Strategije deljenja koda za mikroservise: Osnaživanje skalabilnog i agilnog razvoja softvera

Marko Murić, Bojana Dimić Surla [1]

***Sadržaj* — U ovom radu prikazane su strategije deljenja koda u mikroservisnoj arhitekturi sa ciljem poboljšanja skalabilnosti i agilnosti razvoja softvera. U prvom delu rada razmatra se važnost deljenja koda, kao i izazovi koji postoje prilikom deljenja koda u mikroservisnoj arhitekturi. Zatim se detaljno analiziraju trenutno popularne strategije za deljenje koda. Na kraju kroz studiju slučaja prikazujemo na praktičan način neke od strategija za deljenje koda uz detaljan pregled implementacije.**

***Ključne reči* — mikroservisi, paketi, strategije za deljenje koda, sihrona komunikacija, asihrona komunikacija**

I. UVOD

Mikroservisi predstavljaju popularnu softversku arhitekturu koja nudi brojne pogodnosti, uključujući poboljšanu skalabilnost, izolaciju grešaka i povećanu brzinu razvoja. Međutim kako softverski sistemi postaju sve više distribuirani i modularni, javlja se izazov efikasnog deljenja koda među mikroservisima [1]. Mogućnost efikasnog deljenja koda prilikom razvoja mikroservisa promoviše ponovnu upotrebu koda, smanjenje redudantnosti i omogućava razvojnim timovima da brže isporuči nove funkcionalnosti i poboljšanja.

[1] M. Murić, Računarski fakultet, Beograd, Srbija (e-pošta: mmuric1320m@raf.rs)
B. Dimić Surla, Računarski fakultet, Beograd, Srbija (e-pošta: bdimicsurla@raf.rs)

Deljenje koda između više mikroservisa može biti složen poduhvat koji zahteva pažljivo razmatranje različitih faktora kao što su granice servisa, verzioniranje, kompatibilnost i održavanje. Planiramo da pružimo uvid u komplikacije koje deljenje koda u mikroservisnoj arhitekturi sa sobom nosi, kao i da pregledamo efikasne strategije, koje mogu rešiti ove probleme.

Mikroservisi su servisi koji su modelovani oko pojedinačnog poslovnog domena. Ovo znači da servis treba da enkapsulira jednu funkcionalnost i pruži je na raspolaganje drugim servisima putem jednostavnih interfejsa [2]. Ovo predstavlja jasnu granicu između mikroservisa kao gradivnih blokova aplikacije. Ove granice omogućavaju nezavisno razvijanje, testiranje, deploy i skaliranje svakog servisa.

Promene unutar granica mikroservisa ne bi smele da utiči na druge mikroservise ili krajnjeg korisnika, što omogućava nezavisno isporučivanje funkcionalnosti. Na ovaj način mikroservis radi izolovano i isporučuje se na zahtev. Sa jasnim i stabilnim granicama servisa, koje se ne menjaju kada se promeni unutrašnja implementacija dobijate sistem koji imaju slabu povezanost i jaču koheziju koje predstavljaju ključne pokazatelje kvaliteta mikroservisnih aplikacija.

II. VAŽNOST I IZAZOVI DELJENJA KODA KOD MIKROSERVISA

Deljenje koda predstavlja korišćenje postojećeg koda za razvoj novog softvera. Kada pričamo o deljenju koda obično mislimo na izdvajanje koda koji se ponavlja u zasebne pakete ili bilioteke. Deljenje koda preko paketa podrazumeva instalaciju različitih zavisnosti neophodnih za korišćenje datog paketa. Ovo može otežati održavanje koda. Postoje studije [3] koje uspešno dokazuju da deljenje koda smanjuje tehnološki dug i ističi veliki značaj efkasnog deljenja koda u razvoju softvera.

Ako posmatramo mikroservis kao nezavisnu komponentu razvijenu od strane autonomnog tima, onda možemo uvideti da ta komponenta predstavlja praktičan primer deljenja koda. Ovde deljenje koda uzimamo kao sinonim za deljenje funkcionalnosti. Mikroservis, kao zasebna komponenta komunicira sa ostatkom sistema preko definisanih interfejsa, što razvojnim timovima olakšava razvoj. Pored toga, timovi mogu biti kreativniji jer se mogu fokusirati na samo jednu nezavisnu uslugu, a njihovi komunikacioni zahtevi su minimizirani.

Izazovi deljenja koda u mikroservisnoj arhitekturi se uglavnom svode na pojedine strategije. Izazove koje ćemo ovde razmotriti, pogledaćemo u odnosu

na strategije deljenja koda preko paketa, i strategija deljenja koda preko mikroservisa.

Jedna od ključnih karakteristika mikroservisa je decentralizacija upravljanja. Iako su benefiti ovih karakteristika brojni, možemo istaći brži i agilniji razvoj, smanjena komunikacija između timova, nezavisno održavanje i nezavisno odpremanje [2]. Uspešnost ovih principa dovelo je do stvaranja paterna deli-ništa (engl. share-nothing) koji predviđa da mikroservisi ne dele ni jednu funkcionalnost. Korišćenje ovog paterna zaista dovodi do stvaranja autonomnih servisa. Međutim, absolutna autonomija mikroservisa, kao ekstrem kome treba težiti, često nije moguća. Autonomija mikroservisa često je pogrešno shvaćena. Autonomija se ovde odnosi na nezavisnost podataka, i nezavisnost raspoređivanja i isporuke [5].

