SKLOPKE ZA DOSTUPNOST ZNAČAJKI UNUTAR MIKROSERVISNE OKOLINE

Macan, Marko

SVEUČILIŠTE U SPLITU

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Preddiplomski stručni studij informacijske tehnologije

MARKO MACAN

Z A V R Š N I R A D

SKLOPKE ZA DOSTUPNOST ZNAČAJKI UNUTAR MIKROSERVISNE OKOLINE

Split, srpanj 2020.

SVEUČILIŠTE U SPLITU

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Preddiplomski stručni studij informacijske tehnologije

Predmet: Programiranje u Javi

Z A V R Š N I R A D

Kandidat: Marko Macan

Naslov rada: Sklopke za dostupnost značajki unutar mikroservisne okoline

Mentor: Josip Vrlić, predavač

Split, srpanj 2020.

Sadržaj

1. Sažetak

Tema ovoga završnog rada je sustav pomoću kojeg ostali servisi unutar mikroservisne okoline mogu dinamički mijenjati putanju izvršavanja koda na osnovu klijenta koji mu pristupa. Takav sustav je poznat pod nazivom *feature flags* ili *feature toggles*. *Feature flags* je sistem koji omogućava da se na siguran način konstantno i ubrzano izbacuju i testiraju nove značajke i promjene unutar proizvoda na produkcijskoj okolini bez straha da će nova verzija aplikacije ugroziti izvršavanje aplikacije jer se vrlo lako jednim klikom kroz administracijsko sučelje može obraniti ako taj kod zaista unosi greške u operaciji aplikacije. *Feature flags* sustav je priznat kao dobra programerska praksa s mnogo prednosti, a u svom razvojnom procesu ga koriste velike kompanije kao što su Google i Facebook

Ključne riječi: *sklopke značajki, mikro servisi, Docker, Spring*

2. Summary

Feature flags in microservices environment

Topic of this paper is system trough which other services in microservice architecture can dynamically change way of their code execution based on who is accessing it. That system is known as feature flags or feature toggles. Feature flags is system which allows safe, continuous and rapid releasing and testing of new features and changes inside of product without worrying that new features might threaten execution of application because with one click on administration panel can disable execution of faulty code. Feature flags is acknowledged as a good programming practice with lot of benefits, and big companies as Google and Facebook are using them in their development process.

Keywords: *feature flags, microservices, Docker, Spring*

3. Uvod

Programeri u svom radu često se nalaze u situaciji gdje se od njih traži da što brže razviju neku novu značajku, a također moraju biti odgovorni za svoj rad i umanjiti mogućnost greški u programu. Takvim situacijama programeri su doskočili tako da su popularizirali korištenje sustava *feature flagova* kako bi se zaštitili od greški pri ubrzanom razvoju.

Cilj ovog rada je prikazati prednosti *feature flag* sistema i razviti mikroservis koji bi pružio *endpointe* preko kojih drugi mikroservisi mogu provjeriti smiju li izvršiti neki kod za određenog korisnika ili ne, te sučelje za administraciju (dodavanje, uređivanje, brisanje) *feature flagova*.

Rad je podijeljen na četiri dijela. Prvi dio govori o korištenim tehnologijama kao što su Java, Maven, Spring, React, Docker itd. Drugi dio rada govori o tipovima i korištenju feature flagova. U trećem dijelu je predstavljena arhitektura aplikacije, od baze podataka do backenda i frontenda, te implementacija i način serviranja. U četvrtom dijelu je objašnjena sama instalacija programskog rješenja.

4. Korištene tehnologije

4.1. Java

Java je programski jezik izrađen 1995. godine od strane James Goslinga, programera američke kompanije Sun Microsystems. 2010. godine je Oracle kupio Sun Microsystems i danas održava i unapređuje Javu. Trenutno, zadnja verzija je Java 14. Danas, kao i zadnja dva desetljeća, Java je na samom vrhu popularnosti programskih jezika. Prilikom izrade Jave jedan od najvažnijih zahtjeva bio je da se može izvršavati na bilo kojem OS-u (*operacijskom sustavu*), bez obzira na kojem je kompiliran. To je omogućeno na način da se kompilirani Java *bytecode* izvršava unutar virtualne mašine koja ima zadaću da instrukcije prevodi u strojni kod, koji je specifičan od platforme do platforme.

4.1.1. Specifičnosti jezika

Java spada u skupinu programskih jezika čiji se izvorni kod cijeli kompilira prije nego li se može izvršavati. Druga skupina jezika su interpreteri koji se izvršavaju liniju po liniju. Generalno, kompilirani jezici imaju bolje performanse zbog toga što je cijeli kod spreman na izvršavanje te se ne treba trošiti vrijeme za prevođenje liniju po liniju tijekom izvršavanja.

Java je objektno orijentiran jezik što znači da su tu prisutni svi koncepti objektno orijentiranog programiranja (klase, objekti, apstrakcija, enkapsulacija, nasljeđivanje, polimorfizam...)

Tipovanje u Javi je snažno (eng. *strongly typed*) te statičko (eng. *statically*). Snažno tipovanje znači da se prilikom pisanja koda mora navesti tip varijable te na taj način osigurati da prilikom samog kompiliranja, kompajler može provjeriti je li pridružena vrijednost definiranoj varijabli traženog tipa, te ako nije, odmah izbaciti grešku. Na taj način povećavamo sigurnost koda, i ne strepimo od takve greške prilikom izvršavanja programa. Statično tipovanje znači da se varijabli kojoj je prvotno pridružena vrijednost nekog tipa, kasnije ne može pridružiti vrijednost nekog drugog tipa te također takve greške se mogu uočiti pri samom kompiliranju.

4.1.2. Java ekosustav

Kao što je na početku navedeno, za izvršavanje Java koda, potrebna je specijalna virtualna mašina koja može čitati kompilirane instrukcije. To je JVM (eng. *Java Virtual Machine*). Java kod se ne kompilira direktno u strojni kod nego u *bytecode*. Taj rezultat kompiliranja možemo promatrati kao posredni kod između Java koda i strojnog koda. Iako je Java neovisna o platformi kao programski jezik, JVM to nije. JVM je specifičan za svaku platformu te on zna čitati te prevoditi *bytecode* u strojni kod i time izvršavati Java program.

Za pokretanje Java programa, potrebno je imati instaliran JRE (eng. *Java Runtime Environment*). To je softver koji sadrži u sebi JVM i set biblioteka i dodatnih datoteka koje JVM koristi prilikom svog izvršavanja.

Da bi programer mogao pisati Java kod, mora imati instaliran JDK (eng. *Java Development Kit*). To je programersko okruženje koje u sebi sadrži JRE te ostale alate potrebne prilikom izrade Java aplikacija. Jedan od tih alata je i sam Java kompajler `javac`.

4.2. Maven

Prilikom izrade jednostavne Java aplikacije, dovoljan nam je JDK i jednostavno kompiliramo naš program sa `javac` komandom i možemo ga odmah izvršavati na JVM-u.

Međutim, kod izrade kompleksnijih aplikacija, moramo imati nekakav alat za automatizaciju *build* procesa, dohvaćanje potrebnih biblioteka i radnih okruženja (eng. *frameworks*), izvršavanje automatiziranih testova i integriranje tog procesa u CI/CD (eng. *continuous integration/continuous delivery*) sustav između ostalog.

Najpoznatiji takav alat u Java svijetu je Apache Maven. Maven projekte definira kroz XML datoteku zvanu *Project Object Model* ili pom.xml. To je XML datoteka unutar koje se pomoću definiranih XML tagova definira struktura i meta podaci projekta i modula, zavisne biblioteke i radna okruženja, dodaci (eng. *plugins*), *build* koraci i ostalo. Maven tada prilikom izvršavanja svojih *build* fazi automatski dohvaća potrebne artefakte s nekih od definiranih repozitorija, kompilira kod te ga zapakira u komprimiranu .jar datoteku koja je spremna za pokretanje i *deploy* na servere.

4.3. Spring

Spring je daleko najpopularniji Java *framework* za izradu mrežnih aplikacija. Spring je također besplatan i *open source* alat dostupan na GitHubu.

Spring *framework* kao solucija za izradu robusnih mrežnih aplikacija nudi čitavu paletu svojih modula za rješavanje različitih poslovnih zahtjeva. Na taj način omogućava programerima da se koncentriraju pri programiranju na poslovnu logiku bez puno brige o *low level* aspektima programiranja. Primjer takvih modula može biti modul za rad s bazom podataka (Spring Data), modul za rad sa AMQP (*advanced messaging queueing protocol*) (Spring AMQP), modul za autentikaciju i kontrolu pristupa (Spring Security) i ostali. Često nam u razvoju treba više tih modula da bi razvili kompletno poslovno rješenje i tu je najpoznatiji Springov modul (Spring Boot) objedinio najbitnije module pod jedan i tako ponudio robustan unaprijed određen (eng. *opinionated*) način razvijanja mrežnih aplikacija. Tako prilikom izrade aplikacije u Spring Bootu imamo ugrađen i *servlet container* za Java aplikacije te ne trebamo brinuti o postavljanju naše aplikacije na istoga već samo pokrenemo aplikaciju.

Sam Spring kao *framework* razvijen je u duhu dva uzorka programiranja (eng. *design patterns*). Jedan je *dependency injection*, a drugi aspektno orijentirano programiranje. Jedno i drugo u Springu je riješeno pomoću svojstva u Javi zvanog refleksija.

