analitičke upite i ACID svojstva zajedno sa prednostima jezera podataka u vidu podrške za nestrukturirane podatke, fleksibilnosti i smanjenih troškova. Ovo se postiže uvođenjem dodatnog nivoa apstrakcije nad skladištem jezera podataka pomoću tehnologije formata tabela. Podaci se fizički čuvaju u kolonski orijentisanim fajl formatima, kao što su *Apache Parquet* i *ORC* (*Optimized Row Columnar*), koji omogućavaju efikasnu kompresiju i selektivno čitanje kolona relevantnih za analitičke upite [15]. Formati tabela nadograđuju ove fajl formate tako što datoteke organizuju u logičke tabele, čuvajući metapodatke o pripadnosti datoteka određenim tabelama, informacija o shemi, strukturi particija, kao i evidenciju transakcija. Upravo su navedeni metapodaci oni koji daju ACID garancije za transakcije u *data lakehouse* arhitekturi. Najpoznatiji formati tabela su *Apache Iceberg*, *Delta Lake*, *Apache Hudi*.

Apache Iceberg je format tabela otvorenog koda razvijen od strane kompanije *Netflix* 2017. godine. Ključna razlika između *Iceberg* i ostalih formata tabela jeste u hijerarhijskoj organizaciji metapodataka koja omogućava efikasnije planiranje upita i bolje performanse prilikom obrada velikih skupova podataka. Zahvaljujući činjenici da se informacije o strukturi particija tabela čuvaju u metapodacima, olakšana je promena particionisanja tabele jer se ne izvršava fizička reorganizacija datotaka [4]. Hronološka izmena tabela prati se u vidu slika koje se kreiraju nakon svake transakcije. Slike garantuju atomičnost transakcija i omogućavaju povratak na proizvoljno prethodno stanje tabele. Lokacija metapodataka svake tabele čuva se u eksternom katalogu koji je neophodna komponenta za integraciju *Iceberg* tabela sa tehnologijama za obradu podataka.

Apache Spark

Apache Spark je distribuirana platforma otvorenog koda za obradu podataka većinski u radnoj memoriji sa podrškom za paketnu obradu i obradu tokova. Glavne

komponente Spark arhitekture su rukovodilac (engl. *driver*) i radnici (engl. *worker*). Uloga rukovodioca je da pokrene `main()` funkciju, obezbedi dovoljne resurse za izvršavanje zadataka i podeli posao radnicima. Na svakom radničkom čvoru se pokreće jedan ili više procesa izvršilaca (engl. *executor*) koji izvršavaju dodeljeni kôd i rezultate vraćaju rukovodiocu. *Apache Spark* je distribuirani okvir, što znači da se aplikacije izvršavaju paralelno na više računara (čvorova). Za upravljanje resursima u okviru klastera računara, *Apache Spark* interaguje sa menadžerom klastera. `SparkSession` klasa u okviru rukovodioca se povezuje sa menadžerom klastera i inicijalizuje izvršioce na radničkim čvorovima u okviru klastera. *Spark* rukovodilac aplikaciju deli na više poslova (engl. *job*), a zatim je svaki posao podeljen na jedan ili više stadijuma (engl. *stage*) i na kraju na više zadataka (engl. *task*). `SparkSession` šalje kôd aplikacije i individualne zadatke izvršiocima. Maksimalni broj zadataka na jednom radničkom čvoru jednak je broju jezgara procesora čvora.

Distribuirani skupovi podataka koji se raspoređuju kroz čvorove računarskog klastera u cilju paralelne obrade u okviru Apache Spark platforme apstrahovani su strukturom podataka poznatom kao otporni distribuirani skup podataka (engl. *Resilient Distributed Dataset – RDD*). U Spark okviru prilikom obrade podataka se primenjuje model lenjog izvršavanja podelom operacija na transformacije i akcije. Navođenjem transformacija na podacima opisuje se tok operacija prilikom obrade podataka bez pokretanja računanja i to su obično one koje čitaju, menjaju i agregiraju podatke. *RDD* je imutabilna struktura, pa transformacijama nastaju novi *RDD*. Ovo daje mogućnost rekonstrukcije stanja u slučaju greške ili otkaza u nekom koraku praćenjem niza navedenih transformacija. Navođenjem akcije, kao što su upis podataka, prikazivanje rezultata ili prebrojavanje elemenata u strukturi, pokreće se izvršavanje celokupnog plana obrade.

Strukture koje proširuju *RDD* dodavanjem strukturnih informacija o shemi podataka, definišu izražajniji programski interfejs i koje se češće koriste od *RDD* su tablice podataka (*DataFrame*) i skupovi podataka (*Dataset*) [14]. Obe strukture programerima intuitivnije, u tabelarnom formatu, predstavljaju podatke [14]. Razlika između ove dve strukture je što kod *Dataset* strukture, provera tipova je statička, u vremenu kompilacije, pa samim tim ne postoji podrška za njih u jezicima *Python* i *R*. Korišćenjem *DataFrame* ili *Dataset* apstrakcija, *Apache Spark* može da optimizuje redosled navedenih transformacija u cilju efikasnije obrade, što nije slučaj kod *RDD* jer njegov programski interfejs nižeg nivoa podrazumeva prosleđivanje anonimnih funkcija usled čega namera programa nije poznata u vremenu kompilacije. Listinzi

3 i 4 ilustruju razliku u čitljivosti između *RDD API*-a i *DataFrame API*-a.

```
val rdd = spark.sparkContext.textFile("people.json")
        .map(line => mapper.readValue(line, classOf[Map[String, Any]]))
        .filter(user => user("age").asInstanceOf[Int] >= 18)
        .map(user => (user("country").asInstanceOf[String], 1))
        .reduceByKey(_ + _)
```

Listing 3: Prebrojavanje punoletnih osoba iz fajla pomoću *RDD API*-a

```
val df = spark.read.json("people.json")
        .filter($"age" >= 18)
        .groupBy("country")
        .count()
```

Listing 4: Prebrojavanje punoletnih osoba iz fajla pomoću *DataFrame API*-a

Apache Spark omogućava obradu podataka u okviru različitih domena primene kroz svoje specijalizovane biblioteke: *Spark SQL* za obradu strukturiranih podataka, *Spark Structured Streaming* za obradu tokova podataka u realnom vremenu, *MLlib* za mašinsko učenje i *GraphX* za obradu grafova.

Savremene aplikacije često zahtevaju mogućnost obrade podataka u periodičnim intervalima kao kod paketne obrade, ali i obradu podataka u realnom vremenu. *Spark SQL* i *Spark Structured Streaming* su od ključnog značaja za paketnu obradu i obradu tokova podataka u realnom vremenu. Dok *Spark SQL* učitava podatke iz strukturiranih izvora i interno ih predstavlja u tabelarnom obliku, *Spark Structured Streaming* proširuje mogućnosti *Spark SQL* modula omogućavajući obradu podataka koji kontinuirano pristižu iz sistema za razmenu poruka i tokova podataka. Zahvaljujući tome, isti *Dataset*/*DataFrame* interfejs i *SQL* upiti mogu se koristiti i u paketnoj obradi i u obradi tokova podataka u realnom vremenu. Tokovi podataka se posmatraju kao jedna neograničena tabela na čiji kraj se dodaju novi podaci kako pristižu u sistem [6]. Upiti nad tokovima se primenjuju samo nad novopristiglim podacima u navedenoj neograničenoj tabeli koji se proveravaju u unapred definisanom vremenskom intervalu koji može biti od svega par sekundi do nekoliko minuta. Ulazni podaci dele se u mikro-pakete (engl. *micro-batch*) na osnovu zadatog vremenskog intervala obrade.

Tehnologije za isporuku aplikacija

Docker

Docker je platforma otvorenog koda koja služi za isporuku softvera u okviru kontejnera (engl. *container*). Kontejneri omogućavaju organizaciju koda i njegovih zavisnosti u jednu logičku celinu tako da pružaju minimalni okvir u kome se može pokrenuti kod. Kontejneri pružaju apstrakciju nad različitim okruženjima u okviru kojih se mogu pokrenuti aplikacije, čime se za isporučeni softver garantuje konzistentno funkcionalno ponašanje u svim okruženjima, bilo da je to okruženje za razvoj, testiranje ili produkciju.

Aplikacije pokrenute u okviru različitih kontejnera su izolovane, što znači da aplikacije u izvršavanju nemaju uticaj jedna na drugu i eliminisani su potencijalni konflikti između zavisnosti aplikacija. Kontejneri dele jezgro operativnog sistema mašine na kojoj se pokreću, dok istovremeno mogu sadržati različita radna okruženja (na primer, različite *Linux* distribucije). Zahvaljujući ovoj činjenici, pokretanje kontejnera je značajno brže od drugih rešenja za isporuku aplikacija, kao što su virtuelne mašine koje zahtevaju pokretanje celokupnog operativnog sistema. Kontejneri se takođe mogu skalirati u slučajevima velikog radnog opterećenja obezbeđivanjem dodatnih resursa i pokretanjem više instanci istog kontejnera. Zahvaljujući navedenim karakteristikama olakšane prenosivosti softvera, izolovanosti aplikacija, brzine pokretanja i skalabilnosti, kontejneri su stekli veliku popularnost kao način isporuke mikroservisnih aplikacija.

Kreiranje kontejnera uz pomoć *Docker* alata moguće je zahvaljujući *Docker* serveru (engl. *daemon*), dok korisnici interaguju sa *Docker* klijentom koji prosleđuje korisničke instrukcije serveru. Opisivanje kontejnera počinje definisanjem *Docker* slike (engl. *image*) u datoteci koja se naziva `Dockerfile` i predstavlja šablon koji opisuje strukturu i sadržaj kontejnera. Pri tome, moguće je kreirati slike od početka ili koristiti postojeće slike dostupne putem platforme *Docker Hub*, sa opcionalnim proširivanjem njihovih definicija [9]. Pokretanjem komande `docker run` instanciraju se kontejneri na osnovu specifikacija iz *Docker* slika.

Pomoćni alat uz *Docker* koji se često koristi prilikom razvoja mikroservisnih aplikacija je *Docker Compose*. *Docker Compose* pojednostavljuje isporuku višeservisnih aplikacija čija je celokupna konfiguracija objedinjena u jedinstvenoj datoteci `docker-compose.yml` u YAML[1] formatu. Umesto individualnog pokretanja svakog

[1]YAML (YAML Ain't Markup Language, originalno Yet Another Markup Language) je jezik za

servisa, svi servisi se pokreću zajedno izvršavanjem komande `docker-compose up`. *Docker Compose* je veoma koristan alat za definisanje i pokretanje više servisa odjednom, međutim porastom broja servisa gde svaki ima različite potrebe za resursima ili skaliranjem kako se radno opterećenje menja, konstantne manuelne izmene su neefikasne.

Kubernetes

Kubernetes je platforma otvorenog koda koja služi za upravljanje kontejnerizovanim aplikacijama. *Kubernetes* omogućava automatsko skaliranje i oporavak od grešaka pokretanjem novog kontejnera u slučaju kvara postojećih. Pored automatizacije, *Kubernetes* olakšava otkrivanje servisa i balansiranje opterećenja koje su takođe bitne karakteristike u mikroservisnim aplikacijama. *Kubernetes* pruža apstrakciju nad hardverom tako da upravljanje i isporuka aplikacija izgleda isto u slučajevima rada na jednom lokalnom računaru ili u klasteru računara. *Kubernetes* klaster čine kontrolna ravan (engl. *control plane*) i radni čvorovi (engl. *worker nodes*). Kontrolna ravan upravlja klasterom, definiše aplikacijski interfejs koji programeri koriste za definisanje novih *Kubernetes* objekata, kreira nove *Kubernetes* objekte na osnovu definicije i čuva stanje klastera zajedno sa postojećim objektima u svom internom skladištu podataka. Komponente radnih čvorova pokreću aplikacije, nadgledaju izvršavanje aplikacija, komuniciraju sa kontrolnim čvorom i balansiraju opterećenje aplikacija. *Kubernetes* objekti drže informacije o stanju klastera i opisuju određenu funkcionalnost sistema.

Najmanja jedinica isporuke u okviru *Kubernetes* platforme je čaura (engl. *Pod*) koja može da sadrži jedan ili više kontejnera pri čemu će deliti kritične resurse kao što su skladište i mreža [24]. Iako u jednoj čauri može biti više kontejnera, uobičajena praksa je da sadrži samo jedan kontejner. One se najčešće ne kreiraju samostalno već se njihov životni ciklus prepušta objektima višeg nivoa kao što je objekat isporuke (engl. *Deployment*). Isporuke upravljaju čaurama i primenjuju strategije ažuriranja aplikacije postepenom zamenom novim verzijama čaura bez prekida dostupnosti servisa, uz mogućnost povratka na prethodno stanje u slučaju neuspešne isporuke.

Kubernetes klaster definiše virtuelnu mrežu u kojoj svaka čaura ima svoju jedinstvenu IP adresu čime je omogućena međusobna komunikacija. Međutim, kako je životni ciklus čaura ograničen, a samim tim se IP adrese čaura često menjaju,

obeležavanje koji se često koristi za definisanje konfiguracionih datoteka. Karakteriše ga minimalni stil i lako čitljiva struktura.

definiše se servis objekat (engl. *Service*) koji apstrahuje mrežu tako da se komunikacija izvršava preko servisa. Servisi takođe implementiraju funkcionalnost otkrivanja servisa i balansiranje opterećenja među čaurama.

Objekti koji omogućavaju razdvajanje konfiguracionih parametara u cilju olakšavanja prenosivosti i isporuke aplikacije u različita okruženja su konfiguracione mape (engl. *ConfigMap*). Slično konfiguracionim mapama, objekti tajni (engl. *Secret*) se koriste za definisanje konfiguracionih parametara osetljivog karaktera kao što su lozinke, tokeni za autentifikaciju ili sertifikati. Za razliku od konfiguracionih mapa, tajne se čuvaju enkodirane. Konfiguracije i tajne mogu da se koriste u aplikaciji preko dodeljenih promenljivih okruženja.

Glava 3

Implementacija

Aplikacija *Bonds* [39] je društvena mreža razvijena u okviru master rada, inspirisana postojećim popularnim mrežama kao što su *X* (ranije *Twitter*), *Instagram*, *Facebook*, *LinkedIn*. *Bonds* implementira prethodno navedene koncepte i koristi opisane alate sa ciljem da demonstrira kako zahtevi sistema utiču na definiciju arhitekture i izbor tehnologija u cilju održavanja visokih performansi, skalabilnosti i otpornosti na greške kao faktore od velikog značaja u kontekstu aplikacija koje rade sa velikim količinama podataka. Celokupna arhitektura aplikacije zasnovana je na mikroservisima pri čemu svaki mikroservis ima jasno određenu odgovornost na osnovu funkcionalnosti i podataka iz domena. Aplikacija takođe sadrži elemente arhitekture vođene događajima s obzirom na to da je asinhrona komunikacija implementirana kroz log događaja.