Oštra kritika internet zajednice na deljenje koda u mikroservisima često je povezana sa prevođenjem monolitne arhitekture na mikroservisnu arhitekturu. Nije retka pojava da je određeni deo koda kod monolitne arhitekture već apstrahovan kako bi se ponovo koristio unutar aplikacije. Prilikom migracije na mikroservisnu arhitekturu, ovaj kod često završi kao jedan od paketa. Na ovaj način kratkoročno je ubrzan razvoj mikroservisa, ali se stvara povezanost istih. Korišćenje ovakvih paketa dovodi do povišene komunikacije između timova, često netrivijalnih promena u svim mikroservisima prilikom promene zahteva, i na kraju naduvenosti koda gde mikroservisi imaju pristup kodu koji ne koriste [4].

Iz prethodnog lako možemo zaključiti da netrivijalne promene u mikroservisima, kao posledica promene u paketima, uz dodatnu koordinaciju, narušavaju autonomiju razvoja i raspoređivanja, a samim tim dovode do zavisnih mikroservisa. Da bi se ova zavisnost izbegla nakon inicijalnog razvoja neophodno je uložiti vreme i preuzeti ili implementirati neophodne funkcionalnosti paketa unutar samog mikroservisa kao deo poslovne logike. Ovo bi dovelo do nezavisnih mikroservisa, a paket bi postao zastareo i vremenom bi nestao. U suprotnom dobili bi smo neku vrstu distribuirane monolitne arhitekture.

Iz ovoga zaključujemo da deljenje koda koji predstavlja poslovnu logiku može narušiti principe na kome zasnivamo autonomiju mikroservisa. Previše deljenja stvara previše zavisnosti između mikroservisa, što dovodi do krhkih sistema koje je veoma teško testirati i primeniti [6]. Ovo nas dovodi do antipaterna deli-što-je-moguće-manje, što nam pomaže u održavanju granica između mikroservisa na taj način što nas podstiče na deljenje koda koji je tehnički i apstraktan.

Ukoliko kod koji želimo da delimo nije dovoljno generalizovan da bude otvoreni kod (engl. open source), onda ne bi ni trebao da bude u paketu [7]. Tehnički apstraktnom kodu povećavamo kvalitet strogim testiranjem, pa samim tim možemo očekivati da neće zaustaviti rad drugog mikroservisa.

Jedna od dobrih praksi podrazumeva da paket ima verzioniranje, samim tim ostavlja se mogućnost projektnom timu mikroservisa da izvrši ažuriranje kada to njima odgovara. Na ovaj način smanjujemo koordinaciju između timova.

Antipatern deli-što-je-moguće-manje ne rešava sve izazove koje imamo kada je reč o povezanosti između mikroservisa. Ukoliko smo u obavezi da koristimo kod koji nam je podeljen kroz paket, to dovodi do takozvane tehnološke zavisnosti. U najvećem broju slučajeva smo ograničeni na jezik u kom je paket napisan, kao i na odgovarajući okvir (engl. framework).

Osnovna pretpostavka decentralizovanog upravljanje je da možemo sami birati alat koji nam najviše odgovara za implementaciju poslovne logike. Samostalan izbor alata zasnovan je na činjenici da na taj način možemo izabrati najbolji alat za optimalno rešenje. Nemogućnost da se izabere drugi jezik kao alat, značajno umanjuje fleksibilnost koju mikroservisi imaju u odnosu na ostale arhitekture.

Postoje dve moguće opcije koje ovde možemo uraditi. Prva je da sagledamo benefite, i ako su benefiti veći od problema koje izazivamo upotrebom samo jednog jezika, nastavimo sa upotrebom paketa, a samim tim i odabranog jezika. Druga opcija je da ipak koristimo drugi jezik, što podrazumeva i prepisivanje paketa u jeziku koji želimo da koristimo. Ovaj pristup predstavlja značajan overhed u razvoju tim pre što možemo imati veliki broj mikroservisa, i potencialno svaki može biti napisan u različitom jeziku. U takvim slučajevima treba odustati od deljenja koda preko paketa.

Poslednji izazov koji moramo sagledati tiču se same organizacije. Sa rastom aplikacije dolazi i porast broja članova tima, što komplikuje komunikaciju i pružanje smernica o ponovnoj upotrebi koda. Stvaranje i održavanje kataloga i arhive postaje neophodno kako za pakete koda tako i za zajedničke mikroservise.

Drugi značajan organizacioni izazov je nedostatak jasnoće u vezi sa odgovornošću za razvoj i održavanje zajedničkog koda. Dok je u monolitnoj arhitekturi odgovornost bila jasno dodeljena svim članovima tima, u mikroservisnoj arhitekturi ovaj proces postaje kompleksniji. Podela odgovornosti na sve timove nije optimalna jer može umanjiti autonomiju timova, dovodeći do povezanosti među njima.

Često se susrećemo i sa otporom razvojnih timova prema izmenama u zajedničkom kodu, što može biti posledica nedostatka upoznatosti sa poslovnim domenom drugih timova. Proces izmena zahteva značajan napor, što može rezultirati izbegavanjem izmena u zajedničkom kodu. Stoga, optimalno bi bilo da svaki kod koji se deli, bilo kao paket ili kao zaseban mikroservis, ima svoj razvojni tim.

U slučaju strategije deljenja koda putem mikroservisa, planiranje razvoja je često ostvarivo, jer menadžment obično planira razvoj mikroservisa koji će biti pozivani od strane drugih. Međutim, razvoj paketa zahteva posebno planiranje kako kratkoročno, tako i dugoročno, uključujući pisanje dokumentacije, vođenje arhive i održavanje, što zahteva odgovarajuće resurse i pažljivo planiranje menadžmenta.

