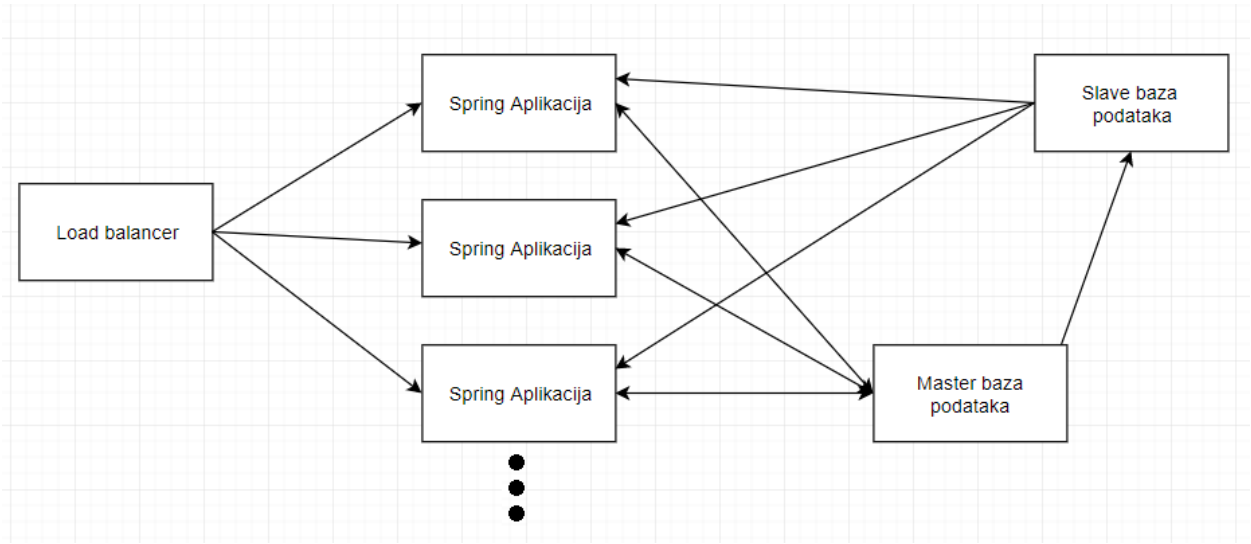
Povećavanje skalabilnosti

Naš projekat je rađen u *Java* jeziku pomoću *Spring Boot framework*-a, koji po *default*-u koristi *Tomcat* web server. S obzirom da je *Java* jezik koji podržava *multithreading*, jedna pokrenuta instanca naše aplikacije istovremeno može da opslužuje više korisnika, jer će se za svaki pristigli zahtev od klijenta napraviti zasebna nit. Ove niti se u *Tomcat*-u ne kreiraju pri svakom pristiglom zahtevu, nego se radi *Thread pooling*, to jest postoji određen broj unapred spremnih niti koje čekaju na klijentske zahteve. Ovo ujedno predstavlja i prvo mesto gde se može vršiti optimizacija radi povećanja skalabilnosti: menjanjem broja niti u *pool*-u u zavisnosti od trenutnog doba dana (noću očekujemo manje saobraćaja i imamo manje niti). Isti koncept se može primeniti i na *pool* konekcija sa bazom podataka.

Problem nastaje kada povećavanjem veličine *pool*-a ne možemo više da povećamo brzinu usluživanja klijentskih zahteva. Ovo se dešava iz prostih hardverskih ograničenja, jer procesor ima ograničen broj jezgara na kojima paralelno može da izvršava sve aktivne niti.

Prvo rešenje ovog problema jeste vertikalno skaliranje, to jest unapređenje postojećeg hardvera. Povećanjem broja jezgara na procesoru, kao i veličine radne memorije, možemo donekle rešiti problem velikog broja istovremenih zahteva.

Drugo rešenje jeste horizontalno skaliranje, to jest pokretanje naše aplikacije na više fizički odvojenih uređaja. Ovo rešenje može drastično više da se skalira nego prvo, ali uvodi probleme sinhronizacije stanja aplikacije između ovih uređaja.



Slika 1 – Skica osnovne arhitekture aplikacije

Horizontalno skaliranje bismo izveli, tako što bismo imali više zasebnih uređaja (servera) na kojima se izvršava naša *Spring* aplikacaija, kao što se vidi na slici 1. *Load balanser* je uređaj koji ima javnu *IP* adresu i njega klijent gađa kada želi da poseti našu veb aplikaciju. Njegova uloga je da prosledi klijentov zahtev serveru koji je trenutno najmanje opterećen. Radi veće bezbednosti, možemo dodati i još jedan redudantni *slave load balancer* koji se aktivira u slučaju da prvi otkaže i tako smanjuje *down time* aplikacije.

