Matematički fakultet
Univerzitet u Beogradu

Master rad

Mreža mikroservisa sa distribuiranom bazom podataka u okruženju Flask

Student:
Marija Ševković

Mentor:
prof. dr Vladimir Filipović

Članovi komisije:
prof. dr Filip Marić
dr Aleksandar Kartelj

Beograd 2020.

Sadržaj

1 Uvod

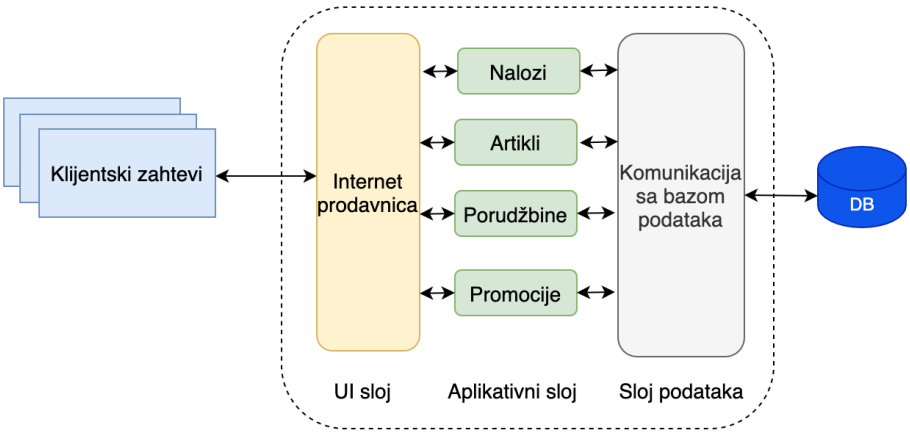
Mikroservisna arhitektura je trenutno najpopularniji model arhitekture kod modernih veb aplikacija. Mirkoservisi su pogodni jer omogućavaju podelu odgovornosti unutar aplikacije i između razvojnih timova koji ih implementiraju, tako da timovi mogu samostalno odlučivati o primeni odgovarajućih tehnologija. Python odlikuje jednostavna sintaksa, relativno kratak period razvoja komponenti (kao što su mikroservisi) u poređenju sa drugim objektno orjentisanim jezicima i veliki skup biblioteka koje su dostupne na zvaničnom indeksu paketa (PyPI). Za ovaj jezik postoji više popularnih razvojnih okruženja (eng. framework), a u ovom radu biće prikazano Flask mikro okruženje.

Radi bolje ilustracije gore pobrojanih pojmova, osmišljena je i razvijena posebna aplikacija, koja kreira mrežu povezanih mini servisa, pri čemu svaki od njih funkcioniše slično kao Twitter. Svaki servis je nezavisan objekat sa sopstvenom bazom podataka, jedinstvenim identifikatorom i svestan je postojanja drugih servisa u mreži, može komunicirati sa njima i pretražiti podatke iz njihovih baza podataka. Takođe, implementacija svakog servisa je ista dok su početne konfiguracije različite.

Aplikacija je nazvana Seven Tweets i na raspolaganju se kao slobodan softver pod licencom Apache 2 nalazi na adresi [11]. Aplikacija je napisana u Python3 programskom jeziku, Flask okruženju i koristi Postgresql bazu podataka. Pokretanje aplikacije moguće je na drugim tipovima servera koji su pogodniji za produkcijsko okruženje, kao što je wsgi http server Gunicorn. Uz softver dolaze testovi aplikacionog interfejsa i jedinični testovi napisani u Pytest-u.