III. STRATEGIJE ZA EFIKASNO DELJENJE KODA

Razumevanje strategija za efikasno deljenje koda između mikroservisa predstavlja veoma bitan aspekt razvoja softvera u mikroservisnoj arhitekturi. U ovom poglavlju predstavićemo najčešće strategije deljenja koda između mikroservisa, kao fundamentalni aspekt koji doprinosi agilnosti i skalabilnosti savremenih aplikacija. Praksa deljenja koda, kao sastavni deo procesa razvoja, može poprimiti različite oblike, svaki sa sopstvenim skupom prednosti i kompromisa. Osnovne strategije za efikasno deljenje koda uključuju izbegavanje deljenja koda, manuelno deljenje koda kopiranjem, automatizovano deljenje koda upravljanim kopiranjem, deljenje koda preko paketa i deljenje koda kroz mikroservise.

A. *Izbegavanje deljenja koda*

Ukoliko je moguće izbeći deljenje koda, ovo je idealna opcija sa stanovišta arhitekture mikroservisa [5]. Ukoliko ne delimo kod sve karakteristike mikroservisa su ispunjene, što znači da će mikroservisi razvijeni na ovaj način biti nezavisni, slabo povezani, bez prelazaka granica između mikroservisa. Na ovaj način dobijamo agilne mikroservise, koji nemaju zavisnosti prilikom raspoređivanja.

Prednosti ove strategije su dobra izolovanost i skalabilnost, čiste i nezavisne granice sistema. Kompromisi ove strategije su narušavanje DRY (don't repeat yourself) principa, što dovodi do redudantnog koda, sklonost ka nedoslednostima i veliko vreme potrebno za izmene izmene koje treba izvršiti na većem broju mikroservisa.

B. *Manuelno deljenje koda kopiranjem*

Strategija manuelnog deljenja koda, prosto kopiranje koda, predstavlja jednu od najlakših načina da se kod podeli između mikroservisa. Ova strategija direktno narušava DRY principe, pa samim tim nosi sa sobom sve probleme dupliranja koda. Ova strategija se ne preporučuje [5], ali se ipak jako često koristi, naročito kada su u pitanju relativno male funkcionalnosti za koje očekujemo da se neće često menjati. Iako ova strategija tehnički predstavlja deljenje koda, jako malo se razlikuje od prethodne strategije. Sam kod egzistira u okviru granica mikroserivsa, u koji smo kod preneli. Samim tim karakteristike mikroservisa nisu narušene, a mikroservisi razvijeni na ovaj način su agilni i nemaju zavisnosti prilikom raspoređivanja.

Ipak, kod koji je podložan čestim promenama, potrebno je kopirati na sve mikroservise koji ga koriste, i eventualno adaptirati i izmeniti u skladu sa potrebama konkretnog mikroservisa. Ovo je relativno izvodljivo ukoliko je broj mikroservisa u aplikaciji mali. Ipak kako raste broj mikroservisa za potrebe aplikacije, raste i vreme potrebno da se određeni kod kopira i adaptira za potrebe mikroservisa, samim tim i zavisnost prilikom raspoređivanja.

Prednosti ove strategije su brza i jednostavna implementacija za male projekte, nije neophodna dodatna infrastruktura. Kompromisi ove strategije su sklonost nedoslednostima i greškama usled redudantnosti, ili kada su potrebne česte promene, ne skalira se dobro kada projekat raste, ograničena kohezija sistema zbog dupliranog koda.

C. *Automatizovano deljenje koda upravljanim kopiranjem*

Izdvajanje koda u samostalne komponente, koje putem određenih mehanizama za sinhronizaciju možemo importovati u mikroservise predstavlja jedan od načina deljenja koda, naročito kada su u pitanju frontend, ili mikro-frontend rešenja. Ovi mehanizmi za upravljanje komponentama nalik su alatima za upravljanje paketima. Oni omogućavaju da se komponente učitaju u kod, i koriste na isti način kao i paketi. Sa druge strane pružaju mogućnost da se iste komponente preuzmu kao kopija koda i učitaju kao komponenta koja egzistira u okruženju mikroservisa, čime se olakšava održavanje komponente. Nakon izvršenih izmena, potrebno je da komponentu izvezemo a mehanizam će se pobrinuti da svaki mikroservis bude sinhronizovan sa najnovijim izmenama (upravljano kopiranje). Na ovaj način je značajno olakšano i ubrzano održavanje.

U trenutku pisanja rada naišli smo na samo jedan mehanizam koji automatizovano deli kod pod nazivom Bit. Nažalost ovaj mehanizam podržava samo typescript jezike. Ukoliko je izabrani jezik typescript ili nodejs, moguće je koristiti ovu strategiju, inače ne.

Prednosti ove strategije su automatizacija određenih aspekata deljenja koda, što smanjuje greške ručnog kopiranja. Kompromisi ove strategije su povećana složenost u održavanju mehanizama sinhronizacije, što sa sobom nosi i rizike u vidu grešaka sinhronizacije.

D. *Deljenje koda preko paketa*

Paketi predstavljaju skup fajlova i direktorijuma zapakovanih zajedno kao kolekcija softvera koja pruža odgovarajuću funkcionalnost kao deo većeg softverskog rešenja. Pretpostavka je da već koristimo alat za manipulaciju paketima, tako da nam nije neophodan dodatni alat niti dodatno znanje. Pored toga, veliki broj materijala koji olakšava implementaciju sopstvenog paketa čini ovu strategiju jako popularnom [5].