Aplikacija se može podeliti u dva dela: transakcioni i analitički deo. Fokus transakcionog dela aplikacije je na implementaciji glavnih funkcionalnosti društvenih mreža od kojih su najbitnije upravljanje korisničkim profilima, kreiranje i interakcija sa objavama korisnika, praćenje korisnika i pretraga postojećih korisnika i objava. Analitički deo koristi podatke o objavama i korisnicima iz transakcionog sistema kako bi izveo analitike kojima korisnici mogu da prate rast svojih pratilaca i performanse objava kroz vreme. Ovaj deo se konceptualno razlikuje od transakcionog po arhitekturi skladištenja i modelu obrade podataka.

3.1 Pregled mikroservisa iz transakcionog dela

1. **Korisnici (User Service)** - Sadrži osnovne informacije o korisnicima i omogućava jednu od glavnih funkcionalnosti na društvenoj mreži - praćenje kori-

snika.

2. **Objave (Post Service)** - Servis koji sadrži najbitnije funkcionalnosti aplikacije. Omogućava kreiranje novih objava, učitavanje objava pratilaca korisnika, reagovanje na objave u vidu komentarisanja ili sviđanja.

3. **Obaveštenja (Notification Service)** - Zadatak ovog servisa je da obavesti korisnika u trenutku kada dobije novo sviđanje ili komentar na objavu.

4. **Pretraga (Search Service)** - Pruža mogućnost pretrage nad postojećim objavama i korisnicima u aplikaciji.

5. **Autentifikacija (Auth Service)** - Neophodan servis za inicijalno kreiranje korisničkih naloga. Osigurava pristup ostalim servisima i njihovim podacima onemogućavajući funkcionalnosti aplikacije za neprijavljene korisnike.

6. **Mrežni prolaz (Gateway)** - Predstavlja ulaznu tačku celog sistema. Definiše dostupne rute i usmerava klijente na odgovarajući servis. Vrši validaciju tokena za autentifikaciju u cilju ograničavanja pristupa servisima neprijavljenim korisnicima.

Dijagram kontejnera C4 modela[1] opisanih servisa predstavljen je na Slici 3.1.

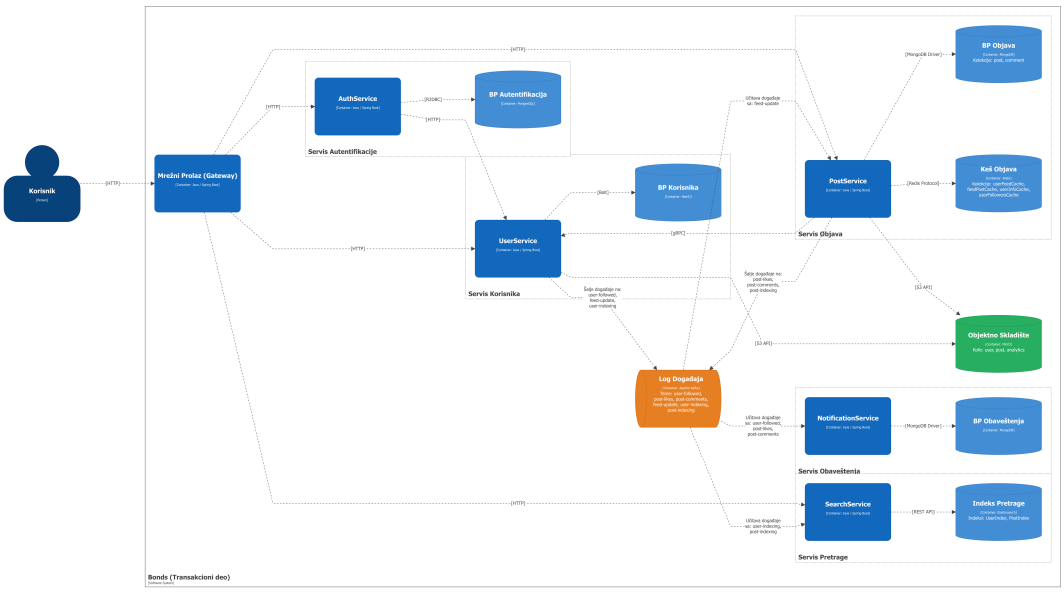
U nastavku sledi opis specifičnosti implementacije svakog od navedenih mikroservisa. Poseban akcenat biće dat na odabiru modela skladištenja podataka ili na specifičnosti implementacije određenih funkcionalnosti.

3.2 Implementacija transakcionog dela aplikacije

Mikroservisi u ovom delu implementirani su kao odvojeni projekti u programskom jeziku *Java*, uz pomoć *Spring Boot* okvira u reaktivnom stilu sa *WebFlux* bibliotekom. Svaki mikroservis definiše *REST API* koji klijenti mogu da pozivaju radi izvršavanja određene funkcionalnosti. Organizacija paketa u okviru projekata odrađena je na sličan način tako da postoje paketi koji se javljaju u svim servisima:

- **config** - Konfiguracione klase koje definišu podešavanja za integraciju eksternih tehnologija i biblioteka koje mikroservis koristi (npr. klijenti za baze podataka, sistemi za razmenu poruka, objektna skladišta)

[1] Zvanična veb stranica: https://c4model.com/



Slika 3.1: Dijagram kontejnera transakcionog dela aplikacije

- **model** - Klase koje predstavljaju preslikavanje domenskog modela za skladištenje podataka
- **repository** - Sloj pristupa podacima koji sadrži interfejse koji implementiraju različite apstrakcije `Repository` interfejsa iz *Spring Data* modula. `Repository` interfejsi služe za direktnu interakciju sa bazom podataka. *Spring Data* stavlja na raspolaganje različite apstrakcije Repository interfejsa u zavisnosti od baze podataka (npr. `R2dbcRepository` za relacione baze podataka, `ReactiveNeo4jRepository`, `ReactiveMongoRepository`)
- **service** - Sadrži interfejse i klase koje implementiraju poslovnu logiku servisa. Klase u okviru ovog paketa koordinišu pozive ka sloju za pristup podacima ili ka eksternim servisima kako bi izvršile jednu poslovnu operaciju, čime predstavljaju posredni sloj između prijema zahteva i pristupa podacima.
- **controller** - Definiše *REST API* za krajnjeg korisnika i poziva odgovarajuće metode iz service paketa radi odgovora na zahtev korisnika.
- **dto** - Klase koje se koriste za komunikaciju između servisa ili slojeva celokupnog sistema

- **resources** - Sadrži `application.yaml` fajl koji centralizuje podešavanja neophodna za okruženje aplikacije. U okviru ove datoteke mogu se izmeniti podrazumevana podešavanja servera aplikacije i definisati osnovna podešavanja za integraciju sa eksternim tehnologijama. Česta praksa je da se osnovna podešavanja definišu u okviru `application.yaml` fajla, dok se programska konfiguracija, kao što su definisanje Spring komponenti i podešavanje ponašanja biblioteka, implementira u okviru `config` paketa.

Pored projekata mikroservisa, definisani su i pomoćni projekti `common` i `common-proto` namenjeni da budu korišćeni kao biblioteke u ostalim mikroservisima. Konkretno, `common` sadrži pomoćne biblioteke za validaciju autentifikacionih tokena i implementira klijenta za interakciju sa *MinIO* objektnim skladištem, dok `common-proto` definiše strukture podataka u vidu proto fajlova koje se koriste u servisima koji komuniciraju putem *gRPC* protokola. Izdvajanjem ovih definicija u zasebne projekte izbegava se dupliranje koda u servisima koji koriste iste funkcionalnosti ili dele iste *gRPC* ugovore.

Korisnici

Servis korisnika čuva osnovne informacije koje opisuju korisnika: jedinstveno korisničko ime (username); alternativno korisničko ime (alias); biografija (bio); zemlja (country).

Svrha alternativnog korisničkog imena jeste da pruži mogućnost korisnicima da promene naziv profila. Kako korisničko ime mora biti jedinstveno u okviru sistema, nije moguće dozvoliti njegovu izmenu, pa alias služi kao promenljiva alternativa. Korisnik može dodati kratak opis o sebi (bio), kao i zemlju u kojoj se trenutno nalazi (country).

Pored navedenih informacija, korisnici mogu da dodaju profilnu sliku. Zbog problema opisanih u prethodnoj glavi oko skladištenja multimedijalnih podataka, profilne slike se čuvaju u objektnom skladištu *MinIO*. Slike mogu da se dodaju preko *REST API* metode kao tip `multipart/form-data`, a zatim se u pozadini koristi klijent koji direktno komunicira sa *MinIO* skladištem kako bi se slika sačuvala kao objekat. U sekciji o *MinIO* objektnom skladištu napomenuto je da je *MinIO* kompatibilan sa *AWS S3* aplikacijskim interfejsom što u ovom slučaju omogućava korišćenje zvanične *Java SDK* biblioteke za *AWS* servise. Iako *MinIO* ima svoju *Java* biblioteku, *AWS Java SDK V2* asinhroni klijent za ulazno-izlazne operacije

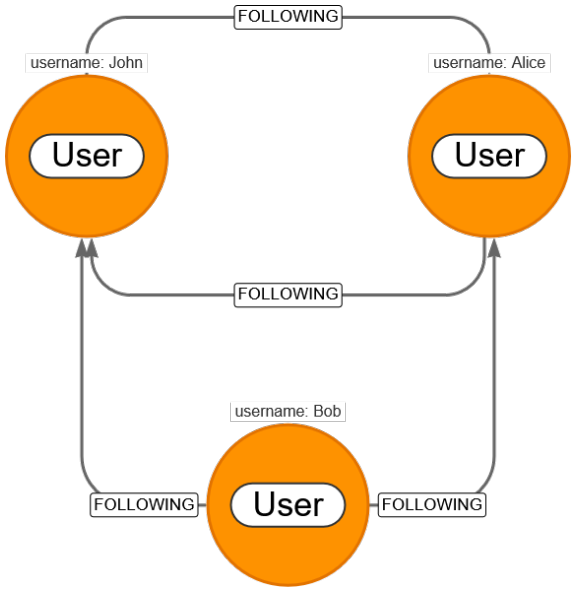
se oslanja na *Netty* server [2], što ga čini boljim izborom za *Spring Boot WebFlux* aplikacije koje se takođe oslanjaju na *Netty*. Pre dodavanja novih objekata u skladište, potrebno je da postoji kofa u koju će se smeštati objekti kreirani u okviru servisa korisnika kako bi se obezbedila izolovanost i kako ne bi došlo do konflikta u nazivima objekata iz različitih servisa.

U bazi podataka je dovoljno da se čuva ključ objekta pomoću kojeg se može dohvatiti objekat iz skladišta. Moguće je dohvatiti objekat iz skladišta na osnovu ključa, naziva kofe i neophodnih kredencijala za skladište, a zatim objekat prevesti u binarni format i takvog ga vratiti klijentu. Međutim, ponovo se nameće problem velikih binarnih objekata koji bi tada trebalo da se prenose preko mreže kao deo odgovora *REST API* poziva. Umesto toga, operacija dohvatanja datoteka može se ostaviti klijentu direktno. Objekti iz objektnog skladišta su po pravilu privatni i direktan pristup van servera aplikacije je onemogućen iz bezbednosnih razloga. U tom slučaju, objekti se dele preko unapred potpisanog URL-a (engl. *presigned URL*) [3]. Kako bi se sprečio neograničen pristup objektu u slučaju da URL dospe u posed neovlašćenog korisnika, svaki unapred potpisan URL važi samo u ograničenom vremenskom intervalu nakon čijeg isteka pristup objektu putem tog URL-a više nije moguć. Svaki put kad klijent zatraži sliku, kao deo odgovora biće uključen unapred potpisan URL koji je tipa obične niske, a samim tim i značajno efikasniji za slanje preko mreže od potencijalno velikih binarnih odgovora.

Sledeća bitna funkcionalnost servisa korisnika je mogućnost praćenja drugih korisnika i čuvanje informacija o odnosima praćenja. Svaki korisnik može da prati veći broj korisnika, ali i jedan korisnik može da bude praćen od strane više drugih korisnika. Ovo ukazuje da će model podataka zahtevati postojanje odnosa više-više. U ranijoj sekciji o nerelacionim bazama podataka opisan je problem modeliranja odnosa više-više i kao dobro rešenje navedene su grafovske baze podataka umesto kreiranja vezne tabele u relacionom modelu. Graf korisnika sadržaće čvorove sa svojstvima koji označavaju korisnike, pri čemu će svojstva činiti do sada navedene informacije o korisnicima. Grane grafa su usmerene i označavaju smer odnosa praćenja između korisnika. Graf korisnika se može grafički predstaviti kao na Slici 3.2.

U implementaciji korišćena je grafovska baza *Neo4J*, a za integraciju Neo4J baze podataka sa Spring okvirom korišćena je *Spring Data Neo4J* biblioteka. Mapiranje grafa u kodu predstavljen je u Listingu 5.

Svaki čvor sadrži jedinstveni identifikator u formatu UUID (*Universally Unique*



Slika 3.2: Graf odnosa između korisnika, sa prikazanim `username` svojstvom.

IDentifier) niske. Odnos `FOLLOWING` se mapira u atribut `following` kao kolekcija korisnika koje trenutni korisnik prati, dok je usmerenje veze izlazno u odnosu na trenutni čvor. Ovakav model omogućava jednostavno dohvatanje svih korisnika koje trenutni korisnik prati. Takođe je moguće dohvatiti sve korisnike koji prate trenutnog korisnika pretragom svih čvorova koji imaju izlaznu vezu ka trenutnom čvoru, što se postiže *Cypher* upitom iz listinga 6.

Neo4J i *Cypher* omogućavaju implementaciju još zanimljivih funkcionalnosti kao što je preporuka korisnika za praćenje. Ova funkcionalnost je u popularnim društvenim mrežama implementirana pomoću algoritama mašinskog učenja i predstavlja složen problem koji bi zahtevao saradnju sa analitičkim sistemom. Za potrebe ove aplikacije, može se iskoristiti moćna izražajnost *Cypher* upita za implementaciju jednostavnog sistema preporuke, prikazan u Listingu 7.

Upit pronalazi korisnike `u3` koje trenutni korisnik `u1` sa identifikatorom `userId` još uvek ne prati, a koje prate korisnici `u2` koje `u1` već prati, pri čemu se `u3` rangira prema broju takvih zajedničkih veza. Međutim, u slučaju novih korisnika koji još uvek ne prate nijednog korisnika, ovaj upit neće biti koristan. U tom slučaju koristi se alternativno rešenje prikazano u Listingu 8.

Drugi upit traži korisnike `u2` sa najviše pratilaca iz iste zemlje kao i `u1`, pod pretpostavkom da bi korisnici bili najzainteresovaniji za popularne korisnike iz svoje zemlje. Oba upita se objedinjuju u odgovornom servisu iz poslovne logike aplikacije pri čemu se drugi upit izvršava samo ukoliko prvi ne vrati nijedan rezultat, što se

```java
@Node
public class User {

    @Id
    @GeneratedValue(UUIDStringGenerator.class)
    private String id;
    private String username;
    private String imageName;
    private String alias;
    private String bio;
    private String country;
    @Relationship(
        type = "FOLLOWING",
        direction = Relationship.Direction.OUTGOING
    )
    private Set<User> following;

    // getter i setter metode
    ...
}
```

Listing 5: Model grafa korisnika u *Java* kodu

```
MATCH (u1:User)-[:FOLLOWING]->(u2:User) WHERE u2.id = $id RETURN u1
```

Listing 6: Dohvatanje pratilaca korisnika u2

```
MATCH (u1: User{id: $userId})-[:FOLLOWING]->(u2:User)-[:FOLLOWING]->(u3:User)
WHERE u1 <> u3 AND NOT (u1)-[:FOLLOWING]->(u3)
RETURN u3, count(DISTINCT u2) as score
ORDER BY score DESC
LIMIT $limit
```

Listing 7: Jednostavan sistem preporuke preko zajedničkih praćenja

postiže korišćenjem `switchIfEmpty` operatora, kao u Listingu 9.

