UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Uroš Milenković

RAZVOJ VEB APLIKACIJE U OBLAKU ZA UPRAVLJANJE PREVODIMA U VIŠEJEZIČNIM APLIKACIJAMA

master rad

Beograd, 2021.

Mentor:

prof. dr Saša Malkov, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

doc. dr Jelena Graovac, docent
Univerzitet u Beogradu, Matematički fakultet

Anđelka Zečević, asistent
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: 28. septembar 2021.

Naslov master rada: Razvoj veb aplikacije u oblaku za upravljanje prevodima u višejezičnim aplikacijama

Rezime: Računarstvo u oblaku se svakog dana sve više razvija a prednosti softvera u oblaku su višestruke. Drugi trend koji se može primetiti u industriji je razvoj softvera u mikroservis arhitekturi. Iako ove dve tehnike imaju svoje prednosti, one sa sobom nose i određenu kompleksnost koja se ne sme zanemariti.

Bitna karakteristika modernih aplikacija je da budu višejezične. Takve aplikacije pružaju bolje korisničko iskustvo prevazilaženjem jezičkih barijera. Razvoj višejezičnih aplikacija zahteva saradnju programera i prevodioca, koja se ogleda u tome da programeri šalju šifrarnike na jednom jeziku prevodiocu, a potom prevodilac vraća popunjen šifrarnik na željenom jeziku. Šifrarnici se uglavnom prosleđuju u čoveku čitljivom formatu za serijalizaciju, kao što su *JSON*, *YML* ili *XML*.

Informacioni sistem koji pruža podršku saradnji programera i prevodioca, a pritom razvijen u oblaku, olakšao bi posao obema stranama u velikoj meri tako pto bi imao visoku dostupnost, standardizovao bi proces razmene prevoda i prevodiocima bi pružao prijatno korisničko iskustvo.

Cilj ovog rada je da predloži i razvije informacioni sistem u oblaku koji bi pomogao programerima i prevodiocima tokom razvijanja višejezične aplikacije.

U okviru rada na tezi su obrazloženi, analizirani i diskutovani arhitektura i dizajn sistema koji podržavaju postavljene zahteve. Uz ovaj rad je napravljen sistem kao veb aplikacija u oblaku uz pomoć *Kubernetes* sistema za automatsko skaliranje i upravljanje kontejnerskim aplikacijama.

Ključne reči: višejezičnost, računarstvo u oblaku, mikroservisi, javascript, react, nestjs, docker, kubernetes

Sadržaj

1 Uvod **1**
1.1 Problem prevođenja 2
1.2 Arhitektura veb aplikacija 4
1.3 Tehnologije korišćene u radu 10

2 Funkcionalni zahtevi aplikacije Interpres **25**
2.1 Uređivač prevoda 25
2.2 Uvoz i izvoz prevoda 26
2.3 Aplikacija u oblaku 27

3 Slučajevi upotrebe aplikacije Interpres **28**
3.1 Prijava korisnika uz pomoć GitHub IdP 28
3.2 Pravljenje projekta 30
3.3 Uvoz prevoda 32
3.4 Izmena prevoda 33
3.5 Izvoz prevoda 34

4 Implementacija aplikacije Interpres **36**
4.1 Komponente sistema 36
4.2 Klijent 39
4.3 Server 40
4.4 Automatizacija 41
4.5 Produkciono okruženje 43

5 Zaključak **45**

Literatura **47**

Glava 1

Uvod

Aplikacije koje su namenjene za globalno tržište treba pripremiti tako da korisnici koji pripadaju različitim govornim područjima i geografskim regionima razumeju sadržaj. Postupak dizajniranja aplikacije tako da podrži različite jezike se naziva internacionalizacija, a osobina aplikacije koja podržava više jezika se naziva višejezičnost.

Proces prilagođavanja softvera za različite jezike može biti kompleksan. Veliku pomoć u tom procesu danas pružaju različita rešenja. Odabir rešenja uglavnom zavisi od odabira programskog jezika u kom će softver biti razvijan, kao i od korišćenja programskog okvira.

Sa druge strane, ne može se očekivati da programer zna sve jezike za koje je softver namenjen. To znači da se u proces internacionalizacije uključuju i prevodioci koji poznaju jezik korisnika. Oni implicitno postaju osobe koje razvijaju softver.

Pomenuta rešenja za olakšavanje internacionalizacije uglavnom su fokusirana samo na tehničke probleme, odnosno probleme programera. Prevodi se čuvaju u fajlovima i obično predstavljaju serijalizovanu strukturu mape, odnosno parove „ključ – vrednost". Takvi fajlovi su neretko teški za korišćenje od strane netehničkih lica, a prevodioci često nisu tehnička lica. Kako bi što bolje obavljali svoj posao, prevodiocima je potrebna neka vrsta alata koja im deluje poznato, nešto na šta su navikli, odnosno nešto što koriste svaki dan. Prevodioce ne treba opterećivati rešenjima koje su programeri izabrali, niti koji format serijalizacije se koristi. Njima je potreban familijaran interfejs za ažuriranje tih fajlova.

Razvojem računarstva u oblaku, ljudi su navikli da im se sve nalazi u oblaku, odnosno da im je sve uvek dostupno, na svakom mestu i u bilo koje vreme.

Paralelno sa razvojem softvera u oblaku, popularnost stiču arhitekture zasnovane na mikroservisima ali i razvoj aplikacija u kontejnerima. Za kontrolisanje velikog broja kontejnera se sve više korsti alat *Kubernetes*. Pored toga, tendencija je da se hardver, odnosno infrastruktura, više ne održava u okviru organizacije, već da se on iznajmljuje od dobavljača računarstva u oblaku (eng. *Cloud Computing Provider*). Ovakvim pristupom se omogućava lako prenošenje aplikacije sa jednog okruženja na drugo. Razvoj takvog softvera nosi sa sobom dodatne izazove.

U ovom radu biće opisan razvoj softvera u oblaku. Fokus će biti stavljen na arhitekturu mikroservisa, korišćenje alata *Kubernetes* i proces pravljenja višejezične aplikacije. Za razvoj klijentskog, ali i serverskog dela kôda, koristiće se programski jezik *JavaScript*. U svrhu ilustracije i boljeg razumevanja biće razvijena aplikacija za upravljanje prevodima. Implementirano rešenje je nazvano *Interpres*. Aplikacija je višejezična, odnosno može prikazati korisnički interfejs u više jezika. Za potrebe uređivanja prevoda je korišćen upravo *Interpres*. Izvorni kod je otvoren i može se naći na adresi `https://github.com/enco164/interpres`.

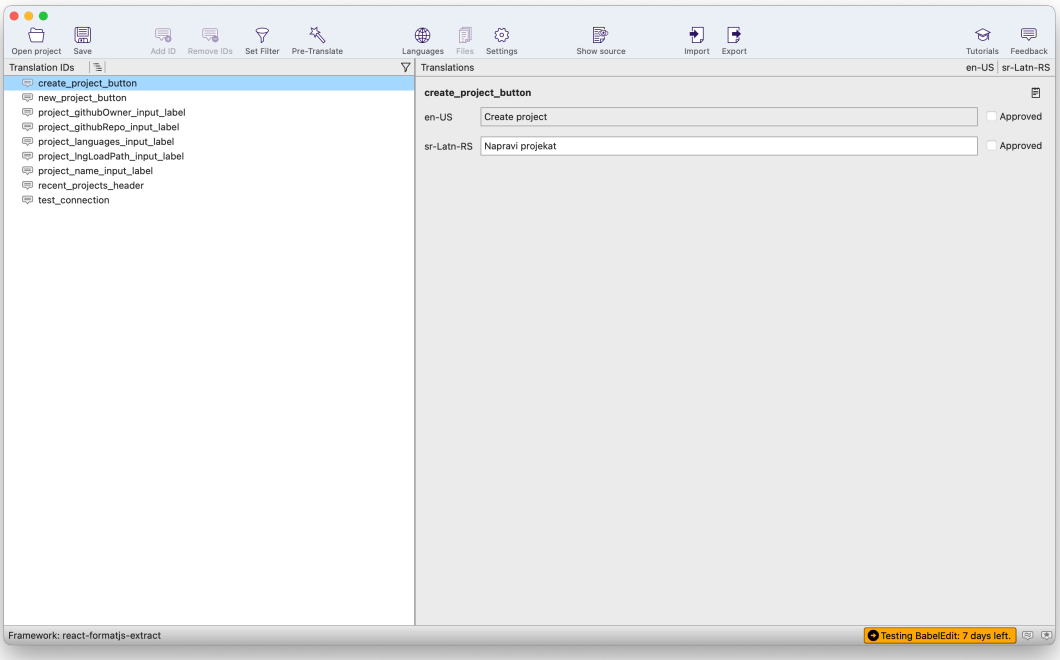
1.1 Problem prevođenja

Analizom postojećih alata za upravljanje prevodima mogu se uočiti prednosti i mane tih alata. Zadatak analize postojećih sistema je upoznavanje sa problematikom kako bi se izbegle greške koje su ti sistemi napravili, ali i prikupljanje dobrih ideja i funkcionalnosti koje pružaju ti sistemi.

Postojeća okruženja

Na tržištu postoji dosta rešenja koji se bave upravljanjem prevodima. Jedno od takvih rešenja je *BabelEdit* [1], editor prevoda za veb aplikacije. Program se instalira na klijentskom računaru. Program podržava pravljenje projekta u koji se kasnije uvoze fajlovi sa prevodima. Kada korisnik završi sa prevođenjem, može da izveze prevode iz projekata u razne formate. Ovaj alat ne nudi mogućnost kolaboracije, a fajlovi se nalaze na lokalnom računaru, i kao takav je pogodan jedino za manje projekte. Ovaj alat ima probni period od nekoliko dana, a kasnije se mora platiti. Na slici 1.1 je prikazan korisnički interfejs alata *BabelEdit*.

Drugo popularno rešenje je *Localazy* [2], platforma koja podržava više od 50 radnih okvira, formata fajlova. Platforma se nalazi u oblaku i pristupa joj se



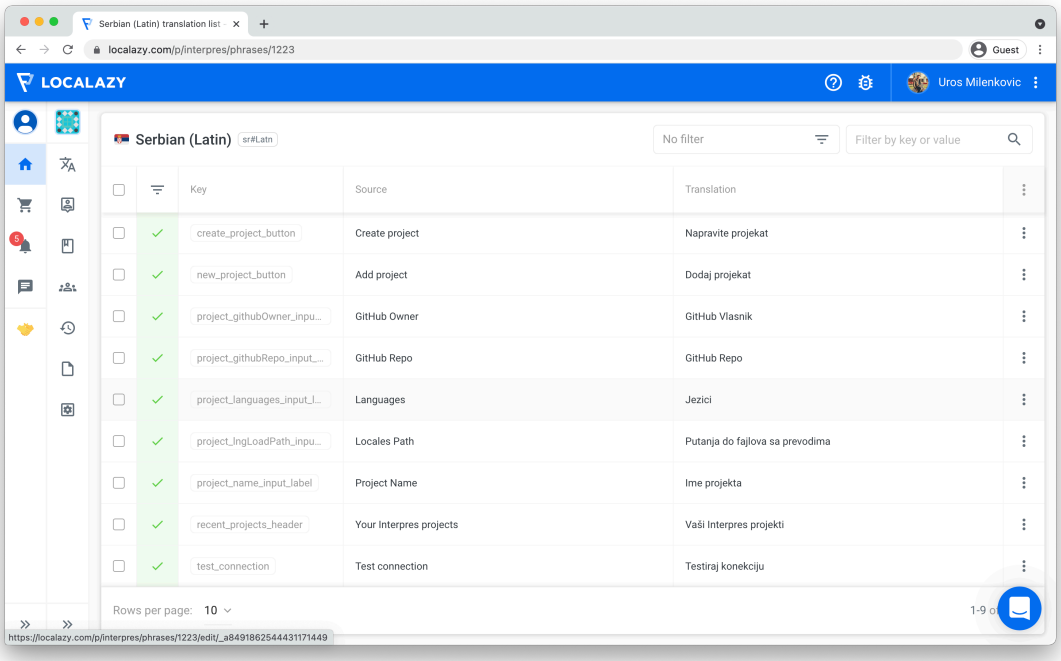
Slika 1.1: Korisnički interfejs alata *BabelEdit*

preko veb pregledača. Na platformi se može napraviti novi projekat u okviru kojeg se mogu uvesti fajlovi sa prevodima za izabrani jezik. Istom projektu može pristupiti više prevodilaca i uređivati prevode. Kada se izabere fajl za prevod, sa leve strane se može videti ključ za nisku koju treba prevesti, a pored njega i sam prevod. Prevodi su kasnije dostupni preko mreže za dostavu sadržaja (eng. *Content Delivery Network, CDN*). *Localazy* omogućava besplatno korišćenje za projekte koji imaju najviše 200 ključeva, što odlikuje manje projekte. Na slici 1.2 je prikazan korisnički interfejs alata *Localazy*.

Navedeni alati poseduju i integraciju sa sistemima za mašinsko prevođenje, što u mnogome olakšava posao prevodiocima. Svakako, na kraju je potrebna provera od strane čoveka kako bi sadržaj bio u većoj meri prilagođen.

Moguća unapređenja

Oba analizirana alata pružaju nekakav oblik obaveštenja da je prevodilac završio sa poslom. U slučaju alata *BabelEdit* potrebno je da prevodilac izveze prevode i kasnije izvezene fajlove da pošalje programeru. Kod *Localazy* je proces malo bolji jer prevodilac ne treba da šalje nikakve fajlove, već samo da obavesti programera,



Slika 1.2: Korisnički interfejs alata *Localazy*

koji će ih kasnije preuzeti sa *CDN*.

Ovde se može primetiti da je moguće automatizovati proces predaje prevoda. Potrebno je samo da klikom dugmeta prevodilac obavesti sistem da je završio svoj posao. Ta akcija bi trebalo da pokrene izgradnju nove verzije aplikacije.