Dependency injection radi po principu da se prilikom konstruiranja objekta, unutar konstruktora kroz parametre proslijede odgovarajuće instance neke klase koja implementira *interface* koji konstruktor očekuje. U Springu *dependency injection* se događa automatski iza programerovih očiju. Naime, kada se Spring pokreće, pokrene nešto što se naziva *application context*. To možemo promatrati kao *container* ili objekt koji u sebi sadrži sve potrebne instance klasa koje smo u kodu anotirali s odgovarajućim anotacijama. Tada su te instance dostupne za umetanje (eng. *injection*) na potrebnim lokacijama u našem kodu.

Aspektno orijentirano programiranje je paradigma pomoću koje se omogućava veća modularnost u kodu kada se želi dodati nekakvo dodatno ponašanje bez da modificiramo postojeći kod. Tako na primjer možemo omogućiti izvršavanje nekog koda prije, poslije ili prije i poslije izvršavanja nekih metoda bez da kod eksplicitno obuhvatimo sa tim dodatnim kodom. Najpoznatiji primjer za to u Springu je deklarativno korištene transakcija. Tako ako

metodu anotiramo s anotacijom `@Transactional` možemo implicitno obuhvatiti SQL naredbu koja se izvršava u danoj metodi unutar transakcije da budemo sigurni da se izvršeni SQL u potpunosti i uspješno izvršio, ili ako je bila nekakva greška da ne ostavimo bazu u nečistom stanju nego da promjene koje su uspjele odbacimo jer cijela transakcija nije uspješno obavljena.

Dakle, kao što je navedeno, gornje je omogućeno upotrebom refleksije, a to je mogućnost softvera da si pregleda vlastitu strukturu tijekom izvođenja. U Javi to je omogućeno preko Java Reflection API-ja preko kojeg možemo dohvatiti imena klasa, metoda, polja, tako i postavljenih anotacija.

4.4. Hibernate

Dodatan bitan dio svake ozbiljnije aplikacije koja komunicira s bazom podataka jest ORM (eng. *object-relational mapping*) *framework*. Najpoznatiji takav u Java svijetu je Hibernate. To je također besplatan *open source* alat koji nam omogućava da možemo preslikati tablice i veze među njima iz baze u kod. To znatno olakšava i ubrzava proces razvijanja aplikacije jer se programeri mogu koncentrirati da pišu tipski siguran i fluentan programski kod bez da pišu čiste SQL naredbe u kodu kao *string* što je prilično nesigurno i rezultira jako teškim održavanjem te aplikacije u slučaju bilo kakvih promjena u bazi.

Java kao jezik definira, ali ne i implementira svoj API (eng. *application programming interface*) za rad s entitetima iz baze podataka zvan JPA (*Java Persistence API*). Ta uloga je prepuštena ORM *frameworcima* i to je upravo ono što Hibernate čini. U Springu zadana implementacija za JPA je upravo Hibernate. Hibernate je zapravo jako kompleksan kotačić, ali u suštini osnovno korištenje nije teško i funkcionira tako da se unutar koda definiraju POJO (eng. *plain old Java object*) klase poznatije kao Java *beans*. Te klase se anotiraju s anotacijama za JPA koje Hibernate skenira i na osnovu njih zna mapirati entitete u bazi na odgovarajuće klase. Pomoću tih anotacija možemo definirati mnoga svojstva i ograničenja stupaca unutar tablice. Na primjer, tip stupca, minimalna i maksimalna vrijednost koju se smije ubaciti u polje, primarni i sekundarni ključ i još mnogo toga. Dodatno, pomoću Hibernatea možemo doslovno izgraditi cijelu shemu baze podataka na osnovu tih klasa. Takvo generiranje baze na osnovu koda, naziva se *code first*.

4.5. Flyway

Kao što je rečeno, Hibernate nam omogućava definiranje cijele baze na osnovu modela klasa u kodu. Međutim, također je bitno naglasiti da unutar produkcijske okoline gdje želimo biti sigurni da je baza točno onakva kakvu želimo, korištenje automatskog generiranja baze podataka preko Hibernatea, nije sto posto sigurno. Na žalost, Hibernate ne može osigurati da će se nekakve specifične naknadne promjene sheme baze i tablica sigurno primijeniti i oslikavati 1 na 1 situaciju iz koda sa situacijom iz baze. Zato posao izmjene baze treba raditi preko migracijskih skripti. To su SQL datoteke u kojima je zapisan čisti SQL koji definira i/ili izmjenjuje strukturu sheme baze podataka. Da bi se te skripte koristile zajedno s aplikacijom, koristi se biblioteka Flyway koja omogućava izvršavanje tih skripti na siguran način prilikom pokretanja aplikacije. To se kontrolira tako da Flyway stvori u bazi tablicu u kojoj drži aplicirane migracije i njihove *checksumove*. Tako možemo biti sigurni da će se određena migracija aplicirati samo jednom, bez obzira koliko god puta pokrenuli aplikaciju.

4.6. PostgreSQL, MS SQL Server

Prilikom izrade završnog rada, zbog toga što su prisutna dva servisa koja se pokreću, namjerno su za potrebe učenja i demonstracije odstupanja u sintaksi SQL-a za različite servere baza podataka odabrane dvije baze podataka. To su PostgreSQL i MS SQL Server.

PostgreSQL, također poznat kao Postgres je besplatan *open source* RDBMS (eng. *relational database management system*) koji se fokusira na što veću usklađenost s definiranim SQL standardom, dok je MS SQL Server Microsoftova solucija za RDBMS koja se plaća u komercijalnim rješenjima dok postoji besplatna verzija za korištenje prilikom razvoja. Za razliku od Postgresa, sintaksa SQL-a za MS SQL Server je malo više rječitija te malo više odstupa od definiranog SQL standarda što će se prikazati dalje u tekstu ovog dokumenta.

4.7. RabbitMQ

U modernim arhitekturama mrežnih aplikacija, mikroservisi su odvojeni i samostalni dijelovi cjelokupnog sustava koji tek zajedno tvore kompletnu aplikacijsku logiku. U takvim arhitekturama često se izdiže pitanje koje nije problem u zastarjelom monolitnom načinu

građenja aplikacija, a to je kako da određeni događaj u jednom mikroservisu utječe da se izvrši neka logika u drugom mikroservisu. Rješavati taj problem sa standardnim HTTP upitom na određeni mikroservis koji treba dobiti obavijest o događaju je neefikasno i komplicirano, jer svaki servis za sebe bi morao voditi računa o tome koji su mikroservisi ti koji ovise o nekim događajima koji se dogode unutar tog mikroservisa. Solucija tog problema nazire se u vidu nekakvih sabirnica poruka na razini cijele mikroservisne okoline. Rješenje tog problema popularizirano je korištenjem servisa koji služe kao brokeri poruka (eng. *message brokers*).

Jedno od popularnijih rješenja u industriji je RabbitMQ. RabbitMQ je *messaging broker* koji implementira AMQP koji možemo lako koristiti s modulom Spring AMQP. U suštini to je servis koji prima i prosljeđuje poruke. Koncepti povezani uz RabbitMQ su *producer, consumer, exchange* i *queue*.

Producer je onaj servis koji šalje poruke u RabbitMQ, *consumer* je onaj servis koji konzumira poruke preko RabbitMQ-a. *Queue* je red poruka koji se treba isporučiti onim *consumerima* koji su pretplaćeni na određeni *queue*, a *exchange* je mjesto na koje *produceri* šalju poruke. Zadaća *exchangea* je rutiranje poruke na određeni *queue*.

S ovakvim solucijom možemo jednostavno propagirati informacije mikroservisima koji ih trebaju. Raspodijeliti nekakav teži posao između više instanci servisa kako bi paralelizirali egzekuciju, ili poslati svima odjednom istu poruku.

4.8. Javascript, Typescript

Kako *feature flags* servis zahtjeva određeno web sučelje za administraciju *feature flagova*, izrada frontenda nemoguća je bez upotrebe Javascripta. To je programski jezik kojeg su internet preglednici sposobni izvršavati i time omogućiti dinamička web sučelja.

4.8.1. Javascript

Javascript je skriptni programski jezik. Suprotno uvriježenom mišljenju Javascript nije u potpunosti interpreterski jezik, već uz pomoć određenih optimizacija prilikom skeniranja koda, neki dijelovi Javascripta mogu se kompilirati i time ostvariti povećanje performansi. Za razliku od Jave, Javascript je slabo tipiziran (eng. *weekly typed*) ali i statički tipiziran. To bi značilo da se za definiranu varijablu u kodu ne treba unaprijed znati njen tip, ali kad se

obavi prvotno pridruživanje, tada više nije moguće pridružiti neki drugi tip vrijednosti toj istoj varijabli. Zbog toga što je Javascript praktički jedina solucija za izvršavanje koda u internetskom pregledniku, time je i jako popularna. Prije se Javascript isključivo izvršavao u internetskom pregledniku, ali zbog popularnosti pogodno je bilo omogućiti i njegovo izvršavanje u drugim okolinama. Stoga je u posljednje vrijeme jako popularan Node.js koji omogućuje njegovo izvršavanje izvan internetskog preglednika.

4.8.2. Typescript

Zbog navedenog slabog tipovanja koje može dovesti do neželjenih grešaka prilikom izvršavanja, u industriji su se pojavila razna rješenja kako bi doskočila tom problemu. Najpopularnije je sigurno Typescript. To je Microsoftova solucija pomoću koje se Javascriptu omogućilo snažno tipovanje. Typescript je zapravo super set Javascripta, to nije direktno programski jezik. On se zapravo prilikom kompiliranja prevodi u Javascript. Međutim, zbog toga što se u tom procesu kompiliranja mogu uočiti greške u tipovanju, time podižemo razinu sigurnosti naše aplikacije i tada znamo da kompilirani Typescript rezultira u sigurno tipovanom Javascriptu.