Ovakva arhitektura se može izvesti na našoj aplikaciji, jer je komunikacija sa klijentom potpuno *stateless*, to jest podaci na serveru se ne vezuju za sesiju. Naša aplikacija dobija podatke o ulogovanom klijentu pomoću tokena koji on šalje u zaglavlju svakog svog zahteva. Ostatak potrebnih podataka dobija pomoću komunikacije sa bazom podataka, koja se horizontalnim skaliranjem može izdvojiti na zaseban uređaj.

Ovo može biti *PostgreSQL* baza podataka, koja, između ostalog, omogućava *master-slave* režim rada. *Master* baza podataka predstavlja glavnu bazu, u kojoj se radi upis i iz koje se podaci iščitavaju. *Slave* baza podataka se nalazi na *standby* serveru i ona postaje *master* baza u slučaju da se server *master* baze sruši. Sve transakcije koje se obavljaju na *master* bazi su uspšeno *commit*-ovane tek ako su uspešno zapisane i u *master* i u *slave* bazu. Ovaj nivo zaštite se naziva *2-safe replication*. Iz tog razloga, sadržaj *slave* baze podataka je uvek sinhronizovan sa *master* bazom i ona se može koristiti za čitanje podataka i tako rasteretiti saobraćaj sa glavnom bazom. Takođe, moguće je dodati više *slave* baza kako bismo još poboljšali performanse.

Dodatne optimizacije

Da bismo dodatno rasteretili opterećenje baze podataka, možemo uvesti keširanja učitanih podataka u našoj aplikaciji. *Spring Hibernate* nam po *default*-u nudi *L1* keširanje, koje nam omogućava da, unutar jedne sesije, svaki zahtev za objektom iz baze uvek vraća istu instancu objekta. Takođe, postoji i *L2* keširanje, koje se odnosi na privremeno čuvanje objekata dobijenih iz baze na serveru, tako da se pri novom upitu ka bazi prvo proverava da li se traženi objekti već nalaze u kešu. Konfigurisanjem *TTL* (*time to live*) se može odrediti dužina trajanja ovog keša i time balansirati između opterećenja baze i ažurnosti podataka.

Našu arhitekturu možemo poboljšati uvođenjem redova poruka (*message queue*) i korišćenjem mikroservisa. Red poruka nam omogućava da se svi zahtevi pristigli od klijenta skladište i čekaju na svoj red da bi bili obrađeni. Na ovaj način garantujemo da će se svi zahtevi obraditi i to u redosledu pristizanja. Takođe, u slučaju da se neki server sruši tokom obrade zahteva, taj zahtev (pošto se ne uklanja iz reda sve dok nije uspešno obrađen) će biti prosleđen drugom serveru.
Mikroservisna arhitektura se odnosi na izdvajanje različitih komponenti našeg sistema u zasebne aplikacije. U našem slučaju, razdvojili bismo aplikaciju na 4 dela: za hotele, za avio-kompanije, za rent-a-car servise i za rezervacije. Pored činjenice da je dekomponovanjem na manje celine olakšan dalji razvoj aplikacije, takođe možemo bolje da skaliramo apliakciju, tako

što ćemo pokrenuti više instanci onog mikroservisa koji je najviše korišćen (u našem slučaju, avio-kompanije su posećenije od hotela i rent-a-car servisa zbog specifikacije projekta).

Brzinu pristupanja sadržaju naše veb aplikacije značajno možemo poboljšati korišćenjem *Content Delivery Network*-a, to jest mreže servera lociranih širom sveta koji služe da keširaju statički sadržaj naše aplikacije (*html* stranice, *JavaScript* kod, slike ...). Kada klijent zatraži neki od statičkog sadržaja, zahtev će biti poslat najbližem *CDN* serveru i tek u slučaju da na njemu taj sadržaj već nije keširan, zahtev se prosleđuje našoj aplikaciji. Ovako se takođe smanjuje i opterećenje naših servera, jer umanjujemo broj pristiglih zateva.

U slučaju da korišćenjem svih prethodnih metoda i dalje imamo preopterećenje *Load Balancer*-a, kao i *Master* baze podataka, možemo probati da napravimo više klastera sa slike 1 i da koristimo *DNS Load Balancing*. Potrebno je da se na *DNS* naziv naše veb aplikacije privežu *IP* adrese *Load Balancer*-a svakog od naših klastera. Tada klijent pri slanju zahteva automatski bira *IP* adresu sebi najbližeg *Load Balancer*-a i time se znatno povećava brzina odgovora aplikacije. Ovo rešenje nam otvara problem sinhronizacije između *master* baza podataka svakog od ovih klastera. Pošto ovi klasteri ne bi bili geografski blizu jedni drugih, smatramo da nije potrebna prevelika ažurnost podataka između njih, te bi se ove master baze sinhronizovale na neki određeni vremenski period, a ne pri svakoj transakciji.