1.2 Arhitektura veb aplikacija

Aplikacije na vebu su dominantno bazirane po modelu klijent–server arhitekture. Server u takvoj arhitekturi obezbeđuje uslugu klijentima koji je zahtevaju. Klijent–server arhitektura je višenamenska i modularna, a sa ciljem unapređenja upotrebljivosti, interoperabilnosti, fleksibilnosti i skalabilnosti.

Veb aplikacije su nezavisne od platforme jer zahtevaju samo veb pregledač, dok se kod standardne klijent–server arhitekture klijent mora instalirati na platformi korisnika. Pored toga, kod veb aplikacija je protokol komunikacije definisan, koristi se *HTTP*, dok se kod klijent–server arhitekture protokol može izabrati po potrebi.

Komponente klijent – server arhitekture, prilagođene za veb, se mogu grupisati u tri kategorije: server, klijent i mreža. Uloga servera je da upravlja zajedničkim

resursima, bazom podataka, da izvršava poslovnu logiku, kao i da kontroliše pristup i bezbednost podataka. Posao klijenta je da upravlja korisničkim interfejsom. Računarska mreža omogućava komunikaciju između klijenta i servera, a komunikacija mora pratiti određene standarde.

Arhitektura teških klijenata

Odlika arhitekture teških klijenata je da server šalje klijentu podatke i meta podatke, i prepušta mu da samostalno pripremi prezentaciju [4]. Ovakvom podelom odgovornosti se smanjuje opterećenje servera, jer više ne mora da priprema prezentaciju. Arhitektura teških klijenata podrazumeva intenzivno korišćenje *JavaScript*-a.

Ekstremni oblik ove arhitekture je arhitektura jedne stranice (eng. *Single Page Application, SPA*). Arhitektura jedne stranice propisuje da se čitava aplikacija učitava kroz jednu stranicu. Primenom ovih pravila se dobija potpuno odvojen klijent od servera. Iscrtavanjem korisničkog interfejsa na klijentskoj strani postiže se bolje korisničko iskustvo iz dva razloga. Prvo, pošto klijent priprema prezentaciju sam, onda nema potrebe čekati prezentaciju da pristigne preko mreže, već će se za korisnika sve brže prikazivati. Drugo, preko mreže se u tom slučaju preuzimaju samo podaci i izvode transakcije, pa je i protok kroz mrežu manji.

Mikroservisi

Tradicionalan način razvijanja poslovnih aplikacija predstavlja razvoj takozvane monolitne arhitekture. Jedan monolit bi sadržao svu poslovnu logiku koju izvršava aplikacija. Kako aplikacija raste vremenom, tako raste i sam monolit, koji se sve teže održava. Problemi koji nastaju kod velikog monolita nisu samo problemi održavanja, već je i skaliranje upitno. Njih karakteriše spor razvojni ciklus i ažuriraju se relativno retko.

Danas se ovako veliki monoliti raščlanjuju na manje, nezavisne komponente koje se nazivaju mikroservisima. Džejms Luis i Martin Fauler, dvojica konsultanata za mikroservise koji rade za *Thoughtworks*, kažu da se izraz „Mikroservisna arhitektura" sve češće pojavlje poslednjih par godina kako bi opisao određeni način projektovanja softverskih aplikacija kao paketa usluga koji se mogu odvojeno isporučiti. Iako ne postoji precizna definicija ovog arhitektonskog stila, postoje

određene zajedničke karakteristike u organizaciji oko poslovne logike, automatskog isporučivanja i decentralizovane kontrole podataka [3].

Mikroservisi predstavljaju aplikaciju koja je strukturirana kao kolekcija slabo vezanih servisa. Glavna ideja iza mikroservisne arhitekture je da se aplikacije lakše razvijaju i održavaju ako su podeljene na manje delove koji nesmetano rade zajedno.

Servisi se mogu posmatrati kao komponente sistema. Martin Fauler definiše komponente kao jedinice softvera koje se mogu nezavisno zameniti i unaprediti [5].

Biblioteke su komponente koje su povezane u program i pozivaju se pomoću funkcija koje se nalaze u memoriji, dok su servisi komponente koje su van procesa i koje komuniciraju preko mreže. Servisi, naravno, mogu da koriste biblioteke. Glavna benefit ovakvog korišćenja servisa je da se mogu nezavisno isporučiti, što nije slučaj ako imamo jednu aplikaciju koja je sastavljena od skupa biblioteka. Sa druge strane, daljinski pozivi preko mreže su skuplji, odnosno sporiji. Druga loša strana je da se logika teže prebacuje iz jednog servisa u drugi, odnosno refaktorisanje je otežano u tom smislu.

Da bi se monoliti lakše razvijali, programeri se uglavnom podele po tehnologiji, na primer u tim za bazu podataka, tim za poslovnu logiku i tim za korisnički interfejs. U arhitekturi mikroservisa je malo drugačije. Timovi se organizuju oko jednog servisa i sastavljeni su od progamera različitih tehnologija. To znači da su timovi nezavisni, kao i što su i sami servisi nezavisni, i organizovani su sa fokusom na poslovnu logiku, a ne na tehnologiju.

Jedna zgodna posledica deljenja monolita na servise je da servisi ne moraju da budu razvijeni u istim tehnologijama. Tako na primer, jedan servis može biti izgrađen u programskom jeziku *C++* a drugi u *NodeJS*, dokle god mogu da komuniciraju jedan sa drugim. Nekada je pogodnije koristiti drugi programski jezik jer je u njemu lakše rešiti problem.

Monolitne aplikacije preferiraju da imaju jednu bazu podataka za čuvanje podataka. Sa druge strane, kod mikroservisnih aplikacija se odluka o bazi podataka prepušta samom servisu. Neke probleme koje servis rešava je pogodnije rešiti relacionom bazom podataka, dok je za neki drugi servis možda pogodnija grafovska baza. Iako i monolitne aplikacije mogu da koriste više tipova baza podataka, ova osobina je češća kod mikroservisa. Decentralizovana odgovornost za čuvanje podataka ima i svoje implikacije na ažuriranje podataka. Ovaj problem se može rešiti korišćenjem transakcija. Korišćenjem transakcija rešava se problem konzistentno-

sti, ali se smanjuje dostupnost. Zato, česta je odluka da se konzistentnost zameni odloženom konzistentnošću, u onim slučajevima gde je ona moguća.

Kada se govori o mikroservisima često se, sa razlogom, postavlja pitanje postoji li razlika između arhitekture mikroservisa i servisno orijentisane arhitekture (eng. *Service Oriented Architecture, SOA*). Glavne karakteristike arhitekture mikroservisa su umnogome slične *SOA*. Problem je taj što *SOA* može da predstavlja mnogo različitih stvari. Preveliki fokus na kanal komunikacije (eng. *Enterprise Service Bus, ESB*) koji se koristi za integraciju monolitnih aplikacija, predstavlja jedan od tih problema. Može se reći da je arhitektura mikroservisa nastala iz stečenog iskustva tokom razvijanja *SOA*, odnosno da predstavlja sledeći iterativni korak u razvoju ove arhitekture. Preuzeti su raznorazni dobri obrasci iz *SOA*. Sa druge strane, delovi koji su bili previše kompleksni zamenjeni su jednostavnijim. Primer je *ESB* koji je zamenjen jednostavnijim veb protokolima.

Razlike između ove dve arhitekture, u suštini, i nema mnogo. Ključne tri razlike su:

- Kod mikroservisa je nivo granularnosti niži. To znači da su pojedinačni mikroservisi suviše mali i u drugačijem kontekstu se retko mogu samostalno ponovo upotrebiti. Jedinice ponovne upotrebe u ovoj arhitekturi predstavljaju grupe mikroservisa. Sa druge strane, u *SOA*, servis predstavlja i jedinicu ponovne upotrebe i jedinicu funkcionalnosti.

- Kod mikroservisa se ne daje preveliki značaj standardizaciji komunikacije koliko kod *SOA*. Naime, kod mikroservisa različiti mikroservisi mogu koristiti različite tehnologije, na primer neki mogu koristiti *REST* a drugi mogu koristiti *GraphQL*. Reprezentacije podataka se isto mogu razlikovati jer jedni mogu serijalizovati podatke u formatu *JSON* a drugi u fromatu *XML*. Pored toga, sami mikroservisi mogu imati neke druge protokole, osmišljene samo za njihove potrebe. *SOA* po ovom pitanju pokušava da standardizuje način komunikacije. U slučaju veb servisa se za protokol bira *SOAP*.

- Mikroservisi podrazumevaju da je komunikacija između njih bez stanja. Komunikacijom bez stanja se postiže lakša horizontalna replikacija, odnosno viša skalabilnost. U *SOA* se komunikacija između servisa često odvija u okviru sesije.

Ljudi koji zagovaraju arhitekturu mikroservisa su iz navedenih razloga krenuli da odbacuju naziv *SOA*, dok drugi smatraju da su mikroservisi samo još jedan oblik *SOA*. Baš zato što *SOA* može da predstavlja toliko različitih stvari, dragoceno je imati izraz koji preciznije opisuje arhitekturu zvanu mikroservisi [3].

REST

Interoperabilnost je sposobnost da različiti sistemi rade zajedno. Da bi se ovo postiglo potrebni su standardi koje će sistemi poštovati. Kako u klijent – server arhitekturi imamo dva različita sistema (klijent i server), oni moraju poštovati određene standarde za komunikaciju. Jedan od takvih standarda za komunikaciju je i *REST* (eng. *Representational State Transfer*). *REST* je stil softverske arhitekture koji definiše skup pravila koja treba da se poštuju tokom pravljenja mrežnih aplikacija. Prvi put je predstavljen u doktorskoj disertaciji Roja Fildinga [6]. Po *REST*-u, fokus se prebacuje sa procedura na resurse (objekte). Za aplikaciju kažemo da je *RESTful* ako poštuje pravila REST-a.

REST definiše šest pravila koja moraju da se ispoštuju [7]:

- **Klijent – server**: Razdvajanjem korisničkog interfejsa od servera briga o podacima ostaje na serveru. Klijenti postaju lakši za portabilnost među različitim sistemima, a server se može lakše skalirati.
- **Sistem bez stanja (eng. *Stateless*)**: Svaki zahtev sa klijenta ka serveru mora da sadrži sve potrebne informacije da bi zahtev bio opslužen. Klijent se ne sme oslanjati na stanje servera. Zapravo, stanje na serveru se zabranjuje. Iz tog razloga stanje sesije se u potpunosti čuva na klijentu.
- **Keširanje**: Server u svom odgovoru sa nekim podacima mora eksplicitno da navede da li podaci mogu da se keširaju ili ne. Ako server odgovori da su podaci kešabilni, klijent ih može koristiti kasnije ako je zahtev za podacima isti. Server može napomenuti i vreme isteka keša.
- **Uniforman interfejs**: Uniforman interfejs definiše interfejs između klijenta i servera. On pojednostavljuje i razdvaja arhitekturu, što omogućava da se svaki deo razvija samostalno. Četiri vodeća principa uniformnog interfejsa su:

1. **Zasnovanost na resursima**: Individualni resursi se mogu identifikovati u zahtevima koristeći *URI* kao identifikator resursa. Resursi kao takvi su konceptualno odvojeni od reprezentacije koja se vraća klijentima. Na primer, server ne šalje svoju bazu podataka, već radije, šalje *HTML*, *XML* ili *JSON* sadržaje koji predstavljaju traženi zapis iz baze podataka.
2. **Upravljanje resursima kroz reprezentaciju**: Kada klijent sadrži reprezentaciju resursa, uključujući meta podatke u prilogu, onda ima dovoljno podataka da izmeni ili obriše resurse sa servera, pod uslovom da ima ovlašćenje da to uradi.
3. **Samoopisne poruke**: Svaka poruka sadrži dovoljno informacija da opiše na koji način treba da bude obrađena. Na primer *Internet media type* (nekada poznat pod nazivom *MIME*) može sadržati informaciju o tome koji parser treba da se pozove. Odgovori eksplicitno označavaju da li imaju sposobnost keširanja.
4. **Hipermedija kao pokretač aplikacije (*HATEOAS*)**: Klijenti treba da znaju što je manje moguće o tome kako da komuniciraju sa serverom. Komunikacija treba da bude što je više moguće generička. Server svoje stanje isporučuje klijentu u vidu hiperteksta, koji unutar sebe sadrži hiperlinkove. To se tehnički naziva hipermedija (hiperlinkovi unutar hiperteksta). *HATEOAS* (eng. *Hypermedia as the Engine of Application State*) znači da, tamo gde je potrebno, veze su sadržane unutar vraćenog odgovora. Tako se isporučuje URI za preuzimanje samog objekta ili srodnih objekata. Uniformni interfejs koji svaki *REST* servis mora pružiti je fundamentalna osnova za njegov dizajn.

- **Slojevit sistem**: *REST* omogućava korišćenje slojevitih sistema, gde klijent ne zna eksplicitno da li komunicira sa krajnjim serverom ili sa posrednikom. Ovako se može povećati skalabilnost uvođenjem balansera opterećenja ili uvođenjem keširanja na strani servera.
- **Kod na zahtev (opciono)**: Serveri mogu obogatiti klijentsku stranu slanjem koda koji će biti izvršen na klijentskoj strani. Ovo može uprostiti klijente jer se smanjuje broj funkcija koje je potrebno da klijent ima. Na primer, u veb aplikacijama bi to bilo slanje *JavaScript* koda.

1.3 Tehnologije korišćene u radu

U ovom poglavlju će biti opisane tehnologije korišćene za razvoj rešenja, a prateći ranije opisanu arhitekturu.

React

React je *JavaScript* biblioteka za građenje korisničkog interfejsa. Koristi se za pravljenje modernih reaktivnih korisničkih interfejsa koji se mogu izgraditi uz pomoć izolovanih delova koda. Ti delovi koda nazivaju se komponente. Korišćenjem komponenti omogućeno je deljenje aplikacije na nezavisne delove koji se prilikom razvoja mogu iznova koristiti [8].