4.9. React

S obzirom na to da uz svaki popularniji programski jezik imamo obično i čitavu lepezu *frameworkova* koji nam čine razvijanje lakšim i bržim, tako imamo i za Javascript. Jedan od njih je React.

React je *open source* biblioteka za lakšu izgradnju web sučelja razvijena od strane Facebooka. React je deklarativan što znači da omogućuje da se programeri ne moraju brinuti o eksplicitnom *renderingu* već se mogu fokusirati na definiranje malih i jednostavnih dijelova prikaza koji se trebaju prikazati na web sučelju, a React će sam efikasno promijeniti prikaz kada je to potrebno.

S Reactom se u pravilu pišu React komponente koje su male, logičke cjeline koje prikazuju određenu komponentu. React je to olakšao tako da je omogućio pisanje JSX-a unutar Javascript koda. To je sintaksa jako slična HTML-u unutar koje možemo definirati HTML elemente i React elemente i time stanje aplikacije zadržati kompletno u kodu, bez da ga dinamički ažuriramo u HTML-u.

Najvažniji koncepti u Reactu su *state*, *props* i životni ciklus React komponente. *State* predstavlja stanje neke komponente. Svaki put kad se promijeni *state*, React automatski ponovno renderira tu komponentu tako da uvijek reflektira najsvježije stanje. *Props* su podaci koje jedna React komponenta može proslijediti drugoj React komponenti koja je ugniježđena unutar nje. Životni ciklus komponente su metode koje nasljeđujemo od React.Component klase i preko njih možemo logički kontrolirati ažuriranje stanja i time osvježavanje prikaza sa željenim podacima.

4.10. Docker

Dugo su se aplikacije postavljale na servere tako da bi server morao imati sve potrebne alate i programe da bi mogao servirati aplikaciju. To bi obično rezultiralo neredom u programskom okolišu servera i znalo je raditi probleme s različitim verzijama potrebnog softvera za pokretanje aplikacija i još dosta sličnih problema. Tomu se doskočilo s uvođenjem virtualnih mašina tako da bi svaka postavljena aplikacija bila postavljena na svoju virtualnu mašinu na tom serveru i time bi imala uredan programski okoliš za sebe. Međutim, virtualne mašine same po sebi su „težak" softver koji mora podignuti čitavi operacijski sustav dodatno na domaćinskom OS-u. Također, nezgodno je bilo skaliranje takvih aplikacija u slučaju povećanog opterećenja.

Tehnologija koja je riješila sve navedene probleme je Docker i to danas postaje *de facto* standard kojim se *deployaju* aplikacije na servere. Docker tehnologija može se činiti na prvi pogled kao virtualna mašina, ali nije baš tako. Naime, za razliku od virtualnih mašina Docker *containeri* su puno „lakši" jer koriste jezgru (eng. *kernel*) operacijskog sustava domaćina, a i dalje pružaju potpunu izolaciju što se tiče programskog okruženja za svaku od postavljenih aplikacija na server.

5. Funkcionalnosti sustava

Tema ovog rada bila je izraditi sustav pomoću kojega će ostali servisi unutar mikroservisne okoline moći dinamički mijenjati putanju izvršavanja koda na osnovu tko je klijent koji mu pristupa. Takav sustav je opće poznat u IT industriji pod nazivnom *feature flags* ili *feature toggles* ili *feature control* i priznat je kao dobra programerska praksa sa mnogo prednosti, te ga velike kompanije kao npr. Google i Facebook koriste u svom razvojnom procesu. Google je konkretno još 2011. godine koristio feature flagove prilikom izrade novog izgleda Gmaila.

5.1. Tipovi *feature flagova*

Razlikujemo nekoliko tipova *feature flagova*, a razlika je ostvarena po načinu korištenja.

5.1.1. *Feature flagovi* za službenu isporuku

Tipičan način razvijanja programa odvija se koristeći neki od sustava verzioniranja, od kojih je najpopularniji Git. Obično ako radimo na nekom većem projektu i jedan tim dobije zadatak razviti neku potpuno novu funkcionalnost za proizvod, za koju je procijenjeno da će trebati duži period da se u potpunosti implementira, kreira se nova grana koja izlazi iz trenutne glavne (eng. *master*) grane i u njoj se razvija tražena funkcionalnost.

Kako prolazi vrijeme, dolazi do sve većeg razilaženja u kodu *master* grane i *feature* grane. Na produkciji se uvijek postavlja verzija aplikacije koja je izrađena na osnovu situacije u *master* grani i time nemamo sigurnost da, kada sjedinimo *feature* granu i *master* granu, će naša aplikacija i dalje stabilno funkcionirati u produkcijskoj okolini.

Tu u pomoć doskaču *feature flagovi* kojima omogućujemo da često sjedinjujemo promjene iz *feature* grane u *master* granu, te korištenjem administracijskog sučelja omogućimo testnom timu da što prije testira funkcionalnost koja se razvija. Na taj način povećavamo sigurnost da dolaskom službenog datuma isporuke sustav i dalje bude stabilan kada se omogući svim korisnicima korištenje novo razvijene funkcionalnosti.

5.1.2. Operacijski *feature flagovi*

Recimo da na sustavu koji razvijamo već postoji određena funkcionalnost koja funkcionira, ali možda smo zamijetili da nam s povećanim brojem klijenata koji ju koriste, upravo ta komponenta opterećuje sustav i usporava ga.

U takvim situacijama, često dolazi do odluke da se ponovo promotri implementacija te funkcionalnosti te vidi postoji li mjesta za optimizaciju. Ukoliko se krene u *refactoring*, moramo se zaštiti da ne pokvarimo trenutnu implementaciju te da novoizgrađena implementacija logički i dalje zadovoljava prethodnu poslovnu logiku.

Upravo s *feature flagom* se štitimo i kada se uvjerimo da je nova implementacije uistinu optimalna tj. bolja od prethodne te da je poslovna logika i dalje sačuvana, tada možemo maknuti *feature flag* i izbrisati stari dio koda. Ovakvi tipovi *feature flagova* su obično kraćeg vijeka trajanja za razliku od prethodno opisane vrste.

5.1.3. Eksperimentalni *feature flagovi*

U IT industriji često se osluškuje reakcija klijenata na određenu promjenu u sustavu. Tako prilikom razvijanja nove ili mijenjanja postojeće solucije, ukoliko s produktne strane nije 100% sigurno kako se želi implementirati nešto, možemo koristiti *feature flag* da napravimo dvije skupine korisnika te jednoj skupini ponudimo jednu verziju rješenja, a drugoj drugu verziju rješenja. Tako možemo vidjeti reakcije korisnika te naposljetku odabrati onu s kojom je više korisnika bilo zadovoljno. To se naziva *A/B testing.*

Ovdje je opet očito da nam takvo što omogućuju upravo *feature flagovi* na osnovu kojih jednoj skupini prikazujemo jednu verziju, a drugoj drugu.

5.1.4. *Feature flagovi* za kontrolu pristupa

Za ovu vrstu *feature flagova* je sporno je li pravo rješenje za dani problem. Naime, jasno je da s *feature flagovima* možemo opet napraviti distinkciju korisnika po raznim kriterijima, pa tako moguće da bi mogli razlikovati *premium* korisnike od osnovnih korisnika. Prema mome mišljenju, ovakve bi probleme trebalo rješavati na način da se ovakav podatak doslovno zapisuje u konfiguraciju korisničkih postavki i čita iz njih kako bi kontrolirali dostupnost funkcionalnosti određenim korisnicima.

5.2. Korištenje *feature flagova*

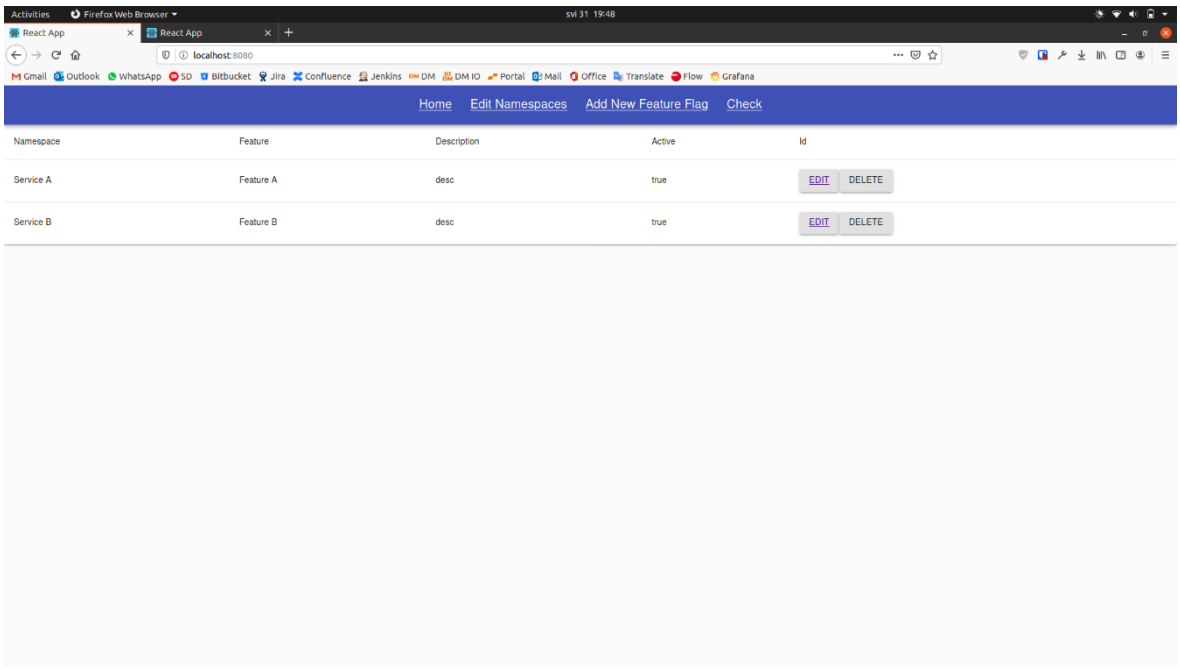
Funkcionalnost *feature flagova* omogućena je kroz dvije komponente:

- Frontend administracijsko sučelje
- Backend API kojeg koriste drugi servisi

5.2.1. Administracijsko sučelje

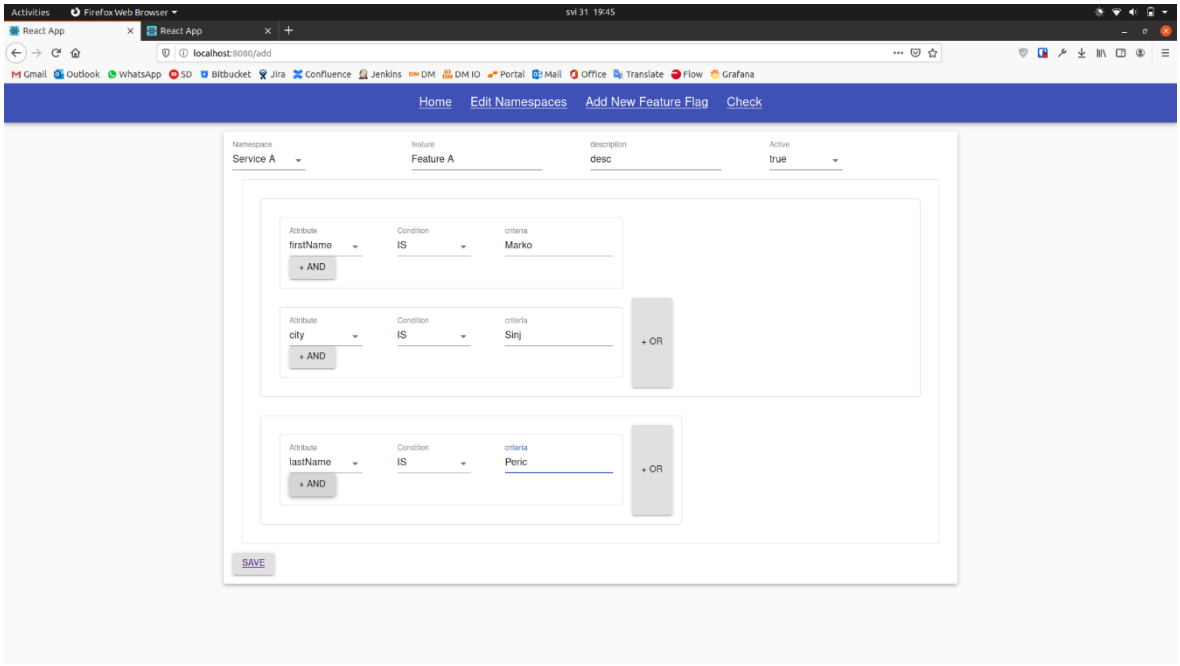
Sami odabir korisnika prema različitim karakteristikama omogućen je korištenjem frontend administracijskog sučelja. Preko njega je omogućeno slaganje kompleksnih logičkih uvjeta kako bi se dobila lista korisničkih identifikatora koji su zadovoljili traženi kriterij pretrage.

Frontend je razvijen tako da na glavnoj stranici se nalazi popis svih kreiranih *feature flagova*. Tu je za brzi pregled prikazano za koji servis je kreiran određeni *feature flag*, naziv samog *feature flaga*, njegov kratki opis, te *true* ili *false* svojstvo koje kazuje je li taj *feature flag* trenutno aktiviran ili ne. Tu se dodatno još nalaze dugmad preko koje možemo izbrisati ili uređivati postojeći *feature flag*.



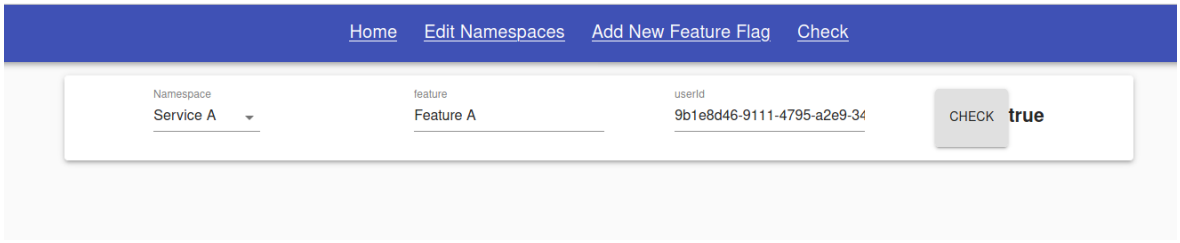
Slika 1 Snimka zaslona koji prikazuje popis postojećih feature flagova

Tu je također stranica preko koje se može kreirati novi *feature flag* sa svim kriterijima pretrage tako da se mogu dinamički, lagano i vizualno složiti logički uvjeti koji se moraju zadovoljiti da se propusti određenog korisnika.



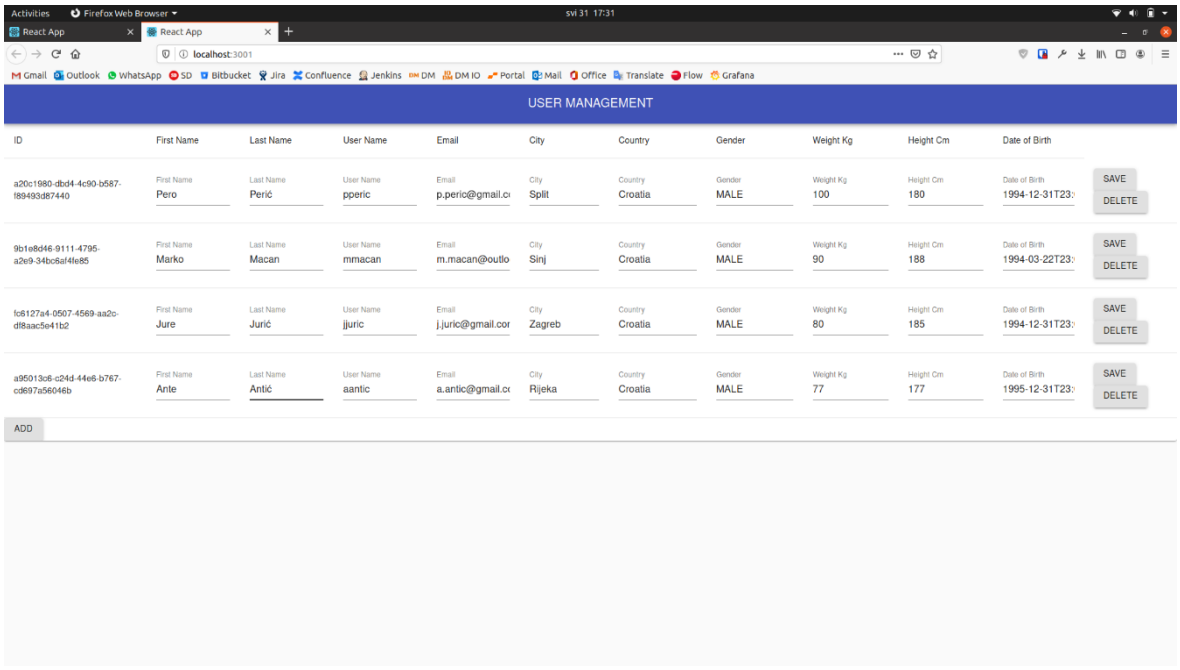
Slika 2 Snimka zaslona koji prikazuje sučelje preko kojeg se kreira novi feature flag

Kako bi za stvarnu demonstraciju bilo potrebno podignuti još nekoliko servisa sa pripadajućim backendom i frontendom, što je izvan okvira ovog rada, kreirana je stranica preko koje se može vizualno testirati kreirani *feature flag*. Tako na toj stranici jednostavno možemo upisati naziv servisa (*namespace*), *feature* i korisnički identifikator te klikom na dugme provjeriti da je li tom korisniku dostupan određeni *feature* ili ne.



Slika 3 Snimka dijela zaslona na kojem je prikazano testiranje nekog feature flaga za dani korisnički identifikator

Kako je dalje opisano u ovom radu, također je implementiran dodatni servis s kojim *feature flag* komunicira kako bi dohvatio listu zadovoljavajućih korisničkih identifikatora, te je također razvijeno jednostavno web sučelje preko kojeg se mogu dodavati, brisati i uređivati korisnici, te uvidjeti kako promjene na tom servisu utječu automatski na ažuriranje liste korisničkih identifikatora unutar *feature flags* baze podataka.



Slika 4 Snimka zaslona koji prikazuje web sučelje preko kojeg se administriraju korisnici

5.2.2. Korištenje u kodu

Feature flagovi su prvenstveno razvijeni kako bi se olakšao svakodnevni programerski rad, tako su i njegovi primarni korisnici programeri. A programeri naravno *feature flagove* koriste u kodu koji pišu. To su prvenstveno `if` uvjeti koji jednostavno pozivaju funkciju u kojoj je implementiran upit na *feature flag* servis. Ta funkcija uvijek vraća `true` ili `false` vrijednost, a kao argumenti joj se prosljeđuju *namespace, feature* i korisnički identifikator.

```
if (featureFlagsClient.isFeatureEnabled("namespaceA","featureB", "userId")) {
...some code
}
else {
...some other code
}
```

Primjer koda 1 Primjer korištenja feature flagova u kodu

U danom primjeru klijentski servis bi napravio upit na API izložen od strane *feature flags* servisa te na osnovu odgovora izvršio ili dio koda unutar `if` bloka, ili dio koda unutar `else` bloka.

