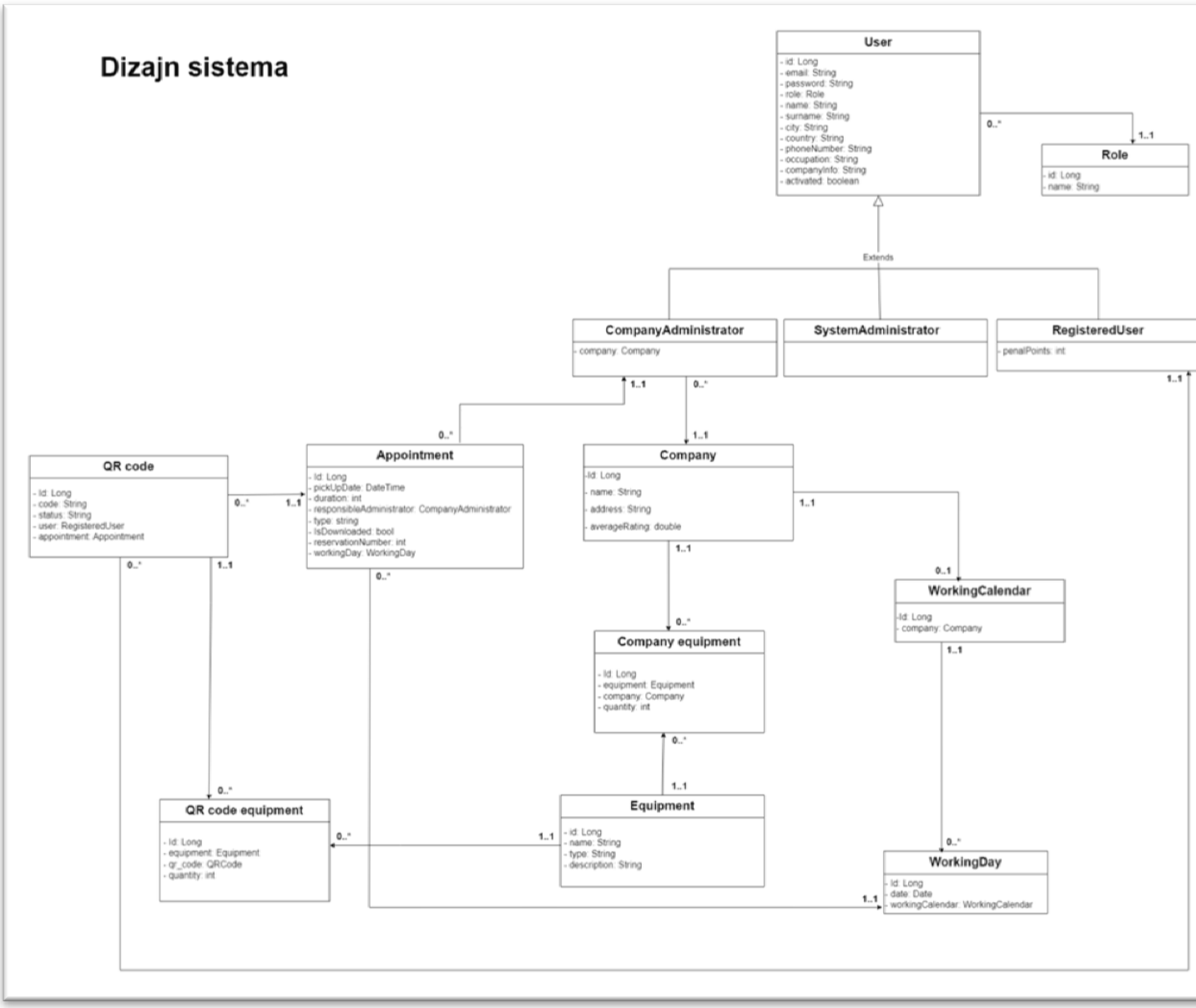