Svaka komponenta sastoji se iz metode za renderovanje i sadrži podatke o tome šta je sve potrebno da se prikaže krajnjem korisniku. Komponente mogu biti klasne ili funkcijske. Funkcijske komponente (eng. *function components*) kao ulazni parametar primaju svojstva i vraćaju *JSX* element. Sa druge strane, klasne komponente su definisane kao *ES6* klase i renderuju *React* elemente koristeći `render()` metodu.

Kako bi komunicirale međusobno sa ostalim delovima aplikacije, komponente moraju imati svoje stanje (engl. *state*) i svojstva (eng. *properties - props*). Svojstva i stanje predstavljaju čist *JavaScript* objekat. Izmena bilo kojeg od ova dva objekta okida ponovno renderovanje komponente.

Svojstva čine *React* komponente fleksibilnim i pogodnim za ponovno korišćenje. Prosleđivanjem različitih svojstava, komponente se mogu ponašati i izgledati drugačije. Svojstva komponente služe isključivo za prikaz i unutar same komponente ih ne bi trebalo menjati. Protok informacija mora biti jednosmeran od višeg nivoa komponente ka nižem. Stoga je komponenta sa višeg nivoa zadužena za prosleđivanje skupa svojstava.

Stanje komponente se koristi za „pamćenje" podataka unutar same komponente. Možemo reći da je stanje privatno, jer se stanju komponente ne može pristupiti iz roditeljske komponente. Može se promeniti na osnovu korisnikove interakcije ili neke druge akcije unutar aplikacije. Za razliku od svojstava, stanjem se može upravljati unutar same komponente.

React kuke (eng. *hooks*) služe za korišćenje stanja u funkcijskim komponentama. U teoriji, kuke su funkcije koje se „kače" za stanja i životne cikluse unutar

funkcijske komponente. Da bi se koristile *React* kuke, moraju se poštovati dva pravila:

- Pozivanje kuka se mora vršiti sa najvišeg nivoa. To znači da se one ne smeju pozivati unutar petlji, postavljenih uslova ili ugnježdenih funkcija.
- Pozivanje kuka se ne može vršiti iz običnih *JavaScript* funkcija, već samo iz funkcija *React* komponente.

Stanje svake komponente je potpuno nezavisno. Korišćenjem kuka, postiže se to da se logika sa stanjem može ponovo koristiti (ali ne i samo stanje). U stvari, svakim pozivom kuke dobijamo potpuno izolovano stanje, pa se stoga jedna kuka može koristiti više puta unutar jedne komponente. Pored toga, kuke omogućavaju pravljenje kompozicije. Unutar jedne kuke je moguće zakačiti drugu kuku i tako napraviti kompoziciju.

Pre uvođenja kuka, u *React*-u se koristio obrazac zvan „komponente višeg reda" (eng. *Higher-Order Components*). Konkretno, komponenta višeg reda je funkcija koja za argument prihvata komponentu i vraća novu komponentu. Kako su i ulaz i izlaz istog tipa (*React* komponenta), lako se može zaključiti da se komponente višeg reda mogu komponovati, a logika i stanje koje one dodaju se mogu iskoristiti na više različitih mesta. Iako je i dalje moguće koristiti ovaj obrazac, jer je nezavisan od toga da li je komponenta napisana kao klasa ili kako funkcija, preporučuje se korišćenje kuka, kad god je moguće. Naime, kuke imaju dve glavne osobine koje imaju i komponente višeg reda: ponovno korišćenje i kompoziciju. Glavna prednost kuka u odnosu na komponente višeg reda je to što su zavisnosti eksplicitne kod kuka. Sa druge strane, kuke se mogu koristiti samo u funkcijskim komponentama pa je upotreba komponenti višeg reda opširnija.

Glavna prednost *React* kuka je enkapsulacija. Odgovornost za prikazivanje korisničkog interfejsa ostaje na samoj komponenti, a logika prikaza za tu komponentu se prebacuje u kuku koja se vezuje za tu komponentu. Na taj način se štiti od neželjenog pristupa unutrašnjim podacima od strane komponente i sakriva se kompleksnost. Uz sve navedeno, testiranje postaje lakše jer se odvojeno mogu testirati prikaz korisničkog interfejsa i logika prikaza. Praktično, testovi za kuke, odnosno logike prikaza, postaju testovi jedinice koda (eng. *Unit Test*) a testovi prikaza korisničkog interfejsa postaju integracioni testovi (eng. *Integration Test*).

Prilikom razvoja aplikacije susrećemo se sa izazovima kao što je pravljenje konzistentnih komponenti koje izgledaju i funkcionišu na isti način, ne samo unutar

jedne aplikacije već i na mnogim drugim aplikacijama koje se razvijaju. Ispis identičnih stilova komponenata iznova i izmena promenljivih vrednosti na više mesta u aplikaciji utiče na vreme razvoja aplikacije.

Umesto da se stilovi koriste globalno, teži se ka tome da se postave standardizovane komponente sa ugrađenim funkcionalnostima i stilovima kako bi se one mogle koristiti iznova i podešavati jednostavno u zavisnosti od slučaja u kojem se koriste. Jedna od popularnijih biblioteka za stilizovanje *React* komponenata je *Material UI*. Ona predstavlja implementaciju *Google*-ovog *Material Design*-a. Uz pomoć ove biblioteke komponente se mogu koristiti zasebno, koristeći samo one stilove koji treba da budu prikazani.

Material UI biblioteka koristi *CSS* u *JS* (eng. *CSS-in-JS*). Korišćenjem globalnih *CSS* fajlova, a zbog kaskadne prirode *CSS*-a (*Cascading Style Sheets*), stilovi se mogu učitati u bilo kom redosledu i pregaziti neki drugi stil. Pisanjem *CSS* u *JavaScript*-u rešava problem zavisnosti jer se mogu koristiti *JavaScript* moduli. Na taj način se komponente mogu pokretati nezavisno, bez oslanjanja na eksterne *CSS* fajlove. Kako danas većina alata za izgradnju projekta podržava pronalaženje i brisanje koda koji se ne koristi, a samim tim i smanjuju veličinu izgrađenog projekta, ova tehnika se može primeniti i na *CSS* u *JS*. Stilovi su opisani preko *JavaScript* objekata i kao takvi mogu da pristupe stanju komponente što povećava fleksibilnost pisanja samih stilova.

NestJS

Razvoj aplikacije na *Node.js* platformi vremenom može postati kompleksan. Što se više svojstava dodaje u aplikaciju, to baza koda postaje sve veća.

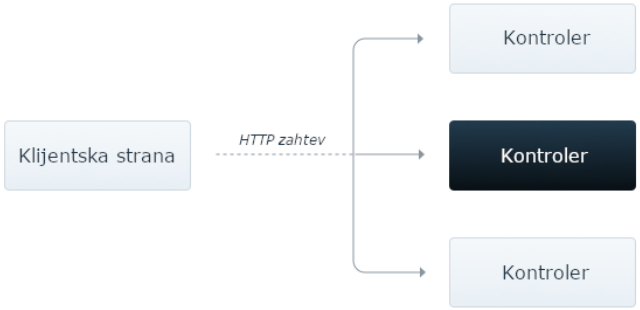
NestJS je radni okvir za građenje efikasne skalabilne *Node.js* aplikacije na serveru. Pruža serverskom delu aplikacije modularnu strukturu za organizaciju koda u odvojene module i služi za eliminisanje neorganizovane baze kodova.

Korišćenjem *NestJS* programerima se pruža mogućnost da koriste sve benefite JavaScript-a i uz to pišu kod za klijentski i serverski deo aplikacije u istom programskom jeziku. Sadrži komponente funkcionalnog, objektno orijentisanog i funkcionalno reaktivnog programiranja.

Inspirisan *Angular*-om, *NestJS* je napisan u programskom jeziku *TypeScript* i koristi *Express.js*, koji ga čini kompatibilnim sa većinom *Express* posredničnih (eng. *middleware*) softvera. Za razvoj podržava JavaScript i TypeScript, kao i PostgreSQL, MongoDB i MySQL baze podataka.

Osnovne gradivne komponente NestJS aplikacije su kontroleri, provajderi i moduli.

Kao i kod većine radnih okvira, kontroleri u *NestJS* su odgovorini za opsluživanje zahteva koji dolaze sa klijentske strane. *NestJS* je struktuiran tako da je mehanizam za usmeravanje, prikazan na slici 1.3, sposoban da kontroliše koji kontroler će biti zadužen za određeni zahtev. Kako bi se napravio osnovni kontroler, koriste se klase i dekoratori. Dekoratori pridružuju meta podatke klasama i omogućavaju *Nest*-u da napravi mapu za rutiranje. Za definisanje HTTP metoda za rute, koriste se dekoratori `@Get()`, `@Put()`, `@Post()` i `@Delete()` [10].



Slika 1.3: Usmeravanje zahteva

Nest provajderi se mogu umetnuti u kontrolere ili u druge provajdere. Nazivaju se i servisi, i dizajnirani su da apstrahuju bilo koji oblik složenosti i logike. Servisni provajder u *Nest* je klasa sa specijalnim dekoratorm `@Injectable()`.

Moduli omogućavaju grupisanje povezanih fajlova. To su TypeScript fajlovi dekorisani sa `@Module` dekoratorom, koji pruža meta podatke koje *Nest* koristi za organizaciju strukture aplikacije. Svaka *Nest* aplikacija mora imati bar jedan modul – osnovni modul. Preporučuje se da se velike aplikacije rastave na više modula kako bi se lakše kontrolisala struktura aplikacije.

Tri osnovna koncepta NestJS su:

- **DTO**: Objekat za prenos podataka (eng. *Data Transfer Object*) je objekat koji definiše kako će se podaci poslati preko mreže;

- **Interfejsi** : *TypeScript* interfejsi se koriste za proveru tipa i definisanje tipova podataka koji se mogu proslediti kontroleru ili *NestJS* servisu i
- **Umetanje zavisnosti** (eng. *Dependency Injection*) je obrazac projektovanja koji se koristi za povećanje efikasnosti i modularnosti aplikaicje. Održava kod čistim i lakšim za korišćenje. U *NestJS* se koristi za pravljenje uvezanih komponenata [9].

GitHub aplikacija

Aplikacije na *GitHub*-u mogu automatizovati i poboljšati tok razvoja. *GitHub* aplikacije se preporučuju za integrisanje sa *GitHub*-om jer nude granularnije permisije za pristup podacima.

GitHub aplikacije se mogu instalirati direktno na organizacione i korisničke naloge i odobriti pristup specifičnim repozitorijima. Može ih instalirati vlasnik naloga organizacije ili korisnik sa administratorskim dozvolama. Kako bi ih promovisao, *GitHub* je napravio stranicu koja predstavlja prodavnicu aplikacija i preko ove prodavnice je moguće instalirati aplikacije i dodatno obogatiti repozitorijume.

GitHub aplikacije se ponašaju kao dodatni korisnici na repozitorijumu. Za svaku aplikaciju se vezuje jedan identitet koji joj odgovara. Registrovanjem aplikacije programer dobija pristupne ključeve. Uz pomoć ovih ključeva i *GitHub*-ovom bibliotekom, zvanom *octokit*, programer može da izvršava akcije u repozitorijumu, kao što su ostavljanje komentara, menjanje datoteka, pravljenje zahteva za promenu ili mogu da reaguju na neke događaje koji su se desili u repozitorijumu preko *Webhook*-ova. Ove aplikacije se ne izvršavaju na *GitHub*-u već za njih mora da se obezbedi server.

Za poboljšanje rada u repozitorijumu, može se napraviti *GitHub* aplikacija koja sadrži više skripti ili celu aplikaciju, i takvu aplikaciju uvezati sa mnoštvo drugih alata [11]. Tako na primer, može se povezati sistem za uređivanje prevoda koji bi mogao da otvori novi „zahtev za promenu" kad su prevodioci završili sa svojim ciklusom prevođenja.

OAuth je standardni protokol koji omogućava korisniku da autorizuje pristup ka *API*-ju od strane aplikacije. Onda kada je pristup omogućen, autorizovana aplikacija može koristiti *API* u ime korisnika.

OAuth 1.0 je delegirana strategija autentikacije koja uključuje više koraka. *OAuth 1.0* se koristi samo za veb. Aplikacija koja zahteva pristup se može iden-

tifikovati preko korisničkog ključa (eng. *consumer key*) i korisničke šifre (eng. *consumer secret*). *OAuth 2.0* je dizajniran da prevaziđe uočene nedostatke u prethodnoj verziji. *OAuth 2.0* podržava klijente koji nisu na vebu. Tok autentikacije je isti kao i kod *OAuth 1.0*. Međutim kod *OAuth 2.0* se postiže bolje upravljanje zahtevima i autorizacijom korisnika. *GitHub*-ova *OAuth* implementacija podržava obe verzije.

Autentikacija korisnika na veb aplikaciji sastoji se iz tri koraka:

- Korisnik bira akciju da prosledi svoj *GitHub* identitet aplikaciji.
- Korisnik se preusmerava na veb stranicu od strane *GitHub*-a. Ako korisnik prihvati zahtev, *GitHub* ga preusmerava nazad sa privremenim kodom i stanjem koje je prosleđeno u prethodnom koraku. Vreme trajanja koda je 10 minuta. U slučaju da se stanja ne poklapaju, proces se obustavlja iz sigurnosnih razloga.
- Aplikacija pristupa *API*-ju sa korisničkim pristupnim tokenom. Pristupni token omogućava pravljenje zahteva prema *API*-ju u korisnikovo ime.

Docker

Pre korišćenja kontejnera, glavni način za izolovanje, organizaciju aplikacije i njenih zavisnosti je bio postavljanje svake aplikacije na zasebnu virtualnu mašinu. Takve mašine su pokretale više aplikacija na istom fizičkom hardveru i takav proces se naziva virtualizacija.

Virtualizacija je imala nekoliko nedostataka:

- Virtualne mašine su bile glomazne;
- Pokretanje više virtualnih mašina uticalo je na performanse i
- Sam proces pokretanja je predugo trajao.