Ovim je prikazano bazično korištenje *feature flagova*. Kao i u svemu uvijek ima prostora za napredak, pa tako i ovdje. U slučaju da se procjeni da je klijentskom servisu preskupo raditi HTTP upit na svaku provjeru *feature flaga*, klijentski servis bi mogao implementirati *caching* na svojoj strani i time čuvati listu korisničkih identifikatora i *featurea* za svoj servis. Također, tu bi se trebalo obratiti pozornost na stanje u bazi servisa koji čuva podatke o korisnicima. Tako bi se taj klijentski servis trebao također pretplatiti na te promjene preko RabbitMQ-a kao i *feature flags* servis kako bi se osiguralo da su *cacheirani* podaci ispravni.

6. Implementacija

Sljedećih nekoliko poglavlja objasnit će kako je ovaj rad praktično implementiran tako što ćemo proći kroz arhitekturu aplikacije i kako bi se ona u praksi mogla uglaviti unutar mikroservisne okoline. Vidjet ćemo shemu baze podataka i kako je ona povezana s backendom samog servisa, te frontend, njegovu implementaciju i način serviranja.

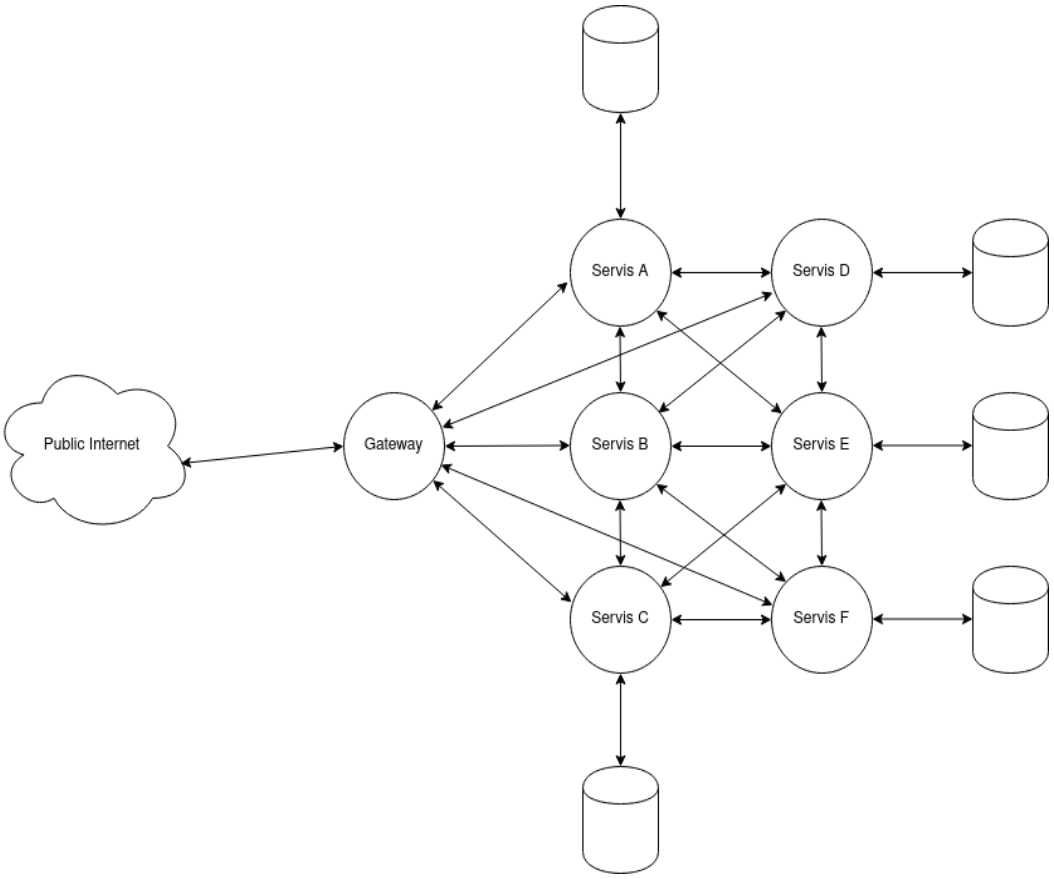
6.1. Arhitektura

Dugo vremena glavni način za razvijanje backend servisa bio je monolitni pristup, što bi u prijevodu trebalo značiti da je sva poslovna logika koju neki proizvod pruža bila zapakirana unutar jednog, velikog servisa. Takav servis bi bio „zakačen" na jednu veliku bazu podataka s velikim brojem tablica i veza između njih. Kod manjih aplikacija i dan danas možemo reći da je ovo preferirani način razvijanja, međutim, ako imate nekakav veći proizvod ili pak više proizvoda koje servirate svojim klijentima, te usput radite u većoj kompaniji s mnogo timova, takav način razvijanja aplikacija brzo će vas sustići. Održavanje koda postat će teško, kolaboracija među timovima također. Usporit će se isporuka novih funkcionalnosti zbog toga što i najmanja promjena u kodu znači da se treba nanovo izvršiti postavljanje čitave aplikacije na server sa tom promjenom. Timovi neće biti produktivni jer praktički moraju poznavati čitav sustav da bi mogli kontribuirati u njega. Riskirate da jedna greška u aplikaciji uzrokuje nedostupnost čitave aplikacije i još mnogo toga.

Tome se doskočilo tako da su se aplikacije počele razvijati tako da aplikacijska logika bude razdvojena na više manjih i logički kompaktnih servisa koji su morali „raditi jednu stvar, i raditi je dobro". Takvi servisi moraju međusobno komunicirati, i tek svi zajedno tvore kompletnu poslovnu logiku. Na taj način osim samog razvijanja logike aplikacije, stvorili su se uvjeti da se određeni timovi bave ne nužno poslovnom logikom nego nekim popratnim servisima kako bi pružili infrastrukturu drugim servisima u vidu sigurnosti, upravljanja korisničkim računima, pravima pristupa, izlaganjem krajnjih dijelova sustava javnosti, upravljanjem i raspoređivanjem tereta između servisa, itd. Takva arhitektura aplikacije se naziva mikroservisnom, gdje se gore navedeni servisi smatraju „mikroservisima". Jedan takav je i *feature flags* servis koji je razvijen u praktične svrhe ovog rada.

6.1.1. Putanja javnog HTTP upita kroz mikroservisni sustav

Tipično kod mikroservisnih arhitektura je da postoji neki servis iznad svih ostalih preko kojeg se izlažu javnosti određeni dijelovi mikroservisnog sustava i preko kojeg se može javnom internetu omogućiti pristup aplikaciji. Takvi servisi nazivaju se *gateway* servisi. Oni primaju sav javni promet te ga rutiraju i balansiraju prema odgovarajućem servisu. Obično ti servisi primljeni upit provjeravaju tako da kontaktiraju nekakav sigurnosni servis unutar mikroservisne okoline koji vrši validaciju dobivenog upita te ga odbacuje ako nije verificiran sa odgovarajućom vrstom autentikacije. Tek kada je validacija upita zadovoljena, *gateway* će proslijediti upit traženom mikroservisu na obradu te će rezultat obrade vratiti nazad klijentu.



Slika 5 Pojednostavljena shema mikroservisne arhitekture

6.1.2. Arhitektura *feature flags* sustava

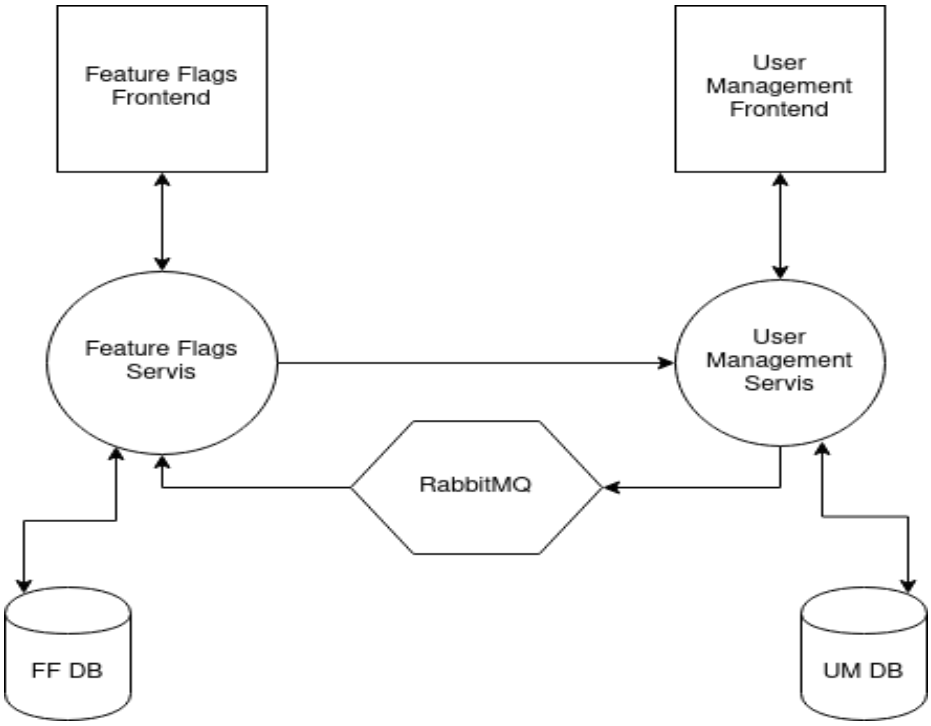
Servis razvijen u sklopu ovog rada zamišljen je da bude dio neke veće mikroservisne okoline. Kao takav fokusiran je na obavljanje logike preko koje možemo na jednostavan način probrati skupinu korisnika po raznim kriterijima pretrage te ostalim servisima omogućiti na jednostavan način provjeru može li neki korisnik pristupiti određenom svojstvu tog servisa.

