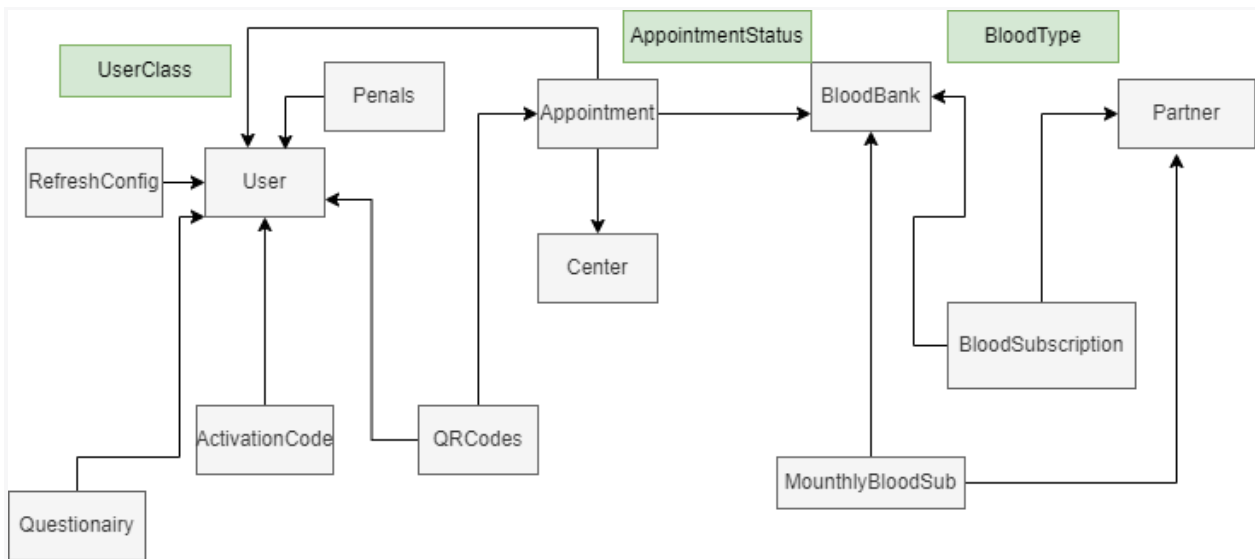


# Skaliranje

## 1. Dizajn šeme baze podataka

- Konceptualni dizajn
- Logički dizajn
- Fizički dizajn

Sistem koji je već implementiran ima sređene veze između klasa. Korišten je spring boot zarad rukovanja entitetima. Korišten je JPA interfejs za postavljanje upita ka bazi, a kao baza korištena je PostgresDB. Svaka tabela ima polje id koji predstavlja ključ. To je jedinstvena vrijednost koja se automatski generiše prilikom pravljenja objekta klase. Većina upita je sinhrona, dok su neki upiti radjeni asinhrono, kao što je dobavljanje perioda ažuriranja. Zarad povećanja performansi, u repozitorijumima su postavljeni *custom* upiti k bazi, koji za cilj dobavljanje već filtriranih listi kako se ne bi izbjeгла implemntacija na nivou servisa. Filtriranja i sortiranja su izmještena na frontend, kako se više puta ne bi slali isti upiti.



## 2. Strategija za particionisanje podataka

Strategija particionisanja podataka igra ključnu ulogu u optimizaciji performansi, upravljanju velikim količinama podataka i skalabilnosti sistema.

Pošto je na početku potrebno obezbjediti resurse koji će zadovoljiti sve zahtjeve korisnika, najbolje je koristiti vertikalno skaliranje. Odnosno, obezbjediti hardver sa dovoljno procesorske moći i sa dovoljno memorije. Nakon toga nakon što se poveća broj korisnika mogu se dodati novi serveri. U slučaju da se broj korisnika počne povećavati, ovo nije dovoljno, pa je potrebno razmisliti i strategiji particionisanja monolitne u mikroservisnu arhitekturu. Tu bi imali mikroservis za autentifikaciju i autorizaciju, mikroservis za upravljanje korisničkim nalogima, mikroservis za upravljanje pregledima i mikroservis za nabavku/iskoristiti klijentima. U zavisnosti od daljeg razvoja treba razmisliti da je bolje imati svaku odvojenu bazu za svaki mikroservis ili dijeljenu bazu. U svakom slučaju najbolje je da imamo jednu bazu u servisu za autentifikaciju i autorizaciju, pošto je ovo jedan od najkorištenijih servisa.

### **3. Strategija za replikaciju i otpornost na grešaka**

Pošto je potrebno osigurati da svi servisi budu visko dostupni, na nivou ove arhitekture je najlakše replicirati service na više servera. U slučaju otkaza jednog, nastavit će sa radom drugi serveri, iako će trpiti povećano opterećenje. Ovo iziskuje dosta više materijalnih resursa. U slučaju kada bi se implementirala mikroservisna arhitektura, bilo bi jednostavnije izanalizirati ranjive service, koji često trpe otkaze i za njih obezbjediti rezervne resurse.

### **4. Strategija za keširanje podataka**

U trenutnom sistemu implementirano je keširanje na nivou *spring boot* frejvorka. Servisi koji često rade sa istim podacima podžavaju keširanje. Trenutno je podržano keširanje sa maksimalno 100 instanci. Smatramo da je ovaj broj dovoljan, da smanji opterećenje ka bazi, a u isto vrijeme da obezbjedi konzistentne podatke. U slučaju da se iskoristi svih 100 mjesta, brišu se najstariji ili najmanje korišteni objekti.