Objave

Funkcionalnosti od ključnog značaja za celu aplikaciju nalaze se u servisu objava. Objavu karakterišu naredna svojstva: korisnik koji je kreirao objavu, sadržaj

```
MATCH (u1: User{id: $userId}), (u2: User)<-[:FOLLOWING]-(:User)
WHERE u1.country = u2.country
  AND u1 <> u2
  AND NOT (u1)-[:FOLLOWING]->(u2)
RETURN u2, count(*) as score
ORDER BY score DESC
LIMIT $limit
```

Listing 8: Jednostavan sistem preporuke preko najpopularnijih profila iz zemlje iz koje je korisnik

```
public Flux<User> getUserRecommendations(String userId, int limit) {
    return userRepository
        .findUserRecommendationsByCommonConnections(userId, limit)
        .switchIfEmpty(
            userRepository
                .findUserRecommendationsFromTheSameCountry(userId, limit)
        );
}
```

Listing 9: Objedinjavanje upita za preporuku u servisu aplikacije

objave koji može biti u tekstualnom ili multimedijalnom formatu, datum i vreme objave, reakcije na objavu u vidu sviđanja ili komentara na objavu. Informacije o korisniku koji je kreirao objavu, kao što su korisničko ime, profilna slika i nadimak, već postoje u servisu korisnika, pa je dovoljno uz objave čuvati samo jedinstveni identifikator korisnika kako bi se izbeglo dupliranje podataka. Iz istog razloga, sviđanja se mogu predstaviti kao kolekcija identifikatora korisnika, čuvana kao svojstvo objave. Komentari su entiteti složenije strukture jer sadrže informacije o korisniku koji je komentarisao objavu, sadržaj komentara, datum i vreme objave komentara. Iz tog razloga se mogu predstaviti kao zasebni entiteti i povezuju se sa objavom na koju se odnose putem reference na njen identifikator. Daljim razvojem aplikacije i dodavanjem novih funkcionalnosti može doći do izmene informacija koje se čuvaju za objave i iz tog razloga je dobro da shema podataka bude fleksibilna. Za bazu podataka odabrana je nerelaciona baza dokumenata *MongoDB*. Pored fleksibilne sheme, *MongoDB* je bolji izbor kad su u pitanju denormalizovani podaci, što je slučaj kod kolekcije sviđanja objave.

Listing 10 i 11 prikazuju mapiranja dokumenata odgovarajućih kolekcija u objek-

te uz pomoć *Spring Data MongoDB* modula.

```
@Document(collection = "post")
public class Post {
    @Id
    ObjectId id;
    @Indexed
    String userId;
    String text;
    String imageName;
    long timestamp;
    List<String> likes;
    int numComments;

    // get i set metode
    ...
}
```

Listing 10: Mapiranje dokumenta objave u objekat

```
@Document(collection = "comment")
public class Comment {
    @Id
    ObjectId id;
    @Indexed
    String postId;
    String userId;
    String text;
    long timestamp;

    // get i set metode
    ...
}
```

Listing 11: Mapiranje dokumenta komentara u objekat

Dodavanje slike uz objavu radi se na isti način kao i u servisu korisnika, uz pomoć objektnog skladišta *MinIO*. Slično, uz model objave biće sačuvan samo ključ po kome može slika da se pronađe u objektnom skladištu (atribut `imageName`).

Anotacija `@Indexed` označava da će *MongoDB* za anotirane atribute kreirati indekse nad odgovarajućim svojstvima radi ubrzanja pretrage. Korisnike često inte-

resuju sve objave od strane specifičnog korisnika, dok se komentari traže uglavnom na osnovu objave kojoj pripadaju.

Prilikom pregleda objava, korisnika najviše zanima sadržaj objave, dok komentari nisu relevantni osim ako korisnik želi da ih pregleda. Iz tog razloga se čuva samo informacija o broju komentara u sklopu modela objave kao atribut `numComments`. Tek kada korisnik zatraži detaljne informacije za određenu objavu, dohvatiće se i svi komentari vezani za nju, čime se smanjuje broj upita ka bazi podataka u uobičajenim scenarijima pregleda sadržaja.

Glavni sadržaj aplikacije je pregled objava korisnika koje trenutni korisnik prati sortirane po vremenu objave, takozvani *feed*. Kako bi se kreirao sadržaj *feed*-a, najpre je neophodno dohvatiti kolekciju korisnika koje trenutni, prijavljeni korisnik prati, a zatim sve objave praćenih korisnika proširiti atributima o informacijama korisnika. Na osnovu navedenog, jasno je da je neophodna komunikacija sa servisom korisnika, pri čemu će komunikacija biti sinhrona jer dalji koraci u formiranju *feed*-a zavise od rezultata komunikacije. Kao tehnologija sinhrone komunikacije između servisa korisnika i servisa objava, odabran je *gRPC* zbog svojih performansi. U komunikaciji se koriste poruke i servisi definisani u okviru `user.proto` fajla (Listing 12), koji je deo zasebnog `common-proto` *Java* projekta namenjenog kao pomoćna biblioteka za servise kojima je neophodna *gRPC* komunikacija.

Servis korisnika implementira metode *gRPC* servisa i postaje *gRPC* server. Servis objava za trenutnog korisnika zahteva listu svih korisnika koje on prati pomoću metode `GetUserFollows(UserIdMsg)`, pri čemu je `userId` koji je u sklopu `UserIdMsg` poruke njegov identifikator. Servis korisnika će dohvatiti neophodne informacije iz Neo4J baze podataka i vratiti odgovor servisu objava u okviru `UserListResponse`, koji predstavlja listu `UserResponse` strukture koja sadrži tražene podatke. Servis objava tada na osnovu odgovora može da dohvati iz svoje baze podataka sve objave korisnika, sortirane po vremenu objave. Na kraju, klijentu se vraća formiran *feed*, strukture iz Listinga 13:

Navedeno rešenje je funkcionalno validno, međutim, efikasnost čitave operacije je upitna. Naime, problem je u činjenici da je neophodna sinhrona komunikacija sa korisničkim servisom, kao i da će odgovor od korisnika biti potencijalno veoma obiman u slučaju korisnika koji prate veliki broj drugih korisnika. Zatim je problem što se za svaki zahtev dohvatanja *feed*-a korisnika izvršava celokupni tok, iako potencijalno nema novih objava ili je broj novih objava minimalan. Umesto formiranja celokupnog *feed*-a za korisnika iz početka, bolje rešenje bi bilo da se prethodni re-

```
service UserService {
  rpc GetUser(UserIdMsg) returns (UserResponse) {}
  rpc GetUserFollows(UserIdMsg) returns (UserListResponse) {}
}

message UserIdMsg {
  string userId = 1;
}

message UserResponse {
  string userId = 1;
  string username = 2;
  string alias = 3;
  string profilePicUrl = 4;
}

message UserListResponse {
  repeated UserResponse users = 1;
}
```

Listing 12: Definicija protokol bafera za servis korisnika

```
public record FeedPost (
    String userId,
    String username,
    String userAlias,
    String profilePicUrl,
    String postId,
    String text,
    String imageUrl,
    int numLikes,
    int numComments,
    long timestamp
) { }
```

Listing 13: Struktura *feed* objave

zultat kešira. Kao tehnologija za keširanje može se koristiti *Redis* skladište ključeva i vrednosti. Pristupanje podacima *Redis* keša značajno je brže nego bazi podataka jer se podaci čuvaju u radnoj memoriji umesto na disku. Keširanjem kolekcije praćenih korisnika i njihovih osnovnih informacija izbegava se potreba komunikacije

sa servisom korisnika i smanjuje se opterećenje na bazu podataka servisa korisnika prilikom svakog zahteva. Slično, keširanjem formiranog *feed*-a smanjuje se opterećenje na bazu podataka servisa objava. Čuvanje kolekcije `FeedPost` objekata kao vrednosti u kešu za svakog korisnika nameće nove probleme. Prvi problem je što će se duplirati potencijalno velike količine podataka, ako se neke od objava javljaju u *feed*-u velikog broja korisnika što istovremeno značajno opterećuje radnu memoriju. Sledeći problem koji se nadovezuje na prethodni jeste činjenica da bi ažuriranje bilo koje od objava zahtevalo ažuriranje iste za *feed* svakog korisnika. Navedeni problemi mogu se prevazići uvođenjem dva odvojena keša. Prvi keš bi čuvao *feed* korisnika u kome je ključ identifikator korisnika, a vrednost je kolekcija identifikatora objava koje pripadaju njegovom *feed*-u. Drugi keš bi čuvao `FeedPost` objekat kao vrednost za ključ identifikator objave. Dohvatanje keširanog *feed*-a za korisnika bi onda činilo dohvatanje kolekcije iz prvog keša, a zatim proširivanje svake objave iz kolekcije informacijama iz drugog keša. Ovim se izbegava dupliranje podataka i olakšava se ažuriranje objava jer je dovoljno samo ažurirati pojedinačne objave.

Kada korisnik kreira novu objavu, *feed* za svakog od njegovih pratilaca mora biti ažuriran. Jedno moguće rešenje je poništavanje keša i ponovno formiranje *feed*-a, što predstavlja neefikasno rešenje jer ostatak *feed*-a ostaje nepromenjen. Bolja alternativa je da se za svakog pratioca kreatora objave ažurira *feed* dodavanjem identifikatora nove objave na vrh kolekcije u sklopu keširanog *feed*-a. Navedeni pristup naziva se emitovanje (engl. *fanout*). Metod u Listingu 14 poziva se nakon kreiranja nove objave. Ukoliko ne postoji keširan *feed* za pratioca autora nove objave, ignoriše se ažuriranje jer će za tog pratioca *feed* svakako biti formiran od nule. Implementacija uvodi novu *gRPC* metodu `getUserFollowers` čiji je zadatak da dohvati identifikatore pratilaca trenutnog korisnika, kao i novi keš, `followersCache`, koji kešira rezultate tog poziva.

Događaj koji takođe zahteva ažuriranje *feed*-a korisnika je praćenje novog korisnika. Objave novog zapraćenog korisnika bi trebalo da se dodaju u *feed* trenutnog korisnika. Kako je praćenje novih korisnika funkcionalnost servisa korisnika, za ažuriranje će takođe biti neophodna komunikacija između servisa korisnika i objava, međutim ovaj put komunikacija je asinhrona jer servisu korisnika nije bitan rezultat ažuriranja. Komunikacija se odvija preko *Kafka* loga događaja, definisanjem `feed-update` teme, tako da će postojanje događaja na toj temi označavati da je neophodno ažurirati *feed*. U ovoj komunikaciji servis korisnika je proizvođač jer piše novi događaj na odgovarajuću temu, dok je servis objava potrošač jer će reagovati

```
private Mono<Post> fanoutNewPostToFollowers(Post post) {
    return followersCache
        .getFollowers(post.getUserId())
        .switchIfEmpty(followersCache.setFollowers(
            post.getUserId(),
            userGrpcClient
                .getUserFollowers(post.getUserId())
                .flatMapIterable(UserIdsListResponse::getUserIdsList)
                .map(UserIdMsg::getUserId)
        ))
        .flatMap(followerId ->
                userFeedCache.addPostToFeed(
                        followerId,
                        post.getId().toString(),
                        post.getTimestamp()
                    )
                )
        .then(Mono.just(post));
}
```

Listing 14: Emitovanje informacije o novoj objavi svim pratiocima korisnika.

na nove događaje.

```
private Mono<Void> sendFollowUpdate(User followed, String user) {
    UserFollowedDTO userFollowed = new UserFollowedDTO(
        user,
        followed.getId(),
        followed.getUsername(),
        followed.getAlias(),
        generatePresignedProfilePicUrl(followed.getImageName())
    );
    return kafkaTemplate.send(FEED_UPDATE_TOPIC_NAME, userFollowed).then();
}
```

Listing 15: Slanje događaja o novom praćenju iz korisničkog servisa.

Metod u Listingu 15 `sendFollowUpdate` u servisu korisnika poziva se nakon što trenutni korisnik zaprati novog. Za slanje informacija o novom praćenju koristi se apstrakcija `ReactiveKafkaTemplate` iz *Spring Kafka* modula koja pojednostavljuje slanje poruka sa proizvođačke strane tako što automatski serijalizuje i deserijalizuje

```
@Service
public class FeedUpdateListener {
    private final PostService postService;

    FeedUpdateListener(PostService postService) {
        this.postService = postService;
    }

    @KafkaListener(topics = FEED_UPDATE_TOPIC_NAME)
    public void consumeFeedUpdateMsg(UserFollowedDTO userFollowedDTO) {
        postService.updateFeedOnFollow(userFollowedDTO).subscribe();
    }
}
```

Listing 16: Čitanje događaja o novom praćenju u okviru servisa objava.

podatke omogućavajući programerima da se fokusiraju na poslovnu logiku apstrahujući detalje komunikacije sa *Kafka* posrednikom. Metod `consumeFeedUpdateMsg` u sklopu `FeedUpdateListener` klase iz servisa objava, prikazan u Listingu 16, anotiran je `@KafkaListener` anotacijom koja označava da će metoda biti pozvana kada se novi događaj pojavi na odgovarajućoj temi. Na kraju, metod `updateFeedOnFollow` će izvršiti ažuriranje *feed*-a tako što će najpre dohvatiti objave zapraćenog korisnika i rezultate dodati u keširan *feed*, ukoliko postoji u kešu.

Kako bi se sprečio uticaj veličine keša na njegove performanse, čuvanje zastarelih podataka ili podataka kojima se ne pristupa često, korisno je definisati dodatne parametre za *Redis*. Servis objava u svom konfiguracionom fajlu `application.yaml` definiše sledeća podešavanja:

- **ttl-seconds** – Vreme važenja keširanih podataka (engl. *Time To Live* – TTL).
- **max-size** – Ograničava veličinu *feed*-a koji se kešira, pod pretpostavkom da većina korisnika pregleda samo ograničen broj objava.
- **page-size** – Korisnicima se ne vraća ceo keširan *feed* odjednom, već je izvršena paginacija, čime se smanjuje količina podataka koja se prenosi pri svakom zahtevu i izbegava nepotrebno učitavanje objava koje korisnik možda neće pregledati.
- **new-follow-post-limit** – Kada se *feed* proširuje objavama novog zapraćenog korisnika, uzima se samo ograničen broj njegovih objava sortiran po vremenu

kreiranja opadajuće, pod pretpostavkom da starije objave nisu relevantne za *feed*.

Parametri su namenjeni da budu konfigurabilni kako bi se njihove vrednosti prilagođavale potrebama sistema.