Feature flags servis sam po sebi ne bi služio svrsi ako nije uparen s nekim servisom koji mu može ponuditi mogućnost pretrage korisnika po kriterijima zadanim unutar pravila definiranog za određeni *feature*. Za tu potrebu uz servis *feature flags* razvijen je i jednostavan pomoćni servis koji drži podatke o korisnicima – *user management*. On kao takav, pomoćni servis u ovom slučaju, nema nikakvu kompleksnu logiku osim jednog dijela koji treba pružiti, a to je mogućnost pretrage korisnika za dane kriterije od strane *feature flags* servisa. Kako je to implementirano, vidjet ćemo kasnije.

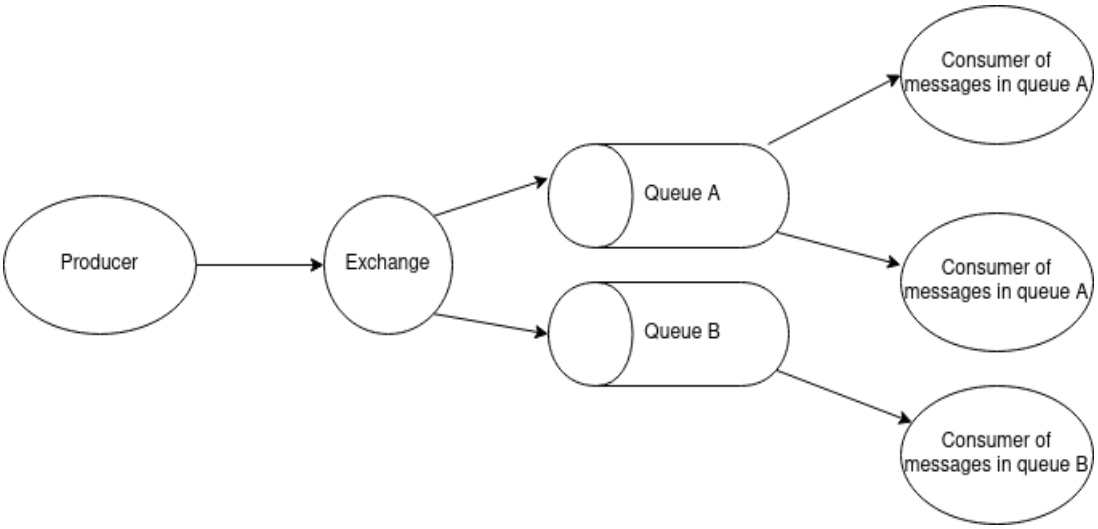
Dakle, da bi sve skladno funkcioniralo, tj. da *feature flags* servis može dobiti listu korisničkih identifikatora koji zadovoljavaju pravilo pristupa definirano kroz web sučelje, mora se izvesti komunikacija *feature flags* servisa s *user management* servisom, što samo po sebi nije problem i predstavlja jednostavan HTTP upit, stvari se kompliciraju ako dođe do promjena unutar baze od *user management* servisa i ispostavi se da neki korisnik ima promijenjena svojstva koja više ne zadovoljavaju kriterije zadane od strane *feature flags* servisa, ili je neki korisnik dodan ili izbrisan. Te je slučajeve također trebalo osigurati da bi bili sigurni da imamo uvijek točne korisničke identifikatore na strani *feature flags* sustava.

Taj problem se riješio tako da smo uveli RabbitMQ kao sabirnicu poruka između ta dva servisa. U suštini što se događa je sljedeće: Svaki put kada se promjeni stanje u bazi na strani *user management* servisa, okida se poruka koja se šalje u RabbitMQ sabirnicu na definirani *exchange*, a s druge strane, *feature flags* je na RabbitMQ prijavljen da osluškuje dolazne poruke u tom *exchangeu*. Tako kada *feature flags* primi poruku o promjeni u bazi *user management* servisa, zna da treba ponovo izvršiti upit na *user management* servis kako bi osigurao da ima svježe podatke i na svojoj strani.

Što se tiče frontenda i jednog i drugog servisa, on servira sa svakog servisa svoj frontend.



Slika 6 Arhitekture razvijene solucije



Slika 7 Shema RabbitMQ sabirnice poruka

6.2. Baza podataka

Osnova oba razvijena servisa je relacijska baza podataka. Preko nje smo omogućili da imamo permanentno mjesto na koje možemo zapisati stanje aplikacije. Uz to, kvalitetno razvijenom shemom tablica i relacija među njima osigurali smo i da to stanje koje mijenjamo zadržava svoj integritet, odnosno ne dopuštamo svakakve modifikacije u smislu nedozvoljenih upisivanja, uređivanja i brisanja podataka ako ne zadovoljavaju sva pravila koja smo postavili.

Prva stvar koju moramo osigurati je da kreiramo sheme u bazi u kojoj će se nalaziti tablice oba servisa. Svaki servis ima svoju shemu, i on je sam kreira u definiranoj bazi na koju je spojen, ukoliko ta shema već ne postoji.

Ovdje možemo vidjeti odstupanja u SQL sintaksi za dva različita korištena RDBMS (eng. *Relational database management system*) – MS SQL Server i PostgreSQL.

```
IF NOT EXISTS(SELECT 1
              FROM sys.schemas
              WHERE name = 'feature_flags')
    EXEC ('CREATE SCHEMA [feature_flags]');
```

Primjer koda 2 SQL naredba za generiranje sheme za MS SQL Server

```
CREATE SCHEMA IF NOT EXISTS user_management;
```

Primjer koda 3 SQL naredba za generiranje sheme za PostgreSQL

Nakon što je shema napravljena, moraju se kreirati tablice unutar nje. Prilikom kreiranja tablica mora se posvetiti pažnja da se zaštitimo od unosa neispravnih podataka. To se postiže postavljanjem različitih ograničenja na tablicu.

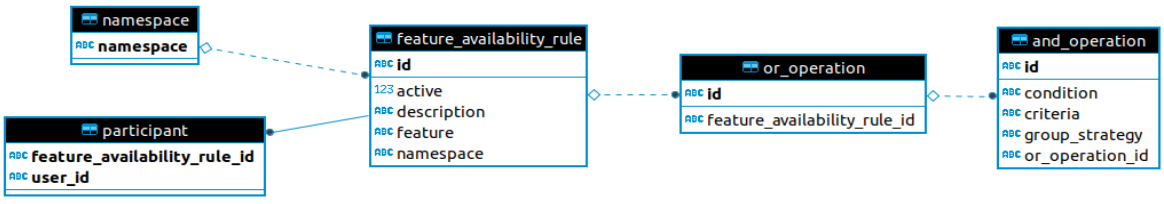
Glavno ograničenje u tablici koje se treba definirati je primarni ključ. To može biti samo jedno polje koje ne mora nositi nikakav značaj osim toga da omogućuje da se pripadni redak u tablici može jednoznačno odrediti, ili ipak može nositi određeni značaj. Također, nismo ograničeni da to bude samo jedno polje tako da možemo osigurati da neki jedinstveni stupac ili jedinstvena kombinacija stupaca budu primarni ključ u toj tablici.

Osim primarnog ključa možemo postaviti još neka ograničenja, neka standardna, a neka naša definirana. Svakako moramo postaviti tip svakog od stupaca u tablici. Tu možemo definirati maksimalnu i minimalnu vrijednost koja se može unijeti u polje, minimalan i maksimalan broj slova koje može sadržavati polje, može li to polje biti ostavljeno prazno ili ne, je li to polje strani ključ iz neke druge tablice ili pak nešto složenije, recimo ne dopustiti unos datuma novijeg od trenutnog za polje „datum rođenja”, ili neko polje može imati samo dvije moguće vrijednosti: „muško” ili „žensko” za polje „spol”.

```
create table user_management.user
(
    id            char(36)     not null,
    city          varchar(255) not null,
    country       varchar(255) not null,
    date_of_birth date         not null check (date_of_birth < now()),
    email         varchar(255) not null,
    first_name    varchar(255) not null,
    gender        varchar(255) not null check (gender = 'Male' OR gender =
'Female'),
    height_cm     int2         not null check (height_cm >= 1 AND height_cm <=
300),
    last_name     varchar(255) not null,
    user_name     varchar(255) not null unique,
    weight_kg     int2         not null check (weight_kg >= 1 AND weight_kg <=
500),
    primary key (id)
)
```

Primjer koda 4 SQL naredba koja generira tablicu s ograničenjima

Gore u primjeru je prikazana tablica kreirana za *user management* servis. Za *feature flags* servis je razvijena dosta kompliciranija shema baze koju ću ovdje prikazati.



Slika 8 Shema baze za feature flags servis

Kao što vidimo definirano je nekoliko tablica, a ono što se željelo postići je da možemo spremiti kriterije po kojima je definiran *feature flag* na način da složimo pravilo korištenjem *i/ili* logičkih operatora. Ono što se može naslutiti, cilj je da se spremljeno pravilo može pročitati i translatirati u validan dinamički SQL upit s `where` uvjetima koji će se izvršavati na *user management* servisu kako bi se dobili identifikatori korisnika koji zadovoljavaju kriterij pretrage. Za spremanje pravila koriste se tri tablice sa slike: `feature_availability_rule`, `or_operation`, te `and_operation`.