Pomenuti nedostaci doveli su do nastanka tehnike korišćenja kontejnera (kontejnerizacije).

Kontejnerizacija je tip virtualizacije koji dovodi virtualizaciju na nivo operativnog sistema. Kao što virtualizacija apstrahuje hardver, tako i kontejnerizacija apstrahuje operativni sistem.

Neke od prednosti kontejnerizacije su sledeće:

- Kontejneri nemaju gostujući operativni sistem i koriste operativni sistem domaćina. Dele relevantne biblioteke i resurse onda kada je to potrebno.
- Procesiranje i izvršavanje aplikacije je veoma brzo jer se kompajlirana aplikacija i biblioteke kontejnera izvršavaju na kernelu domaćina.
- Pokretanje kontejnera traje samo delić sekunde. Takođe, kontejneri su lakši i brži od virtualnih mašina.

Docker je platforma koja aplikaciju i sve njene zavisnosti pakuje u formi kontejnera. Ovakav aspekt obezbeđuje da aplikacija radi na svim okruženjima.

Svaka aplikacija se pokreće na zasebnom kontejneru i ima svoj skup zavisnosti i biblioteka. Zbog toga možemo biti sigurni da svaka aplikacija radi nezavisno od ostalih aplikacija, dajući programerima sigurnost da grade aplikacije čije se zavisnosti neće međusobno sukobljavati.

Docker fajl je tekstualni dokument koji sadrži komande koje je potrebno pokrenuti u komandnoj liniji kako bi se sastavila *Docker* slika. *Docker* može izgraditi sliku automatski na osnovu pročitanih instrukcija iz *Docker* fajla.

Docker sliku možemo uporediti sa šablonom koji se koristi za pravljenje *Docker* kontejnera. To su šabloni koji se ne mogu menjati i predstavljaju gradivni element kontejnera. *Docker* slike se čuvaju u *Docker* registrima.

Docker kontejner je pokrenuta instanca *Docker* slike. Sadrži sve što je potrebno da bi se pokrenula aplikacija. To je u osnovi spremna aplikacija napravljena iz *Docker* slike, što ujedno predstavlja i krajnji proizvod *Docker*-a [14].

Docker je trenutno najpopularnija implementacija kontejnera.

Kubernetes

Razvojem mikroservisnih arhitektura dovelo je do povećane upotrebe kontejner tehnologija jer kontejneri predstavljaju savršeno rešenje za male, nezavisne aplikacije, kao što su mikroservisi. To je dalje dovelo do toga da se aplikacije sada nalaze u velikom broju kontejnera. Upravljanje tim kontejnerima, kroz različita oruženja, uz pomoć skripti ili alata koji su nastali u okviru kompanije koja proizvodi aplikaciju postaje ubrzo jako kompleksno.

Obzirom da su mikroservisi odvojeni jedni od drugih, mogu se razvijati, instalirati, ažurirati i skalirati svaki ponaosob. Ovakva osobina omogućava češće promene na komponentama. S druge strane, povećanjem broja komponenti koje

treba instalirati postaje sve teže konfigurisati, upravljati i očuvati ceo sistem u radnom stanju. Pored navedenog, mnogo je teže shvatiti kako i gde postaviti ove komponente kako bi se postigla veća iskorišćenost resursa, a samim tim i smanjiti cenu potrebnog hardvera. Odatle postoji potreba za automatizacijom, koja uključuje automatsku konfiguraciju, nadzor i rešavanje problema. Iz ovih razloga je razvijen *Kubernetes*.

Porastom kompleksnosti sistema raste i broj alata koji se koriste za upravljanje. Vremenom postaje nemoguće da tim programera isprati sve tehnologije i alate. Odatle prirodno nastaje potreba za izdvojenim timom tehničkih lica koja poznaju alate i tehnologije za upravljanje sistemom. Ovaj tim se naziva operacioni tim i zadužen je za održavanje sistema i raspoređivanje novih verzija. Tim programera je u tom slučaju rasterećen jer mu je fokus samo na rešavanju poslovne logike.

Kubernetes omogućava programerima da sami instaliraju svoju aplikaciju, bez dodatne pomoći operacionog tima. Ali s druge strane, nemaju samo programeri benefit. Ovaj alat takođe pomaže operacionom timu tako što automatski nadgleda, i u slučaju greške pokreće nove instance aplikacija. To znači da se fokus operacionog tima preusmerava sa nadgledanja pojedinačnih aplikacija na nadgledanje i upravljanje infrastrukture i *Kubernetes* alata, dok se *Kubernetes* stara o samim aplikacijama.

Kubernetes apstrahuje hardversku infrastrukturu i pruža privid da je ceo centar za podatke jedan veliki resurs. To omogućava velikim kompanijama koje pružaju usluge računarstva u oblaku da ponude programerima jednostavnu platformu za pokretanje raznih tipova aplikacija, a da pritom administratori sistema tih kompanija ne znaju koje su aplikacije pokrenute na njihovom hardveru. Kako velike kompanije sve više prihvataju *Kubernetes* model kao jedan od boljih načina za pokretanje aplikacija, tako *Kubernetes* postaje standardan model za računarstvo u oblaku [13].

Kubernetes je softver otvorenog koda koji služi za orkestraciju kontejnera, a razvijen je od strane *Google*-a. Pomaže pri upravljanju aplikacijama koje su razvijene u velikom broju kontejnera. Može da se primeni na različita okruženja za isporuku, kao što su fizički hardver, virtualne mašine ili oblak.

Prednosti korišćenja *Kubernetes* su mnogobrojne. Visoka dostupnost aplikacije je jedna od tih prednosti. To zači da će korisnici moći (skoro) uvek da pristupe aplikaciji. Druga prednost je horizontalna skalabilnost aplikacije, odnosno po potrebi se lako dodaju novi čvorovi sistemu. Treća prednost koja dolazi uz korišćenje

Kubernetes je oporavak od otkazivanja, što praktično znači da, ako je došlo do greške u infrastrukturi, *Kubernetes* ima mehanizme da bekapuje podatke i da nastavi sa radom od poslednjeg sačuvanog stanja.

Arhitektura Kubernetes-a

Arhitektura *Kubernetes*-a je zasnovana na klasterima. Klaster predstavlja skup čvorova. Svaki klaster sadrži jedan glavni čvor koji je povezan sa jednim ili više radnih čvora. Radni čvorovi na sebi imaju takozvani „kublet" proces koji je pokrenut na njima. Ovaj proces služi da klaster može da komunicira sa radnim čvorovima i izvšava određene poslove na njima, kao što je pokretanje procesa za aplikacije. Svaki radni čvor ima različite *Docker* kontejnere, različitih aplikacija, koje su isporučene na njemu. Raspored kontejnera u radnim čvorovima zavisi od opterećenja sistema. Ako je opterećenje za određeni servis veće, *Kubernetes* može da pokrene veći broj kontejnera za taj servis.

Dok se aplikacija izvršava na radnim čvorovima, glavni čvor služi da se na njemu pokrenu bitni procesi bez kojih *Kubernetes* ne može da radi. Jedan od tih procesa je *API Server*, koji služi za komunikaciju sa različitim *Kubernetes* klijentima, kao što je korisnički interfejs ili alat u komandnoj liniji. Drugi proces koji se nalazi na glavnom čvoru je *Controller manager* koji prati šta se dešava u klasteru, da li nešto treba da se popravi, ili da otkrije da je došlo do greške u kontejneru. Dalje, na glavnom čvoru se nalazi proces pod imenom *Scheduler* koji je zadužen za podizanje kontejnera na različitim čvorovima u odnosu na opterećenje i dostupne resurse na svakom čvoru. *Scheduler* je proces koji odlučuje na kom čvoru će biti podignut koji kontejner. Još jedna bitna komponenta na glavnom čvoru je *etcd*, skladište „ključ – vrednost", koje služi da čuva stanje *Kubernetes* klastera. On čuva sve konfiguracije za svaki čvor, aplikaciju, ali i statusne podatke o svakom kontejneru. Poslednja, ali nimalo manje važna komponenta u arhitekturi *Kubernetes*-a, je virtualna mreža. Preko nje čvorovi mogu međusobno da komuniciraju.

Može se primetiti da će glavni resursi biti raspoređeni na radne čvorove, jer oni služe za pokretanje aplikacije. Za glavni čvor nije potrebno toliko resursa, jer se na njemu pokreću ne toliko zahtevni procesi koji služe samo za rad *Kubernetes*-a. Ako nekim slučajem dođe do otkazivanja radnog čvora, glavni čvor će se pobrinuti da se podigne novi radni čvor sa istom konfiguracijom. S druge strane, ako dođe do otkazivanja glavnog čvora, gubimo konekciju sa svim ostalim čvorovima. Iz tog razloga, u produkcionom okruženju se uvek drži barem 2 pokrenuta glavna čvora,

tako da, ako jedan padne, drugi preuzima posao na sebe.

Osnovni koncepti Kubernetes-a

Čaura (eng. *Pod*) u *Kubernetes*-u predstavlja najmanju jedinicu koja može da se konfiguriše i sa kojom može da se ostvari interakcija. Ona praktično predstavlja omotač oko kontejnera. U okviru jednog radnog čvora može se naći više čaura, a u jednoj čauri se može naći više kontejnera. Uobičajeno je da se jedan kontejner nalazi u jednoj čauri, ali postoje slučajevi kada jedan kontejner zahteva pomoćne kontejnere i tada se može naći više njih u jednoj čauri. To zači da će jedna aplikacija, odnosno jedan servis, biti u jednoj čauri. Čaura predstavlja apstrakciju za upravljanje nad kontejnerom koji se pokreće unutar čaure. Na primer, ako se kontejner ugasi, čaura će ga ponovo podići za nas.

Čaure predstavljaju privremene komponente, što znači da često mogu da otkažu iz različitih razloga. Na primer, kada je potrebno isporučiti novu verziju aplikacije, prvo će se napraviti nove čaure sa novom verzijom, a potom će se stare ukloniti. Pomenuta virtualna mreža koja se nalazi nad celim klasterom će svakoj čauri dodeliti po jednu *IP* adresu. To znači da je svaka čaura zaseban server sa svojom *IP* adresom preko koje međusobno komuniciraju.

S obzirom na to da čaure predstavljaju privremene komponente, i da se pravljenjem nove čaure dodeljuje nova *IP* adresa, prirodno se uvodi pojam *Servisa*. Servis predstavlja zamenu za *IP* adrese, tako da umesto da čaure komuniciraju međusobno preko *IP* adrese, one mogu da komuniciraju preko Servisa koji dalje prosleđuju komunikaciju čauri. Tako da, ako se čaura ponovo napravi, ostali će znati da komuniciraju s njom kada ponovo bude dostupna. Pored zamene *IP* adrese, Servisi služe i kao balanseri opterećenja.

Može se reći da servisi predstavljaju apstrakciju čauri, kao i da čaure predstavljaju instance servisa.

Konfiguracija Kubernetes-a

Konfigurisanje sistema *Kubernetes* se odvija deklarativno, uz pomoć konfiguracionog fajla u formatu *YAML*. U njemu deklarišemo koji kontejner treba da se podigne, od koje *Docker* slike treba da se napravi, koliko čaura treba da bude u svakom trenutku. Pomenuto skladište *etcd*, pored stanja klastera, može da sačuva i dodatne promenljive. U konfiguracionom fajlu za čaure se te promenljive

mogu navesti, a potom će, prilikom izvršavanja, one biti dostupne kroz sistemske promenljive kontejnera. Na ovaj način se kontejneri mogu lako konfigurisati bez prekida rada. *Kubernetes* se stara da ovi zahtevi budu ispoštovani, i za to je zadužen *Controller Manager*, koji proverava konfiguraciju i upravlja ostalim procesima.

Kontinualna integracija, isporuka i raspoređivanje

CI/CD je metod za često dostavljanje aplikacija korisnicima kroz predstavljanje automatizacije u fazama razvoja aplikacije. Glavni koncept *CI/CD* su kontinualna integracija (eng. *Continuous Integration*), kontinualno dostavljanje (eng. *Continuous Delivery*) i kontinualno raspoređivanje (eng. *Continuous Deployment*).

CI/CD uvodi automatizaciju i kontinualno praćenje kroz životni ciklus aplikacija, od integracije i faze testiranja do dostavljanja i raspoređivanja. Zajedno, ove povezane prakse često se nazivaju „*CI/CD* tok" (eng. *CI/CD pipeline*), prikazan na slici 1.4, a podržan je i od strane razvojnih i od strane operacionih timova.

Slika 1.4: *CI/CD* tok

Od slučaja do slučaja, na šta se termini konkretno odnose zavisi od toga koliko je automatizacije ugrađeno u *CI/CD* tok. Mnoga preduzeća započinju automatizaciju dodavanjem kontinualne integracije, nakon čega uvode automatizaciju dostavljanja i raspoređivanja.

Kontinualna integracija

U razvoju modernih aplikacija, cilj je da više programera istovremeno radi na različitim delovima aplikacije. Međutim, ukoliko je organizacija postavljena tako da se spajanje koda sa svih grana vrši u jednom danu, takav posao može biti monoton, manuelan i dugotrajan. To se dešava u slučajevima kada programer vrši izmene na aplikaciji i na taj način povećava šansu za nastajanje konflikta sa izmenama koje istovremeno prave drugi programeri.

Kontinualna integracija (*CI*) pomaže programerima da spoje izmene na kodu na deljenu granu češće, čak i na dnevnom nivou. Nakon spajanja izmenjenih delova koda, izmene se validiraju tako što se automatski gradi aplikacija i pokreće se više nivoa automatskog testiranja. To znači da se testira sve, od klasa i funkcija do različitih modula koji su deo aplikacije. Ako automatski testovi pronađu konflikt između novog i postojećeg koda, *CI* olakšava rešavanje konflikata jer pomaže u pronalaženju uzroka konflikta. Uspešan *CI* označava da su nove izmene koda na aplikaciji regularno izgrađene, testirane i pripojene deljenom repozitorijumu.