Obaveštenja

Uloga servisa obaveštenja je da stavi do znanja korisniku da ima novog pratioca, novo sviđanje ili komentar na objavi. Ova funkcionalnost zahteva asinhronu komunikaciju između servisa korisnika i obaveštenja u slučaju događaja praćenja, a u slučaju sviđanja i komentara na objavi, između servisa objava i obaveštenja. Komunikacija se odvija preko *Kafka* posrednika na tri teme: `user-followed` za nova praćenja, `post-likes` za sviđanja, `post-comments` za komentare. Slično kao u prethodnom poglavlju, proizvođači će uz pomoć `ReactiveKafkaTemplate` poslati događaje na odgovarajuću temu, dok će servis obaveštenja reagovati na novopristigle događaje. Klijentu je neophodno proslediti obaveštenje u realnom vremenu. Zahtev za dohvatanje obaveštenja razlikuje se od do sada viđenih *HTTP* zahteva jer se od klijenta ne očekuje da konstantno upućuje zahtev serveru kako bi proveravao da li postoje nova obaveštenja, već je zadatak servera da obavesti klijenta. U ovom slučaju pogodno je koristiti *Server Sent Events* (*SSE*), kojima se uspostavlja *HTTP* konekcija sa klijentom i komunikacija se odvija jednosmerno - od servera ka klijentu. Konekciju inicijalizuje klijent, običnim *HTTP* zahtevom, a odgovori servera biće *MIME* tipa `text/event-stream`. Konekcija ostaje otvorena dok je klijent ne zatvori ili dok ne dođe do greške na serveru ili klijentu.

Događaji koji se pročitaju sa *Kafka* teme biće prosleđeni klijentima samo ukoliko je otvorena *SSE* konekcija, u suprotnom će podaci biti izgubljeni. Iz tog razloga bi trebalo trajno sačuvati pristigla obaveštenja kako bi neaktivni korisnici dobili sva propuštena obaveštenja kad budu bili aktivni. Obaveštenja mogu da se čuvaju u *MongoDB* bazi podataka radi jednostavnih izmena u slučaju promene sheme. Implementirana `NotificationListener` klasa iz Listinga 17 podržava čitanje sa prethodno navedenih *Kafka* tema, pri čemu je poslovna logika ista za sve tipove događaja.

Na pristigla obaveštenja se dodaje polje `read` koje označava da li je korisnik pročitao obaveštenje. Nakon čuvanja, događaj se dodaje na reaktivni tok tipa `Sinks.Many<Notification>` koji dozvoljava čitanje od strane više pretplatnika.

```java
@Service
public class NotificationListener {
    private final Sinks.Many<Notification> sink =
        Sinks.many().multicast().onBackpressureBuffer();
    private final NotificationService notificationService;

    NotificationListener(NotificationService notificationService) {
        this.notificationService = notificationService;
    }

    @KafkaListener(topics = {
        POST_LIKES_TOPIC,
        POST_COMMENTS_TOPIC,
        USER_FOLLOWED_TOPIC
    })
    public void listen(Notification notification) {
        notification.setRead(false);
        notificationService
            .saveNotification(notification)
            .map(sink::tryEmitNext)
            .then()
            .subscribe();
    }

    public Flux<Notification> getNotifications() {
        return sink.asFlux();
    }

}
```

Listing 17: Učitavanje događaja o obaveštenjima sa *Kafka* tema

Ovaj tok funkcioniše kao posrednik koji prima obaveštenja sa *Kafka* teme i omogućava višestrukim pretplatnicima da istovremeno čitaju emitovane vrednosti. Podaci na ovom toku čuvaju se isključivo u radnoj memoriji, što znači da bi u slučaju otkaza servisa došlo do njihovog gubitka, što je još jedan od razloga za čuvanje obaveštenja u bazi podataka. Metoda `getNotifications` izlaže tok kao `Flux<Notification>`, što omogućava pretplatu na pristigla obaveštenja. Na kraju, metoda iz definicije *REST API*-a, prikazana u Listingu 18 pretplaćuje se na reaktivni tok, filtriraju se obaveštenja tako da se izdvajaju samo ona koja su namenjena za trenutnog korisnika i formira se *SSE* odgovor.

```
@GetMapping(value = "/", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
public Flux<ServerSentEvent<Notification>> emitNotificationsToUser(
    @RequestHeader(HttpHeaders.AUTHORIZATION) String bearerToken
) {
    String userId = JwtUtil.extractUserId(bearerToken.split(" ")[1]);

    return notificationListener.getNotifications()
        .filter(notification -> notification.getToUserId().equals(userId))
        .map(notification -> ServerSentEvent
            .<Notification>builder()
            .id(notification.getId().toString())
            .event(notification.getType().name())
            .data(notification)
            .build()
        );
}
```

Listing 18: Prosleđivanje obaveštenja korisniku putem *SSE*

Pretraga

Servis za pretragu daje mogućnost korisnicima da pretražuju korisnike i objave na osnovu zadatih ključnih reči. Za potrebe pretrage koriste se sistemi pretrage, baze podataka specijalizovane za efikasnu pretragu. *Elasticsearch* čuva podatke o korisnicima i objavama za potrebe pretrage, definisani kao indeksi.

Indeksi se popunjavaju podacima iz servisa korisnika i servisa objava, asinhrono. Nakon kreiranja, ažuriranja ili brisanja podataka u izvornim servisima, podaci neophodni za indeksiranje šalju se na odgovarajuće *Kafka* teme: `user-index` i `post-index`. Indeksi će biti eventualno konzistentni u odnosu na izvorne podatke, što je dovoljan uslov za potrebe servisa pretrage jer pretraga ne zahteva strogo ažurne podatke. Indeksi su definisani u Listingu 19 i 20 pomoću *Spring Data Elasticsearch* modula na način naveden u nastavku, tako da sadrže neophodne atribute za pretragu i korisne informacije za prikazivanje rezultata:

Osnovna struktura upita *Elasticsearch Query DSL* je *JSON* objekat koji počinje navođenjem ključne reči `query`, kao što je prikazano u Listingu 21.

Rezultat klauzule `match` su svi dokumenti koji sadrže reč koja se tačno poklapa sa datim terminom, bez obzira gde se termin nalazi u tekstu. Na primer, zadati upit će se poklapati sa dokumentima koji sadrže tekst *„plan i program”* kao i *„program za rad sa dokumentima”*. Međutim, za dokument koji sadrži reč *programi-*

```java
@TypeAlias("post")
@Document(indexName = "post_index")
public class PostIndex {
    @Id
    private String id;
    private String content;
    private int numLikes;
    private int numComments;
    private String photoUrl;
    private long timestamp;
    // ... getter i setter metode
}
```

Listing 19: Definicija indeksa objava

```java
@TypeAlias("user")
@Document(indexName = "user_index")
public class UserIndex {
    @Id
    private String id;
    private String username;
    private String alias;
    private String bio;
    private String avatarUrl;
    private long timestamp;
    // ... getter i setter metode
}
```

Listing 20: Definicija indeksa korisnika

```
{
  "query": {
    "match": {
      "message": {
        "query": "program"
      }
    }
  }
}
```

Listing 21: Primer `match` upita za ključnu reč „program”

ranje, neće biti poklapanja već je potrebno koristiti klauzulu `prefix` umesto `match` za poklapanje po prefiksu. U klauzulama moguće je dodati opciju `fuzziness` koja uzima u obzir moguće slovne greške. Ukoliko je dati termin niska koja se sastoji od više reči (u nastavku – tokena), `match` i `prefix` traže poklapanje sa bilo kojim od datih tokena u dokumentima. Opisano ponašanje ekvivalentno je upitu oblika `match token1 OR token2 OR ...`. Ukoliko se zahteva poklapanje i sa redosledom tokena u terminu pretrage, koriste se `match_phrase` i `match_phrase_prefix`. Ponašanje je ekvivalentno upitu oblika `match token1 AND token2 AND ...`. Klauzula `match_phrase_prefix` se razlikuje od `match_phrase` po tome što se poslednji token u terminu poklapa po prefiksu. Na primer, termin *„master rad"* poklapa se sa tekstom *„master radovi"* ukoliko se koristi `match_phrase_prefix`, ali ne i u slučaju `match_phrase`. Uz ove klauzule moguće je dodati opciju `slop`, broj koji označava koliko je tokena koji se ne poklapaju sa tokenima u terminu pretrage dozvoljeno da se nalazi između reči u tekstu. Na primer, u slučaju `match_phrase_prefix` klauzule sa opcijom `slop=1` za termin *„student fakulteta"*, pretraga vraća rezultat poklapanja sa tekstom *„student fakulteta dramskih umetnosti"* ali i za *„student matematičkog fakulteta"*, pri čemu se zbog `slop` opcije ignoriše reč *„matematičkog"* u drugom tekstu. Pretragu nad objavama moguće je izvršiti samo nad tekstualnim sadržajem objave. Korišćene su klauzule `match_phrase_prefix` i `match`, pri čemu prva ima veći prioritet jer je bolji rezultat onaj koji se poklapa i po redosledu reči u tekstu. Operatori `bool` i `should` omogućavaju kombinovanje rezultata pretrage po različitim klauzama sumirajući njihove skorove. Za `match_phrase_prefix` dozvoljeno je odstupanje koje je jednako broju tokena u datom terminu. Sortiranje rezultata je izvršeno u opadajućem redosledu tako da su najpopularnije objave sa najboljim poklapanjem pri vrhu. Na kraju je izvršena paginacija čime se ograničava broj rezultujućeg skupa. Upiti se mogu definisati programski putem modula *Spring Data Elasticsearch*, umesto direktnim navođenjem *Query DSL JSON* strukture, kao što je prikazano u Listingu 22.

Pretraga nad korisnicima ima sličnu strukturu, samo se pretraga izvršava nad korisničkim imenom, nadimkom ili opisom profila. Korisničko ime i nadimak traže se po klauzama `match` i `prefix` uz `fuzziness` opciju, pri čemu `match` ima veći prioritet, dok se opis profila pretražuje slično kao i sadržaj objave, pomoću `match_phrase_prefix`.

```
public Mono<PostSearchResultDTO> findPostsMatchingQuery(
    String query,
    int pageNumber
) {
    int numTokens = query.split(" ").length;
    NativeQuery nativeQuery = NativeQuery
        .builder()
        .withSort(sort -> sort.score(score ->
            score.order(SortOrder.Desc)))
        .withSort(sort -> sort.field(f ->
            f.field("numLikes").order(SortOrder.Desc)))
        .withSort(sort -> sort.field(f ->
            f.field("numComments").order(SortOrder.Desc)))
        .withQuery(q -> q.bool(
            boolQuery -> boolQuery
                .should(should -> should.matchPhrasePrefix(phrasePrefix ->
                    phrasePrefix
                        .field("content")
                        .query(query)
                        .slop(numTokens))
                )
                .should(should -> should.match(match ->
                    match
                        .field("content")
                        .query(query)
                        .fuzziness("AUTO"))
                )
        ))
        .withPageable(PageRequest.of(pageNumber, paginationProperties.pageSize()))
        .build();
 return operations
        .searchForPage(nativeQuery, PostIndex.class)
        .map(searchPage ->
            new PostSearchResultDTO(
                searchPage.map(SearchHit::getContent).getContent(),
                searchPage.getNumber(),
                searchPage.getTotalPages()
            )
 );
}
```

Listing 22: Pretraga objava po sadržaju

Autentifikacija

Kako bi klijenti mogli da koriste funkcionalnosti aplikacije, moraju biti prijavljeni. Servis za autentifikaciju obezbeđuje pristup sistemu proverom identiteta klijenta koji zahteva resurs. Provera identiteta najčešće podrazumeva prilaganje korisničkog imena i lozinke ili proveru tokena kao alternativnog mehanizma autentifikacije. U implementaciji korišćen je modul *Spring Security* koji olakšava implementaciju funkcionalnosti kreiranja i provere identiteta. Zadatak servisa je da čuva informacije o korisničkom nalogu neophodne za prijavu i izdavanje pristupnog tokena (engl. *access token*) neophodnog za naredne zahteve. Zahtevi za registraciju i prijavu se ređe pozivaju u odnosu na ostale funkcionalnosti sistema, a nameće se potreba da podaci o prijavi budu konzistentni. Iz tih razloga, servis za autentifikaciju koristi relacionu bazu podataka *PostgreSQL* za skladištenje. Servis za autentifikaciju razdvaja odgovornost upravljanja identitetima od upravljanja korisničkim profilima i relacijama iz domena servisa korisnika. Prilikom kreiranja naloga korisnik navodi korisničko ime, e-mail adresu i lozinku kao osnovne podatke za prijavu, ali i atribute profila kojima upravlja servis korisnika. Po prijemu zahteva najpre se poziva odgovarajuća metoda iz servisa korisnika za kreiranje korisničkog čvora u grafovskoj bazi podataka na sinhron način, uz pomoć reaktivnog *HTTP* klijenta `WebClient` *Spring WebFlux* okvira. Identifikator kreiranog korisničkog objekta čuva se zajedno sa podacima o prijavi u *PostgreSQL* bazi podataka, čime je uspostavljena veza između prijave i korisničkog profila.

Nakon registracije i prijave, servis kao odgovor vraća pristupni token kreiran po *JWT* (*JSON Web Token*) standardu, koji definiše način bezbedne razmene informacija u vidu *JSON* objekata. Bezbednost *JWT*-a garantuje digitalni potpis kreiran primenom kriptografskog algoritma i tajne vrednosti koju definiše i čuva server. Token se sastoji iz tri dela: zaglavlja koje navodi naziv algoritma za potpisivanje, sadržaja koga čine tvrdnje (engl. *claim*) kojima se opisuje klijent, i potpisa. Kako bi se smanjila opasnost zloupotrebe u slučaju da token dospe u pogrešne ruke, definiše se vreme važenja pristupnog tokena koje obično iznosi od nekoliko sati do nekoliko dana. Servis za autentifikaciju implementira logiku kreiranja tokena, dok biblioteka `common` implementira pomoćnu klasu `JWTUtils` zaduženu za validaciju na osnovu vremena trajanja tokena i tvrdnji, kao i za ekstrakciju identifikatora korisnika ili korisničkog imena iz tvrdnji tokena radi identifikacije korisnika prilikom obrade zahteva u drugim mikroservisima (Listing 23).

Kako korisnici ne bi bili primorani da se ponovo prijavljuju nakon isteka pristup-

```
public static String extractUserId(String token) {
    return extractClaim(token, claims -> claims.get("user-id", String.class));
}

public static boolean isTokenValid(String token) {
    try {
        Claims claims = extractAllClaims(token);
        return claims != null && !isTokenExpired(token);
    } catch (IllegalArgumentException | JwtException e) {
        return false;
    }
}
```

Listing 23: Validacija pristupnog tokena

nog tokena, uz njega se najčešće kreira i token za obnavljanje (engl. *refresh token*) [7]. Token za obnavljanje koristi se za izdavanje novog pristupnog tokena umesto da korisnik ponovo unosi svoje kredencijale. Vreme važenja tokena za obnavljanje je značajno duže od pristupnog tokena, što ponovo predstavlja bezbednosni rizik jer kompromitovanjem tokena za obnavljanje zlonamerni akter stiče mogućnost kreiranja novih pristupnih tokena. Moguće rešenje je rotacija tokena za obnavljanje nakon svakog korišćenja. Kako bi servis vodio evidenciju važećih tokena za obnavljanje, oni se čuvaju u *Redis* kešu koji se ažurira svakom rotacijom. Tokeni za obnavljanje se čuvaju u kešu po ključu koji je jednak njegovoj `jti` (*JWT* identifier) tvrdnji, čija je primarna uloga sprečavanje napada ponovnom upotrebom tokena (engl. *replay attack*). Vrednost `jti` tvrdnje se postavlja prilikom kreiranja tokena za obnavljanje u vidu *UUID* niske, čime je praktično isključena mogućnost kolizije između dva tokena.