Ova strategija nije bez mana, od kojih možemo istaći da nije moguće da se besplatno podele privatni paketi. Moguće je zakupiti privatno skladište i podeliti privatni paket, međutim ovo iziskuje ili određeno znanje ili skuplje opcije zakupa. Pored podešavanja privatnih paketa, potrebno je podesiti i CI/CD da preuzima pakete sa privatnog skladišta. Olakšavajuće je da jednom postavljena podešavanja za privatni paket možemo koristiti zauvek.

Kada pričamo o mikroservisima, strategija deljenja koda preko paketa je jedna od najkritikovanijih strategija. Deljenje poslovne logike kroz paket predstavlja jednu od osnovnih kritika strategije deljenja koda preko paketa [4], jer se na taj način narušava autonomija mikroservisa.

Zbog visoke povezanosti između paketa i mikroservisa, dolazi do visoke povezanosti i između mikroservisa. Ovo za posledicu ima međusobnu zavisnost mikroservisa prilikom postavljanja, što pre podseća na distribuiranu monolitnu arhitekturu, nego na arhitekturu mikroservisa.

Bez obzira na to da li kritikuju deljenje koda preko paketa [4] ili podržavaju deljenje koda preko paketa [8], internet zajednica podržava dijeljenje zajedničkih opštih funkcionalnosti kao što su logovanje, nadzor, sigurnosni mehanizmi, asihrona komunikacija i upravljanje greskama preko paketa. Implementacijom ovih rešenja na jednom mestu utičemo na doslednost mikroservisa. Nisu nam potrebni testovi integracije. Kada ispravimo grešku u paketu, to se propagira na sve mikroservise, što olakšava ispravljanje grešaka.

Razvoj paketa bi trebao da prati praksu razvoja softvera otvorenog koda. Ukoliko kod nije dovoljno generalizovan da bude otvorenog kod, ne bi ni trebao da bude u paketu. Jedna od smernica najbolje prakse je da treba razdvojiti pakete po funkcionalnostima. Npr. logovanje, sigurnost i nadzor pripadaju jednom paketu, upravljanje greškama drugom, asihrona

komunikacija trećem paketu [8]. Prosto nije svakom mikroservisu neophodna svaka funkcionalnost iz pomenutih paketa, zbog čega delimo pakete u manje celine, tako da mikroservisi koriste samo one pakete čija im je funkcionalnost potrebna.

Paketi bi morali da budu kompatibilni unazad. Ukoliko je kompatibilnost unazad prekinuta, velike su šanse da je neophodno uraditi izmene na mikroservisu koje nisu trivijalne. Ovakve izmene zahtevaju dodatno vreme, čime se gubi smisao upotrebe paketa. Izmene paketa bi trebale da podržavaju semantičko verzioniranje koje prati obrazac GLAVNI.MANJI.ZAKRPA (engl. MAJOR.MINOR.PATCH). Verzioniranje daje mogućnost svakom timu da se na vreme informiše i odluči u svom temu o nadogradnji paketa [8].

Jedan od većih problema koje ova strategija sa sobom nosi, je tehnološka zavisnost između mikroservisa. Ipak kada uzmemo u obzir prednosti koje ova strategija donosi, možemo zaključiti da prednosti nadjačavaju mane ove strategije [9].

Prednosti ove strategije su centralizovan kod koji se lakše održava, konzistentnost prilikom ponovne upotrebe, povećana skalabilnost. Kompromisi ove strategije su povećana zavisnost eksternih paketa, što sa sobom donosi i određene izazove kada su u pitanju verzije i kompatibilnosti, takođ može uvesti dodatne troškove kada postoji potreba za ažuriranjem paketa.

E. *Deljenje koda kroz mikroservise*

Mikroservisi predstavljaju nezavisne komponente od kojih gradimo aplikaciju, stoga ne čudi ideja da podelimo funkcionalnost preko mikroservisa. U ovom slučaju deljenje funkcionalnosti predstavlja specifičan oblik deljenja koda. Granice između mikroservisa su jasno definisane pa samim tim mikroservisi predstavljaju nezavisne i izolovane komponente koje enkapsuliraju određenu funkcionalnost. Ova funkcionalnost može biti i deo poslovne logike, što daje prednost ovoj strategiji u odnosu na strategiju deljenje koda preko paketa. Osnovna kritika deljenja poslovne logike preko paketa bila je rigidnost i stvaranje zavisnosti između mikroservisa. Ove mane nisu prisutne ako delimo poslovnu logiku preko mikroservisa, pa je samim tim poželjno kad god je moguće podeliti poslovnu logiku preko mikroservisa.

Ako delimo kod kroz mikroservis, većina karakteristika u startu nije narušena. Funkcionalnost koju želimo da podelimo sa ostatkom aplikacije, enkapsulirana je u samom mikroservisu što predstavlja nezavisnu komponentu. Tim koji razvija mikroservis, organizovan je oko poslovnog

domena, i odgovoran je za mikroservis tokom celog životnog ciklusa mikroservisa. Testiranje samog mikroservisa nezavisno je isto kao i raspoređivanje samog mikroservisa. Tim sam za sebe može odabrati tehnološku platformu, i odabrati kako želi da upravlja podacima.

