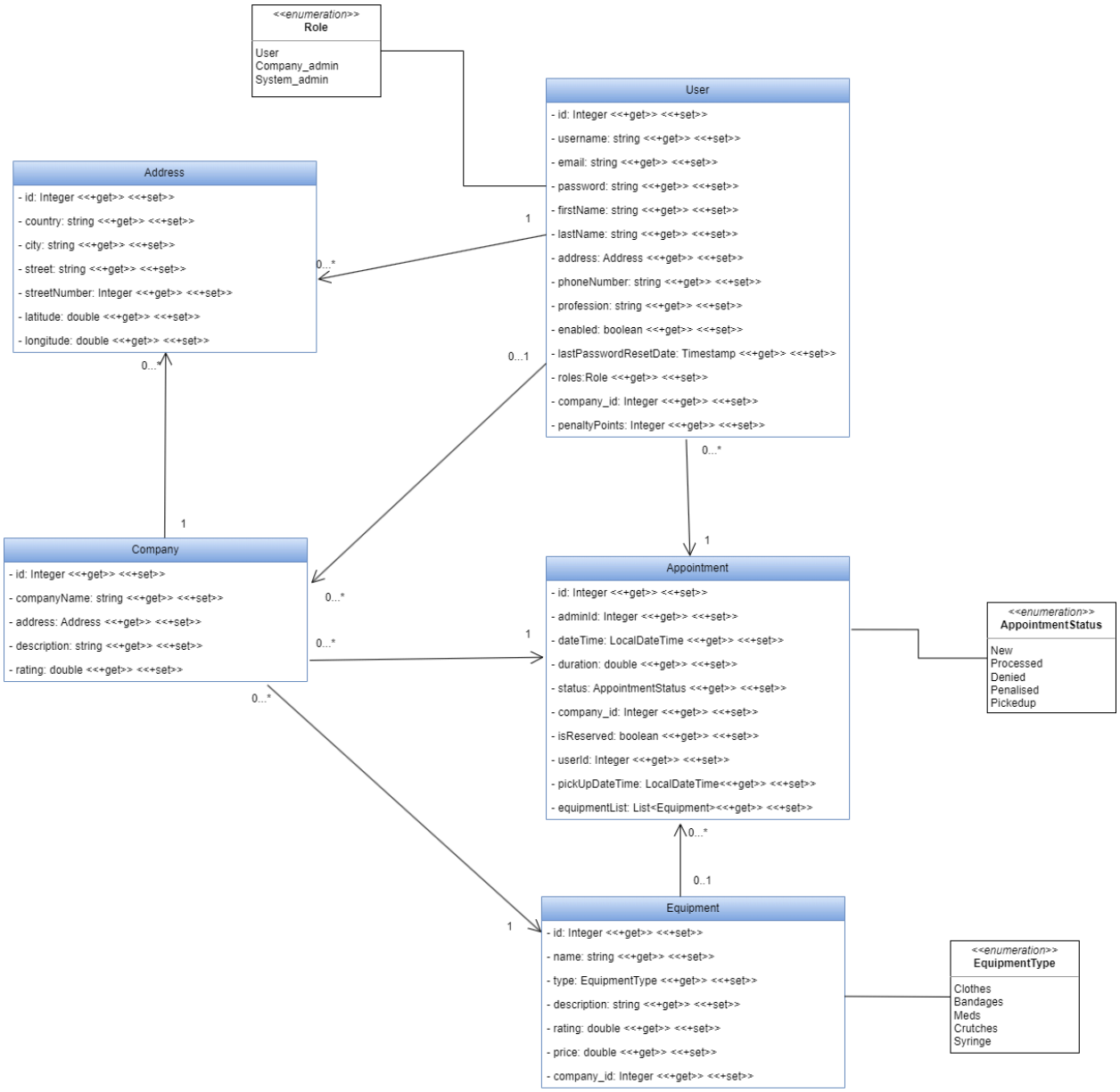
Proof of Concept

Dizajn seme baze podataka



Predlog strategije za particionisanje podataka

Ako bi broj korisnika vremenom prestigao mogucnosti nase aplikacije, dobru pomoc bismo nasli particionisanjem podataka i to podelom velikih tabela na vise manjih. Postoji mogucnost da particioniramo podatke po ucestalosti koriscenja i to ako pretpostavimo da nam nece uvek u svakom trenutku biti potrebni svi podaci o useru, samim tim bismo i tu tabelu mogli particionisati.

