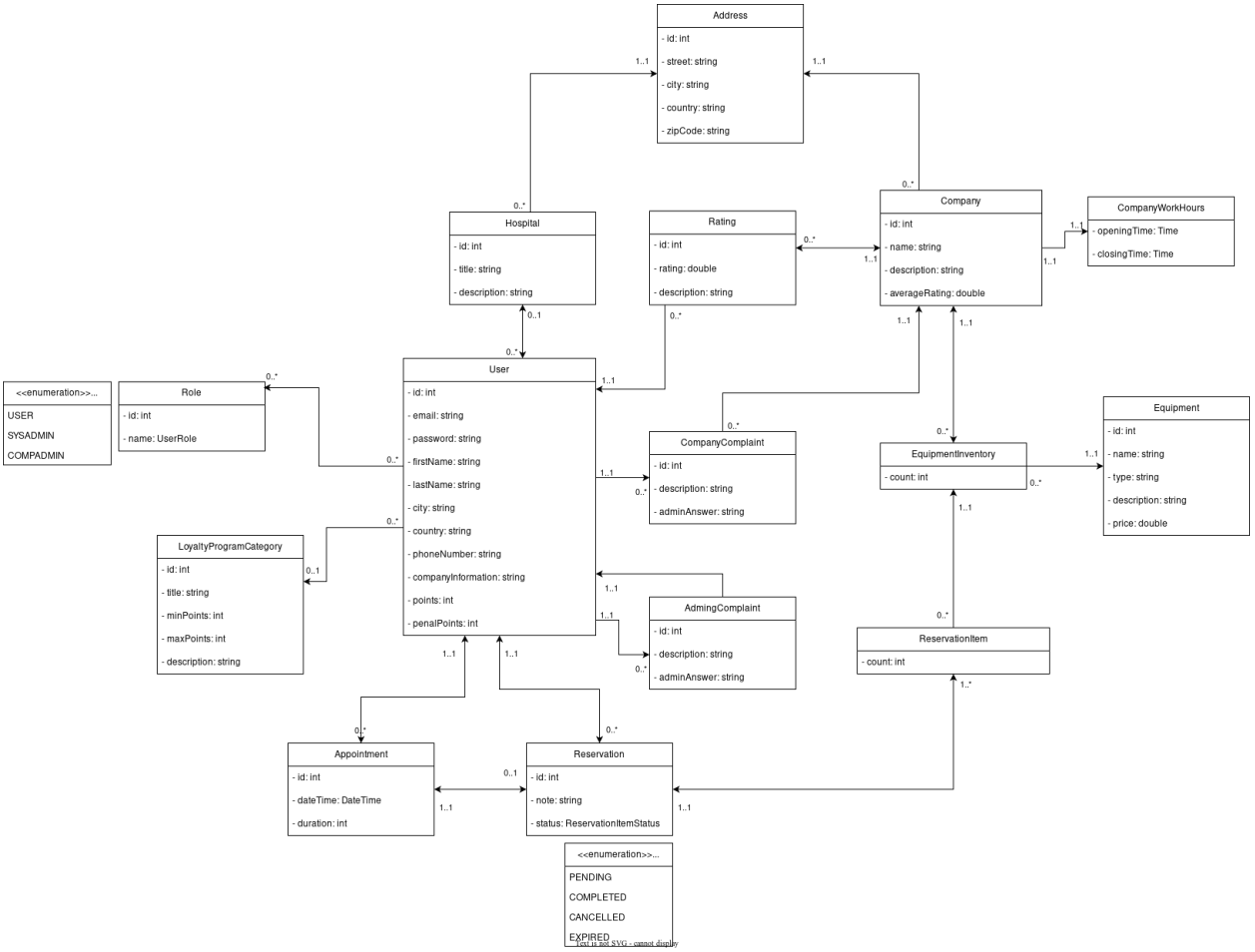
Predlog za skalabilnu arhitekturu aplikacije Tim 46

Uvod: Ovaj dokument predstavlja *Proof of Concept* arhitekturu koja treba da obezbedi skalabilnost aplikacije za upravljanje medicinskom opremom. Pretpostavke:

- Ukupan broj korisnika aplikacije: 100 miliona
- Broj rezervacija svih entiteta na mesečnom nivou je 500.000
- Sistem mora biti skalabilan i visoko dostupan

1. Dijagram klasa sistema



2. Predlog strategije za particionisanje podataka

Kada sagledamo model podataka uviđamo podelu na podatke koji se redovno ažuriraju i podatke koji se retko ažuriraju. Na primer: Podatke o korisniku (tabela: *User*) možemo vertikalno particionisati na podatke o korisničkim informacijama (ime, prezime i slično) i na tabelu sa korisničkim poenima koji se redovno ažuriraju u toku korišćenja

aplikacije. Sa druge strane, s obzirom da sistem opslužuje veliki broj korisnika, uvodimo dodatnu pretpostavku u kojoj smatramo da su korisnici rasprostranjeni na više geografskih regiona. U tom slučaju strategija za horizontalno particionisanje bi bila jednostavna, particionisali bi podatke po regionima.

3. Predlog strategije za replikaciju baze i obezbeđivanje otpornosti na greške

Za replikaciju baze podataka predlažemo read-only način replikacije ili *Single Master Replication*. Ovaj način replikacije određuje jedan server baze podataka koji će imati ulogu primarnog servera i više drugih servera koji imaju ulogu servera replike. Primarni server ima dozvolu za ažuriranje baze, dok serveri replike imaju samo pristup za čitanje podataka. Ova strategija rasterećuje primarni server od *read-only* upita i poboljšava performanse procesiranja transakcija. Održavanje konzistentnosti je obezbeđeno propagacijom podataka sa primarnog servera na severe replike. U želji da imamo robustan sistem predlažemo *full-table* tip replikacije koji je sporiji ali kreira repliku svih podataka iz tabela odabranih za replikaciju.

Replikacijom obezbeđujemo otpornost na greške jer eliminišemo SPoF (*Single point of failure*) jer uvodimo redundantnost podataka kojima možemo oporaviti primarni server u slučaju havarije, dok server replika preuzima odgovornost primarnog servera. Kako naša odabrana baza podataka - *PostgreSql* nema ugrađenu detekciju grešaka, moramo koristiti alate kao što su *EDB Postgres Failover Manager*.

4. Predlog strategije za keširanje podataka

Pored pozitivnog uticaja na dobro korisničko iskustvo, keširanje podataka rasterećuje servere sa bazama podataka što pozitivno utiče na skalabilnost i dostupnost aplikacije. Kako naša aplikacija implementira *REST* arhitekturu, ona je potpuno *stateless* zbog čega će se naš predlog fokusirati na *L2* tip keširanja. Naš predlog je implementacija *L2* keširanja koristeći *EhCache*, jednog od najpopularnijih eksternih provajdera za *L2* keširanje u java aplikacijama. Podaci koje smatramo da su pogodni za keširanje su podaci o privatnim kompanijama (**napomena:** ovo ne uključuje njihov inventar raspoložive opreme), podaci o korisnicima, kao i ocenama aplikacije. Što se tiče strategije keširanja prelažemo *cache-aside* raspored, u kojem aplikacija prvo proverava da li su traženi podaci keširani, ako jesu preuzimaju se iz keša, a ako nisu pristupa se bazi podataka i ažurira se keš. Mana ovog pristupa je što prilikom ažuriranja baze podataka postoji prozor u kojem se keširani podaci ne ažuriraju dok ne isteknu. Kako su podaci koje keširamo većinski statički ili se retko se ažuriraju smatramo da je ovaj *trade-off* vredan jednostavnosti implementacije. Kako bi umanjili mane ovakvog rasporeda smo se odlučili za *NONSTRICT READ WRITE* strategiju ažuriranja keša koja radi ažuriranje po završetku transakcije koja menja keširane podatke.