2 Mikroservisna arhitektura

Arhitektura programskog sistema se može definisati kao skup komponenti i način na koji su one povezane. Kreiranjem arhitekture treba razdvojiti zahteve i funkcionalnosti programskih komponenti i njihove veze, koristeći neki od razvojnih pristupa. Kod razvoja veb aplikacija uobičajeno je aplikaciju strukturirati u više delova ili slojeva. Najčešći izbor je troslojna arhitektura koja aplikacijske komponente razdvaja na prezentacijski (http), aplikativni ili biznis sloj i sloj podataka. Prezentacijski sloj je u direktnom kontaktu sa korisnikom, pruža mu potrebne informacije i zadužen je za celu interakciju sa njim. Sloj u sredini ili aplikacijski sloj, zadužen je za upravljanje aktivnostima koje pojedina veb aplikacija treba da izvrši. On je takođe zadužen i za komunikaciju prezentacijskog i sloja podataka pa on upravlja tokom operacije. Najniži sloj troslojne arhitekture je sloj podataka. On sadrži informacije o podacima i logiku pristupa tim podacima. Slojevito razdvajanje aplikacije olakšava razvoj funkcionalnosti i njeno održavanje. Broj slojeva arhitekrure može biti proizvoljan mada je 3-slojna arhitektura najčešća i ona je prikazana na slici 1. Primer proizvoljnih slojeva može biti sloj osnovnih servisa koji se često nazivaju platforma.

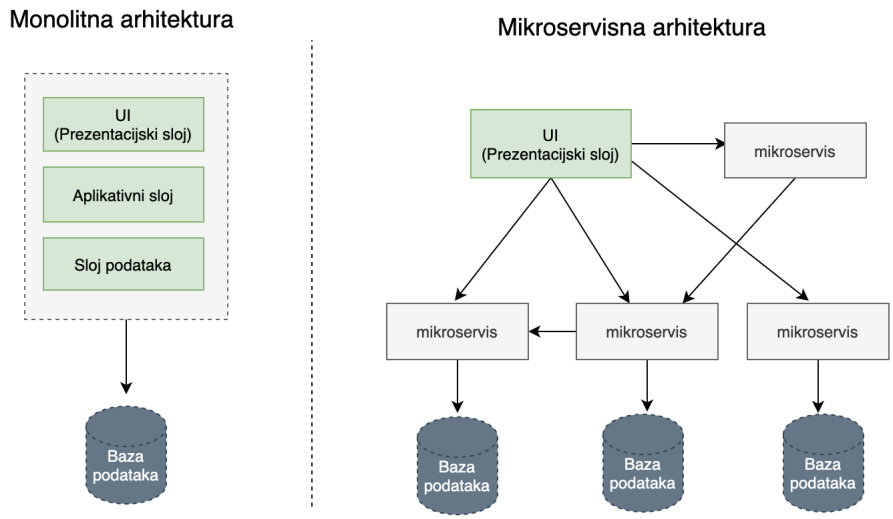


Slika 1: Primer troslojne arhitekture aplikacije

Mikroservisnu arhitekturu čini više malih servisa, zvanih mikroservisi. To su nezavisne programske komponente koje pružaju usluge drugim servisima i sa njima komuniciraju. Cilj razvoja servisa u mikroservisnoj arhitekturi je da konačni servis bude što manji, optimizovaniji i fokusiran isključivo na svoju funkcionalnost koja treba da bude što atomičnija. [1] Servisi su alternativno rešenje monolitnim aplikacijama koje u sebi sadrže sav kod zadužen za kompletnu interakciju sa korisnikom i bazom podataka. Vremenom, kako se aplikacija razvija, količina programskog koda sve više raste. Svaka promena postaje otežana i troši više vremena zbog sve većeg broja linija koda. Iako se pri razvoju koristi

neki alat za strukturiranje koda unutar programske komponente, komponente i njihovi odnosi postaju sve komplikovaniji. Obično funkcije i metode povezuju više domena odjednom, što dosta otežava implementaciju nove funkcionalnosti ili otklanjanje grešaka.