Ipak, u nekim slučajevima mogu se javiti zavisnosti i kada koristimo mikroservise za deljenje koda. Ukoliko mikroservis preko koga delimo funkcionalnost prestane sa radom, postoji opasnost da mikroservis koji ga poziva neće pravilno raditi. Ovo je naročito izraženo kod mikroservisa koji pružaju svoje funkcionalnosti sihrono. Iz tog razloga bitno je da svi mikroservisi budu dizajnirni za neuspeh i da graciozno upravljaju greškama. Ukoliko mikroservis koji implementiramo može narušiti kompatibilnost unazad, postoji opasnost da možemo uvesti greške kod mikroservisa koji ga koriste.

Iz prethodnog uočavamo da može doći do potencijalnih problema povezanosti između mikroservisa koji dele funkcionalnosti i mikroservisa koji koriste ove funkcionalnosti. Nažalost, ako želimo da delimo funkcionalnosti kroz mikroservise, ovu povezanost ne možemo u potpunosti eliminisati.

Strategija deljenja koda preko mikroservisa nije bez svojih kompromisa. Jean od kompromisa na koji računamo je potreba za dodatnom infrastukturom i resursima za korišćenje mikroservisa. Da bi smo pozvali funkcionalnosti koje delimo preko mikroservisa najčešće moramo uputiti ili sihroni poziv prema API-ju, ili asihrono slanje poruka. Shodno tome mikroservisi koji upućuju pozive moraju voditi računa o greškama, što unosi određenu složenost u komunikaciji.

Takođe, korišćenjem ove strategije, kada pozivamo funkcionalnost koju delimo, imamo određenu vrstu kašnjenja dok čekamo na odgovor. Ova kašnjenja nisu nužno povezana sa kvalitetom implementacije funkcionalnosti u mikroservisima, već sa okruženjem u kom mikroservisi egzistiraju i izvršavanje u odvojenim procesima Iz tog razloga ova strategija nije preporučena ukoliko nam je potrebno malo vreme kašnjenje (engl. small latency).

Prednosti ove strategije su odlična izolacija koda što daje mogućnost lakšeg održavanja, visoku skalabilnost i efikasnost resursa, jasne granice sistema i nezavisnost. Kompromisi ove strategije su dodatna infrastruktura za primenum, dodaje složenost u smislu komunikacije između mikroservisa, potencijalno kašnjenje u komunikaciji između mikroservisa, zahteva robustne alate za praćenje i upravljanje za distribuirani sistem.

IV. STUDIJA SLUČAJA I NAJBOLJA PRAKSA ZA DELJENJE KODA

U ovoj sekciji prikazana je praktična primena strategije za deljenje koda na primeru implementacije deljenja koda preko paketa i strategija deljenja koda preko mikroservisa

A. *Implementacija strategije deljenja koda preko paketa*

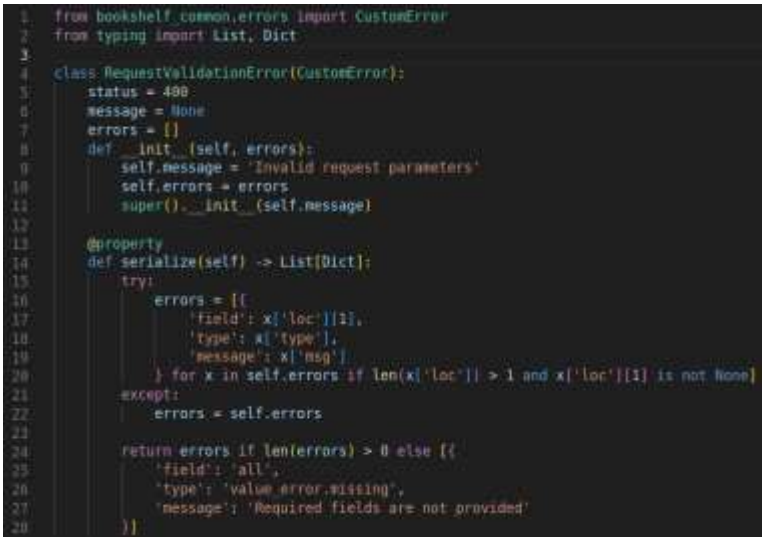
Za implementaciju paketa koristićemo Python-packaging pristup [10]. Ovaj pristup pruža mogućnost da implementiramo određeni deo koda, i na vrlo jednostavan način taj kod zapakujemo u Python paket koji se dalje može instalirati preko pip ili easy_install alata za upravljanje paketima.

Paket koji će se deliti između mikroservisa implementira upravljanje izuzecima i asihronu komunikaciju. Izvorni kod paketa je javno dostupan i može se pronaći na lokaciji: https://github.com/mmuric/bookshelf-common.

Slika 1. Konfiguracija paketa - setup.py

Na slici 1 vidimo prikazan setup.py fajl, kao glavni konfiguracioni fajl za objavu paketa u Pythonu. Metoda setup prihvata različite argumente. Argument **name** predstavlja ime paketa. Argument **version** predstavlja verziju paketa. Verzija mora biti ažurirana kada objavljujete paket, pošto nije moguće objaviti paket sa istom verzijom. Argument **packages** predstavlja listu svih paketa koja se nalazi u projektu. Poslednji važan argument je **install_requires** i predstavlja listu zavisnosti za izvršenje paketa.

Na slici 2 prikazan je primer implementacije jednog izuzetka koji će se deliti između mikroservisa preko paketa. Izuzetak je implementiran klasom *RequestValidationError* koja nasleđuje kalsu *CustomError*, takođe razvijenu u okviru paketa u kojoj su implementirane zajedničke funkcionalnosti svih izuzetaka.