Mrežni prolaz

Mrežni prolaz je ulazna tačka za celu aplikaciju. Svi klijentski zahtevi upućeni su mrežnom prolazu, čiji je zadatak da proveri pravo pristupa traženom resursu i preusmeri zahtev na odgovarajući mikroservis. Provera pristupa na mrežnom prolazu eliminiše potrebu da svaki mikroservis zasebno implementira tu logiku. Za jednostavno podešavanje mrežnog prolaza, koristi se *Spring Cloud Gateway* projekat. *Spring Cloud Gateway* omogućava definisanje ruta koje određuju način i lokaciju servisa kome će klijentski zahtev biti prosleđen. Sve rute definišu se u `application.yaml`

datoteci, kao što je prikazano u Listingu 24.

```
spring:
  application:
    name: gateway
  cloud:
    gateway:
      routes:
        - id: user-service
          uri: http://${USER_SERVICE_HOST:localhost}:${USER_SERVICE_PORT:8082}
          filters:
            - name: Authentication
          predicates:
            - Path=/api/users/**, /user-service/v3/api-docs
    ...
```

Listing 24: Definisanje ruta za servise u okviru mrežnog prolaza

Primer prikazuje samo podešavanje ruta za servis korisnika, ali je definicija slična za ostale mikroservise. U polju `filters` navode se nazivi filtera koji će biti primenjeni na klijentski zahtev pre usmeravanja zahteva mikroservisu. *Spring Cloud Gateway* daje opciju korišćenja predefinisanih filtera [35], ali u ovom slučaju primenjuje se *Authentication* filter čija logika je implementirana u posebnoj klasi. Filter najpre proverava da li postoji zaglavlje za autentifikaciju, a zatim validira token ukoliko postoji. U slučaju odsustva zaglavlja ili nevalidnog tokena, zahtev se odbija.

Spring Cloud Gateway takođe integriše sve opise *REST* interfejsa mikroservisa, u vidu *OpenAPI* specifikacije na jednom mestu, što omogućava pregled celokupnog API-a aplikacije putem jedinstvenog *Swagger UI* interfejsa. Kako bi ovo bilo moguće, neophodno je u individualnim servisima u `application.yaml` fajlu navesti na kojoj putanji se nalazi API specifikacija, kao u Listingu 25 za servis korisnika.

```
springdoc:
  api-docs:
    enabled: true
    path: /user-service/v3/api-docs
  swagger-ui:
    path: /user-service/swagger-ui.html
```

Listing 25: Konfiguracija putanja za dokumentaciju *REST API*-a servisa

Isporuka mikroservisa

Implementirana aplikacija se isporučuje na lokalnoj mašini. Isporuka je realizovana na dva načina - pomoću *Docker Compose* i *Kubernetes*-a. Radi jednostavnosti, neophodna infrastruktura kao što su baze podataka, *Kafka* posrednik i objektno skladište definisani su u okviru `docker-compose.yaml` datoteke korišćenjem zvaničnih *Docker* slika. Za svaku komponentu definisani su portovi i kredencijali, a u slučaju *Kafka* posrednika i *MinIO* objektnog skladišta, dodatno su definisane skripte koje kreiraju teme i kofe.

Sa druge strane, implementirani *Spring Boot* mikroservisi isporučuju se u okviru *Kubernetes* klastera. Pre definisanja isporuke za svaki mikroservis, neophodno je definisati *Kubernetes* klaster sa čvorovima kontrolne ravni i radnim čvorovima u kome će se pokretati aplikacije. Lokalno okruženje može da podrži samo jedan čvor u okviru Kubernetes klastera pri čemu taj čvor preuzima ulogu kontrolne ravni i radnih čvorova. Za realnije testiranje aplikacija u lokalnom okruženju, neophodno je kreirati veći broj čvorova što je moguće ukoliko se čvorovi definišu kao *Docker* kontejneri. Alat koji olakšava kreiranje *Kubernetes* klastera na lokalnoj mašini u kome su čvorovi *Docker* kontejneri naziva se *kind* (***K**ubernetes **IN D**ocker*)[2]. Za potrebe aplikacije Bonds, komandom `kind create cluster --config <definicija-klastera>.yaml` kreiran je klaster `kind-bonds-dev` sa dva radna čvora i jednom kontrolnom ravni. Radi izolovanja resursa pojedinačne aplikacije kako bi se izbegle kolizije u nazivima resursa, definiše se prostor imena (engl. *namespace*) u okviru klastera, tako da je kreiran prostor imena `ns-bonds-local` pomoću `kubectl` alata komandne linije komandom `kubectl create namespace ns-bonds-local`.

Mikroservise je najpre neophodno izgraditi u vidu *Docker* slika. S obzirom na to da su svi mikroservisi implementirani u Spring Boot okviru u programskom jeziku Java, definicija *Docker* slike izgledaće slično za sve mikroservise. Primer definicije *Docker* slike za servis objava prikazan je u Listingu 26. Putanja `../../` se koristi jer se kontekst izgradnje postavlja na koreni direktorijum projekta, odakle je moguć pristup zajedničkom `pom.xml` i izvornom kodu modula, dok se sam *Dockerfile* nalazi u `infra/docker` direktorijumu.

Pre izgradnje mikroservisa, potrebno je izgraditi pomoćne `common` i `common-proto` biblioteke kako bi bile uključene kao zavisnosti servisa.

Za definisanje *Kubernetes* objekata neophodnih za isporuku mikroservisa ko-

[2]https://kind.sigs.k8s.io/

```
1  FROM maven:3.9-eclipse-temurin-21-alpine AS post-build
2
3  WORKDIR /usr/src/post
4  COPY ../../pom.xml ./parent-pom.xml
5  COPY ../../post/pom.xml ./pom.xml
6  COPY ../../post/src ./src
7  COPY --from=bonds/common-proto-build:latest /root/.m2 /root/.m2
8  COPY --from=bonds/common-build:latest /root/.m2 /root/.m2
9  RUN mvn install -N -f parent-pom.xml && mvn clean package -DskipTests
10
11 FROM eclipse-temurin:21-jdk-alpine
12
13 WORKDIR /app
14
15 COPY --from=post-build /usr/src/post/target/*.jar post.jar
16 ENTRYPOINT ["java", "-jar", "post.jar"]
```

Listing 26: Definicija *Docker* slike za izgradnju *Java Spring Boot* aplikacije.

risti se alat *Helm*[3]. Alat *Helm* koristi se za upravljanje *Kubernetes* aplikacijama definisanjem paketa (engl. *chart*) koji sadrže sve neophodne definicije objekata za kreiranje instance aplikacije. *Helm* paketi takođe olakšavaju verzionisanje i deljenje definicija *Kubernetes* aplikacija. Pozivanjem `helm create <naziv-mikroservisa>` iz komandne linije kreira se *Helm* paket koji u sebi sadrži šablone željenih *Kubernetes* objekata u okviru `templates` direktorijuma i datoteke `Chart.yaml` za verzionisanje i `values.yaml` za definisanje promenljivih vrednosti koje su zajedničke za veći broj šablona. Takođe ovako kreiran `values.yaml` već sadrži sve podrazumevane vrednosti podesivih svojstava *Kubernetes* objekata koje programer po potrebi može da izmeni ili ukloni. U slučaju pojedinačnih definicija *Kubernetes* objekata bez *Helm* šablona, potrebno je manuelno definisati sva svojstva karakteristična za taj objekat i navesti vrednosti koje su često zajedničke za sve objekte, kao na primer, naziv prostora imena ili metapodaci aplikacije. Navođenje vrednosti na ovaj način predstavlja nepotrebno ponavljanje istog postupka, čime se usporava proces isporuke.

Umesto toga, odlomak iz datoteke `values.yaml` za servis korisnika prikazan u Listingu 27, definiše svojstva koja se zatim koriste za definiciju isporuke definisane u `templates/deployment.yaml` datoteci prikazanoj u Listingu 28.

Na ovaj način sva promenljiva svojstva nalaze se odvojeno od objekata čime je

[3]https://helm.sh/

```yaml
namespace: "ns-bonds-local"
image:
  repository: bonds/user-service
  pullPolicy: Always
  tag: latest

nameOverride: ""
fullnameOverride: "user-service"

replicaCount: 1
autoscaling:
  enabled: false
env:
  - name: MINIO_SERVICE_HOST
    value: host.docker.internal
  - name: MINIO_SERVICE_PORT
    value: "9000"
  - name: MINIO_USERNAME
    valueFrom:
      secretKeyRef:
        name: minio-secrets
        key: username
  - name: MINIO_PASSWORD
    valueFrom:
      secretKeyRef:
        name: minio-secrets
        key: password
  - name: NEO4J_HOST
    value: host.docker.internal
  - name: NEO4J_PORT
    value: "7687"
  - name: NEO4J_USERNAME
...
```

Listing 27: Definicija svojstava u `values.yaml` datoteci

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "user.fullname" . }}
  namespace: {{ .Values.namespace }}
  labels:
    {{- include "user.labels" . | nindent 4 }}
spec:
  {{- if not .Values.autoscaling.enabled }}
  replicas: {{ .Values.replicaCount }}
  {{- end }}
  selector:
    matchLabels:
      {{- include "user.selectorLabels" . | nindent 6 }}
  template:
    metadata:
      labels:
        {{- include "user.selectorLabels" . | nindent 8 }}
    spec:
      containers:
        - name: {{ .Chart.Name }}
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag
          ↪  | default .Chart.AppVersion }}"
          imagePullPolicy: {{ .Values.image.pullPolicy }}
          ports:
            - name: http
              containerPort: {{ .Values.service.port }}
              protocol: TCP
            - name: grpc
              containerPort: 9090
              protocol: TCP
          env:
            {{- with .Values.env }}
              {{ toYaml . | trimSuffix "\n" | nindent 12 }}
            {{- end }}
...
```

Listing 28: Objekat isporuke za servis korisnika kreiran u okviru *Helm* šablona

olakšano razumevanje konfiguracije aplikacije i olakšane su buduće izmene u definiciji isporuke servisa jer će se one uglavnom vršiti nad `values.yaml` datotekom bez potrebe izmena šablona. Komandom `helm template <naziv-mikroservisa>` prikazuje se rezultat zamene vrednosti svojstava iz dvostrukih vitičastih zagrada u šablonima odgovarajućim vrednostima definisanim u `values.yaml` ili `Chart.yaml` datotekama. Komanda `helm install <naziv-mikroservisa>` zatim primenjuje ove definicije na Kubernetes klaster čime je upotpunjena isporuka instance aplikacije. Definisanje Helm paketa slično je za sve ostale mikroservise, jedina razlika je u definisanju isporuke za mrežni prolaz koji dodatno definiše *Ingress* objekat. Objekat *Ingress* omogućava povezivanje i upućivanje zahteva od strane eksternih klijenata, što je neophodno s obzirom na to da trenutna konfiguracija za ostale mikroservise omogućava komunikaciju isključivo unutar klastera. Odlomak iz `values.yaml` za mrežni prolaz iz Listinga 29 opisuje podešavanje za *Ingress* objekat, pri čemu svojstvo `host` definiše osnovnu adresu aplikacije, `http://bonds.localhost/`, na koju se nadovezuju putanje ka pojedinačnim funkcionalnostima.

```yaml
ingress:
  enabled: true
  className: nginx
  annotations: {}
  hosts:
    - host: bonds.localhost
      paths:
        - path: /
          pathType: Prefix
  tls: []
```

Listing 29: Odlomak iz values.yaml za mrežni prolaz koji opisuje podešavanje za *Ingress* objekat

Poslednji korak isporuke je automatizacija procesa izgradnje *Docker* slika i primene definisanih isporuka na *Kubernetes* klaster. Jedno moguće rešenje je definisanje skripti koje bi automatizovale ovaj proces, ali alat *Tilt*[4] je dobar izbor u slučaju aplikacije *Bonds* jer je namenjen za automatizaciju isporuke lokalnih aplikacija. *Tilt* je alat otvorenog koda koji je zvanično preuzet od strane *Docker* korporacije 2022. godine [16]. Tilt prati potencijalne izmene u kodu aplikacije i sam pokreće proces izgradnje i isporuke kako bi nova verzija aplikacije odmah bila dostupna. Ceo

[4]Zvanična veb stranica: https://tilt.dev/

proces od izgradnje do isporuke se definiše u okviru `Tiltfile` datoteke, uz pomoć odgovarajućih funkcija za rad sa *Docker*-om, *Kubernetes* klasterima i *Helm* paketima. Odlomak iz Listinga 30 prikazuje kako izgleda definicija `Tiltfile` za isporuku servisa objava. Na sličan način dodaju se i ostali mikroservisi.

Najpre se pomoću `docker_build` funkcije izgrađuju *Docker* slike za pomoćne biblioteke i servis objava. Funkcija `helm` izvršava `helm template` operaciju za konstruisanje *Kubernetes* objekata na osnovu *Helm* šablona, a `local_resource` izvršava skriptu koja primenjuje konfiguracionu mapu na odgovarajući prostor imena u klasteru. Na kraju, funkcija `k8s_yaml` primenjuje konstruisane definicije *Kubernetes* objekata na klaster, čime se pokreće instanca aplikacije.

3.3 Pregled arhitekture i servisa analitičkog dela

Zadatak analitičkog dela aplikacije jeste da korisnicima omogući uvid u statistike koje opisuju performanse profila. Praćenjem ovakvih statistika korisnici mogu da prilagode sadržaj objava kako bi povećali svoj uticaj na društvenoj mreži. Kako bi podaci bili od veće vrednosti za korisnika, nameće se potreba da budu dostupni što bliže realnom vremenu. Informacije o novim pratiocima i reakcijama na objave već se prenose kao događaji preko *Kafka* posrednika za potrebe obaveštavanja korisnika, ali mogu dodatno da se iskoriste za formiranje analitika. Centralnu ulogu u obradi podataka u ovom delu imaće *Spark Structured Streaming* koji prikuplja podatke sa *Kafka* posrednika. Arhitektura analitičkog dela aplikacije organizovana je u vidu arhitekture medalje (engl. *medallion architecture*).

Arhitektura medalje je obrazac projektovanja koji opisuje organizaciju podataka u vidu slojeva u okviru *data lakehouse*-a [1]. Slojevi nastaju inkrementalnom obradom podataka iz prethodnog sloja čime se postiže rastući nivo kvaliteta i strukture podataka. Arhitekturu čine tri sloja: bronzani, srebrni i zlatni, pri čemu svaki sloj ima jasno definisanu odgovornost u procesu obrade.

Bronzani sloj je prvi korak namenjen za efikasno prikupljanje i skladištenje podataka u *data lakehouse*-u sa različitih izvora, bez transformacija ili uz minimalne transformacije. Podržani izvori podataka mogu biti izvori paketne obrade kao što su baze podataka, ili izvori za obradu tokova kao što je *Kafka*. Učitavanjem podataka što bliže izvornom obliku kreira se arhiva podataka koja po potrebi može da se iskoristi za ponovnu obradu ili debagovanje procesa obrade. Ovaj sloj je posebno značajan za izvore tokova podataka jer obezbeđuje nezavisnost između izvora i