Ukoliko se pri razvoju monolitne aplikacije razmišljalo o tome da su zajednički koncepti čvrsto povezani i da čine smislenu celinu, tada je prelazak iz postojeće monolitne arhitekture u mikroservisnu arhitekturu nešto jednostavniji. Da bi izdvojili mikroservis iz monolitne arhitekture treba prvenstveno precizirati njegov domen tj izdvojiti najmanju atomičku funkcionalnost koju bi mogao da obavlja. Kako bi lakše odredili taj domen, prvo treba sagledati postojeći sistem i njega podeliti na smislene logičke celine. Iz dobijenih celina treba nastaviti sa daljim izdvajanjem manjih celina sve dok se ne dođe do određenog nivoa kompleksnosti. Tada se može razmatrati kako celine izdvojiti i napraviti male nezavisne servise koji međusobno komuniciraju. Na slici 2 može se videti uporedni prikaz monolitne i mikroservisne arhitekture.



Slika 2: Primer monolitne i mikroservisne arhitekture aplikacije

Autonomija servisa se čuva kontrolom svega što pružaju servisi aplikacije. Oni trebaju imati nezavisan mehanizam isporuke koda (eng. deploy) i nezavisnu komunikaciju sa bazom podataka. Ukoliko servis deli previše svoje funkcionalnosti, može doći do negativnog efekta da klijenti postanu zavisni od interne logike tog servisa. Servisi koriste aplikacioni interfejs (eng. application programming interface-API) za komunikaciju sa drugim servisima putem mreže. On se može predstaviti i kao zajednički jezik između servisa. Mikroservisi u okviru jednog sistema mogu biti napisani u različitim programskih jezicima ali bi trebalo da imaju API klijente napisane u jezicima drugih servisa koji sa njime komuniciraju. Odabir tehnologije u kojoj će biti napisan specifican servis nije lak posao jer treba uzeti u obzir dosta faktora. Neki od najbitnijih su prilagođenost odre-

đenog programskog jezika samom domenu servisa - na primer ne treba koristiti interpreterski jezik za servis koji treba dosta brzo da izvršava kod i da odgovara na veliki broj zahteva od strane drugih servisa. Dužina implementacije servisa i održavanje su takođe bitni faktori naročito u situaciji kada su dati fiksni vremenski okviri. Omogućavanje odabira odgovarajuće tehnologije za svaki zadatak, skraćivanje vremena razvoja i poboljšane performanse aplikacije su velika prednost u odnosu na tradicionalnu monolitnu arhitekturu gde tehnologija koja se koristi u celoj aplikaciji možda nije najbolji alat za neki zadatak ili ta funkcionalnost uopšte nije podržana. U tom slučaju potrebno je implementirati odgovarajući algoritam u korišćenoj tehnologiji. [2]

2.1 Prednosti mikroservisne arhitekture

Aplikacija sa mikroservisnom arhitekturom je podeljena na nezavisne komponente zvane mikroservisi koji zajedno sarađuju da bi omogućili izvršavanje svih funkcionalnosti aplikacije. Svaki mikroservis obično enkapsulira svoju biznis logiku koja je zadužena za pojedinačnu funkcionalnost aplikacije. Oni se mogu nezavisno skalirati (na gore ili dole), testirati i mogu imati nezavisan mehanizam isporuke koda na produkcijskom okruženju. Glavne prednosti ovog tipa arhitekture su:

- Agilnost.
- Autonomnost servisa.
- Atomička funkcionalnost - visoka kohezija i slabo povezivanje.
- Nezavisnost od OS platforme.
- Moguće je različite servise implementirati u različitim tehnologijama i programskim jezicima.
- Razvoj koda i isporuka su nezavisni za svaki servis.
  - Ovo omogućava lakše obezbeđivanje neprekidne integracije (eng. continuous integration) i bržu isporuku novih funkcionalnosti.
  - Omogućava pravljenje velikih skalabilnih sistema i granularno planiranje isporuke nove verzije aplikacije.
- Skaliranje na nivou servisa.
- Jednostavnije baze podataka koje nisu deljene između servisa.