Kontinualno dostavljanje

Nakon automatskog građenja aplikacije i testiranja u *CI*, kontinualno dostavljanje automatizuje puštanje prethodno validiranog koda u repozitorijum. Kako bismo imali efektivan proces kontinualne dostave, važno je da imamo već ugrađen *CI* u protoku. Cilj kontinualnog dostavljanja jeste postojanje baze koda koja je uvek spremna za raspoređivanje na produkciono okruženje.

U kontinualnom dostavljanju, svaka faza, počevši od spajanja izmenjenog koda do dostavljanja verzija spremnih za produkciju, podrazumeva automatsko testiranje i automatizaciju raspoređivanja koda. Može se smatrati odgovorom na slabu preglednost i komunikaciju između tima programera i poslovnog tima. Iz tog razloga, svrha kontinualnog dostavljanja jeste da omogući raspoređivanje novog koda brzo, jednostavno i uz minimalne napore.

Kontinualno raspoređivanje

Poslednja faza *CI/CD* toka je kontinualno raspoređivanje. Kao dodatak kontinualnom dostavljanju, koji automatizuje isporuku verzija spremnih za produkciju, kontinualno raspoređivanje automatizuje puštanje aplikacije u produkciju. U velikoj meri se oslanja na dobro osmišljeno automatsko testiranje.

U praksi, kontinualno raspoređivanje znači da izmene na aplikaciji mogu biti na produkciji za samo nekoliko minuta (pod pretpostavkom da su automatski testovi uspešno završeni).

Kontinualno dostavljanje i kontinualno raspoređivanje su povezani koncepti i mogu se koristiti naizmenično. I jedan i drugi tiču se automatizacije određenih faza u distribuiranju aplikacije, međutim, nekada se koriste i odvojeno u cilju prikaza količine automatizacije koja se odvija.

Sve povezane prakse *CI/CD* čine raspoređivanje aplikacije manje rizičnim, stoga je lakše pustiti promene u aplikaciji u delovima, pre nego odjednom [15].

GitHub Akcije

Za verzionisanje koda se može koristiti *GitHub*. Pored verzionisanja, on pruža i alate za automatizaciju kontinualne integracije, isporuke i raspoređivanja, kroz alat koji su nazvali *GitHub Akcije* (eng. *GitHub Actions*). *GitHub Akcije* su zasnovane na događajima, što znači da mogu da pokrenu niz komandi koje će se desiti posle navedenog događaja. Na primer, svaki put kada neko napravi *zahtev za promenu* (eng. *Pull Request*) u repozitorijumu, može se automatski pokrenuti komanda koja pokreće testove.

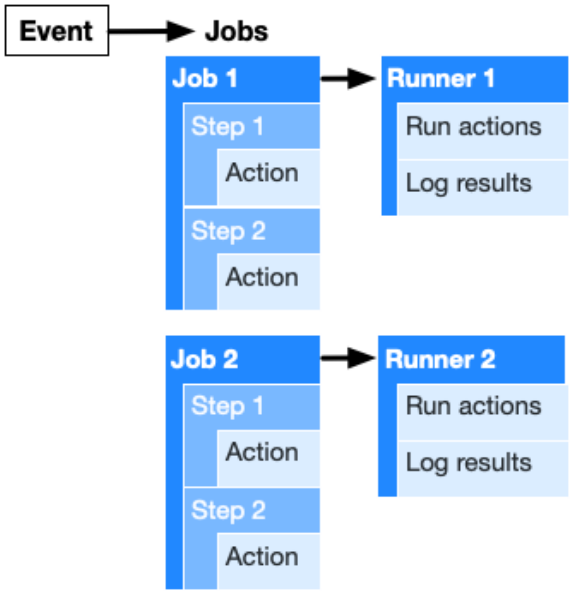
GitHub Akcije koriste fajlovi u formatu *YAML* kako bi se definisale komponente toka rada. Ovi fajlovi se čuvaju u repozitorijumu, u folderu pod nazivom `.github/workflows`.

Komponente koje postoje u *GitHub Akcijama* su:

- Izvršilac (eng. *Runner*);
- Tok rada (eng. *Workflow*);
- Događaj (eng. *Event*);
- Posao (eng. *Job*);
- Korak (eng. *Step*) i
- Akcija (eng. *Action*).

Izvršilac Izvršilac je mašina nad kojom se *GitHub Akcije* izvršavaju. Izvršilac osluškuje pokrenute poslove, pokreće jedan posao za drugim, i šalje izveštaje *GitHub*-u o napretku, logovima i rezultatima. Ove mašine mogu da budu bazirane na *Linux*, *Windows* i *macOS* operativnim sistemima, a svaki posao unutar toka rada se pokreće iz novog virtualnog okruženja [16].

Tok rada Tok rada je automatizovana procedura koja se definiše u repozitorijumu. Tokovi rada su sastavljeni od jednog ili više poslova i mogu se pokrenuti na određeni događaj ili biti zakazani u određeno vreme. Tok rada se može koristiti da se aplikacija izgradi, testira, spakuje, isporuči ili rasporedi na različita okruženja.



Slika 1.5: Prikaz interakcija komponenata *GitHub Akcija*

Događaji Događaj je specifična aktivnost koja pokreće tok rada. Na primer, aktivnost može nastati kad neko napravi zahtev za promenu. Aktivnost ne mora da bude u okviru *GitHub*-a. Ona može da predstavlja i poziv od strane nekog eksternog sistema.

Posao Posao predstavlja skup koraka koji treba da se izvrše u okviru istog izvršioca. Tok rada sa više poslova će pokrenuti ove poslove u paraleli. Naravno, tok rada se može podesiti tako da izvršava poslove sekvencijalno. Na primer, tok rada može imati dva sekvencijalna posla koji izgrađuju i testiraju kod, gde je posao za testiranje zavisan od toga da li kod uopšte može da se izgradi. Ako posao za izgradnju ne uspe, posao za testiranje se neće ni pokretati.

Korak Korak predstavlja jedan zadatak koji može pokrenuti komandu unutar posla. Korak može biti ili akcija ili komandni skript. Svaki korak unutar posla se izvršava nad istim izvršiocem, što dozvoljava akcijama da dele podatke između sebe.

Akcija Akcije predstavljaju samostalne komande koje se mogu kombinovati u korake kako bi sačinile jedan posao. Akcije su najmanje portabilne jedinice građe

toka rada. Korisnici mogu sami da naprave svoje akcije ili da koriste akcije koje su napravljene od strane *GitHub* zajednice.

Glava 2

Funkcionalni zahtevi aplikacije Interpres

Glavni zadatak sistema je da omogući lako upravljanje prevodima. To podrazumeva da prevodilac, koji nije tehničko lice, može sa lakoćom da obavlja svoj posao a da ne zna interne procese rada programerskog tima. S druge strane, programerima treba omogućiti što veći stepen automatizacije. Jednostavno, treba razdvojiti što je više moguće poslove prevodilaca i programera. Tu se mogu uočiti dve celine sistema, jedna celina koja podržava proces rada prevodilaca, i druga celina koja podržava proces rada programera.

2.1 Uređivač prevoda

Uređivač prevoda predstavlja korisnički interfejs za uređivanje i obradu prevoda, a koristi ga prevodilac. Kako se svaki prevod čuva u obliku „ključ – vrednost", pored instance prevoda, potrebno je prikazati i sam ključ kako bi se mogla jednoznačno prikazati svaka instanca.

Ključ se može posmatrati i kao referenca na jednu grupu instanci prevoda koje predstavljaju istu nisku u različitim jezicima. Osim toga, on nema neku dodatnu vrednost za prevodioca. Njima veću vrednost predstavlja instanca prevoda u jeziku koji je zajednički za prevodioca i programera. Na primer, kada programer želi da dobije prevode za srpski jezik, on će prevodiocu isporučiti instance prevoda na engleskom a prevodilac će uz odgovarajuće ključeve i instance prevoda na engleskom jeziku dodati prevode na srpskom jeziku. To znači da je na korisničkom interfejsu potrebno prikazati sve instance prevoda za određeni ključ.

Biblioteke za internacionalizaciju uglavnom pružaju nekakvu vrstu grupisanja ključeva. Ako se prevodi čuvaju u fajlovima, onda se jedan fajl može smatrati jednom grupom prevoda. Grupisanje je korisno jer se onda prevodi za određeni deo aplikacije obrađuju u istom kontekstu. Sa druge strane, ako se prevodi u aplikaciju dovlače preko mreže, inicijalno učitavanje aplikacije se može ubrzati jer nije potrebno učitati sve prevode, već se mogu učitati po potrebi. Na korisničkom interfejsu treba prikazati te grupe i omogućiti laku navigaciju kroz grupe.

Pored grupisanja prevoda, sistem treba da pruža mogućnost pravljenja više projekata. Na primer, prevodilac može uređivati prevode za više aplikacija. Svaki projekat može sadržati različite skupove jezika na koje treba prevesti aplikaciju, ali i različita podešavanja za uvoz, odnosno izvoz prevoda.

2.2 Uvoz i izvoz prevoda

Proces uvoza i izvoza prevoda iz sistema treba da bude što je više moguće automatizovan. Zapravo, integracija sa sistemom za kontrolu verzija, kao što je *GitHub*, bi pružala najudobniji rad. Integracija bi radila i za uvoz prevoda, ali i za izvoz.

Prevodilac bi mogao preko dugmeta za sinhronizaciju da uveze nove promene koje su se pojavile u repozitorijumu. Promene koje mogu da se pojave u repozitorijumu su dodavanje novih ključeva ili brisanje postojećih. Za te potrebe, promene će se uvek preuzimati sa glavne grane iz sistema za kontrolu verzija.

Izvoz prevoda bi isto mogao da se uradi jednim klikom na dugme. U tom slučaju sistem bi napravio novi zahtev za promenu (eng. *Pull Request*) na *GitHub*-u, koji bi zahtevao odobrenje od strane programera da se spoji sa glavnom granom koda.

Ovakvim razdvajanjem dva tima mogu raditi nesmetano. Prevodioci će uvek imati najsvežiju verziju prevoda, odnosno ključeva za niske koje treba prevesti i neće zavisiti od procesa rada programera. Sa druge strane, programeri će na kraju dobijati zahteve za promene i mogu sami da odluče u kom trenutku žele da primene nove prevode.

2.3 Aplikacija u oblaku

Aplikacije koje je potrebno prevesti na što veći broj jezika zahtevaju i timove prevodilaca. Kolaboraciju između više prevodilaca treba omogućiti preko mreže radi što veće udobnosti. Svaki prevodilac treba da ima mogućnost da pristupa aplikaciji, odnosno prevodima, sa svog računara. Kako bi se omogućila kolaboracija između više korisnika, potrebno je podatke skladištiti u oblaku. Veb aplikacije u oblaku pružaju najveću udobnost u tom smislu. Dovoljno je otvoriti aplikaciju u veb pregledaču i korisnik je spreman za rad.

Jedna od važnih osobina aplikacija na vebu je dostupnost. Dostupnost aplikacije predstavlja stepen u kojem je aplikacija operativna, funkcionalna i upotrebljiva za ispunjavanje zahteva korisnika. Računarstvo u oblaku pruža rešenja da aplikacija bude dostupna u skoro svakom trenutku.

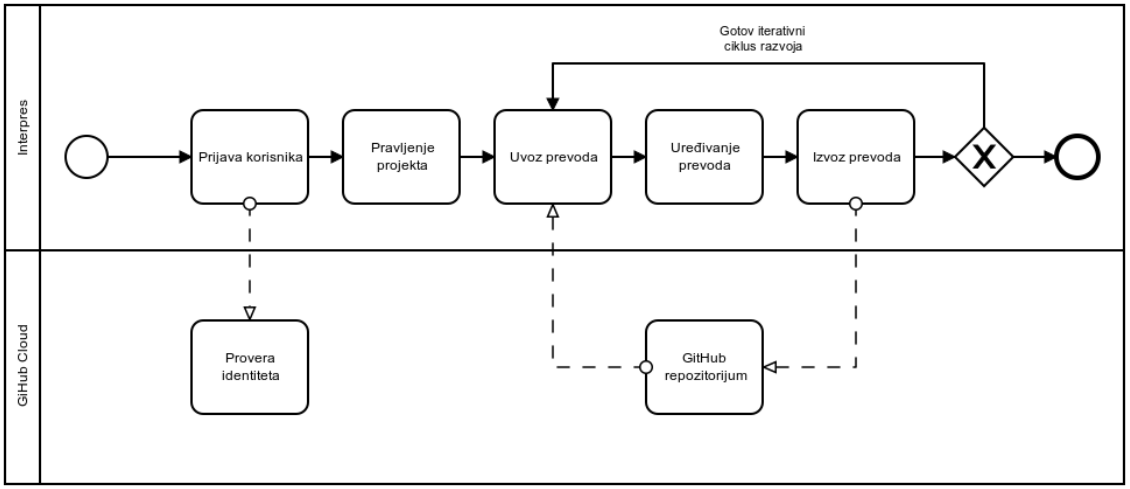
Aplikacije u oblaku su podrazumevano javno dostupne preko veb adrese. Kako bi se podaci zaštitili, potrebno je implementirati i sistem za autentikaciju. S obzirom da se već radi integracija sa *GitHub*-om, radi uvoza, odnosno izvoza prevoda, *GitHub* se može iskoristiti kao provajder identiteta (eng. *Identity Provider*). To znači da je potrebno da prevodici naprave svoje naloge na *GitHub*-u s kojima će se kasnije prijavljivati na sistem.

Glava 3

Slučajevi upotrebe aplikacije Interpres

Razvoj softvera iterativnom metodom diktira i uređivanje prevoda za aplikaciju. Na kraju svakog ciklusa, programeri obaveste prevodioce da su završili kako bi prevodioci mogli da započnu svoj ciklus prevođenja.

U ovom poglavlju će biti opisani slučajevi upotrebe. Na dijagramu 3.1 je prikazana šira slika svih slučajeva upotrebe.