Delimo tabelu na jednu koja ce sadrzati podatke koji ce se cesto koristiti, i na tabelu sa onim redje koriscenim podacima.
Horizontalno particionisanje bismo mogli da upotrebimo za appointment-e i istoriju preuzimanja, gde bi mozda najbolje bilo staviti ih u particije za odredjen vremenski period, recimo za svaki mesec ili za po polovinu meseca.
Mogli bismo mozda koristiti i geografsko particionisanje, jer ce najcesce opremu narucivati iz svog kontinenta, a jako retko iz nekog drugog. Tako ce korisnici moci brze da pristupaju podacima, koji ce se nalaziti na serverima koji su blizu.

Predlog strategije za replikaciju baze i obezbedjivanje otpornosti na greske

Replikacija nam moze pomoci u vise aspekata. Ako pored ‘glavne’ baze podataka imamo jos par replika, odnosno kopija te baze, dakle imamo vise identicnih baza, Ovo bi predstavljalo neku vrstu sigurnosti za nase podatke u slucaju da nam padne baza. Takodje, ovo povecava mogucnosti naseg sistema da rukuje sa visestrukim zahtevima za citanje podataka.
Dodatno ubrzanje i rasterecenje bi se moglo postici koriscenjem master-slave replikacije, gde bi master bio koriscen za write naredbe, i master bi onda prosledio slav-u te promene, ali to bi bilo asinhrona replikacija. To bi dosta ubrzalo performanse, jer upisivanja u bazu ne bi bila usporena citanjem iz nje. Slave nam sluzi kao backup podataka. Takodje, ako je master baza zauzeta pisanjem, master uloga baze se redirektuje na neku od slave baza ako je potrebno upisivanje podataka u tom trenutku. Master-slave replikacija je podrzana u PostgreSQL-u.
Mana asinhronog master-slave-a je u tome sto ako master padne pre nego sto je prosledio promene slave-rima, tada se te promene zauvek gube, dok bi se kod singronog to izbeglo.

Predlog strategije za kesiranje podataka

Za kesiranje podataka bismo mogli koristiti Redis. Ovaj pattern nam omogucava cache-aside, odnosno lazy-loading strategiju, kao i write-through strategiju. Cache-aside (lazy-loading) strategija bi nam bila korisna jer bi kes sadrzao samo podatke koje aplikacija zapravo koristi, i na taj nacin bi se cuvala ‘memorija’ u kesu. Ovo bi moglo da se koristi za sve podatke u nasoj aplikaciji. Redis takodje podrzava i write-through strategiju koji omogucava da nam u kesu uvek budu azurni podaci, odnosno, ako se u kesu nalazi neki podatak i u medjuvremenu nam se taj podatak azurira, automatski ce biti azurirani podatak i u samoj bazi. To ce omoguciti bolje iskustvo korisnika i bolje performanse nase aplikacije, jer ce podatak biti idalje u kesu, sto ce smanjiti pristup bazi, radi citanja podataka. Mana ovoga je sto bi kes morao biti obimniji, samim tim i skuplji. Ako bismo iskombinovali lazy-loading i write-through strategije, postigli bismo lepe performanse cele nase aplikacije. Postigli bismo veliku konzistentnos podataka.