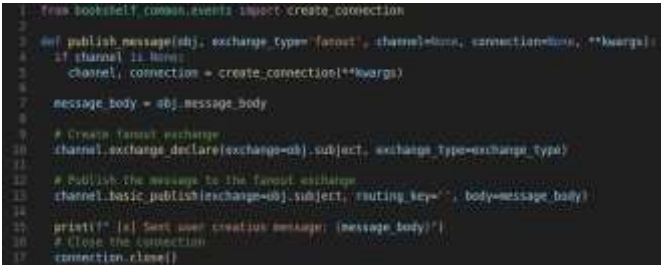


```python
from bookshelf_common.errors import CustomError
from typing import List, Dict

class RequestValidationError(CustomError):
    status = 400
    message = None
    errors = []
    def __init__(self, errors):
        self.message = 'Invalid request parameters'
        self.errors = errors
        super().__init__(self.message)

    @property
    def serialize(self) -> List[Dict]:
        try:
            errors = [{
                'field': x['loc'][1],
                'type': x['type'],
                'message': x['msg']
            } for x in self.errors if len(x['loc']) > 1 and x['loc'][1] is not None]
        except:
            errors = self.errors

        return errors if len(errors) > 0 else [{
            'field': 'all',
            'type': 'value_error.missing',
            'message': 'Required fields are not provided'
        }]
```

Slika 2. Izuzetak RequestValidationError

Druga funkcionalnost koja se deli između mikroservisa putem paketa je implementacija asinhrone komunikacije korišćenjem RabbitMQ brokera za razmenu poruka. Za potrebe ove implementacije dodat je još jedan paket u setup.py fajlu, to je *pika* [11]. Za slanje poruka potrebno je implementirati funkcionalnost za otpremanje poruka, za prijem i obradu poruka. Pored ovoga, sa ciljem olakšnja implementacije komunikacionog protokola, implementirane su klase koje definišu koji podaci se mogu poslati preko poruka.

Na slici 3 prikazana je implementacija metode za objavu poruka. Ova metoda prihvata argument channel, u slučaju da kanal za slanje poruka želimo da definišemo van ove metode.



```python
from bookshelf_common.events import create_connection

def publish_message(obj, exchange_type='fanout', channel=None, connection=None, **kwargs):
    if channel is None:
        channel, connection = create_connection(**kwargs)

    message_body = obj.message_body

    # Create fanout exchange
    channel.exchange_declare(exchange=obj.subject, exchange_type=exchange_type)

    # Publish the message to the fanout exchange
    channel.basic_publish(exchange=obj.subject, routing_key='', body=message_body)

    print(f" [x] Sent user creation message: {message_body}")
    # Close the connection
    connection.close()
```

Slika 3. Implementacija objave poruka

Automatizacija objave i preuzimanja paketa izvršena je korišćenjem GitHub akcija. Na slici 4 prikazana je specifikacija GitHub akcije u YAML formatu koju koristimo da automatizujemo objavu Python paketa na repozitorijum PyPI. Ova akcija se pokreće kada se kreira nova objava na GitHub repozitorijumu.

Dokument prikazan an slici 4 sadrži ime akcije (linija 1), zatim definiciju kada se akcija izvršava (linija 4) i dozvolu pristupa repozitorijumu. Od linije 10 definisani su poslovi (engl. jobs), što i predstavlja glavni deo akcije. Ovde su definisani koraci koji su neophodni kako bi paket bio objavljen. U liniji 18 definisana je verziju pythona neophodna za izvršenje ovog paketa, u liniji 24 navedena je komanda kojom se izvršava izgradnja paketa, i u liniji 26 komanda koja vrši objavu paketa na PyPI.

Slika 4. Objava paketa

B. Implementacija strategije deljenja koda preko mikroservisa

Da bi smo delili funkcionalnosti između mikroservisa, potrebno je da mikroservisi komuniciraju jedan sa drugim. Dva najćešća metoda komunikacije između mikroservisa su sihrona komunikacija i asihrona komunikacija. Sihrona komunokacija kod mikroservisa se odnosi na obrazac gde klijent koji šalje zahtev ka mikroservisu čeka odgovor od strane mikroservisa pre nego što nastavi dalje sa izvršenjem akcije. Asihrona komunikacija kod mikroservisa se odnosi na obrazac gde klijent koji upućuje zahtev mikroservisu, ne čeka na njegov trenutni odgovor, već nastavlja sa

izvršenjem akcije, bez blokiranja. Mikroservis ovaj zahtev dalje izvršava nezavisno od mikroservisa koji je uputio poziv.

Jedna od funkcionalnosti koja se može podeliti između mikroservisa u vidu odvojenog mikroservisa je funkcionalnost za otpremanje slika. Ovaj mikroservis će se pozivati sihrono. Krajnja tačka za pristup servisu je "upload_photo" (slika 5).