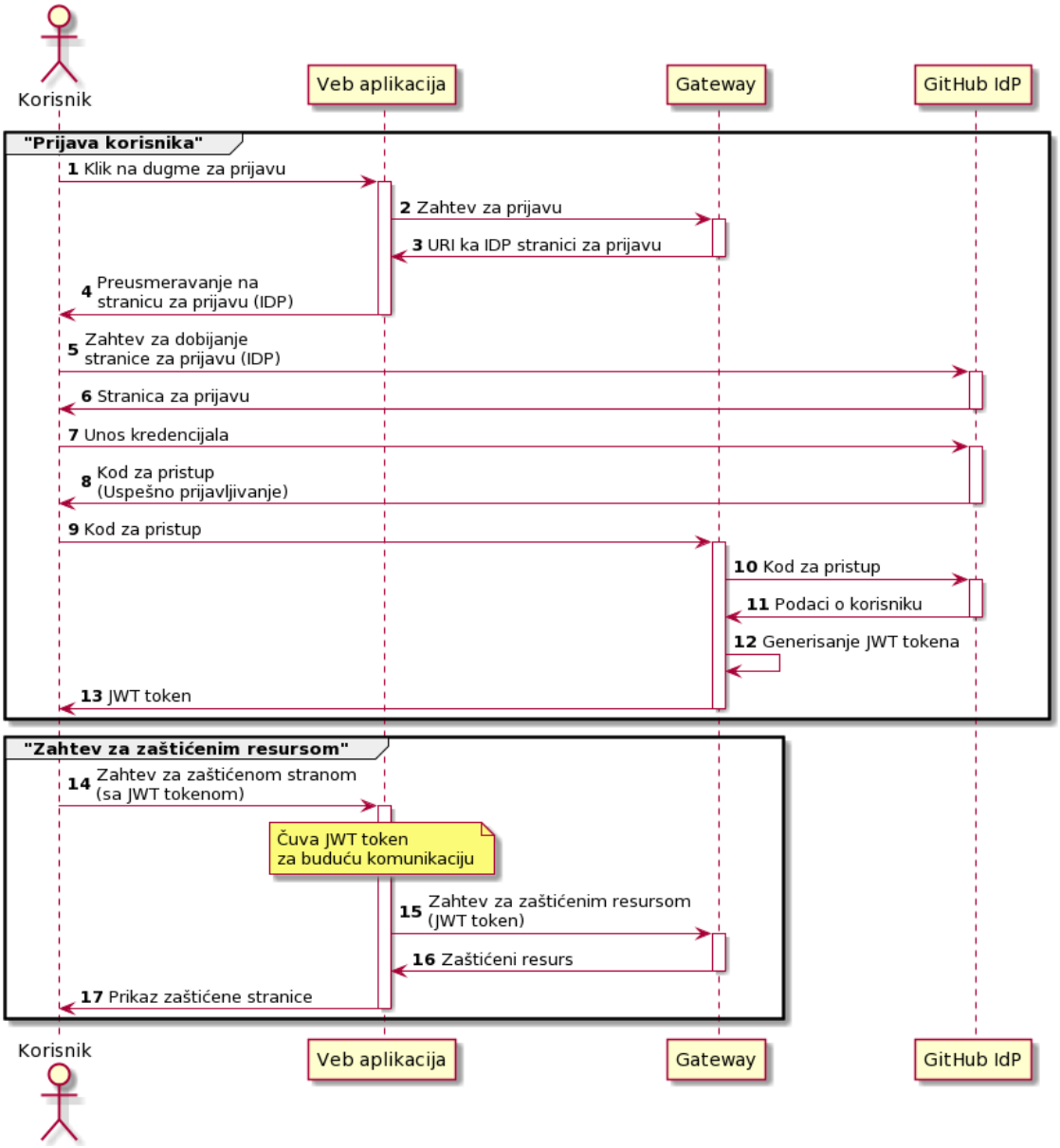
Slika 3.1: BPMN dijagram slučajeva upotrebe

3.1 Prijava korisnika uz pomoć GitHub IdP

Kratak opis: Korisnik se prijavljuje sa sopstvenim kredencijalima

Učesnici: Korisnik
Postuslov: Korisnik je prijavljen
Glavni tok:

1. Korisnik klikne na dugme za prijavu
2. Sistem šalje zahtev prema *Gateway*-u za prijavu
3. *Gateway* vraća preusmeravanje (302 *Redirect*) na stranicu za prijavu provajdera identiteta (u ovom slučaju *GitHub*)
4. Šalje se zahtev za dobijanje stranice za prijavu provajdera identiteta
5. *GitHub* šalje stranicu za prijavu korisnika
6. Korisnik unosi svoje kredencijale
7. Korisnik šalje zahtev za prijavu klikom na dugme
8. *GitHub* dobija podatke za prijavu i nakon uspešnog prijavljivanja šalje kod za pristup veb aplikaciji
9. Veb aplikacija prosleđuje *Gateway*-u kod za pristup
10. *Gateway*, uz zahtev za dobijanje podataka o korisniku, prosleđuje kod za pristup *GitHub*-u
11. *GitHub Gateway*-u vraća podatke o korisniku
12. Na osnovu dobijenih podataka *Gateway* generiše *JWT* token
13. *Gateway* šalje *JWT* token veb aplikaciji
14. Veb aplikacija omogućava prikaz zaštićene stranice korisniku i čuva *JWT* token za buduće komunikacije



Slika 3.2: Dijagram sekvence – prijava korisnika

3.2 Pravljenje projekta

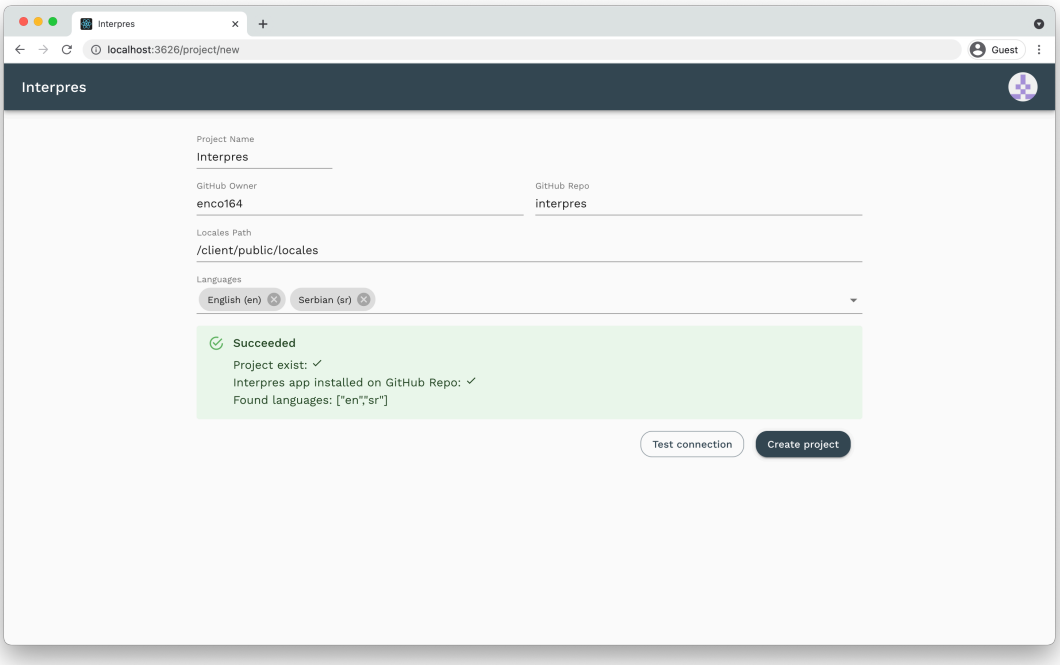
Kratak opis: Prevodilac ima mogućnost da napravi novi projekat

Učesnici: Prevodilac

Postuslov: Projekat je napravljen

Glavni tok:

1. Sistem prikazuje opciju za pravljenje novog projekta
2. Prevodilac bira akciju za pravljenje novog projekta
3. Sistem prikazuje formu sa tekstualnim poljima za unos podataka o projektu
4. Prevodilac unosi
   - naziv projekta
   - vlasnika *GitHub* repozitorijuma
   - *GitHub* repozitorijum
   - putanju na kojoj će se čuvati fajlovi sa prevodima i
   - skup jezika za koje su potrebni prevodi.
5. Prevodilac klikne na dugme za testiranje konekcije
   - Ukoliko su podešavanja pravilno unešena i *GitHub* aplikacija uspešno instalirana u repozitorijum, sistem prikazuje uspešnu poruku zajedno sa jezicima koje je pronašao u repozitorijumu
   - Ukoliko podešavanja nisu pravilno unešena ili *GitHub* aplikacija nije uspešno instalirana u repozitorijum, sistem prikazuje poruku sa greškom.
6. Prevodilac klikne na dugme za pravljenje projekta
7. Sistem čuva projekat
8. Sistem prevodiocu vraća stranicu sa podacima o projektu



Slika 3.3: Korisnički interfejs – pravljenje projekta

3.3 Uvoz prevoda

Kratak opis: Prevodilac želi da uveze prevode sa *GitHub*-a
Učesnici: Prevodilac
Postuslov: Prevodilac se nalazi na projektu u koji želi da uveze prevode
Glavni tok:

1. Sistem prikazuje opcije za uvoz i izvoz prevoda
2. Prevodilac bira akciju za uvoz prevoda
3. Sistem prikazuje modal sa porukom da će se uvozom prevoda sve izmene koje nisu izvezene izgubiti
4. Prevodilac klikne na dugme „*OK*” ukoliko nema prevoda koji nisu sačuvani
5. Sistem uvozi prevode
6. Sistem zatvara modal

Alternativni tok: Ukoliko prevodilac u koraku 4. klikne na dugme „*Cancel*", sistem zatvara modal i prevodi neće biti uvezeni.

3.4 Izmena prevoda

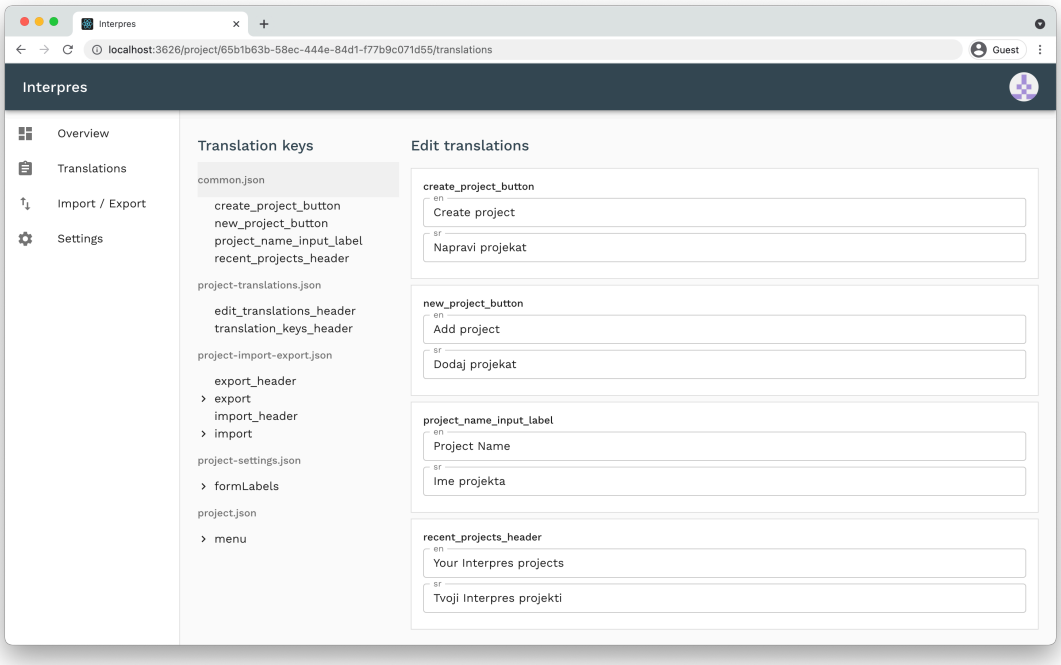
Kratak opis: Prevodilac vrši izmene kako bi dodao novi prevod ili izmenio postojeći

Učesnici: Prevodilac

Postuslov: Uspešno izvršena izmena prevoda

Glavni tok:

1. Sistem prikazuje listu prevoda sa odgovarajućim ključevima
2. Prevodilac klikne na željeni ključ za koji želi da unese ili izmeni prevod
3. Sistem vraća tekstualna polja za unos za onoliko jezika koliko je u tom trenutku dostupno za izabrani ključ
4. Prevodilac unosi ili menja jedan ili više prevoda
5. Prevod se automatski šalje na čuvanje u trenutku kada tekstualno polje izgubi fokus
6. Sistem čuva izmenjeni prevod



Slika 3.4: Korisnički interfejs uređivača prevoda

3.5 Izvoz prevoda

Kratak opis: Prevodilac želi da izveze prevode izabranog projekta i na taj način napravi zahtev za promenu na *GitHub*-u

Učesnici: Prevodilac

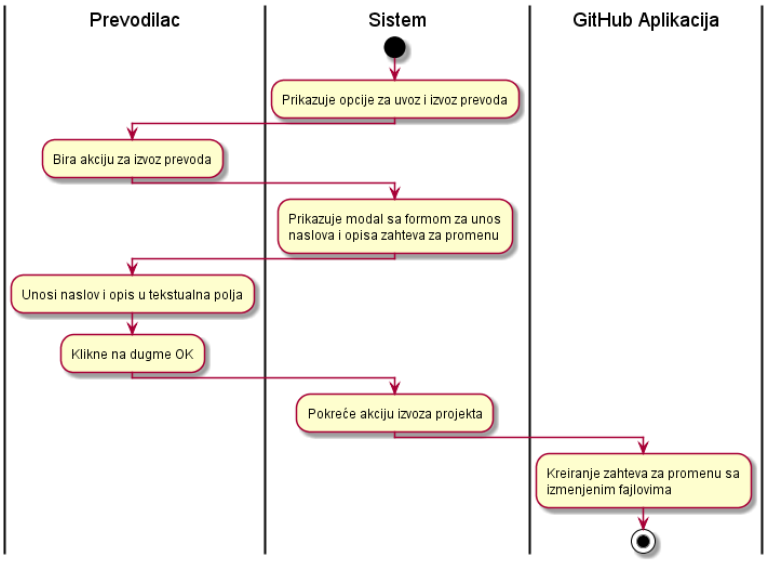
Preduslov: Prevodilac se nalazi na projektu koji želi da izveze