Jedna od velikih prednosti je i to što razvojni timovi mogu biti manji i orjentisani na pojedinačne servise. Za rešavanje problema mogu koristiti tehnologiju za koju se proceni da se najbolje uklapa u potrebe specifičnog servisa.

2.2 Razlike i sličnosti sa SOA

Kao glavna razlika između mikroservisne arhitekture i SOA obično se navodi to što SOA ne daje programerima i ostalim razvojnim inženjerima smernice kako od monolita napraviti servisnu arhitekturu. Ne definišu se striktne granice i predviđena veličina pojedinačnog servisa, niti koje bi granulacije trebala da ima nova arhitektura. Na sve te odluke delom utiču i proizvođači programskih okruženja svojim izborom tehnologija i implementacijom SOA-e. Kod SOA-e su često izostavljeni praktični primeri, kako pravilno odvojiti servise jedne od drugih, a upravo iz toga proizilaze problemi povezani sa SOA-om. Mikroservisna arhitektura se razvila iz stvarnih primera te se može, na neki način, smatrati posebnim SOA pristupom. [2] U tabeli 1 mogu se videti osnovne razlike između ove dve arhitekture.

Mikroservisna arhitektura	**SOA**
Razdvajanje funkcionalnosti i ograničavanje konteksta servisa	Maksimalna iskorišćenost istog servisa
Dodavanje nove funkcionalnosti uglavnom znači dodavanje novog servisa	Uglavnom se zahteva izmena već postojecih servisa
Veliki fokus na DevOps i Continuous Delivery	Popularno ali ne i neophodno
Koristi jednostavne protokole poput HTTP-a, REST-a ili RPC-a i jednostavne poruke. Često se koriste i brokeri poruka kao sto je RabbitMQ	Za komunikaciju se koristi Enterprise Service Bus i podržava nekoliko protokola poruka
Obično se koriste Cloud platforme i kontejneri za pokretanje aplikacija	Zajednička platforma za sve servise
Svaki servis može imati svoju bazu podataka	Deljenje zajedničke baze podataka

Tabela 1: Prikaz razlika između SOA i mikroservisne arhitekture

2.3 Dizajniranje mikroservisa

Posmatranjem servisa u mikroservisnoj arhitekturi može se zaključiti da su im zajedničke osobine slaba povezanost i visoka kohezija.

Slabo povezivanje (eng. Loose coupling) u mikroservisnoj arhitekturi se koristi da bi imali mogućnost menjanja jednog servisa bez uticaja ili izmena drugog servisa pa je to jedna od najvažnijih prednosti. Slabo povezani servisi ne znaju puno jedni o drugima, čak i kada neposredno sarađuju. Ako se pri razvoju ne uzme u obzir da se sva komunikacija odvija kroz mrežu, tada može doći do problema s performansama i bitno je pripaziti na učestalost razmene informacija između servisa. Velika fluktuacija informacija između dva servisa dovodi do čvršće povezanosti između njih, što je isto suprotno od onoga što se želi postići. Cilj je postići visoku koheziju (eng. High cohesion) i grupisanje logike programa u povezane celine. Svaka celina sadrži logiku i ponašanje koje je usko vezano za njen kontekst, dok je sve ostalo, što nije direktno vezano za taj kontekst, u drugoj celini. Granice celina kreiraju se prema njihovim kontekstima. Te celine pretvaraju se u servise pa se zahtev za promenom ponašanja pojedinog servisa menja samo na jednom mestu i to ubrzava isporuku promene na produkcijsko okruženje (eng. production environment). U slučaju da se funkcionalnost proteže kroz više servisa, kao što je to često slučaj kod lošije grupisanih celina, za promenu funkcionalnosti treba izmeniti nekoliko servisa i možda ih pustiti u produkciju u isto vreme. Menjanje većeg broja servisa odjednom drastično povećava rizik od greške pa je time postupak izmene puno teži, budući da se izvodi na nekoliko servisa.