Dodatno, dobivene identifikatore moramo spremiti u bazu *feature flags* servisa kako bi bili dostupni prilikom provjere nekog drugog servisa koji koristi *feature flagove*. Ti identifikatori moraju biti povezani s određenim pravilom. Tako njih spremamo u tablicu participant koja je povezana s tablicom `feature_availability_rule`.

Također prilikom kreacije *feature flaga* želimo odabrati mikroservis na kojem će određeni *feature flag* biti u funkciji. Za to nam služi tablica *namespace* koja jednostavno služi za pohranu imena mikroservisa za koje je neki *feature flag* definiran.

6.3. Backend

6.3.1. Postavljanje projekta

Kada je definirana shema baze podataka, možemo krenuti s konkretnom implementacijom servisa. Kako smo spomenuli, rad je razvijen u radnom okviru Spring pa se za početak mora podići standardni kostur Spring Boot projekta za izradu web aplikacija. To se može najlakše izvesti tako da odemo na službenu stranicu Spring-a, te skinemo standardni kostur projekta i otvorimo ga u IDE-u u kojem programiramo. (eng. *integrated development environment*).

Nakon toga dodane su potrebne biblioteke u pom.xml datoteku, definirala su se specifična svojstva projekta u `application.yml` datoteci. Tipa URL, korisničko ime i lozinka za spajanje na bazu i slično.

6.3.2. Entiteti i modeli

Nakon što smo postavili projekt možemo krenuti s programiranjem. Prvo što bi trebali isprogramirati su klase koje predstavljaju entitete u bazi podataka. S obzirom na to da u nekim slučajevima ne želimo na *endpoint* vratiti sva polja koja su prisutna u klasi entiteta, bilo iz sigurnosnih razloga ili nečeg trećeg, praksa je definirati modele – klase koje blisko odgovaraju entitetima, ali mogu recimo izbaciti neka polja koja se nalaze u entitetu, ili recimo napraviti model koji odgovara rezultatu jednog retka SQL upita u kojem je izvršeno spajanje tablica. Da bi lakše koristili modele, trebamo i osigurati metode – *mappere* koji jedan tip objekta mapiraju u drugi.

```
@Data
@EqualsAndHashCode(callSuper = true)
@Builder
@AllArgsConstructor
@NoArgsConstructor
@Entity
@Table(name = "feature_availability_rule",
        indexes = {@Index(columnList = "namespace")},
        uniqueConstraints = {@UniqueConstraint(columnNames = {"namespace",
"feature"})})
public class FeatureAvailabilityRule extends AbstractEntity {

    @JoinColumn(name = "namespace", nullable = false)
    @ManyToOne(fetch = FetchType.LAZY)
    private Namespace namespace;

    @Column(nullable = false)
    private String feature;

    private String description;

    @Column(nullable = false)
    private Boolean active;

    @ToString.Exclude
    @OneToMany(mappedBy = "availabilityRule", cascade = CascadeType.ALL)
    private List<OrOperation> orOperations;

    @ToString.Exclude
    @OneToMany(mappedBy = "featureAvailabilityRule", cascade = CascadeType.ALL)
    private List<Participant> participants;
}
```

Primjer koda 5 Primjer klase koja definira entitet iz baze podataka

6.3.3. Slojevi aplikacije

Kada imamo definirane modele i entitete s kojima želimo raditi, spremni smo kodirati logiku. Standardno aplikacije imaju slojeve u koje se stavljaju određeni dijelovi koda radi bolje organizacije. Tako imamo sloj koji radi s bazom, sloj u kojem je poslovna logika i sloj u kojem se nalaze konkretne rute koje servis izlaže prema van.

Sloj za rad s bazom

Sloj za rad s bazom ili *repository layer* definira metode koje izvršavaju SQL upite na bazu. U Springu kreiranje takvih metoda je olakšano do krajnjih granica jer doslovno Spring može na osnovu naziva metode u *interfaceu* shvatiti kakvu metodu i s kakvim SQL upitom mora definirati prilikom kompiliranja.

```
@Transactional(readOnly = true)
public interface FeatureAvailabilityRuleRepository extends
JpaRepository<FeatureAvailabilityRule, String> {

    boolean existsByNamespace_NamespaceAndFeature(String namespace, String
feature);

    Optional<FeatureAvailabilityRule>
findByNamespace_NamespaceAndFeatureAndActiveIsTrue(String namespace, String
feature);

}
```

Primjer koda 6 Primjer interfacea na osnovu kojega Spring generira željenu metodu

Servisni sloj

U servisnom sloju kodiramo logiku servisa. Klase servis metoda unutar Spring radnog okvira moraju biti anotirane sa `@Service` anotacijom kako bi ih Spring registrirao prilikom podizanja aplikacijskog konteksta na samom startu aplikacije. Također time nam je omogućen automatski *dependency injection* kroz konstruktor *repository*, *service* ili drugih Spring *beanova* kako bi mogli složiti kompleksnu aplikacijsku logiku.

```
@Service
@Validated
@AllArgsConstructor
@Transactional(readOnly = true)
public class FeatureFlagService {

    private final FeatureAvailabilityRuleRepository ruleRepository;

    public List<String> getAllForFeatureAvailabilityRule(@NotBlank String
namespace, @NotBlank String feature) {
        FeatureAvailabilityRule rule =
ruleRepository.findByNamespace_NamespaceAndFeatureAndActiveIsTrue(namespace,
feature)
                .orElseThrow(EntityNotFoundException::new);

        return rule.getParticipants()
                .stream()
                .map(participant -> participant.getParticipantId().getUserId())
                .collect(Collectors.toList());
    }

    public boolean isUserAllowed(@NotBlank String namespace, @NotBlank String
feature, @NotBlank String userId) {
        Optional<FeatureAvailabilityRule> rule =
ruleRepository.findByNamespace_NamespaceAndFeatureAndActiveIsTrue(namespace,
feature);

        return rule.map(featureAvailabilityRule ->
featureAvailabilityRule.getParticipants()
                .stream()
                .map(participant -> participant.getParticipantId().getUserId())
                .anyMatch(userId::equals)).orElse(false);
    }
}
```

Primjer koda 7 Primjer servisne klase unutar Spring radnog okvira

Sloj za izlaganje ruta

Na najvišem vrhu imamo sloj za izlaganje ruta ili *controller layer*. Te klase trebaju biti anotirane sa `@Controller` anotacijom te bi u principu trebale biti jednostavne i samo pozivati metode iz odgovarajućih servisa na odgovarajućoj ruti.

```
@RestController
@RequestMapping("/api/feature-availability-rule")
@AllArgsConstructor
public class FeatureAvailabilityRuleController {

    private final FeatureAvailabilityRuleService service;

    @GetMapping("/{id}")
    public FeatureAvailabilityRuleModel get(@PathVariable String id) {
        return service.get(id);
    }

    @GetMapping
    public List<FeatureAvailabilityRuleModel> getAll() {
        return service.getAll();
    }

    @GetMapping("/attributes")
    public List<String> getAttributes() throws IOException {
        return service.getAttributes();
    }

    @PostMapping
    public void save(@RequestBody FeatureAvailabilityRuleModel model) throws
IOException {
        service.save(model);
    }

    @PatchMapping
    public void update(@RequestBody FeatureAvailabilityRuleModel model) throws
IOException {
        service.update(model);
    }

    @DeleteMapping("/{id}")
    public void delete(@PathVariable String id) {
        service.delete(id);
    }
}
```

Primjer koda 8 Primjer klase za izlaganje ruta

6.4. Frontend

Frontend je razvijen pomoću React biblioteke i Typescripta. Kako frontend komunicira s backendom korištenjem istih modela definiranih na backendu, kako bi se ubrzao razvoj, postoji koristan dodatak za Maven pomoću kojega se mogu generirati odgovarajući Typescript tipovi onima koji su korišteni na backendu. Tako da taj je dio generiran automatski.

Tipično u React + Typescript aplikaciji imamo dio s tipovima, dio s metodama u kojima su pripremljeni upiti na backend te dio sa samim React komponentama koje definiraju izgled stranice.

```
export class FeatureFlagsTable extends React.Component<{}, State> {

    async componentDidMount() {
        const rules = await getAllFeatureAvailabilityRules();
        this.setState({rules});
    }

    render() {
        return (
            <TableContainer component={Paper}>
                <Table aria-label="simple table">
                    <TableHead>
                        <TableRow>
                            <TableCell>Namespace</TableCell>
                            <TableCell>Feature</TableCell>
                            <TableCell>Description</TableCell>
                            <TableCell>Active</TableCell>
                            <TableCell>Id</TableCell>
                        </TableRow>
                    </TableHead>
                    <TableBody>
                        {this.mapRows()}
                    </TableBody>
                </Table>
            </TableContainer>
        );
    }

    private mapRows = () => {
        if (this.state) {
            return this.state.rules
                .map(rule => <RuleRow key={rule.id}
updateStateOnDelete={this.updateStateOnDelete} {...rule}/>);
        }
        return null;
    };

    private updateStateOnDelete = async () => {
        const rules = await getAllFeatureAvailabilityRules();
        this.setState({rules});
    }
}
```

Primjer koda 9 Primjer React komponente napisane u Typescriptu

7. Instalacija

Kako bi se instalirala programska solucija razvijana za potrebe ovog završnog rada može se ići u nekoliko različitih smjerova.