Kako smo već uveli pretpostavku o geografskoj rasprostranjenosti naših korisnika, poželjno je da obezbedimo keširanje statičkih podataka frontend dela aplikacije (html,

javascript, css, slike i slično) korišćenjem *CDN* (*Content delivery network*) sistema. Glavni benefiti ovog vida keširanja su povećana brzina učitavanja web stranica, ali i potencijalno unapeđenje bezbednosti naše aplikacije kroz *DDoS* zaštitu koju pružaju mnogi provajderi *CDN*-ova.

5. Okvirna procena za hardverske resurse potrebne za skladištenje podataka za narednih 5 godina

Kako naš sistem većinski obrađuje podatke o korisnicima, opremi i rezervacijama opreme fokus naše procene će biti na podacima vezanim za ove entitete.

Entitet	Prosečna veličina koju zauzima jedan red u tabeli	Procena resursa potrebnih za skladištenje podataka za narednih 5 godina	Pojašnjenje
Korisnik	~200 B	19 - 20 GB	Aplikacija opslužuje 100 miliona korisnika
Rezervacija	~280 B	7.82 GB	Pretpostavljamo: - Mesečno je kreirano 500.000 rezervacija i da svaka rezervacija ima 5 stavki
Oprema i inventar	~73 B oprema ~43 B inventar	0.7 MB oprema 0.5 GB invenar	Pretpostavljamo: - Prosečna kompanija u invenatru ima oko 300 vrsta opreme - Postojaće oko 10.000 vrsta opreme u sistemu
Ostalo	-	5 - 10 GB	-

Dodatna pojašnjenja: Na sajtu *Europages* smo pronašli informaciju da trenutno u Evropi ima oko 12.000 kompanija koje prodaju medicinsku opremu, naša procena je da se ovaj broj neće mnogo menjati tokom sledećih 5 godina, ali smo u procenama računali da imamo 15.000 kompanija.

Napomena: Planirano je da se u bližoj budućnosti aplikacija unapredi tako da podržava čuvanje slika vezanih za korisnike, kompanije, opremu i slično. Ovo će imati uticaj na zahtev hardverskih resursa.

6. Predlog strategije za postavljanje *load balancera*

Kako očekujemo da korisnici duži vremenski period provode povezani na aplikaciju odlučili smo da to bude merilo po kojem će *load balancer* da određuje kojem serveru će proslediti zahtev. *Least connection* algoritam, kako mu ime nalaže, prosleđuje zahtev serveru sa najmanjim brojem aktivnih konekcija. U slučaju da želimo vertikalno da skaliramo određeni broj naših servera, možemo izabrati da im dodelimo težine (eng: *weight*) i uvesti *weighted least connection* algoritam. Ovaj algoritam je nadogradnja na prethodni u smislu da u obzir dodatno uzima i dodeljene težine servera. Ako imamo hardverski jači server dodelićemo mu veću težinu kako bi algoritam više posla dodeljivao njemu. Kako je jedan od softvera koji su industrijski standard odlučili smo se da posao *load balancera* izvršava *NGINX*.

Kao dodatan način za smanjenje troškova održavanja servera, predlažemo da se na funkcionalnosti za koje je procenjeno da su najzahtevnije postavi *rate limiter* koji će sprečiti korisnika da šalje previše zahteva za ove funkcionalnosti u kratkom vremenskom periodu.

7. Predlog operacija značajnih za nadgledanje u cilju poboljšanja sistema

Operacije koje obrađuju podatke o rezervacijama (zakazivanje, praćenje, preuzimanje) poseduju najviše poslovne logike i kao takve predstavljaju primarni cilj za nadgledanje. Praćenjem performansi aplikacije tokom izvršavanja ovih zahtevnih operacija možemo bolje da razumemo kako se sistem ponaša i kako možemo da ga optimizujemo. Pored ovoga, želimo da prikupljamo podatke o ponašanju korisnika i povratne informacije o potencijalnim načinima kako možemo da unapredimo sistem. Alati kao što su *Prometheus* i *CheckMK* su *open source* i naša su preporuka za rešenje problema nadgledanja.

8. Dijagram predložene arhitekture