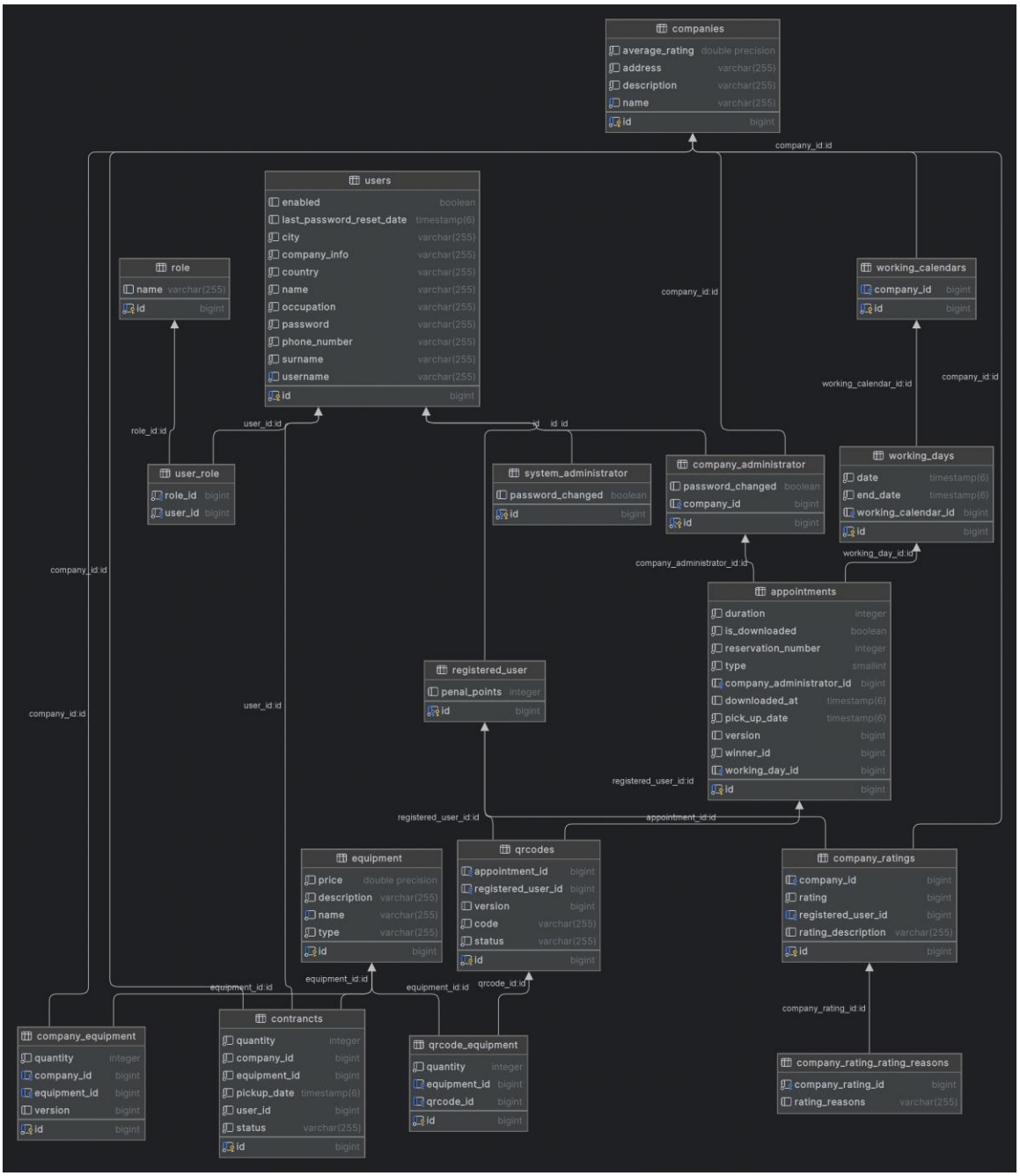
Proof of concept