```
allow_k8s_contexts('kind-bonds-dev')

docker_build(
  'bonds/common-proto-build',
  context='.',
  dockerfile='./infra/docker/Dockerfile.commonproto'
)

docker_build(
  'bonds/common-build',
  context='.',
  dockerfile='./infra/docker/Dockerfile.common'
)

docker_build(
  'bonds/post-service',
  context='.',
  dockerfile='./infra/docker/Dockerfile.post'
)

post_yaml = helm(
    './infra/helm/post',
    name='post-service',
    namespace='ns-bonds-local',
    values=['./infra/helm/post/values.yaml'],
)

local_resource(
  'post-service-configmap',
  'helm template post-service ./infra/helm/post ' +
  '--namespace ns-bonds-local ' +
  '--values ./infra/helm/post/values.yaml ' +
  '--show-only templates/configmap.yaml | kubectl apply -f -',
  deps=['./infra/helm/post'],
)

k8s_yaml(post_yaml)
```

Listing 30: Definisanje neophodnih funkcija u `Tiltfile` za isporuku servisa objava

ostatka sistema zahvaljujući činjenici da otkazi na izvorima neće uticati na dostupnost podataka. Za potrebe aplikacije Bonds, bronzani sloj putem *Spark Structured*

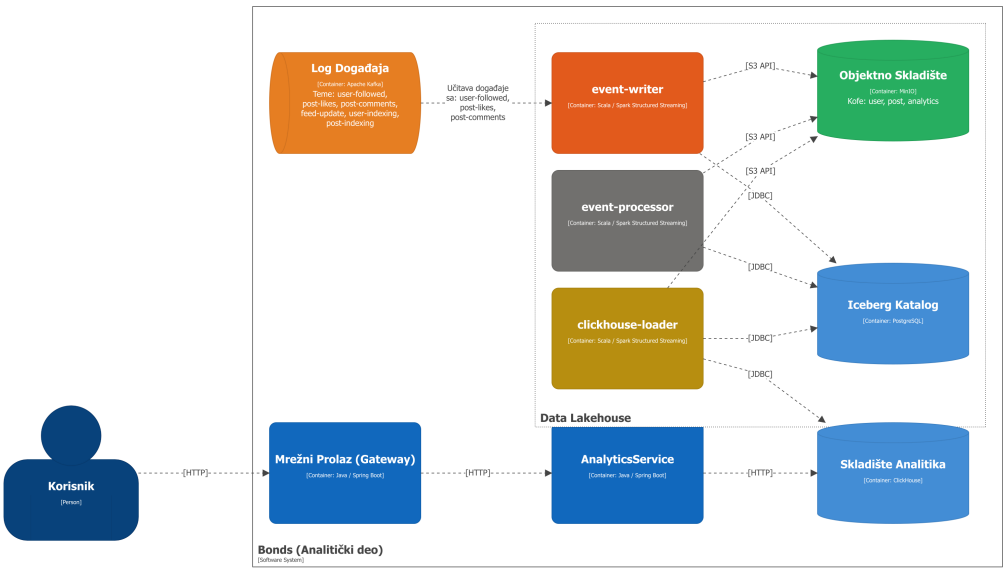
Streaming aplikacije učitava događaje sa odgovarajućih *Kafka* tema i čuva podatke uz minimalne transformacije kao *Iceberg* tabele u objektnom skladištu *MinIO*.

Srebrni sloj je sledeći korak u obradi podataka kojim se vrši dalje filtriranje, prečišćavanje i validacija. Ova faza priprema podatke za zahtevnije zadatke kojima je neophodan veći nivo preciznosti podataka kao u slučaju projekata mašinskog učenja. U ovom sistemu, *Spark Structured Streaming* aplikacija učitava *Iceberg* tabele iz prethodnog sloja, filtrira relevantne kolone i uklanja duplikate. Prečišćeni podaci takođe se čuvaju kao *Iceberg* tabele u *MinIO* objektnom skladištu.

Zlatni sloj čine podaci najvećeg kvaliteta, optimizovani za čitanje koji su prilagođeni za podršku odlučivanja, pisanje izveštaja ili za korisničke analitike. Implementaciju ovog sloja čini *Spark Structured Streaming* aplikacija koja učitava tabele iz prethodnog sloja i prilagođava shemu podataka kako bi mogli da se učitaju u *ClickHouse* skladište za potrebe uvida u korisničke analitike.

Poslednju komponentu analitičkog dela čini mikroservis za analitike koji korisnicima omogućava dohvatanje podataka iz *ClickHouse* skladišta preko *REST API*-a.

Opisana arhitektura se može vizuelizovati dijagramom kontejnera C4 modela kao na Slici 3.3, uz napomenu da su mrežni prolaz, *Apache Kafka* i *MinIO* skladište zajednički za oba dela aplikacije.



Slika 3.3: Dijagram kontejnera analitičkog dela aplikacije

3.4 Implementacija analitičkog dela aplikacije

Implementaciju analitičkog dela aplikacije čine tri *Apache Spark* aplikacije napisane u programskom jeziku *Scala* za svaki sloj arhitekture medalje i jedna *Java Spring Boot* aplikacija kao mikroservis koji korisnicima omogućava uvid u analitike. Sve tri *Apache Spark* aplikacije koriste *Spark Structured Streaming* biblioteku za obradu podataka u realnom vremenu, podržavaju interakciju sa *MinIO* objektnim skladištem i koriste *Iceberg* biblioteku sa *PostgreSQL* bazom podataka kao eksterni katalog za skladištenje metapodataka.

Projekti su strukturno organizovani tako da sadrže sledeće zajedničke pakete:

- `resources` - Sadrži `application.conf` fajl u kome su definisana podešavanja za komponente kao što su *Kafka*, *MinIO* i *PostgreSQL*. Takođe definiše nazive izvora podataka (*source*), nazive tabela i njihove lokacije za čuvanje podataka (*sink*).
- `config` - Definiše pomoćne klase koje omogućavaju pristup svojstvima definisanim u okviru `application.conf` datoteke. Biblioteka *TypeSafe Config*[5] korišćena je za učitavanje konfiguracija.
- `model` - Ovaj paket se nalazi samo u aplikacijama srebrnog i zlatnog sloja sa ciljem da definiše klase koje će označavati tip *Dataset* strukture podataka. Ove klase predstavljaju sheme *Iceberg* tabela koje čuvaju navedeni slojevi.
- `source` i `sink` - Sadrže klase koje implementiraju logiku učitavanja podataka sa izvora i upisivanja podataka na odredišta, redom.

Učitavanje i skladištenje podataka sa Kafka tema

Apache Spark aplikacija `event-writer` čita događaje sa istih *Kafka* tema kao i servis obaveštenja, kako bi se informacije o novim praćenjima i interakcijama sa objavama sačuvale za potrebe analitika. Svaki tip događaja sačuvaće se u zasebnoj *Iceberg* tabeli, pri čemu se događaj čuva u originalnom formatu kao *JSON* niska u jednoj koloni sa dodatnim kolonama koje sadrže informacije o vremenu učitavanja u sistem i *Kafka* svojstvima. Pre početka obrade neophodno je kreirati tabele u kojima će se podaci čuvati po definisanoj shemi.

[5]Zvanični repozitorijum: https://github.com/lightbend/config

```
val schema = StructType(Seq(
  StructField("rawEvent", StringType),
  StructField("kafkaKey", StringType),
  StructField("topic", StringType),
  StructField("partition", IntegerType),
  StructField("offset", LongType),
  StructField("kafkaTimestamp", TimestampType),
  StructField("ingestionTime", TimestampType)
))

val queries = appConfig.queries.map(q => QueryConfig(q, ConfigFactory.load()))
queries.foreach(q => icebergSink.createTableIfNotExists(q.sink, schema))

queries
  .map { queryConfig =>
    val df = kafkaSource.readFromTopic(schema, queryConfig.source)

    df
      .writeStream
      .foreachBatch { (dfBatch: DataFrame, _: Long) =>
        icebergSink.write(dfBatch, queryConfig.sink)
      }
      .option(
        "checkpointLocation",
        s"${icebergConfig.checkpointDir}/${queryConfig.checkpoint}"
      )
      .trigger(Trigger.ProcessingTime("5 seconds"))
      .start()
  }
  .foreach(q => q.awaitTermination())
```

Listing 31: Kreiranje sheme tabele i tok obrade u bronzanom sloju.

U Listingu 31, pozivom `.start()` metode kreira se `StreamingQuery`, apstrakcija biblioteke *Spark Structured Streaming* koja predstavlja jedan aktivni tok obrade, pri čemu će za svaku *Kafka* temu biti pokrenut po jedan upit. Metod `trigger` navodi vremenski interval tokom kojeg se prikupljaju podaci i formiraju jedan mikro-paket za obradu. Pozivom `awaitTermination()` sprečava se terminacija aplikacije pre završetka obrade, što u ovom slučaju znači da će aplikacija raditi dok ne dođe do greške ili do manuelne terminacije.

Transformacija i prečišćavanje podataka

Aplikacija `event-processor` prilikom čitanja iz *Iceberg* tabele uzima samo `rawEvent` kolonu, parsira *JSON* nisku i filtrira samo neophodne kolone definisane listom `selectColumns`. Na učitani *DataFrame* se dodaje kolona `day` koja označava datum izveden iz postojeće `timestamp` kolone.

Pre pisanja *Iceberg* tabele, uklanjaju se potencijalni duplikati pomoću `MERGE INTO` upita koji se koriste kada je potrebno ažurirati podatke tabele. Upit iz Listinga 32 ispituje da li u sačuvanoj tabeli postoji red sa istim identifikatorima kao u dolaznim podacima. Naredba `WHEN NOT MATCHED THEN INSERT *` upisuje novi red u tabelu ukoliko nisu ispunjeni uslovi navedeni u `ON` naredbi.

```
def deduplicatePostEventsAndWrite(
    dataset: Dataset[PostEngagementEvent],
    tableName: String,
    sourceViewName: String)(implicit sparkSession: SparkSession): Unit = {
  val tablePath = s"${icebergConfig.catalogConfig.name}.$tableName"
  val dsSession = dataset.sparkSession
  dataset
    .dropDuplicates("userId", "fromUserId", "postId", "engagementType")
    .createOrReplaceTempView(sourceViewName)

  dsSession.sql(s"""
      MERGE INTO $tablePath
      USING $sourceViewName
      ON $sourceViewName.userId = $tablePath.userId
      AND $sourceViewName.fromUserId = $tablePath.fromUserId
        AND $sourceViewName.postId = $tablePath.postId
        AND $sourceViewName.engagementType = $tablePath.engagementType
      WHEN NOT MATCHED THEN INSERT * """)
}
```

Listing 32: Uklanjanje duplikata za tabelu objava u srebrnom sloju.

Upisivanje podataka u ClickHouse skladište

Pre implementacije *Apache Spark* aplikacije, potrebno je definisati shemu *ClickHouse* tabela. Shema interakcija (engl. *engagements*) sa objavama i praćenja korisnika zahteva izmene u nazivima pojedinih kolona kako bi njihov kontekst bio jasniji, kao i preciznije definisanje tipa vremenske oznake kao `DateTime64(3)` što

označava broj milisekundi od epohe. Mašina za skladištenje tabela koja se koristi je `MergeTree`, čije su prednosti objašnjene ranije zajedno sa teorijskim osnovama *ClickHouse* skladišta. Ključ sortiranja čine identifikatori korisnika i objave (ili samo korisnika u slučaju tabele praćenja), kao i vremenska oznaka pošto će ove kolone kasnije prilikom agregiranja biti korišćene u `GROUP BY` naredbi, a sortiranjem po neophodnim kolonama se poboljšavaju performanse spajanja fragmentisanih tabela u pozadinskom procesu `MergeTree` mašine. Shema je definisana u Listingu 33.

```
CREATE TABLE IF NOT EXISTS analytics.post_engagements (
    userId          String,
    engagingUserId  String,
    postId          String,
    engagementType  Enum8('LIKE', 'COMMENT'),
    timestamp       DateTime64(3),
    day             String
)
ENGINE = MergeTree
ORDER BY(userId, postId, timestamp);

CREATE TABLE IF NOT EXISTS analytics.user_follows (
     userId      String,
     followerId  String,
     timestamp   DateTime64(3),
     day         String
)
ENGINE = MergeTree
ORDER BY(userId, timestamp)
```

Listing 33: Shema analitičkih tabela skladišta ClickHouse zlatnog sloja

Apache Spark aplikacija `clickhouse-loader`, koja čita *Iceberg* tabele i čuva podatke u *ClickHouse* skladištu, imaće sličnu strukturu kao i prethodna aplikacija, pri čemu će jedine transformacije biti promena naziva kolona tako da se poklapaju sa kolonama *ClickHouse* sheme.

Definisanje agregacija u ClickHouse skladištu

Pre definisanja ostalih *ClickHouse* shema i agregacije podataka, potrebno je precizno definisati kakve analitike se stavljaju korisnicima na raspolaganje.

Nova praćenja korisnika prebrojava pratioce trenutnog korisnika u određenom vremenskom intervalu i daje korisnicima uvid u rast njihovih profila. Ova analitika realizuje se prebrojavanjem jedinstvenih identifikatora pratilaca, kolone `followerId`, grupisanjem po identifikatoru trenutnog korisnika.

Najpopularnije objave po broju sviđanja ili po broju komentara korisnicima pokazuju koje objave su stekle najveću pažnju njihovih pratilaca. Realizuju se prebrojavanjem redova po tipu interakcija, kolone `engagementType`, grupisanjem po identifikatoru trenutnog korisnika i identifikatoru objave.

Stopa interakcija (engl. *engagement rate*) predstavlja koliki udeo korisnika koji je video objavu je interagovao sa objavom korisnika. Pod interakcijom sa objavom podrazumeva se reagovanje na objavu u vidu sviđanja, ili komentarisanje objave. U kontekstu aplikacije Bonds, objave mogu da se pojavljuju u *feed*-u korisnika koji prate autora objave, pa samim tim ova analitika za korisnika u u vremenskom intervalu t može da se izrazi formulom [21]:

$$stopaInterakcija_u^t = \frac{brojJedinstvenihKorisnikaKojiSuInteragovali_u^t}{brojPratilaca_u^t}$$

S obzirom na to da se meri udeo pratilaca koji su reagovali, a ne ukupan broj interakcija, neophodno je uzeti u obzir jedinstvene korisnike koji su ostvarili makar jednu interakciju sa objavom. Za potrebe izračunavanja stope interakcija biće neophodno spajanje tabela interakcija i praćenja, dok će takođe biti neophodno prebrojavanje jedinstvenih identifikatora korisnika koji su reagovali na objave, kolona `engagedUserId`, i prebrojavanje jedinstvenih identifikatora pratilaca. *ClickHouse* za potrebe čuvanja agregiranih podataka definiše `AggregatingMergeTree` mašinu za skladištenje koja nasleđuje `MergeTree`. Pomoću `AggregatingMergeTree` moguće je definisati agregirajuće funkcije koje će se primenjivati nad redovima istog ključa za sortiranje prilikom procesa spajanja fragmentisanih delova tabela [10]. Ovako definisane tabele čuvaju samo međustanje agregacije čime se poboljšavaju performanse upita nad često agregiranim podacima. U nastavku je data agregatna tabela interakcija koja sadrži kolone tipa `AggregateFunction(<nazivFunkcije>, <tipKolone>)`, gde `<tipKolone>` označava tip kolone koja se prosleđuje agregatnoj funkciji. Funkcija `countIf` prebrojava redove koji zadovoljavaju dati uslov, dok funkcija `uniq` prebrojava redove jedinstvene vrednosti zadate kolone. Tabela praćenja, prikazana u Listingu 34 imaće sličnu strukturu, samo će definisati kolonu *Followers* sa `uniq` agregatnom funkcijom.