Slika 5. Servis za otpremanje slika

Slika 6. Servis za slanje elektronske pošte

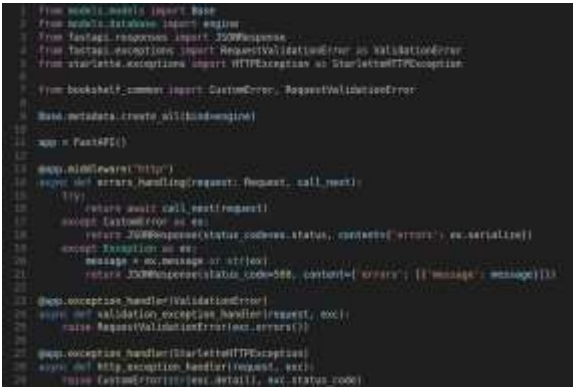
Primer deljenog mikroservisa kome se pristupa asinhrono je mikroservis koji se koristi za centralizovano slanje elektronske pošte. Implementacija ovog mikroservisa prikazana je na slici 6. Pošto se EmailWorker koristi asihrono, potrebno je da implementiramo pretplatu na poruke (engl. consumer), kao i funkcije povratnog poziva (engl. callback functions) koje pripremaju sadržaj email-a.

C. *Primena deljenog koda za hvatanje izuzetaka*

Middleware je koncept u razvoju softvera koji se koristi za obradu zahteva između različitih slojeva aplikacije. On je srednji sloj (engl. middleware) koji se nalazi između različitih komponenti softverskog sistema. Glavna svrha

middleware-a je da omogući dodatnu funkcionalnost ili manipulaciju podacima bez promene same aplikacije.

U ovoj sekciji opisaćemo hvatanje izuzetaka na nivou middleware-a za FastApi i Flask okvire. Kada je reč o hvatanju izuzetaka, middleware je koristan alat jer omogućava centralizovanu obradu grešaka na nivou aplikacije.



Slika 7. Implementacija middleware-a za FastApi okvir.

Na slici 7. prikazana je implementacija middleware-a za FastApi okvir. FastApi okvir ima specifičnu ugrađenu validaciju, zbog koje je neophodno pregaziti RequestValidationError i HTTPException middleware. U ovim middleware-ima podigli smo naše prilagođene izuzetke RequestValidationError i CustomError i na taj način obezbedili doslednost u upravljanju greškama.

Na slici 8 prikazana je implementacija middleware za Flask okvir. Da bi smo implementirali upravljanje izuzecima u Flask-u, neophodno je da registrujemo rukovaoca (engl. handler), što možemo uraditi korišćenjem dekoratora iznad funkcije u kojoj implementiramo middleware. Kako možemo videti ukoliko je izuzetak instance CustomError, tada vraćamo serijalizovani json dokument sa statusom izuzetka i porukom o grešci. Ukoliko je izuzetak instance Exception, tada vraćamo serializovani json sa porukom o grešci i statusom 500.

```
import uuid, os
from flask import Flask, jsonify
from bookshelf_common import CustomError

app = Flask(__name__)
app.secret_key = os.environ.get('SECRET_KEY')

@app.errorhandler(Exception)
def handle_exception(ex):
    if isinstance(ex, CustomError):
        return jsonify({'errors': ex.serialize()}), ex.status

    if isinstance(ex, Exception):
        message = str(ex)
        return jsonify({'errors': [{'message': message}]}), 500

    return ex

if __name__ == '__main__':
    app.run(debug=True)
```

Slika 8. Flask exception handling

D. *Poziv podeljenog koda od strane mikroservisa*

U ovom poglavlju prikazaćemo razvoj dve kranje tačke, a za razvoj oslonićemo se na kod podeljen kako preko paketa, tako i preko mikroservisa.

Na slici 9 prikazana je krajnja tačka mikroservisa preko koga se korisnik registruje. Ovaj mikroservis je implementiran u FastAPI okviru, zbog čega klasa CreateUser (linija 2) preuzima validaciju podataka koju korisnik šalje.

Nakon validacije, proveravamo postojanje korisnika. Ukoliko postoji, vraćamo BadRequestError (linija 11). U suprotnom, inicijalizujemo novog korisnika i postavljamo neophodne parametre. Zatim pokušavamo sačuvati korisnika (linija 24). U bloku od linije 25 do 34, inicijalizujemo objekat UserExchange za prenos podataka i šaljemo asihronu poruku. Ovo omogućava slanje podataka novog korisnika drugim mikroservisima, uključujući EmailWorker servis koji šalje mail dobrodošlice korisniku, što predstavlja asihroni poziv deljenog koda. U slučaju greške prilikom čuvanja korisnika ili slanju poruke, sistem se vraća na prethodno stanje i izbacuje BadRequestError.

Kroz ovaj primer videli smo kako možemo koristiti prilagođene izuzetke koje delimo preko paketa. Takođe videli smo da možemo izvršiti asihroni poziv ka mikroservisu slanjem poruke u nekoliko linija koda, čime se značajno olakšava i ubrzava razvoj.

Slika 9. Krajnja tačka servisa za registraciju korisnika

Na slici 10 prikazana je kranja tačka mikroservisa za kreiranje knjiga. Ova kranja tačka koristi se za kreiranje nove knjige. Prvo kreiramo objekat Books, i postavljamo neophodne vrednosti za ovaj objekat, a zatim dodajemo ovaj objekat u sesiju (od linije 6 do linije 15). Ukoliko postoji lista slika i šaljemo sihroni zahtev za svaku sliku ka photo-upload mikroservisu. Ukoliko otpremanje slike nije moguće, photo-upload mikroservis vraća status 201, nakon čega kreiramo objekat BookImages, i čuvamo metapodatke za tu sliku. Dalje podižemo izuzetak BadRequestError sa porukom koju dobijamo od photo-upload mikroservisa. Kada su slike otpremljene, a metapodaci dodati u sesiju, prelazimo na čuvanje podataka, što možemo videti na liniji 37. Nakon toga pripremamo podatke i šaljemo poruku da je kreirana nova knjiga drugim mikroservisima (od linije 38 do linije 45). Ukoliko dođe do nepredviđenog izuzetka, podižemo BadRequestError sa odgovarajućom porukom.