Tim 25

- **Dizajn šeme baze podataka**





- **Predlog strategije za particionisanje podataka**

Parcionisanje podataka podrazumeva logičku podelu velikih skupova podataka na manje, upravljive segmente. Kroz strategijsko deljenje podataka na osnovu relevantnih kriterijuma možemo značajno poboljšati celokupne performanse sistema.

Za horizontalno particionisanje bih napravio podelu po geografskoj lokaciji, vremenskim zonama i vremenskim periodima (meseci ili periodi dana) koji bi imali velike varijacije računajući da naša aplikacija treba da podrži 100 miliona korisnika i 500.000 rezervacija na mesečnom nivou. Na osnovu rasprostaranjenosti i angažovanosti naših korisnika bi mogli izabrati optimalan odnos između ovih strategija.

Što se tiče vertikalnog particionisanja izvršio bih podelu po poslovnoj logici i odvojio podatke koji se koriste manje frekventno unutar klasa, kao što su npr. detalji o kompaniji, adresi i zaposlenju korisnika.

[NEW]
Horizontalno Particionisanje (Sharding)

Horizontalno particionisanje podrazumeva deljenje tabela po redovima. Ovo znači da se jedna velika tabela deli na više manjih tabela, pri čemu svaka tabela sadrži podskup redova originalne tabele.

Imamo tabelu Users sa milionima redova. Ovu tabelu možemo podeliti na više manjih tabela, kao što su User_1, User_2, itd. Ključ za particionisanje može biti korisnički ID, gde se korisnici sa ID-evima 1-1 000 000 nalaze u User_1, ID-evima 1000001-2,000,000 u User_2, i tako dalje.

Prednosti:

Povećava performanse upita: Budući da se operacije izvode na manjim tabelama, upiti se izvršavaju brže.

Skalabilnost: Lakše je skalirati aplikaciju jer se podaci raspoređuju na više manjih particija, što omogućava bolje upravljanje resursima.

Izazovi:

Složena implementacija: Potrebna je dodatna logika za ruteiranje upita ka odgovarajućoj particiji, što može povećati složenost koda.

Održavanje: Upravljanje više tabela može biti izazovno, posebno kada su potrebne promene u šemi podataka ili migracija podataka.

Vertikalno Particionisanje (Sharding)

Vertikalno particionisanje podrazumeva deljenje tabela po kolonama. To znači da se jedna tabela sa mnogo kolona deli na više tabela, pri čemu svaka nova tabela sadrži podskup originalnih kolona.

Imamo tabelu Users koja sadrži informacije o profilu korisnika i njihove aktivnosti. Ovu tabelu možemo podeliti na dve manje tabele: User_Profile (koja sadrži kolone vezane za profil, kao što su ime, adresa, email) i User_Activity (koja sadrži kolone vezane za aktivnosti, kao što su poslednji login, istorija kupovina).

Prednosti:

Poboljšava performanse: Učitavaju se samo potrebni podaci, što smanjuje veličinu upita i ubrzava njihovo izvršavanje. Na primer, upit za korisnički profil neće uključivati podatke o aktivnostima, što smanjuje vreme izvršavanja.

Efikasnost memorije: Manje kolone po tabeli znače manje podataka koji se čuvaju u memoriji prilikom izvršavanja upita, što može poboljšati efikasnost.

Izazovi:

Potencijalno složeniji upiti: Kada su potrebni podaci iz više particionisanih tabela, upiti postaju složeniji jer je potrebno pridruživanje (JOIN) tabela. Ovo može povećati kompleksnost upita i potencijalno usporiti njihovo izvršavanje.

Održavanje: Povećava se broj tabela koje treba održavati i osigurati da su konzistentne, što može zahtevati dodatne napore pri upravljanju bazom podataka.

• Predlog strategije za replikaciju baze i obezbeđivanje otpornostina greške

Računajući da naša aplikacija trenutno funkcioniše samo sa jednom bazom podataka, da bi skalirali i obezbedili otpornost na greške moramo uvesti nekoliko instanci baze podataka.

Međusobni odnos ovih baza podataka bi bila master-slave, gde bi se sve izmene tj. write komande izvršavale na master bazi podataka. Te izmene bi onda onda bile replicirane na ostale slave baze podataka, sa kojih bi se izvršavale read komande.

Ovim pristupom bi bili otporniji na scenario otkazivanja jedne od baza podataka, jerbi imali njene replike i na taj način mogli da sačuvamo veliku većinu podataka.

Druga strategija koju bih takođe pomenuo je da napravimo multi-master sistem gde bi sve baze međusobno razmenjivale podatke i dozvoljavale i read i write. Na ovaj način bi mogli rasporediti baze podataka po različitim kontinentima i pobojšati brzinu komunikacije sa različitih geografskih lokacija. Morali bi i koristiti softver koji bi koordinisao baze podataka i rešavao konflikte u komunikaciji više mastera kao npr.SymmetricDS ili Bucardo.