Postuslov: Uspešno izvezen sadržaj prevoda

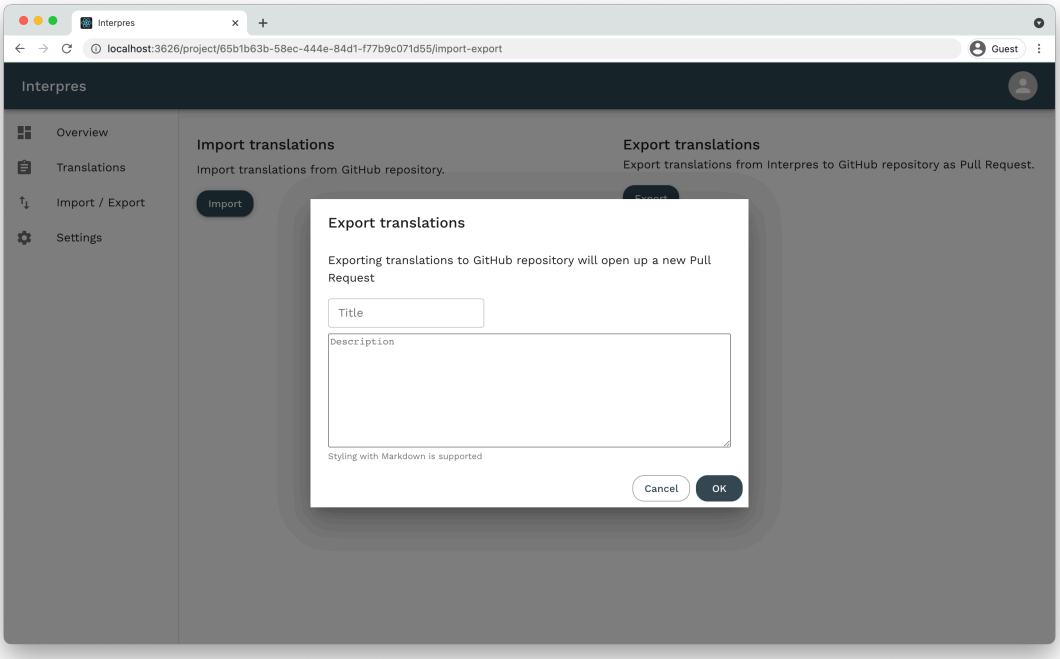
Glavni tok:

1. Sistem prikazuje opcije za uvoz i izvoz prevoda
2. Prevodilac bira akciju za izvoz prevoda
3. Sistem prikazuje modal sa formom za unos naslova i opisa zahteva za promenu
4. Prevodilac unosi naslov i opis u tekstualna polja
5. Korisnik klikne na dugme *OK*

6. Sistem pokreće akciju izvoza prevoda
7. Na *GitHub*-u se pravi zahtev za promenu sa izmenjenim fajlovima za prevode



Slika 3.5: Dijagram aktivnosti – izvoz prevoda



Slika 3.6: Korisnički interfejs za izvoz prevoda

Glava 4

Implementacija aplikacije Interpres

Implementacija prati opisanu arhitekturu iz prethodnih poglavlja. Korisnički interfejs je zasnovan na arhitekturi jedne stranice, uz pomoć okruženja *React*. Komunikacija između klijenta i servera je preko *REST API*-a. Server je implementiran u arhitekturi mikroservisa uz pomoć *NestJS*. Za bazu podataka je izabrana *PostgreSQL*. Za potrebe pravljenja zahteva za promenu na kodu, server komunicira preko *GitHub* aplikacije. Autentikacija je implementirana preko *OAuth* protokola, a za provajdera identiteta je izabran *GitHub IdP*. Kao alat za *CI/CD* se koriste *GitHub Akcije*.

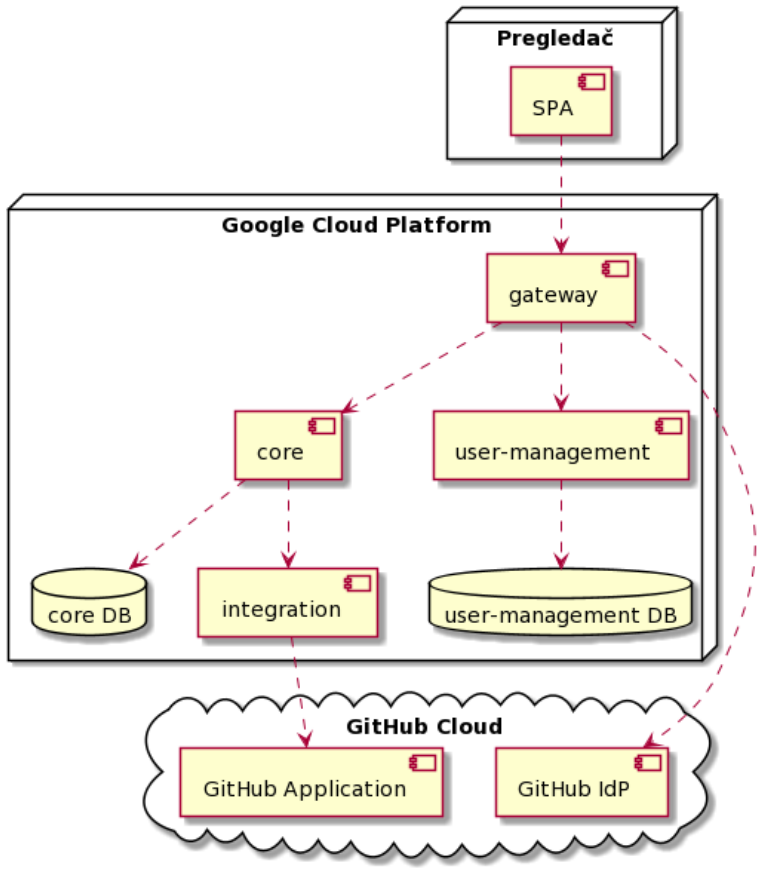
4.1 Komponente sistema

Na slici 4.1 su prikazane komponente sistema kao i njihove zavisnosti. Važno je napomenuti da, iako mikroservisi *core* i *user-management* koriste različite baze podataka, obe baze se zapravo nalaze u istom sistemu za upravljanje bazama podataka. Ova odluka je donešena radi boljeg iskorišćenja resursa.

U nastavku će biti opisana svaka komponenta, kao i njena zavisnost sa drugim komponentama.

SPA

Komponenta *SPA* predstavlja klijentsku stranu aplikacije, implementiranu u stilu arhitekture jedne stranicne. Izgrađena je uz pomoć razvojnog okruženja



Slika 4.1: Komponente sistema

React. Za stilizovanje korisničkog interfejsa korišćena je biblioteka *Material UI*.

Kada se pokrene izgradnja *React* projekta, artifakti koji se dobiju su *HTML*, *JavaScript* i *CSS* fajlovi. Da bi korisnik dobio fajlove, potreban je veb server, i u te svrhe je izabran *Nginx*.

Komponenta *SPA* dobija podatke sa servera. Komunikacija sa serverom je preko protokola *REST*, a ulazna tačka je mikroservis *gateway*.

Gateway

Komponenta *gateway* jedina „otvara kapiju" ka spoljnom svetu. Ona služi da prihvati zahteve sa klijentske strane i prosledi ih drugim mikroservisima. S obzirom da ona predstavlja svojevrsnu „kapiju", autentikacija je implementirana baš tu.

Provera pristupa se radi preko *GitHub IdP*. Ako se korisnik prvi put prijavljuje na sistem, njegovi podaci o imenu i prezimenu će biti poslati mikroservisu *user-management*.

Komunikacija sa ostalim mikroservisima (*core* i *user-management*) se odvija preko protokola *TCP* podržanog od strane *NestJS*.

User Management

Cilj ove komponente je da se brine o podacima korisnika. Pošto se podaci o korisnicima preuzimaju od eksternih servisa, u ovom slučaju od *GitHub*-a, kao dobra praksa se pokazalo da ne treba zavisiti od ključeva eksternih sistema. Ovde se konkretno misli na jedinstveni identifikator korisnika. Kao poboljšanje sistema, mogao bi da se implementira pristup sistemu preko nekog drugog provajdera identiteta. U tom slučaju može doći do kolizije ključeva, odnosno ne možemo da budemo sigurni da će različiti provajderi identiteta davati različite ključeve.

Pored brige o primarnim ključevima za korisnike, ovaj mikroservis služi i kao svojevrsna optimizacija. Naime, mnogo je brže kontaktirati mikroservis koji je u *Kubernetes* klasteru nego neki eksterni servis.

Core

Mikroservis *core* predstavlja srž aplikacije. On se bavi čuvanjem podataka o prevodima i o projektima. Tu se nalazi i poslovna logika za grupisanje i preslikavanje prevoda u strukturu koja je pogodna za klijentsku stranu aplikacije ili za mikroservis *integration*.

Integration

Ova komponenta ima funkciju integracije sa sistemom za verzionisanje koda. Na zahtev mikroservisa *core*, ona može preko *GitHub Aplikacije*, da dohvati prevode sa repozitorijuma i da napravi zahtev za promenu sa novim izmenama. Ako bi se u budućnosti implementirala integracija sa nekim drugim sistemom za verzionisanje koda, ovaj mikroservis je pravo mesto za to.

GitHub Aplikacija i GitHub IdP

Da bi se napravila integracija sa *GitHub*-om, za potrebe menjanja koda, potrebno je napraviti *GitHub Aplikaciju*. Aplikacija u stvari daje samo pristupne ključeve, a na programeru je dalje da implementira i postavi aplikaciju na neki

server. Implementacija aplikacije u *NodeJS* je preko *GitHub*-ove biblioteke, nazvane *octokit*. Preko ove biblioteke se dobija interfejs za sve akcije koje je moguće uraditi u repozitorijumu. Dve glavne akcije koje su implementirane su čitanje fajlova sa prevodima i pravljenje zahteva za promenu, i koriste se za uvoz i za izvoz prevoda.

GitHub IdP je eksterna komponenta i služi kao provajder identiteta. Slično kao i za *GitHub Aplikaciju*, potrebna je samo registracija za dobijanje pristupnih ključeva kojima se pristupa interfejsu provajdera. Za implementaciju pristupa korićena je biblioteka *Passport.js*, preporučena od strane *NestJS*, koja ima implementaciju protokola *OAuth2*, koji koristi *GitHub IdP*.

4.2 Klijent

Klijentski deo je izgrađen uz pomoć radnog okvira *React*, a početna organizacija koda uz pomoć *create-react-app*, preporučenog alata za generisanje *React* projekta od strane *Facebook*-a. Projekat je izgenerisan na jeziku *TypeScript*.

Organizacija projekata je podeljena po funkcionalnostima, i one su *auth* (odgovorna za autorizaciju), *projects* (odgovorna za podešavanje projekta), *import-export* (odgovorna za uvoz, odnosno izvoz prevoda) i *translations* (odgovorna za uređivanje prevoda).

Sve *React* komponente su pisane kao funkcijske komponente uz korišćenje kuka. Programeri koji po prvi put koriste *React* uglavnom nalaze da je ovako napisan kod nečitljiv jer deluje da je pomešana poslovna logika komponente sa prikazom korisničkog interfejsa. U stvari, mišljenje da je poslovna logika pomešana sa prikazom korisničkog interfejsa nastaje iz toga da u zvaničnoj dokumentaciji nije predložena arhitektura, već se samo opisuje tehnologija. Programeru je ostavljeno na razmišljanje kako da organizuje svoju aplikaciju. Prilikom razvoja projekta poštovan je princip da se sva poslovna logika nalazi u kuki, a da funkcijska komponenta koristi kuku i prikazuje sadržaj. Ovim principom se postiže veća čitljivost, razdvajaju se odgovornosti, lakše se testira automatskim testovima, a samim tim se i kôd lakše održava. Ovaj princip je prikazan na primeru koda 1.

Za upravljanje stanjem u aplikaciji korišćen je *redux*. *Redux* je biblioteka koja čuva stanje aplikacije u centralizovanom skladištu. Kada nekoj komponenti zatreba neki podatak iz stanja, ona može da mu pristupi sa sigurnošću da će podaci biti konzistentni i da neće biti particionisani. To znači da ako dve komponente

```
// kuka: usePrimer.ts
export const usePrimer = () => {
  const [count, setCount] = useState(0);

  return {
    count,
    handleClick: () => setCount(count + 1),
  };
}

// komponenta: primer.tsx
export const Primer = () => {
  const {count, handleClick} = usePrimer();

  return (
    <div>
      <p>Kliknuli ste {count} puta</p>
      <button onClick={handleClick}>
        Klikni me
      </button>
    </div>
  );
}
```

Primer koda 1: Princip pisanja funkcijskih komponenata sa kukama

zahtevaju isti podatak, one ga neće čuvati u svom stanju već će ga potraživati sa istog mesta. Komponente mogu promeniti centralizovano skladište, odnosno stanje aplikacije, prosleđivanjem podataka kroz objekat *Action*. Promenom stanja aplikacije se okida ponovno iscrtavanje komponenata koje su pretplaćene na deo stanja koji je promenjen. Tako, na primer, ako postoje dve komponente koje pristupaju istom podatku iz stanja, i jedna od njih ga promeni, druga će automatski biti obaveštena.