Okvirna procena za hardverske resurse potrebne za skladistenje svih podataka u narednih 5 godina

- Sladistenje korisnika:
  Posto podaci o korisniku zauzimaju u proseku 2.2 KB, a treba cuvati 100 miliona korisnika, bice nam potrebno priblizno 220GB
- Skladistenje kompanija:
  Posto podaci o kompaniji zauzimaju u proseku 9.6 KB, a treba cuvati 100 miliona kompanija, bice nam potrebno priblizno 960 GB
- Skladistenje rezervacija:
  Posto podaci o rezervaciji zauzimaju u proseku 2.7 KB, a da je prosecna mesecna kvota oko 500 000 rezervacija, za period od 5 godina bice nam potrebna priblizno 78 GB
- Skladistenje opeme:
  Posto podaci o opremi zauzimaju u proseku 3.6 KB, a treba cuvati 100 miliona razlicite opreme, bice nam potrebno priblizno 360 GB

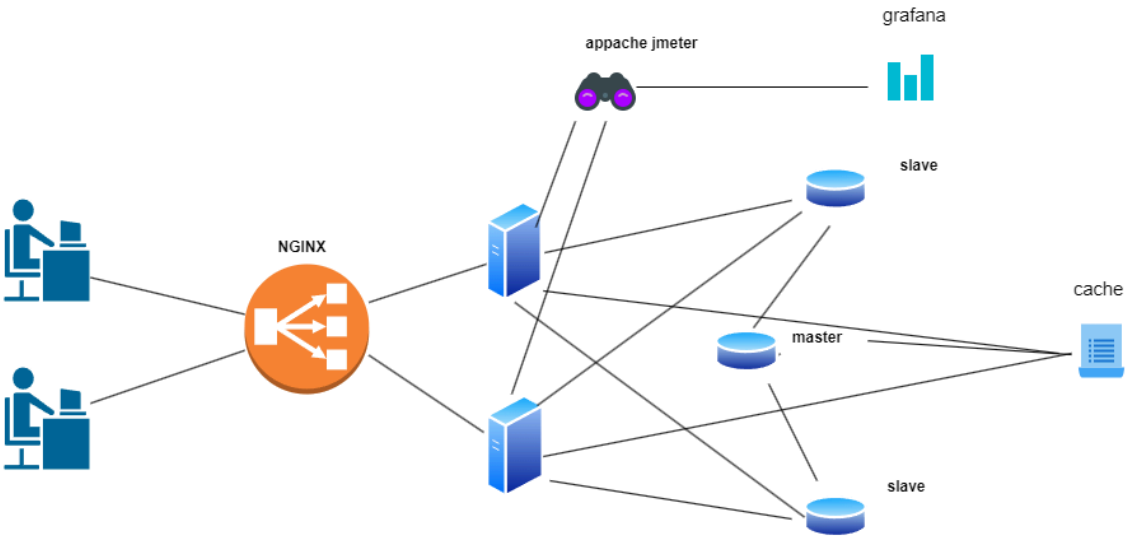
Predlog strategije za postavljanje load balansera

Najbolje za nasu aplikaciju bi bilo da koristimo NGNIX open source http server za load balansera. Razlog zasto ga uzimamo je taj sto ima mogucnost balansiranja izmedju vise servera i podrzava JWT autentifikaciju, a nasa aplikacija upravo koristi JWT tokene. Najbolju od metoda koje NGNIX nudi za load balansiranje za nasu aplikaciju bi bila round robin (rotacija) koja ima jednostavnu implementaciju i ravnomernu raspodelu opterecenja izmedju servera.

Predlog koje operacije korisnika treba nadgledati u cilju poboljsanja sistema

Pracenjem korisnika i njegovih akcija mozemo uvek unaprediti nas sistem. Mozemo pratiti koju opremu korisnici najcesce pretrazuju i narucuju, iz koje kompanije najvise narucuju, tako bismo mogli da pravimo predloge na osnovu njegovog ponasanja. Pracenje njegovih aktivnosti, pretraga, narucivanja bi nam pomoglo da steknemo uvid kakve nove funkcionalnosti bismo mogli da kreiramo. Propusnost same aplikacije bismo mogli pratiti koriscenjem open source alata Appache JMeter, dok bi za graficki prikaz te propusnosti koristili Grafana-u.

Komplet crtez dizajna predlozene arhitekture (aplikativni serveri, serveri baza, serveri za kesiranje, itd)


grafana
appache jmeter
slave
NGINX
cache
master
slave