- **Predlog strategije za keširanje podataka**

Obzirom da koristimo Hibernate, to podarazumijeva da je L1 keš već uključen u našu aplikaciju. Samim tim, poboljšane su perfomanse pristupa podacima u okviru jedne sesije. Sa druge strane, kada je u pitanju drugi nivo keširanja, L2, i on je podržan od strane Hibernate-a. Njegovom uključenošću, možemo riješiti problem keširanja na nivou cjelokupne aplikacije.

Međutim, potreban je jedan eksterni provajder, poput EhCache, kako bismo do kraja konfigurisali L2 keš. Upravo je EhCache dobro podržan od strane Hibernate-a, te bi on sam bio naš izbor za tu ulogu, kao i kod najvećeg broja Spring aplikacija. EhCache može znatno ubrzati pristup podacima i smanjiti opterećenje na bazi podataka. Osnovna svrha i njegov zadatak bi bilo keširanje podataka koji se samo ponekad modifikuju i vrlo često čitaju iz baze. U našem slučaju, to bi bili podaci o kompanijama i raspoloživoj opremi u sistemu. Za strategiju keširanja bismo izabrali Transactional Read-Write. Ona bi omogućila da se svi objekti u kešu

osvježavaju nakon završetka transakcije, čime se održava konzistentnost sa bazom podataka.

[NEW]

Redis je brza in-memory baza podataka koja skladišti podatke u RAM-u za brzi pristup. Koristi se za keširanje podataka radi poboljšanja performansi aplikacija. Podržava razne strukture podataka kao što su stringovi, liste, setovi i heševi. Omogućava visokoučinkovite operacije čitanja i pisanja. Podržava persistenciju podataka putem RDB snapshot-a i AOF logovanja. Takođe podržava replikaciju podataka između master i slave servera radi povećane dostupnosti i otpornosti na greške. Redis koristi politike izbacivanja podataka (eviction policies) za upravljanje memorijom i omogućava postavljanje vremena života (TTL) za keširane podatke.

Dakle, Transactional Read-Write donosi efikasno ažiriranje i čitanje keširanih objekata, te oni odgovaraju stvarnom stanju u bazi. Samim tim, korišćenjem adekvatnih keširanih podataka, broj upita ka bazi u okviru jedne transakcije bi bio smanjen.

Kada je riječ o front-end keširanju, tu bismo se opredijelili za CDN keširanje, koje bi ubrzalo učitavanje sadržaja i, ujedno, rasteretilo server, čime bismo unaprijedili skalabilnost i pouzdanost naše aplikacije. Kako se CDN sastoji iz više servera ili čvorova širom svijeta, svaki od njih bi sadržao par keširanih kopija određenih resursa, poput HTML stranica i CSS stilova, slika i slično.

- **Okvirna procena za hardverske resurse potrebne za skladištenjesvih podataka u narednih 5 godina**

Appointment:

- id [long] - 8 bytes
- pickUpDate [LocalDateTime] - 8 bytes
- duration [int] - 4 bytes

- type [int] - 4 bytes
- isDownloaded [boolean] - 1 bit
- reservationNumber [int] - 4 bytes
- winnerId [long] - 8 bytes
- downloadedAt [LocalDateTime] - 8 bytes
- companyAdministratorId [long] - 8 bytes
- workingDayId [long] - 8 bytes

  TOTAL: 61 bytes

QRcode:

- id [long] - 8 bytes
- code [String] - 6* bytes
- status [int] - 4 bytes
- registeredUserId [long] - 8 bytes
- appointmentId [long] - 8 bytes

  TOTAL: 34 bytes

QREquipment:

- id [long] - 8 bytes
- qrCodeId [long] - 8 bytes
- equipmentId [long] - 8 bytes
- quantity [int] - 4 bytes

  TOTAL: 28 bytes

WorkingCalendar:

- id [long] - 8 bytes
- companyId [long] - 8 bytes

  TOTAL: 16 bytes

WorkingDay:

- id [long] - 8 bytes
- date [Date] - 8 bytes
- workingCalendarId [long] - 8 bytes

  TOTAL: 24 bytes

Company:

- id [long] - 8 bytes
- name [String] - 10* bytes
- address [String] - 40 bytes
- averageRating [double] - 8 bytes
- workingCalendarId [long] - 8 bytes