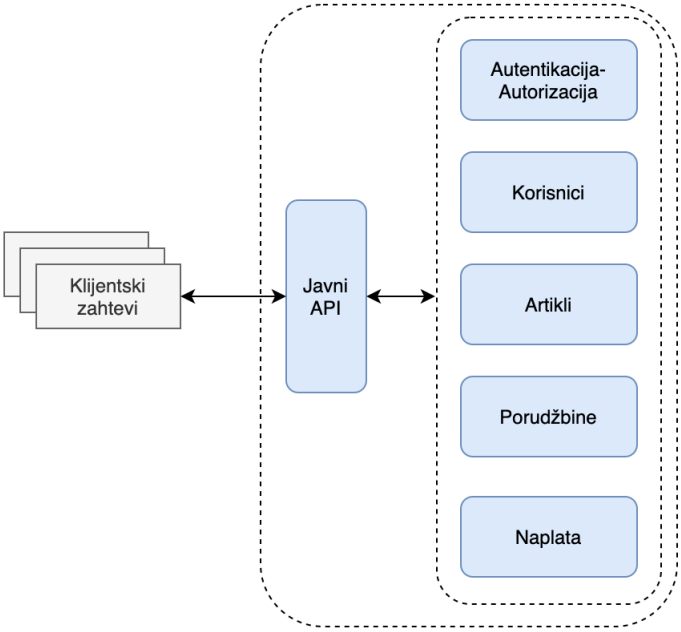
2.4 Ograničenje domena servisa

Dobra definicija konteksta je bitan korak prilikom dizajniranja servisa. Treba izdvojiti informacije koje su bitne samo u okviru pojedinačnog konteksta i one koje treba da budu deljene izvan njega. Skup funkcija kojima se bavi svaki kontekst treba da bude sveden na minimum, idealno bi to bio jedan srodan skup metoda.
Za primer se može uzeti platforma za prodaju stvari preko interneta. Definisanje konteksta kod jednostavnog scenarija internet prodavnice može izgledati ovako:

- Domen logovanja korisnika i njegovih prava pristupa - autentikacija i autorizacija.
- Domen podataka o korisnicima.
- Domen podataka o artiklima.
- Domen podataka o porudžbinama korisnika.
- Domen naplate usluga i artikala.

Iz gore navedenog može se zaključiti da buduća platforma treba da ima pet mikroservisa od kojih svaki implemetira po jedan domen. Klijentima ne bi

trebalo dopustiti direktan pristup ovim mikroservisima i zbog toga treba dodati Javni API platforme koji će biti mesto kontakta za klijente. Slika 3 prikazuje jednu takvu platformu.



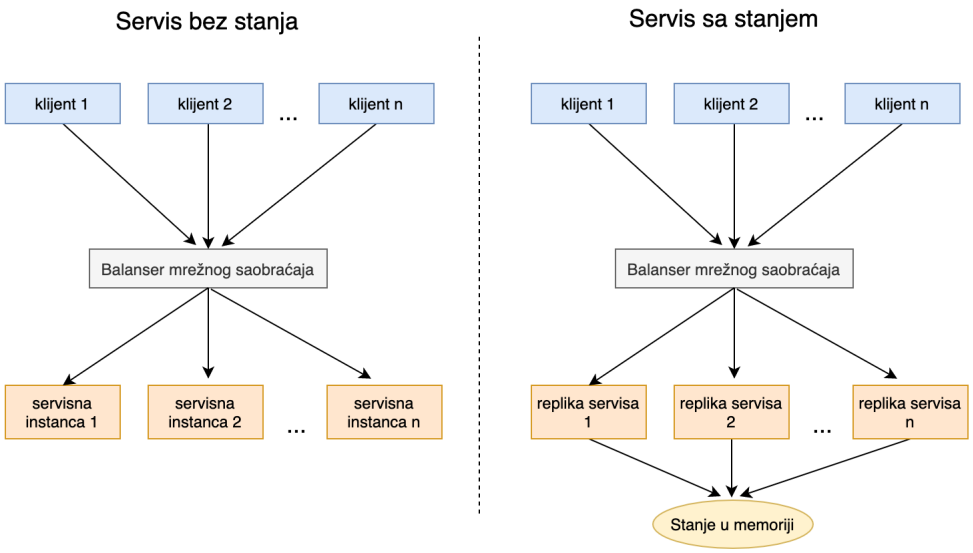
Slika 3: Primer platforme za prodaju stvari preko interneta