4.3 Server

Svaki mikroservis je napravljen kao zasebna aplikacija i generisan uz pomoć *NestJS* interfejsa za komandnu liniju. Ako neki mikroservis treba da zna za postojanje nekog drugog, informacija o lokaciji će biti prosleđenja kroz konfiguraciju, odnosno kroz sistemske promenljive. U konfiguraciji se pored toga čuvaju pristupni ključevi za *GitHub Aplikaciju*, *GiHub IdP* i lokacija i kredencijali za bazu podataka. Konfiguracija mikroservisa se učitava pri svakom podizanju.

Mikroservisi koji čuvaju stanje u bazi podataka su *core* i *user-management*. Ka-

ko *NestJS* koristi u pozadini *TypeORM*, iskorišćena je njegova funkcija migriranja baze podataka. Migracije služe za promenu sheme baze podataka u produkcionom okruženju. One osiguravaju da će prebacivanje na novu verziju sheme biti sigurno i da neće doći do gubitka podataka. Migracioni fajl sadrži klasu koja implementira `MigrationInterface`, a potrebno je implementirati dve metode: `up` i `down`. Prva služi da se baza migrira na višu verziju, a druga služi ako je u nekom slučaju potreban povratak na prethodnu. Unutar tih metoda treba napisati *SQL* naredbe za migracije. Pored ručnog pisanja migracija, *TypeORM* pruža i mogućnost generisanja migracionih klasa, jer može izračunati prethodni oblik modela, a odatle i razliku koju treba primeniti na bazu podataka kako bi podržala novi model.

Stil pisanja koda na serveru je reaktivan uz *rx.js* biblioteku koja implementira obrazac „posmatrač". Server je napisan u stilu mikroservisa, a komunikacija među servisima je sinhrona, odnosno po principu „zahtev – odgovor". To znači da dok neki servis čeka na odgovor drugog servisa, prvi ostaje blokiran dok ne dobije odgovor. Kako se programi napisani u *JavaScript*-u izvršavaju u jednoj niti ovo postaje veliki problem. Iz tog razloga *NestJS* pruža implementaciju *HTTP* klijenta koji prima odgovore asinhrono. On će poslati *HTTP* zahtev koji će biti razrešen kada stigne odgovor. Na taj način mikroservis koji je poslao *HTTP* zahtev može obavljati i neki drugi posao dok odgovor ne stigne. Korišćenjem obrasca posmatrač se ova asinhronost lakše apstrahuje. Potrebno je napraviti zahtev i onda se pretplatiti na odgovor. Dok se čeka odgovor, pogram je slobodan da obavlja neki drugi posao. Kada odgovor stigne biće pozvana funkcija za obradu pretplate i tok izvršavanja će se nastaviti. Pisanje reaktivnog koda je umnogome olakšano jer je i sam *NestJS* napisan uz pomoć *rx.js*. Na primeru koda 2 je prikazano korišćenje *rx.js* na metodi uvoza prevoda.

4.4 Automatizacija

Kontinualna integracija, isporučivanje i raspoređivanje je implementirano uz pomoć *GitHub*-a. Na glavnoj („*main*") grani je postavljeno pravilo zaštite, odnosno onemogućeno je direkto slanje koda na tu granu. Za svaku izmenu koda potrebno otvoriti zahtev za promenu. Pored toga, postavljeno je pravilo da, grana koju treba spojiti na glavnu granu, mora sadržati sve izmene koje se nalaze na glavnoj grani.

Na događaj otvaranja novog zahteva za izmenu koda, pokreće se niz *GitHub*

```
importProject({ projectId }: ImportRequest) {
  return this.projectRepository.getProjectById(projectId)
    .pipe(
      throwIfEmpty(
        () => new NotFoundException(`Project with id ${projectId} not found`)
      ),
      concatMap((project) =>
        forkJoin([
          this.integrationMicroserviceClientSend(
            ''import'',
            {
              owner: project.githubOwner,
              repo: project.githubRepo,
              translationsLoadPath: project.lngLoadPath,
            }
          ),
          this.translationRepository.remove(project.translations)
        ])
      ),
      concatMap(([dataFromGithub, ]) =>
        this.importParsedTranslations(dataFromGithub, projectId)
      )
    );
}
```

Primer koda 2: Metoda za uvoz prevoda napisana u reaktivnom stilu

akcija. Za klijentski deo i za svaki mikroservis se pokreće naredba izgradnje i naredba testiranja. Uz navedena ograničenja za spajanje grana, i sa ovom *GitHub* akcijom, osigurava se da će spojeni kôd biti istestiran pre spajanja. To znači da na glavnoj grani ne bi trebalo da se pojavi neka greška koja bi inače mogla da se otkrije testiranjem.

Kao i za zahtev za izmenu koda, kada se grana spoji u glavnu granu postoji niz *GitHub* akcija, za svaku komponentu po jedna akcija. Tu se izgrade *Docker* slike koje se kasnije isporučuju na *Docker* javni registar. Potom, kada su sve slike isporučene na *Docker* registar, pokreće se *GitHub* akcija koja započinje raspoređivanje na *Kubernetes*.

Na primeru koda 3 je prikazan tok kontinualne isporuke mikroservisa *gateway* na *Docker* javni registar.

4.5 Produkciono okruženje

Razne kompanije pružaju usluge iznajmljivanja računara u oblaku. Neki od poznatijih proizvoda su *Amazon Web Services*, *Microsoft Azure* i *Google Cloud Platform*. Sa druge strane, *Kubernetes* je nezavisan od platforme. Iako ga je moguće instalirati i pokrenuti na sopstvenom serveru, taj posao je mukotrpan pa je pametnije izabrati neki proizvod gde se *Kubernetes* može podići uz par klikova.

Kako je *Google* razvio *Kubernetes*, pretpostavka je da je na *Google Cloud Platform* uvek malo prednjači, pa je iz tog razloga on izabran za aplikaciju *Interpres*.

Podešavanje na *Google Cloud Platform* je jednostavno. Preko korisničkog interfejsa je potrebno napraviti klaster. Za klaster je potrebno izabrati tip virtualne mašine, kao i lokaciju servera na kojoj će ta virtualna mašina biti podignuta. Nadalje se sve može konfigurisati i preko komandne linije uz *Kubernetes*-ov alat `kubectl`. Preko ovog alata se izvršavaju komande za pravljenje čaura.

Potrebno je napomenuti da se korišćenje *Google Cloud Platform*-e, naravno, naplaćuje. U trenutku pisanja ovog rada *Google* za nove korisnike obezbeđuje besplatnih 300$, koji su dovoljni za testiranje. Svakako treba voditi računa prilikom podešavanja kako ne bi došlo do nepotrebnih troškova.

```yaml
name: Build & Publish API Gateway docker image

on:
  push:
    branches:
      - main

jobs:
  build-docker-and-publish:
    runs-on: ubuntu-latest
    environment: main

    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Use Node.js 14
        uses: actions/setup-node@v1
        with:
          node-version: '14'

      - name: Determine Docker Tag
        run: echo ''DOCKER_TAG=${{ github.sha }}''  >> $GITHUB_ENV
      - run: |-
          echo DOCKER_TAG:  $DOCKER_TAG

      - name: DockerHub login
        env:
          DOCKERHUB_USERNAME: ${{ secrets.DOCKERHUB_USERNAME }}
          DOCKERHUB_PASSWORD: ${{ secrets.DOCKERHUB_PASSWORD }}
        run: |
          docker login -u $DOCKERHUB_USERNAME -p $DOCKERHUB_PASSWORD

      - name: Build docker image
        run: |
          docker build \
            -t interpres/api-gateway:latest \
            -t interpres/api-gateway:$DOCKER_TAG \
            ./server/api-gateway

      - name: Push docker image
        run: |
          docker push --all-tags interpres/api-gateway
```

Primer koda 3: Tok kontinualne isporuke mikroservisa *gateway*

Glava 5

Zaključak

Cilj ovog rada je da predloži i obrazloži informacioni sistem u oblaku koji bi pomogao programerima i prevodiocima tokom razvijanja višejezične aplikacije. Korišćenjem ovog sistema smanjuje se jaz između ove dve grupe koje zajedničkim snagama poboljšavaju korisničko iskustvo aplikacije na kojoj rade. Zbog sve prisutnije digitalizacije, ne sme se zaboraviti na grupe ljudi koje ne razumeju određeni jezik i zbog toga ne mogu da koriste aplikaciju.

Predloženo rešenje je u velikoj meri fleksibilno što se tiče podrške alata za prevođenje. Bitno je da alat, za skladištenje prevoda, radi sa datotekama u formatu *JSON*. U budućnosti se ostali formati mogu podržati jednostavnom implementacijom novog parsera u mikroservisu *integration* i poštovanjem interfejsa za komunikaciju sa mikroservisom *core*. Sa druge strane, projekat *Interpres* se previše oslanja na *GitHub*. Glavne mane su nemogućnost uvoza i izvoza prevoda u fajlove i zavisnost od provajdera identiteta. Aplikacije koje je potrebno prevesti mogu koristiti neke druge sisteme za verzionisanje koda, a *Interpres* to ne dozvoljava. Iz tog razloga bi uvoz i izvoz preko fajlova bio poželjniji. Isto tako, nije teško zamisliti da korisnici ne žele da prave naloge na *GitHub*-u, već da žele da koriste neki drugi provajder identiteta kao što je *Google*, *Facebook*, ili pak žele da naprave nalog na samom sistemu.

Tema koja je ostala neobrađena, a koja je jako bitna, je autorizacija. U predloženom rešenju, korisnik koji je autentifikovan ima pristup svim projektima. Šta više, svi korisnici imaju ista prava. Ovaj nedostatak predstavlja veliki rizik za bezbednost podataka. U narednoj verziji projekta trebalo bi osmisliti i implementirati prava pristupa. Korisnici bi se u tom slučaju dodavali u projekte sa određenim pravima, kao što su prava za čitanje, prava za menjanje sadržaja ili

prava za podešavanje samog projekta.

Prilikom razvoja sistema korišćena je arhitektura mikroservisa. Ideja vodilja za odabir ove arhitekture je bila da informacioni sistem bude što je više moguće skalabilan, pouzdan i održiv. Sve veći trend korišćenja mikroservisa u oblaku za velike aplikacije potvrđuje da je to dobar izbor.

Kako se računarstvo u oblaku razvija velikom brzinom, tako se razvijaju i novi alati koji pokušavaju da reše probleme koji nastaju na oblaku. S obzirom na sve veću upotrebu arhitekture mikroservisa, glavni problem predstavlja upravljanje većim brojem komponenti. *Kubernetes*, kao alat za rešavanje ovog problema, u potpunosti zadovoljava potrebe operacionog tima, ali i tima programera.

Alati za automatizaciju u mnogome pomažu razvoj softvera. Kontinuirano isporučivanje i kontinuirano raspoređivanje ubrzavaju ciklus razvoja i korisnici ranije mogu dobiti nove verzije softvera. Pored samih korisnika, benefit imaju i programeri jer mogu ranije da uoče probleme i brže da ih reše.

Potrebno je napomenuti da mikroservisi ne predstavljaju rešenje za sve. Iako se takvi sistemi lakše skaliraju i imaju bolju razdvojenost poslovne logike, sam razvoj ume da bude kompleksan. Prerano razmišljanje o optimizaciji i skaliranju sistema može usporiti razvoj i potencijalno dovesti do gašenja projekta jer funkcionalnosti sistema nisu isporučene na vreme. Sa druge strane, može se desiti da sam sistem uopšte nema potrebe da opsluži toliko korisnika. Onda se dobija kompleksan sistem bez prevelikog iskorišćenja, što je potpuno suprotno od polazne pretpostavke. Potrebno je voditi se principom da se sistem optimizuje samo onda kada je to i zaista potrebno.

Literatura

[1] BabelEdit, https://www.codeandweb.com/babeledit. 30.8.2021.

[2] Localazy, https://localazy.com/. 30.8.2021.

[3] Džejms Luis i Martin Fauler, Microservices, https://martinfowler.com/articles/microservices.html. 30.8.2021.

[4] Saša Malkov, beleške sa predavanja, Programiranje za veb. 2014. godina.

[5] Martin Fauler, Software Component, https://martinfowler.com/bliki/SoftwareComponent.html. 30.8.2021.

[6] Roy Thomas Fielding, Architectural Styles and the Design of Network-based Software Architectures, Doktorska disertacija. University of California, Irvine, 2000.

[7] REST API Tutorial https://restfulapi.net/. 30.8.2021.

[8] React https://reactjs.org/. 30.8.2021.

[9] Olususi Kayode Oluyemi, Getting Started with NestJS https://www.digitalocean.com/community/tutorials/getting-started-with-nestjs. 30.8.2021.

[10] NestJS https://docs.nestjs.com/. 30.8.2021.

[11] GitHub, GitHub Applications, https://docs.github.com/en/developers/apps/getting-started-with-apps/about-apps. 30.8.2021.

[12] TypeORM, https://typeorm.io/. 30.8.2021.

[13] Marko Lukša, Kubernetes in action. *Manning*, 2017, 978-1-61729-372-6

[14] Sahiti Kappagantula, Docker Explained – An Introductory Guide To Docker, `https://www.edureka.co/blog/docker-explained/`. 30.8.2021.

[15] Redhat, What is CI/CD?, `https://www.redhat.com/en/topics/devops/what-is-ci-cd`. 30.8.2021.

[16] GitHub, GitHub Actions, `https://docs.github.com/en/actions/learn-github-actions/introduction-to-github-actions`. 30.8.2021.

Biografija autora

Uroš Milenković (*Bor, 29. maj 1992.*) Završio prirodno–matematički smer, XIII beogradske gimnazije 2010. godine i iste godine upisao Matematički fakultet u Beogradu. Osnovne studije, na smeru „Informatika", završava 2015, kada je upisao i master studije na istom smeru. Aprila 2016. počinje da radi kao softverski inženjer u firmi *CallidusCloud*, do septembra 2019. Tada prelazi u firmu *3ap*. Avgusta 2021. prelazi u firmu *Nutanix*.