  TOTAL: 74 bytes

CompanyEquipment:

- id [long] - 8 bytes

- companyId [long] - 8 bytes
- equipmentId [long] - 8 bytes
- quantity [int] - 4 bytes

  TOTAL: 28 bytes

Equipment:

- id [long] - 8 bytes
- name [String] - 20* bytes
- type [String] - 10* bytes
- description [String] - 50* bytes
- price [double] - 8 bytes

  TOTAL: 96 bytes

CompanyAdministrator:

- userId [long] - 8 bytes
- companyId [long] - 8 bytes
- isDownloaded [boolean] - 1 bytes

  TOTAL: 17 bytes

RegisteredUser:

- userId [long] - 8 bytes
- penalPoints [int] - 4 bytes

  TOTAL: 12 bytes

Role:

- id [long] - 8 bytes
- name [String] - 15* bytes

  TOTAL: 23 bytes

SystemAdministrator:

- passwordChanged [boolean] - 1 bit
- userId [long] - 8 bytes

  TOTAL: 9 bytes

User:

- id [long] - 8 bytes
- username [String] - 30* bytes
- password [String] - 8 bytes
- name [String] - 10* bytes
- surname [String] - 15* bit
- city [String] - 10* bytes
- country [String] - 15* bytes
- phoneNumber [String] - 13* bytes
- occupation [String] - 15* bytes
- companyInfo [String] - 10* bytes

- enabled [boolean] - 1 bit
- lastPasswordResetDate [Timestamp] - 8 bytes

  TOTAL: 143 bytes

Contract:

- userId [long] - 8 bytes
- companyId [long] - 8 bytes
- equipmentId [long] - 8 bytes
- quintity [int] - 4 bytes
- pickupDate [LocalDateTime] - 8 bytes
- status [int] - 4 bytes

  TOTAL: 40 bytes

ESTIMACIJA:

- Appointment 500 hiljada na mjesečnom nivou, godišnje 6 miliona
- QRcode rezervaciju pravi 50% registrovanih korisnika, 25 miliona
- QREquipment svaki od njih prosječno rezerviše dvije vrste opreme, 50 miliona
- WorkingCalendar obzirom na broj kompanija, sveukupno 15 miliona
- WorkingDay 22 prosječno u mjesecu, oko 250 godišnje, 3 750 000 000
- Company obzirom na broj korisnika, sveukupno 15 miliona
- CompanyEquipment svaka kompanija ima ponudu, sveukupno 200 miliona
- Equipment obzirom na broj korisnika, 100 miliona
- CompanyAdministrator 30% korisnika, 30 miliona
- RegisteredUser 50% korisnika, 50 miliona

- Role 3 moguće uloge
- SystemAdministrator 20% korisnika, 20 miliona
- User vrlo korišćeno, ukupan broj korisnika je 100 miliona
- Ugovre pravi 25% registrovanih korisnika, 25 miliona [NEW]

KONAČNO:

- Appointment 6 000 000 * 61 bytes * 5 = 1 800 000 000 = 1.8 GB
- QRcode 25 000 000 * 34 bytes = 850 000 000 = 0.85 GB
- QREquipment 50 000 000 * 28 bytes = 1 400 000 000 = 1.4 GB
- WorkingCalendar 15 000 000 * 17 bytes = 255 000 000 = 0.25 GB [NEW]
- WorkingDay 3 750 000 000 * 24 bytes * 5= 450 000 000 000= 450 GB
- Company 15 000 000 * 74 bytes = 1 110 000 000 = 1.11 GB
- CompanyEquipment 200 000 000 * 28 bytes = 5 600 000 000 = 5.6 GB
- Equipment 100 000 000 * 96 bytes * 5 = 48 000 000 000= 48 GB
- CompanyAdministrator 30 000 000 * 16 = 480 000 000 = 0.48 GB
- RegisteredUser 50 000 000 * 12 bytes = 600 000 000 = 0.6 GB
- Role 3 * 23 bytes = 69 bytes
- SystemAdministrator 20 000 000 * 9 bytes = 180 000 000 = 0.18 GB
- User 100 000 000 * 143 bytes = 14 300 000 000 = 14.3 GB
- Contract 25 000 000 * 40 = 1 000 000 000 = 1GB [NEW]