Otpremanje slika u ovom primeru predstalja sihronu komunikaciju sa mikroservisom, preko koga delimo funkcionalnost. Vidimo da je sihrono pozivanje deljene funkcionalnosti nešto složenije od asihrone. Međutim, sihrona komunikacija je ovde poželjna pošto iz odgovora gradimo metapodatke o slikama koje smo otpremili, i vezujemo ih za knjigu koju želimo da kreiramo.

Slika 10. Servis za kreiranje knjiga

V ZAKLJUČAK

U ovom radu analizirali smo različite strategije deljenja koda u kontekstu arhitekture mikroservisa kako bismo osnažili skalabilan i agilan razvoj softvera. Kroz pregled arhitekture mikroservisa, istakli smo važnost efikasnog deljenja koda kao ključnog elementa ovakvog razvojnog pristupa.

Izazovi i razmatranja predstavljeni u radu ukazuju na kompleksnost procesa deljenja koda u okviru mikroservisne arhitekture, uključujući potencijalne prepreke poput konzistentnosti, kontrole verzija i upravljanja zavisnostima.

Kroz detaljan pregled različitih strategija, uključujući izbegavanje deljenja koda, manuelno deljenje, automatizovano deljenje, deljenje preko paketa i kroz mikroservise, identifikovali smo prednosti i nedostatke svakog pristupa. Ovo nam omogućava da prilagodimo odabir strategije u skladu sa specifičnim zahtevima projekta i kontekstom.

Studije slučaja i primena najbolje prakse pružile su praktične uvid u implementaciju deljenja koda. Kroz analizu tih slučajeva, utvrdili smo koristi od deljenja koda u poboljšanju skalabilnosti, agilnosti i održivosti softvera.

Jedan od problema koji nismo uspeli da rešimo u okviru ovog rada je problem tehnološke zavisnosti. Implementacija paketa preko kojih se deli kod često stvara određenu vrstu jezičke zavisnosti. Na primer, ako koristimo Python za implementaciju paketa, ograničeni smo samo na Python jezik za implementaciju mikroservisa. Ovo može predstavljati izazov u višeznačnim okruženjima gde se preferiraju različiti jezici za različite aspekte sistema.

Tehnološka zavisnost predstavlja predmet daljeg istraživanja u oblasti premošćavanja jezičkih barijera pri deljenju koda u kontekstu mikroservisne arhitekture. Buduće studije bi mogle da istraže mogućnosti za rešavanje ovog problema, kao što su razvoj interoperabilnih rešenja ili standardizacija pristupa deljenju koda preko jezičkih granica.

LITERATURA

[1] Claus Pahl and Jamshidi Pooyan, *Microservices: A Systematic Mapping Study*, 2016, Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016) - Volume 1, pages 137-146

[2] Sam Newman, *Izgradnja mikroservisa* 2022, CET i Računarski fakultet

[3] Daniel Feitosa, Apostolos Ampatzoglou, Antonios Gkortzis, Stamatia Bibi, Alexander Chatzigeorgiou, *CODE reuse in practice: Benefiting or harming technical debt*, Journal of System and Software, Vol 167, 2020

[4] Bellis, Scott, and Chunxin Xu, *Don't Share Libraries among Microservices*, Philipp Hauer's Blog, 2016, dostupno na: https://phauer.com/2016/dont-share-libraries-among-microservices/

[5] Aviv Carmi, *Code Sharing in Microservices Architectur*e, 2022, HUMAN Security, dostupno na: https://www.humansecurity.com/tech-engineering-blog/code-sharing-in-microservices-architecture

[6] Mark Richards, Microservices AntiPatterns and Pitfalls, 2016, O'Reilly Media

[7] Tom Černý, Abdullah Černý, Andrea Černý, Davide Taibi, Microservice Anti-Patterns and Bad Smells. How to Classify, and How to Detect Them. A Tertiary Study, 2023, SSRN Electronic Journal. 10.2139/ssrn.4328067.

[8] *Shared Libraries - Design and Best Practices*, Duda Medium, 2022, dostupno na: https://medium.com/duda/shared-libraries-design-and-best-practices-710774ae0bdc

[9] Using Shared Libraries in a Microservice Architecture: Dos and Don'ts, Dialectica, 2022, dostupno na: https://dialecticanet.com/blog/using-shared-libraries-in-a-microservice-architecture-dos-and-donts/

[10] How To Package Your Python Code — Python Packaging Tutorial, dostupno na:

https://python-packaging.readthedocs.io/en/latest/index.html
[11] Introduction to Pika - pika 1.3.2 documentation, dostupno na: https://pika.readthedocs.io/en/stable/

ABSTRACT

Content — This paper presents code sharing strategies in microservices architecture aimed at improving scalability and software development agility. In the first part of the paper, we discuss the importance of code sharing and the challenges of code sharing in microservices architecture. Then, we analyze in detail currently popular methods for code sharing. Finally, through a case study, we demonstrate some of the code sharing strategies in a practical manner.

Keywords — microservices, packages, code sharing strategies, synchronous communication, asynchronous communication.

CODE SHARING STRATEGIES FOR MICROSERVICES: EMPOWERING SCALABLE AND AGILE SOFTWARE DEVELOPMENT

Name of authors:
MARKO MURIĆ, BOJANA DIMIĆ SURLA