```
CREATE TABLE IF NOT EXISTS analytics.post_engagements_agg (
    userId       String,
    postId       String,
    timestamp    DateTime64(3),
    Likes        AggregateFunction(countIf, UInt8),
    Comments     AggregateFunction(countIf, UInt8),
    Engagements  AggregateFunction(uniq, String)
)
ENGINE = AggregatingMergeTree
ORDER BY(userId, postId, timestamp);
```

Listing 34: Tabela agregacija za objave u skladištu ClickHouse zlatnog sloja.

Kako bi se agregatne tabele popunjavale pri svakom unosu podataka u izvorne tabele, kreira se materijalizovani pogled (Listing 35) koji funkcioniše kao okidač. Za svaki novi red unet u tabelu, pogled primenjuje prethodno definisane agregatne funkcije i upisuje prelazno stanje agregacije u agregatnu tabelu. Sufiks `State` za agregatne funkcije znači da se izračunava prelazno stanje, ali ne i konačan rezultat.

```
CREATE MATERIALIZED VIEW IF NOT EXISTS analytics.post_engagements_agg_mv
    TO analytics.post_engagements_agg
AS SELECT
       userId,
       postId,
       timestamp,
       countIfState(engagementType = 'LIKE')    AS Likes,
       countIfState(engagementType = 'COMMENT') AS Comments,
       uniqState(engagingUserId)                AS Engagements
FROM analytics.post_engagements
GROUP BY (postId, userId, timestamp);
```

Listing 35: Materijalizovani pogled za agregacije tabele objava.

Na kraju je preostala definicija upita koji vraćaju tražene analitike. Upit za izračunavanje stope interakcija na dnevnom nivou u zadatom vremenskom intervalu za zadatog korisnika definiše se na način prikazan u Listingu 36.

Ovaj put koristi se sufiks `Merge` za agregatne funkcije koji označava da će biti vraćen konačni rezultat agregacije. Funkcija `multiIf` koristi se kako bi se izbegle greške usled deljenja nulom.

```
SELECT
    p.userId,
    toStartOfDay(p.timestamp) as dateTime,
    multiIf(
        uniqMerge(Followers) = 0,
        0.0,
        uniqMerge(Engagements) / uniqMerge(Followers)
    ) as engagementRate
FROM post_engagements_agg p join user_follows_agg u
    on p.userId = u.userId
WHERE p.userId = {userId:String}
AND p.timestamp
    BETWEEN fromUnixTimestamp64Milli({startDate:UInt64})
    AND fromUnixTimestamp64Milli({endDate:UInt64})
AND u.timestamp
    BETWEEN fromUnixTimestamp64Milli({startDate:UInt64})
    AND fromUnixTimestamp64Milli({endDate:UInt64})
GROUP BY p.userId, toStartOfDay(p.timestamp)
ORDER BY dateTime
```

Listing 36: Upit za izračunavanje stope interakcija za korisnika.

Ostali upiti, od kojih su neki definisani u Listingu 37, imaće jednostavniju strukturu, uz napomenu da upiti za najpopularnije komentare izgledaju identično, osim što se `countIfMerge` funkcija poziva nad `Likes` kolonom za sviđanja, a za komentare nad `Comments` kolonom.

Implementacija mikroservisa za analitike

Spring Boot mikroservis *AnalyticsAPI* koristi *ClickHouse Java Client* biblioteku za komunikaciju sa skladištem podataka. Korisnici preko *REST API*-a zahtevaju određenu analitiku tako što eksplicitno navode vremenski interval ili koriste neki od predefinisanih intervala. Za tražene analitike, mikroservis zatim koristi jedan od prethodno opisanih upita u komunikaciji sa *ClickHouse* skladištem kako bi formirao odgovor. Servis takođe vrši proveru raspona vremenskog intervala tako da nije moguće definisati vremenski interval kraći od jednog dana. Ukoliko se definiše vremenski interval manji od `MAX_DAYS_FOR_HOURLY_QUERYING`, grupisanje podataka se radi po satu umesto po danu kako bi rezultat činilo više tačaka.

```
-- Broj novih pratilaca korisnika na dnevnom nivou
-- u zadatom vremenskom intervalu
SELECT
    userId,
    toStartOfDay(timestamp) as dateTime,
    uniqMerge(Followers) as numNewFollowers
FROM user_follows_agg
WHERE userId = {userId:String}
    AND timestamp BETWEEN fromUnixTimestamp64Milli({startDate:UInt64})
    AND fromUnixTimestamp64Milli({endDate:UInt64})
GROUP BY userId, toStartOfDay(timestamp)
ORDER BY numNewFollowers

-- Pet najboljih objava korisnika sa najvise svidjanja
-- u zadatom vremenskom intervalu
SELECT
    userId,
    postId,
    countIfMerge(Likes) as numLikes
FROM post_engagements_agg
WHERE userId = {userId:String}
    AND timestamp BETWEEN fromUnixTimestamp64Milli({startDate:UInt64})
    AND fromUnixTimestamp64Milli({endDate:UInt64})
GROUP BY userId, postId
ORDER BY numLikes DESC
LIMIT 5
```

Listing 37: Upiti za izračunavanje analitika novih praćenja i najboljih objava korisnika po broju sviđanja.

Isporuka analitičkog dela aplikacije

Slično kao u slučaju isporuke mikroservisa, infrastruktura koja podrazumeva *Kafka* posrednika, *MinIO* objektno skladište i *ClickHouse* bazu podataka definisani su u okviru `docker-compose.yaml` datoteke uz pomoć zvaničnih *Docker* slika. Servis analitika napisan u *Spring Boot* okviru, isporučuje se na identičan način opisan u poglavlju o isporuci mikroservisa.

Isporuka *Apache Spark* aplikacija u okviru *Kubernetes* klastera se značajno razlikuje od mikroservisa usled činjenice da rukovodilac *Spark* aplikacije zahteva resurse od klastera što nije bio slučaj kod mikroservisnih aplikacija gde se manuelno definiše broj replika čaura. Takođe, ukoliko dođe do kvara rukovodioca, mora da postoji

```
public Flux<PostEngagementsMetric> getPostEngagementsAnalytics(
    String userId,
    PresetTimeRange presetTimeRange,
    Instant startDateInstant,
    Instant endDateInstant
) {
    TimeRange timeRange =
        getTimeRangeFromRequest(presetTimeRange, startDateInstant, endDateInstant);

    boolean useHourlyAnalytics = PresetTimeRange.PAST_DAY.equals(presetTimeRange)
        || TimeUnit.MILLISECONDS.toDays(
            timeRange.endTimeMillis() - timeRange.startTimeMillis()
        ) <= MAX_DAYS_FOR_HOURLY_QUERYING;

    return useHourlyAnalytics
        ? postEngagementsRepository.fetchPostEngagementsMetricForUserHourly(
            userId, timeRange.startTimeMillis(), timeRange.endTimeMillis()
        )
        : postEngagementsRepository.fetchPostEngagementsMetricForUserDaily(
            userId, timeRange.startTimeMillis(), timeRange.endTimeMillis()
        );
}
```

Listing 38: Metoda za dohvatanje analitike stope interakcija definisana u mikroservisu.

sačuvana informacija o poslednjem pročitanom ofsetu sa *Kafka* teme kako bi čitanje moglo da se nastavi, dok je u slučaju kvara mikroservisa dovoljno da se pokrene nova instanca jer nema potrebe za čuvanjem stanja aplikacije. Iz navedenih razloga, za isporuku *Spark* aplikacija neophodno je koristiti *Kubernetes* operator za *Apache Spark* aplikacije.

Kubernetes operatori su ekstenzije koje definišu resurse u okviru *Kubernetes* aplikacijskog interfejsa čime se na raspolaganje stavljaju novi objekti pored postojećih. Novi resursi povezuju se sa kontrolerima koji prate stanje aplikacije. *Spark Operator* razvijen u okviru *Kubeflow*[6] projekta definiše novi *Kubernetes* objekat pod nazivom `SparkApplication`. Na osnovu definicije objekta, kontroler prosleđuje argumente procesu koji se naziva *submission runner* koji pokreće *Spark* rukovodioca u vidu čaure. Rukovodilac zatim kreira radne čvorove takođe u vidu čaura. U okviru opera-

[6]Zvanična veb stranica: `https://www.kubeflow.org/`

tora se takođe nalazi proces koji nadgleda stanje aplikacije i informacije prosleđuje kontroleru.

Pre definisanja `SparkApplication` objekta, potrebno je učitati zvanični *Helm* paket za *Spark Operator*, kao što je prikazano u Listingu 39 u okviru `Tiltfile`-a.

```
local_resource(
    'spark-operator',
    'helm repo add spark-operator https://kubeflow.github.io/spark-operator && ' +
    'helm repo update && ' +
    'helm upgrade --install spark-operator spark-operator/spark-operator ' +
    '--namespace=ns-bonds-local --create-namespace ' +
    '--set spark.jobNamespaces[0]=ns-bonds-local'
)

k8s_kind('SparkApplication', image_json_path='{.spec.image}')
```

Listing 39: Učitavanje i instalacija *Spark* operatora unutar `Tiltfile`-a

Naredba `helm repo add` učitava zvanični *Helm* paket, dok `helm upgrade -install` primenjuje definiciju operatora na klaster čime je `SparkApplication` objekat stavljen na raspolaganje. *Spark* aplikacije mogu da se definišu na način prikazan u Listingu 40.

U okviru `driver` i `executor` svojstava definišu se promenljive okruženja i tajne neophodne za izvršavanje *Spark* aplikacije. Referenca `commonEnv` koristi se za referisanje na liste u *YAML* datotekama kako bi se izbeglo dupliranje koda. *Docker* slika koja se koristi u definiciji objekta sadrži *JAR* fajl koji se sastoji od klasa izvornog koda, ali u sebi sadrži i klase svih zavisnih biblioteka kako bi verzije biblioteka bile usklađene u svim čvorovima *Spark* klastera. Ovakav *JAR* fajl naziva se *über JAR* ili *fat JAR* i u okviru *sbt* alata za izgradnju *Scala* koda, koji se koristi u *Bonds* aplikaciji, dodatak *sbt-assembly* daje mogućnost izgradnje ovakvih *JAR* fajlova. Ostale *Spark* aplikacije se definišu na sličan način.

Na kraju se u `Tiltfile` dodaju funkcije koje izvršavaju izgradnju *Scala* koda pomoću skripte `build-spark-apps.sh` kao i izgradnju *Docker* slike, zatim učitavaju konfiguracionu mapu koja sadrži zajedničke promenljive okruženja za *Spark* aplikacije i na kraju se objekat `SparkApplication` primenjuje na *Kubernetes* klaster. Listing 41 prikazuje dopunu `Tiltfile`-a na primeru `event-writer` aplikacije.

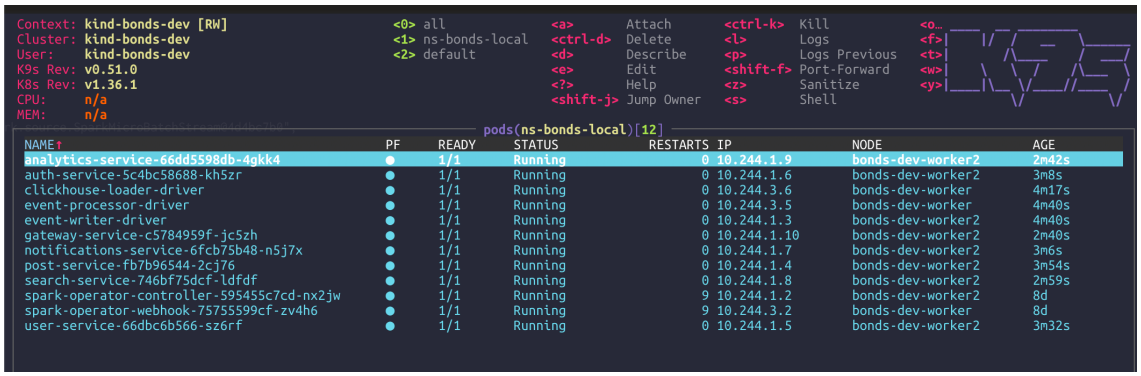
```yaml
apiVersion: sparkoperator.k8s.io/v1beta2
kind: SparkApplication
metadata:
  name: event-writer
  namespace: ns-bonds-local
spec:
  type: Scala
  mode: cluster
  image: bonds/event-writer:latest
  mainClass: com.bonds.analytics.app.MainApp
  mainApplicationFile: local:///opt/event-writer.jar
  sparkVersion: "3.3.0"
  driver:
    env: &commonEnv
      - name: MINIO_URL
        valueFrom:
          configMapKeyRef:
            key: MINIO_URL
            name: "spark-env-config"
      - name: MINIO_USERNAME
        valueFrom:
          secretKeyRef:
            name: "minio-secrets"
            key: "username"
      - name: MINIO_PASSWORD
        valueFrom:
          secretKeyRef:
            name: "minio-secrets"
            key: "password"
      - name: KAFKA_HOST
        valueFrom:
          configMapKeyRef:
            key: KAFKA_HOST
            name: "spark-env-config"
      - name: KAFKA_PORT
        valueFrom:
          configMapKeyRef:
            key: KAFKA_PORT
            name: "spark-env-config"
      - name: POSTGRES_DB_HOST
...
  executor:
    env: *commonEnv
```

Listing 40: Definicija `SparkApplication` objekta za `event-writer` aplikaciju. (Sadržaj datoteke je redukovan radi čitljivosti)