TOTAL: približno 525 GB [NEW]

- **Predlog strategije za postavljanje load balansera**

Imajući u vidu cilj da naša aplikacija bude visoko dostupna i skalabilna, neophodno je da ispravno postavimo i strategiju load balansera. Time bismo poboljšali i horizontalno proširili našu aplikaciju. Vertikalno širenje je uvijek izvodljivo adekvatnom nadogradnjom hardvera, memorije odnosno jačeg procesora.
Horizontalno širenje, koje ne bi narušilo stabilnost naše aplikacije, bi podrazumijevalo postepeno uvođenje dodatnih servera. Shodno tome, važnost izabranog algoritma load balansera dolazi do izražaja. Mi bismo se opredijelili za round-robin algoritam, koji bi ravnomjerno raspoređivao zahtjeve ka svim serverima, te nijednom ne bi došlo do momenta da je neki server nedovoljno korišćen, dok je jedan preopterećen.
Takođe, round-robin algoritam bi ubrzao rad naše aplikacije, ali bi i pored toga riješio još jedan potencijalni problem, a to je otkazivanje servera. Naime, load balanser sa round-robin algoritmom bi omogućio nesmetano i neprekidno funkcionisanje naše aplikacije u tom slučaju, što bi se odrazilo na opšti utisak korisnika.

[NEW]
Trenutna aplikacija i nije preterano horizontalno skalabilna niti je pogodna da se skalira horizintalno. Ipak je reč o monolitnoj aplikaciji koja na kraju krajeva ima samo jedan modul. U slučaju da je neka funkcionalnost aplikacije opterećena morali bismo podići čitav monolit u više instanci pa čak i one funkcionalnosti koje nam nistu potrebne.

Samim ti trebace nam više hardverskih resursa da podignemo više instanci monolitne aplikacije gde će većina funkcionalnosti biti neiskorišćena, a hardverski resursi nepotrebno zauzeti. Naravno vertikalno skaliranje je uvek izvodljivo ali i ne baš profitabilno jer bi često morali da nadogradjujemo našu hardversku infrastrukturu.

Load balancer je svakako dobra praksa koju bi trebalo koristiti. Sam load balancer bi vodio računa i sadražao pamet da kada je servisna instanca opterećena podigne novu instancu bez da korisnik bude svestan da koju instancu aplikacije korsiti, što znači da load balancer treba da sadrži i pamet za rutiranje zahteva ka instancama.

Definitivno bih razmislio da vremenom predjem na mikroservisne arhitekture, gde

bi horizontalna skalabilnost bila dosta izvodljivija jer je sama mikroservisna arhitektura pogodna za horizontalno skaliranje i korišćenje load balancer-a.

Verevatno bi vremenom trebale primeniti 'Strangler' pattern gde bi postepeno identifikovali poslovne funkcionalnosti koje bi izovlovali u nezavisne mikroservise sa istim end-pointim-a i tako postepeno 'gušili' našu monolitnu aplikaciju i prelazili na mikroservisnu arhitekturu.

Prelaskom na mikroservisnu arhitekturu bi poboljšali horizontalnu skalabilnost aplikacije, gde bit po potrebi podizali vise instanci samo onog mikroservisa koji nam je potreban u tom trenutku. Kada opterećenost jednog mikroservisa opadne srušili bi smo suvišne instance.

Takođe bih uveo API gateway koji bi rutirao saobraćaj, a ukoliko bi bilo potrebna i brza komunikacija verovatno bi bilo pogodno korisititi i gRPC (Remote Procedure Call)

Ovo bi poboljšalo dostupnost našeg sistema, bolju iskorišćenost servirskih resursa, takodje bi nam omogućilo nezavistnost tehnologija u kojoj razvijamo servis i proširilo mogućnost izbora skladišta podataka.

Takođe bi smo mogli da koristimo usluge cloud-a (Azure, Amazon AWS) i naša aplikacija bi vremenom postala SaaS (Softare as a Service) čemu većina kompanija danas teži.