7.1. Serverska instalacija

Svakako prva je nabavka servera ili virtualne mašine koja po mogućnosti koristi operacijski sustav Linux jer je Linux uvjerljivo najrasprostranjeniji OS za servere, besplatan je i najbolja je solucija ako se koristi Docker kao solucija za postavljanje aplikacija na server.

U nekakvom profesionalnom produkcijskom okruženju imali bi *source kod* koji je spremljen na nekom Git repozitoriju. Imali bi također CI alat koji je povezan s Git repozitorijem i automatski na svaki novi *commit* u repozitorij zavrti *build* proces. Također, taj alat bi trebao biti povezan na nekakav *artifactory* (repozitorij s unaprijed kompiliranim artefaktima) i nakon uspješno odrađenog *builda* kompilirani artefakt bi se trebao podići na *artifactory*. Primjer takvog Git repozitorija može biti GitHub, BitBucket, GitLab, itd., primjer CI alata može biti Jenkins, a *artifactioryja* Jfrog.

Server na koji postavljamo našu aplikaciju trebao bi biti povezan s *artifactoryjem* tako da jednostavno možemo skinuti aplikaciju i pokrenuti je na serveru. Također, moramo imati na umu da port na kojem se servira aplikacija mora biti propušten od strane *firewalla* kako bi upiti na servis mogli stizati.

Također, server baze podataka mora biti pokrenut na URL-u i portu koji je definiran kao *connection string* unutar aplikacijskih postavki. Isto tako, definirana baza podataka mora postojati unutar servera baze podataka.

7.2. Lokalna instalacija

Gore navedeni koraci su aplikabilni u profesionalnom produkcijskom okruženju. Međutim, ako želimo ovu soluciju pokrenuti lokalno na vlastitom računalu moramo učiniti sljedeće.

S obzirom da je rad izrađen na Linux operacijskom sustavu, u nastavku su prikazani koraci specifični za taj OS.

Prvo moramo instalirati Git klijentski program, po mogućnosti službeni koji se koristi direktno iz komandne linije. Za to se treba otvoriti terminal i upisati naredbu `sudo apt install git.` Sljedeće je potrebno posjetiti GitHub web na kojem je služen aplikacijski kod te kopirati link preko kojeg možemo klonirati projekt na vlastito računalo. Naredba za kloniranje git repozitorija je `git clone https://github.com/mmacan00/zavrsni-rad.git`.

Također, servis je razvijen s verzijom Jave 14 tako da bi lokalno trebali instalirati Javu 14. Tu možemo odabrati službenu verziju koju održava Oracle ili neki drugi JDK kojih ima više. Najpoznatiji je besplatni OpenJDK verzije 14. Nakon što smo preuzeli datoteku sa službene oracle.com stranice moramo instalirati preuzetu datoteku preko terminala sa naredbom `sudo dpkg -i <putanja do preuzete datoteke>`. Po predodređenim postavkama lokacija instaliranog JDKa je `/usr/lib/jvm`. Nakon instalacije moramo kreirati varijablu okruženja `$JAVA_HOME`, te dodati putanju do JDKa u varijablu okruženja `$PATH` tako da na kraj datoteke `/etc/profile` dodamo sljedeće dvije linije:

```
export JAVA_HOME=/usr/lib/jvm/<jdk-14.0.1>
export PATH=${PATH}:${JAVA_HOME}/bin
```

Nakon što smo skinuli i instalirali Javu, trebali bi instalirati Maven. To možemo tako da posjetimo službenu stranicu i preuzmemo instalacijsku datoteku ako smo na Windowsu ili možemo preko ugrađenog *package managera* ako smo na Linuxu tako da izvršimo naredbu `sudo apt install maven`.

Također, trebali bi napraviti račun na Docker Hub te instalirati Docker zato što se obje baze lokalno pokreću svaka u svom Docker *containeru*.

Docker na Linuxu instaliramo sa sljedećim naredbama:

```
sudo apt-get update

sudo apt-get install apt-transport-https ca-certificates curl
gnupg-agent software-properties-common

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo
apt-key add -
```

```
sudo add-apt-repository "deb [arch=amd64] https://down-
load.docker.com/linux/ubuntu $(lsb_release -cs)stable"

sudo apt-get update

sudo apt-get install docker-ce docker-ce-cli containerd.io

sudo groupadd docker

sudo usermod -aG docker $USER
```

Nakon što smo izvršili navedene naredbe moramo se odjaviti i ponovo prijaviti kako bi sve promjene bile prihvaćene na Osu.

Nakon što smo instalirali Docker trebamo povući Docker *image* za PostgreSQL verziju 12, MS SQL Server 2019, te RabbitMQ. Za to su potrebne sljedeće tri naredbe:

```
docker pull postgres

docker pull mcr.microsoft.com/mssql/server

docker pull rabbitmq
```

Kada smo zadovoljili gornje zahtjeve, trebali bi pokrenuti sva tri Docker *containera* tako da se mapira isti port iz *containera* na isti port na našem lokalnom računalu. PostgreSQL se standardno servira preko porta 5432, MS SQL Server preko porta 1433 i RabbitMQ na 15672. Kada smo to učinili trebali bi se spojiti na oba servera baza podataka kako bismo kreirali praznu bazu podataka za *feature flag* servis i *user management* servis. Besplatan i dobar program za spajanje na mnogo različitih baza je DBeaver. Preko njega otvorimo uređivač SQL naredbi i napišemo naredbu `create database FeatureFlagsDB` i izvršimo tu naredbu nad MS SQL Serverom. Zatim otvorimo novi uređivač SQL naredbi i napišemo `create database UserManagementDB` i izvršimo tu naredbu nad PostgreSQL serverom.

Nakon što smo postavili cijelo okruženje, možemo pristupiti pokretanju obje aplikacije. Da bi to učinili moramo ih prvo kompilirati, te zapakirati u standardnu `.jar` komprimiranu datoteku. Druga solucija je odabrati kompiliranje i pakiranje u .war datoteku, i treća i najbolja solucija je kreiranje Docker *imagea* na osnovu aplikacije. Ukratko ćemo proći sva tri scenarija.

7.2.1. .jar

Moramo se preko komandne linije pozicionirati u *root* direktorij jednog od modula kloniranog projekta. Tu moramo izvršiti naredbu `mvn clean install` koja će prvo očistiti ako ima ostataka starih kompiliranja te će povući sve potrebne biblioteke i kompilirati projekt te ga zapakirati u izvršnu `.jar` datoteku. Tada tu datoteku možemo pokrenuti tako da iz komandne linije izvršimo naredbu `java -jar <putanja do .jar datoteke>`. Time bi se aplikacija pri pokretanju spojila na bazu, izvršila migracijske skripte koje bi kreirale shemu i tablice. Tu istu proceduru bi trebali ponoviti i za drugi servis.

7.2.2. .war

Za razliku od pakiranja naše aplikacije u `.jar` datoteku, ako je želimo zapakirati u `.war` samo bi trebali dodati `<packaging>war</packaging>` u `pom.xml` datoteku od oba servisa i ponovno izvršiti `mvn clean install`. Razlika `.war` i `.jar` komprimiranih datoteka je ta što `.jar` datoteku možemo direktno pokrenuti s `java -jar` naredbom i sve bi radilo jer nam je Spring unutar zapakirao Java servlet container. Definirano je to automatski Apache Tomcat, a druge popularne solucije se mogu postaviti, tipa Jetty ili Undertow. Kod pakiranja u `.war` datoteku ne možemo samostalno pokrenuti servis već tu `.war` datoteku moramo postaviti na lokalno instalirani neki od Java servelt containera.

7.2.3. Docker image

Najmodernija solucija za lokalno pokretanje aplikacije bi bila kreiranje Docker *imagea* aplikacije i pokretanje Docker *containera* na osnovu generiranog *imagea*. Srećom i od najnovije verzije Springa 2.3.0. to je jako jednostavno za izvesti. Jednostavno se pozicioniramo u *root* direktorij jednog od modula projekta i izvršimo naredbu `spring-boot:build-image` i time će se lokalno kreirati Docker *image* servisa na osnovu kojeg možemo kreirati Docker *container* i pokrenuti ga. Isti proces samo ponovimo i za drugi modul projekta.

8. Zaključak

Ovim završnim radom prikazano je korištenje *feature flagova* u modernom razvoju softvera. Prikazani su mnogi benefiti, od ubrzanog i sigurnog izbacivanja nove vrijednosti na produkciju, do solucije za osluškivanje reakcije klijenata na proizvod. Pomažu programerima kod otežanog verzioniranja, te testerima omogućuju uranjeno testiranje kako bi sve bilo spremno za globalnu dostupnost. *Feature flagovi* su prepoznati kao dobra praksa te ga sve veće kompanije koriste u svom razvojnom procesu.

Također vidjeli smo kako se implementiraju moderne mikroservisne arhitekture u modernim tehnologijama, kao što su Spring, RabbitMQ, React i Docker. Uvidjeli smo kako i kada se koriste sabirnice za poruke kako bi određeni servisi mogli pravovremeno reagirati na događaje u nekim drugim servisima i slično.

Prikazane su zanimljive specifičnosti kod Java programskog jezika te odstupanja u SQL sintaksi za različite RDBMS kao MS SQL Server i PostgreSQL. Također prikazani su različiti načini instalacije Java aplikacija, bilo lokalno bilo na servis, te je ukratko prikazan CI/CD proces kakav se danas koristi u IT industriji.

Popis slika

Popis primjera koda

Literatura

1. *https://www.baeldung.com/java-reflection-class-fields*
2. *https://docs.docker.com/engine/install/ubuntu/*
3. *https://howtodoinjava.com/java/basics/java-tutorial/*