```
local_resource(
    'spark-apps-assembly',
    cmd='./infra/scripts/build-spark-apps.sh',
    deps=[
        './analytics/event-writer/src',
        './analytics/event-writer/build.sbt',
    ],
)

docker_build(
    'bonds/event-writer',
    context='./analytics',
    dockerfile='./analytics/event-writer/Dockerfile'
)

k8s_yaml('./infra/k8s/spark-env-config.yaml')

k8s_yaml('./infra/k8s/event-writer.yaml')
```

Listing 41: Dopuna `Tiltfile`-a za izgradnju i primenu `event-writer` aplikacije

Pokretanjem `tilt up` iz komandne linije, započinje se izgradnja i isporuka aplikacije *Bonds*. Na kraju, celokupna isporučena aplikacija sadrži čaure prikazane na Slici 3.4 iz alata K9s [7] za upravljanje *Kubernetes* klasterom iz komandne linije.



Slika 3.4: Pregled isporučenih čaura u `kind-bonds-dev` klasteru u okviru `ns-bonds-local` prostora imena.

[7] Zvanična veb stranica: `https://k9scli.io/`

Glava 4

Diskusija

Tokom izrade rada, uočene su određene mane i nedostaci u implementaciji rešenja. Ovo poglavlje ističe aspekte koji se mogu izmeniti ili doraditi kako bi predstavljena arhitektura bila još više u skladu sa zahtevima koje nameće problem obrade velikih podataka.

Servis objava u trenutnom rešenju implementira funkcionalnost dohvatanja *feed*-a korisnika, što ima smisla sa domenskog aspekta. U slučaju implementacije složenijih algoritama za formiranje *feed*-a umesto sadašnje logike koja je na osnovu praćenja korisnika, ovu funkcionalnost bi trebalo izdvojiti u poseban servis, a servisu objava prepustiti samo odgovornost kreiranja, brisanja i interakcija sa objavama. Jedan od razloga za razdvajanje ovih funkcionalnosti je u tome što je formiranje *feed*-a značajno zahtevnija operacija od ostalih, pa bi razdvajanjem bilo omogućeno nezavisno skaliranje u odnosu na opterećenje i potrebne resurse, kao i veća otpornost servisa objava na greške. Drugi razlog je što je kreirana spregnutost između servisa objava i servisa korisnika usled potrebe za dohvatanjem informacije o praćenjima korisnika. Ovaj servis bi čuvao neophodne podatke iz servisa objava i korisnika koje bi dobijao na asinhron način nakon kreiranja, izmena ili brisanja podataka iz izvornih servisa, na sličan način na koji su formirani indeksi u servisu pretrage. Princip keširanja već formiranog *feed*-a ostao bi nepromenjen. Ovim pristupom se izbegavaju sinhroni pozivi ka drugim servisima, što bi pozitivno uticalo na performanse i otpornost servisa. Mana ovog servisa je dupliranje postojećih podataka i eventualna konzistentnost, ali su ti kompromisi prihvatljivi u korist smanjene spregnutosti među servisima.

Servis za autentifikaciju takođe uspostavlja jaku spregnutost sa servisom korisnika usled potrebe za istovremenim kreiranjem profila za prijavu i korisničkog profila.

Jedno rešenje bi podrazumevalo spajanje servisa korisnika i autentifikacije u jedan, koji bi sadržao dve baze podataka. Ovim pristupom bi se samo eliminisala potreba za sinhronim pozivom ka servisu korisnika. Drugo moguće rešenje je implementacija Saga obrazaca za distribuirane transakcije [30]. U Saga obrascu, operacija izvršena u okviru jednog servisa kreira događaj ili poruku koja se na asinhron način šalje sledećem servisu u lancu poziva. Ukoliko dođe do greške, primenjuju se inverzne operacije koje poništavaju izmene kreirane do trenutka greške, čime se obezbeđuje eventualna konzistentnost transakcije. U slučaju prijave na aplikaciju Bonds, servis za prijavu bi sačuvao podatke o prijavi u bazi podataka, a zatim slao događaj servisu korisnika kako bi kreirao novi čvor u grafovskoj bazi podataka. Relacija između prijave i korisničkog profila bi se uspostavila preko korisničkog imena kao atributa koji je jedinstven u okviru celokupnog sistema, umesto čekanja na identifikator korisničkog čvora u grafu.

Analitike koje se čuvaju u *ClickHouse* skladištu sadrže samo identifikatore korisnika i objava, što za posledicu ima potrebu dohvatanja dodatnih informacija iz izvornih servisa na osnovu tih identifikatora. Kako bi se podržalo automatsko proširivanje analitika dodatnim podacima, mogu se koristiti *ClickHouse* rečnici [12] koji predstavljaju bazu podataka ključeva i vrednosti koji se čuvaju u radnoj memoriji. Baze podataka servisa korisnika i servisa objava mogu da se koriste kao izvori podataka za rečnik i time bi se eliminisali dodatni pozivi ka izvornim servisima. Kako bi podaci u rečniku ostali ažurni, podešava se vremenski interval osvežavanja sadržaja rečnika.

Infrastruktura za isporuku aplikacije, definisana u okviru `docker-compose.yaml` datoteke, predstavlja jednostavno rešenje za testiranje aplikacije u lokalnom okruženju. Međutim, ukoliko se aplikacija isporučuje u produkciono okruženje, potrebni su fleksibilniji alati namenjeni za isporuku u različita okruženja. Takođe je poželjno da definicija isporuke za infrastrukturu, mikroservise i *Spark* apikacije bude objedinjena zajedničkim alatom kako bi sva podešavanja bila objedinjena na jednom, centralizovanom mestu. Alati koji se koriste za ovakve namene su *Terraform*[1] ili *OpenTofu*[2] za definisanje infrastrukture kao kôd, koja može biti lokalna ili u oblaku. Dodatno poboljšanje isporuke moguće je postići korišćenjem alata za automatsku integraciju i automatsku isporuku (engl. *continuous integration/continuous deployment* -

[1] `https://developer.hashicorp.com/terraform`
[2] `https://opentofu.org/`

CI/CD) umesto alata *Tilt*, kao što su *Jenkins*[3] i *ArgoCD*[4]. Iako *Tilt* podržava definisanje koraka koji se često opisuju u procesu kontinualne integracije i kontunualne isporuke, on je prvenstveno namenjen ubrzavanju lokalnog razvojnog ciklusa i ne poseduje mehanizme neophodne za produkciono okruženje, kao što su trajno praćenje stanja klastera, kontrola pristupa i podrška za isporuku kroz više odvojenih okruženja.

[3] `https://www.jenkins.io/`
[4] `https://argoproj.github.io/cd/`

Glava 5

Zaključak

U ovom radu predstavljen je problem implementacije sistema za obradu velikih podataka pridajući podjednaku važnost na implementaciji izvornih sistema i analitičkih sistema. Istaknute su karakteristike mikroservisnih i analitičkih sistema relevantne za fazu projektovanja softvera, uz naglasak da je duboko razumevanje koncepata bitan preduslov pre odabira alata za razvoj i implementacije softvera. Pored toga, projektovanje sistema za obradu velikih podataka podrazumeva kompromise u vidu stroge konzistentnosti i minimalne redundantnosti podataka u korist boljih performansi, veće dostupnosti i skalabilnosti sistema.

Spring Boot kao okvir za razvoj mikroservisa kroz svoju bogatu podršku za različite tehnologije i biblioteke pojednostavljuje realizaciju projektovane aplikacije. Predstavljeno je kako funkcionalni zahtevi aplikacije utiču na odabir modela skladištenja podataka. Asinhrona komunikacija između mikroservisa realizovana je posredstvom *Apache Kafka* platforme, u vidu događaja što je omogućilo nisku spregnutost sistema.

Data lakehouse i arhitektura medalje obezbeđuje pouzdanost, validnost i kvalitet podataka za potrebe analitičke obrade velikih podataka. *Apache Spark* kao okvir za obradu velikih podataka omogućio je efikasnu obradu tokova podataka u realnom vremenu, dok *ClickHouse* skladište podataka u realnom vremenu pruža podršku za izvršavanje zahtevnih analitičkih upita.

Razvijena aplikacija društvene mreže služi kao primer koje aspekte bi trebalo razmatrati prilikom definisanja arhitekture sistema velikih podataka i kako savremeni koncepti i alati olakšavaju implementaciju. Dalji rad bi podrazumevao implementaciju funkcionalnosti razmene poruka između korisnika u realnom vremenu i dublju integraciju izvornog i analitičkog dela sistema u cilju serviranja personalizovanog

sadržaja u skladu sa interesovanjima korisnika uz pomoć modela mašinskog učenja. Implementirano analitičko rešenje predstavlja dobru osnovu za proširenje skupa analitika kao i za integraciju sa sistemima za odlučivanje kako bi postojao uvid u sistem sa poslovnog aspekta.

Bibliografija

[1] Derar Alhussein. *Databricks Certified Data Engineer Associate Study Guide*. O'Reilly Media, 2024.

[2] Amazon Web Services. Programming asynchronously using the AWS SDK for Java 2.x, 2026. online at: `https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/asynchronous.html#advanced-operations`.

[3] Amazon Web Services. Sharing objects using presigned urls, 2026. online at: `https://docs.aws.amazon.com/AmazonS3/latest/userguide/ShareObjectPreSignedURL.html`.

[4] Apache Iceberg. Iceberg Table Spec, 2026. online at: `https://iceberg.apache.org/spec/`.

[5] Apache Software Foundation. Introduction, 2026. online at: `https://kafka.apache.org/42/getting-started/introduction/#main-concepts-and-terminology`.

[6] Apache Spark. Structured Streaming Programming Guide: Programming Model, 2026. online at: `https://spark.apache.org/docs/latest/streaming/getting-started.html#programming-model`.

[7] Auth0. Refresh tokens: What they are and when to use them, 2024. online at: `https://auth0.com/blog/refresh-tokens-what-are-they-and-when-to-use-them/`.

[8] Adam Bellemare. *Building Event-Driven Microservices: Leveraging Organizational Data at Scale*. O'Reilly Media, 2025.

[9] John Carnell and Illary Huaylupo Sánchez. *Spring Microservices in Action, Second Edition*. Manning, 2021.

[10] ClickHouse Inc. AggregatingMergeTree table engine, 2026. online at: `https://clickhouse.com/docs/engines/table-engines/mergetree-family/aggregatingmergetree`.

[11] ClickHouse Inc. Architecture Overview: On-Disk Format, 2026. online at: `https://clickhouse.com/docs/academic_overview#3-1-on-disk-format`.

[12] ClickHouse Inc. Dictionaries, 2026. online at: `https://clickhouse.com/docs/dictionary`.

[13] Iuliana Cosmina, Rob Harrop, Chris Schaefer, and Clarence Ho. *Pro Spring 6: An In-Depth Guide to the Spring Framework*. Apress, 2023.

[14] Jules S. Damji, Brooke Wenig, Tathagata Das, and Denny Lee. *Learning Spark: Lightning-fast Data Analytics*. O'Reilly Media, 2020.

[15] DeltaLake. Understanding Open Table Formats, 2026. online at: `https://delta.io/blog/open-table-formats/`.

[16] Docker Inc. Docker Acquires Tilt to Help Fix the Pains of Microservices Development for Kubernetes, 2026. online at: `https://www.docker.com/press-release/docker-acquires-tilt-to-help-fix-the-pains-of-microservices-development-for-kubernetes/`.

[17] Google Cloud. What is a REST API?, 2026. online at: `https://cloud.google.com/discover/what-is-rest-api`.

[18] Katya Gorshkova. *Kafka for Architects: Event-driven architecture, logs, microservices, real-time event processing*. Manning, 2026.

[19] gRPC. Introduction to gRPC, 2026. online at: `https://grpc.io/docs/what-is-grpc/introduction/`.

[20] IBM Think. What is object storage?, 2026. online at: `https://www.ibm.com/think/topics/object-storage`.

[21] Seungbae Kim, Jyun-Yu Jiang, Jinyoung Han, and Wei Wang. Influencerrank: Discovering effective influencers via graph convolutional attentive recurrent neural networks. In *Proceedings of the International AAAI Conference on Web and Social Media*, volume 17, pages 482–493, 2023.

[22] Rob Kitchin and Gavin McArdle. What makes Big Data, Big Data? Exploring the ontological characteristics of 26 datasets. *Big Data & Society*, 3, 02 2016.

[23] Martin Kleppmann and Chris Riccomini. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, 2026.

[24] Kubernetes Documentation. Pods, 2026. online at: `https://kubernetes.io/docs/concepts/workloads/pods/`.

[25] Microsoft Docs. Compare gRPC services with HTTP APIs, 2025. online at: `https://learn.microsoft.com/en-us/aspnet/core/grpc/comparison?view=aspnetcore-10.0`.

[26] MongoDB Inc. An Introduction to Search Indexes, 2026. online at: `https://www.mongodb.com/resources/basics/search-index#what-is-a-search-index`.

[27] MongoDB Inc. JSON and BSON, 2026. online at: `https://www.mongodb.com/resources/basics/json-and-bson`.

[28] Neo4J. *Graph Databases For Beginners - Ebook*, 2018. online at: `https://neo4j.com/wp-content/themes/neo4jweb/assets/images/Graph_Databases_for_Beginners.pdf`.

[29] Neo4j. *The Neo4j Cypher Manual v25*, 2025. online at: `https://neo4j.com/docs/pdf/neo4j-cypher-manual-25.pdf`.

[30] Sam Newman. *Building Microservices*. O'Reilly Media, 2021.

[31] Project Reactor. Introduction to Reactive Programming, 2026. online at: `https://projectreactor.io/docs/core/release/reference/reactiveProgramming.html`.

[32] Redis. What is a Key-Value Database?, 2026. online at: `https://redis.io/nosql/key-value-databases/`.

[33] Joe Reis and Matt Housley. *Fundamentals of Data Engineering: Plan and Build Robust Data Systems*. O'Reilly Media, 2022.

[34] Mark Richards and Neal Ford. *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media, 2020.

[35] Spring IO. Gatewayfilter factories reference, 2026. online at: `https://docs.spring.io/spring-cloud-gateway/reference/spring-cloud-gateway-server-webflux/gatewayfilter-factories.html`.

[36] Spring IO. Spring framework overview, 2026. online at: `https://docs.spring.io/spring-framework/reference/overview.html`.

[37] Spring IO. Spring webflux - overview, 2026. online at: `https://docs.spring.io/spring-framework/reference/web/webflux/new-framework.html`.

[38] Stack Overflow. Developer Survey 2025, 2025. online at: `https://survey.stackoverflow.co/2025/technology/`.

[39] Tatjana Kunić (GitLab). Bonds, 2026. online at: `https://gitlab.com/kunict11/bonds/`.

Biografija autora

Tatjana Kunić rođena je 05.08.1998. godine u Beogradu. Osnovnu školu „Duško Radović" završila je 2013. godine, nakon čega upisuje Devetu gimnaziju „Mihailo Petrović Alas" koju završava 2017. godine. Nakon toga upisuje osnovne studije informatike na Matematičkom fakultetu, Univerziteta u Beogradu. Osnovne studije završava 2022. godine i odmah potom upisuje master studije na smeru informatika. Svoju karijeru započinje 2022. godine u kompaniji *Endava* na poziciji praktikanta i nakon završene prakse biva primljena za stalno na poziciji *Java* softverskog inženjera. Od 2026. godine radi u kompaniji *Zühlke* kao inženjer podataka. Glavnu oblast interesovanja uključuje razvoj platformi za obradu velikih podataka koje prate savremene standarde kvaliteta, pouzdanosti i održivosti softverskih sistema.