- **Predlog koje operacije korisnika treba nadgledati u ciljupoboljšanja sistema**

Da bismo dijagnostifikovali sposobnosti naše aplikacije koje imaju prostora za unapređivanje, trebamo sprovesti detaljno praćenje i analizu korisničkih zahtjeva. U tome nam može pomoći koncept monotoringa, gdje bismo, računajući različite vrste metrika, poput aktivnosti sistema ili brzine odgovora na zahtjeve, detektovali nedostatke koji su se pojavili u međuvremenu. To bi bio preduslov za kontinuirano i kvalitetno funkcionisanje naše aplikacije. Uzimajući u obzir težnju ka proširenju

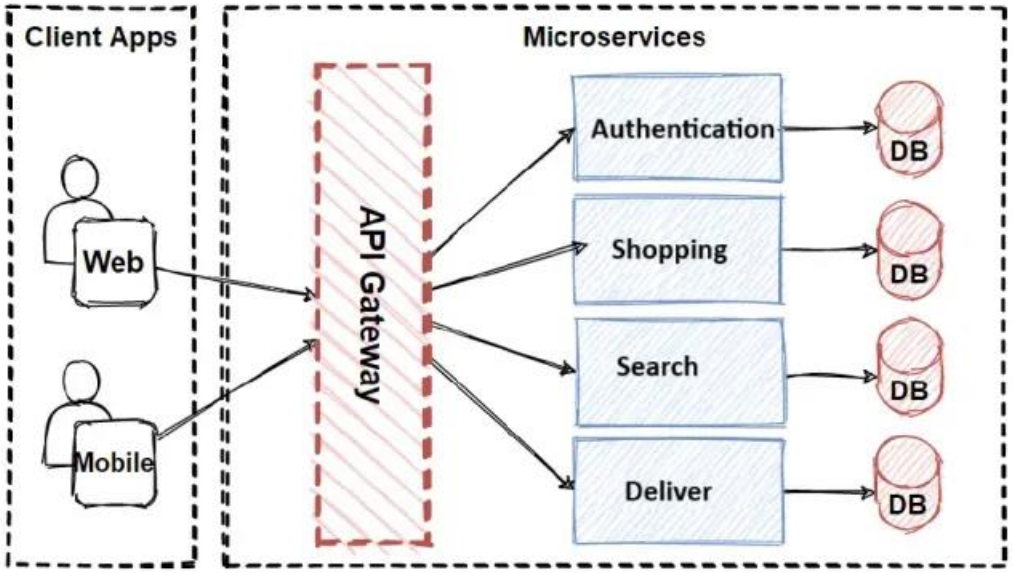
upotrebe naše aplikacije od strane korisnika, računanje takvih metrika bi bio siguran korak ka poboljšanju korisničkih usluga. U našem primjeru, ideja da se u aplikaciju implementira pametni sistem koji korisniku nudi opremu koja je najčešće pretraživana, ili kompanije koje su najčešće rezultat pretrage, apsolutno ima utemeljenje. Takođe, imalo bi smisla upustiti se u analizu statistike termina, zarad informacije o tome koji termini su najposjećeniji i koji stvaraju najveću gužvu i zadržavanje pri realizaciji. Bilo bi interesantno razmotriti mogućnost prikazivanja kvaliteta opreme u nekim kompanijama na osnovu ocjena registrovanih korisnika. Na kraju, naš sistem ima ugrađenu Prometheus datoteku, koja prikuplja osnovne informacije o aplikaciji, poput zauzeća memorije ili iskorišćenosti procesora; dok je zagrafički prikaz zadužena Grafana.