U slučaju potrebe može se implementirati *Redis* baza podataka koja će služiti za keširanje veće količine podataka. Ovo je pogodno, jer se očekuje povećanje broja korisnika, a samim tim i količina podataka i količina upita k bazi.

### **5. Procena hardverskih resursa**

Prije nego što se sistem smjesti na neku hardversku platformu najbolje, je implementirati neko početno rješenje. Nakon toga potrebno je izanalizirati trenutno stanje sa podacima, brojem zahtjevai slično. Nakon toga potrebno je odrediti strategiju skaliranja, da li se radi vertikalno ili horizontalno skaliranje.

## 6. Strategija za postavljanje load balansera

U pocetnim fazama razvoja nema potrebe za load balanserima. Nakon što se uvedu nove instance servera, javiće se potreba za loud balanserom. Na samom početku potreno je uraditi analizu saobraćaja i otkriti kitične dijelove kako bi se na njima implementirao loudbalanser kako bi pmanjio opterećenje na nekim dijelovima. Jedan od narednih koraka je da se izabere pogodan loud balanser i da se konfiguriše. On prije svega treba biti robusan, pošto mora da podrži sve zahjeve koji stižu k aplikaciji. Takođe, treba razmotriti, da li će load balanseri slati zahtjeve ka svim serverima, ili će postojati rezervni serberi koji se aktiviraju kada jedan od aktivnik otkáže. Pa će na osnovu potrebnih zahtjeva biti implementiran jedan od sledećih način aloud balansinga *Round Robin*, *Least Connections* ili *Weighted Load Balancing*.

## 7. Nadgledanje korisničkih operacija

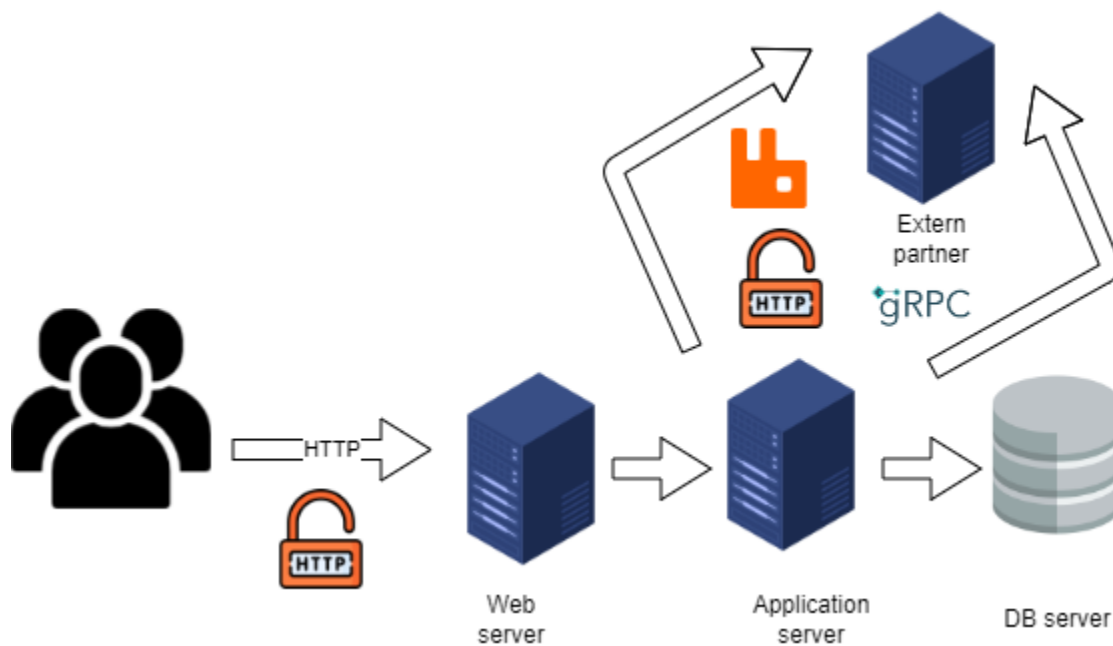
Samom upotrebom spring boot frejmworka urađen je ispis logova. U slučaju pojave neke greške ispisaće se ta informacija u konzoli. U metodama je implementiran ispis poruka u slučajevima kada se desi neki izuzetak ili kada je neka aktivnost uspješno odrađena. Nedostaje trajno čuvanje logova, jer bi se u slučaju kraha servera izgubila inormacija zbog čega je to došli. Zarad bolje analize performansi, praćeja opterećenja i daljih strategija optimizacije sistema najbolje uvesti sistem monitriga koji bi pratio korisnčke zahtjeve, upotrebu resursa i sve što to podrazumjeva. Svakako je ova komponenta korisna kod daljih analiza sistema zarad unaprijeđenja sistema. Svakako bi bilo korisno da se uspostavi i nadgledanj clijeneta-korisnika usluga banke koji često otkazuju termine. Trebalo bi pokušati odrediti šta je moćući uzrok čestog otkazivanja i da li im sistem može ponuditi termine koji im više odgoavara.

## 8. Zaključak

U radu je predstavljen prijedlog strategije za prvi period rada, kao i mogući prijedlozi koji se mogu koristi u slučaju povećanja broja korisnika. Dalju strategiju je na početku teško odrediti, moguće je raditi na osnovu prethodnig iskustva ili iskustva

drugih. Nakon određenog vremena, kada se upotreba sistema poveća, najbolje je raditi analizu zahtjeva, potreba, trenutnih performansi i na osnovu toga implementirati neku od navedenih strategija.

### Trenutna arhitektura



### Unaprijeđena arhitektura