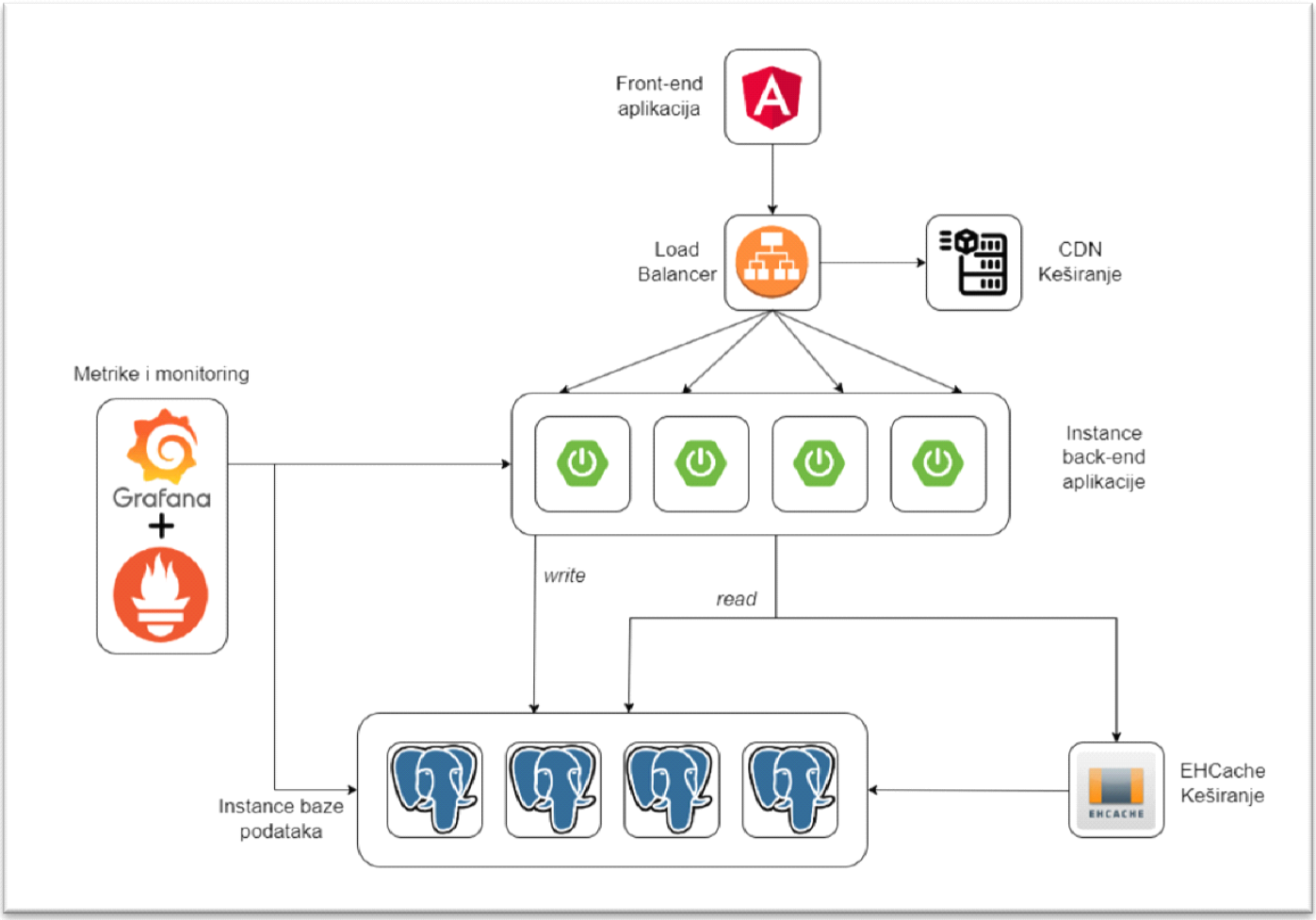
[NEW]

U nadašnje vreme praćenje zdravlja aplikacije je od velikog značaja pogotovo ukoliko predjenmo na mikroservisnu ahritekturu. Naš sistem bi mogao da koristi Loki logging kako bi u realnom vremenu mogli da nadgledamo i vršimo upite nad logovima naše aplikacije. To bi nam omogućilo da pratimo i bezbednosno kritične događaje. Takođe možemo dodati i metrike u realnom vremenu koristeći na primer cAdvisor koji prikazuje metrike i CPU mašine kao i Docker kontenjera.

Operacije koje treba nadgledati, pa u dobrom sistemu gotovo sve operacije treba nadgledati i logovati, ali od izuzetnog značaja je logovati i nadgledati bezbednso rizične operacije kao i operacije koje nisu idempotente i menjaju stanje servera.

- **Kompletan crtež dizajna predložene arhitekture**
  [NEW]


Client Apps
Microservices
Web
Mobile
API Gateway
Authentication
Shopping
Search
Deliver
DB
DB
DB
DB


Front-end aplikacija
Load Balancer
CDN Keširanje
Metrike i monitoring
Grafana
Instance back-end aplikacije
write
read
Instance baze podataka
EHCache Keširanje
EHCACHE