2.5 Servisi sa i bez stanja

U ovom poglavlju biće prikazane osnovne razlike između Servisa bez stanja (eng. Stateless) i Servisa sa stanjem (eng. Statefull). Ključna razlika je u tome što Servisi sa stanjem zahtevaju čuvanje podataka na samom servisu koji opslužuje zahteve i to je preduslov za njihovo ispravno funkcionisanje. Sa druge strane, Servisi bez stanja obrađuju informacije dostupne u samom zahtevu (eng. request payload) kao i informacije iz posvecenog Servisa sa stanjem, npr spoljašnje baze podatka. [3]

Servisi bez stanja (eng. Stateless) Ovo je pristup u razvoju mikroservisa koji je jednostavniji za implementaciju i većinski je zastupljen. Mogu postojati problemi sa sporijim odgovorom na zahteve jer se podaci dohvataju sa eksternog izvora ali oni se u većini slučajeva daju rešiti uvođenjem keš memorije. Važne karakteristike ovog pristupa:

- Procesiranje na osnovu podataka u samom zahtevu. Servis ne mora da čuva informacije između zahteva ali to može uraditi u spoljašnjem servisu kao što je baza.



Slika 4: Skaliranje servisa kod servisa bez stanja i servisa sa stanjem

- Moguće je imati više instanci servisa koje paralelno obrađuju zahteve.
- Različiti zahtevi mogu biti procesirani od strane različitih instanci servisa.
- Balansiranje zahteva (eng. Load balancing) tj raspodela zahteva ravnomerno po instancama značajno olakšava skaliranje i poboljšava performanse servisa. Za ovo se koriste posebne aplikacije kao što je na primer Nginx.

Servisi sa stanjem (eng. Statefull) Ovaj tip pristupa može da ima odlične performanse u specifičnim scenarijima gde se zahteva brzina u obradi podataka. Primer za to su baze podataka i kompjuterske igre. Postoji određeni nivo kompleksnosti prilikom skalarinja jer je potrebno napraviti replike servisa i podataka. Karakteristike:

- Procesiranje zahteva na osnovu sačuvanih podataka iz prethodnih zahteva.
- Obično ista instanca servisa mora da procesira više zahteva koji su povezani zajedničkim stanjem ili to stanje mora biti podeljeno sa drugim replikama servisa koje traže pristup.
- Skaliranje je moguće pravljenjem vise replika servisa.

Na slici 4 prikazan je način skaliranja oba tipa servisa.

3 Dizajniranje API-ja

3.1 REST

Većina modernih veb aplikacija ima API (eng. Application Programming Interface) koji klijenti koriste prilikom interakcije sa aplikacijom. U 2000-tim Roy Fielding je predstavio novi pristup u dizajnu veb servisa koji je nazvao REST (eng. Representational State Transfer). Taj dizajn bio je namenjen prvenstveno za distribuirane sisteme zasnovane na hipermediji. REST nije nužno zavistan od HTTP-a i drugih protokola ali najčešća njegova implementacija koristi HTTP kao aplikacisjki protokol. Velika prednost REST-a je što koristi otvorene standarde i ne vezuje implementaciju API-ja ili klijenta aplikacije za specifične tehnologije. Na primer, REST veb servis može biti napisan u Pythonu, a klijentske aplikacije mogu iskoristiti bilo koji jezik ili alat koji generiše HTTP zahteve da isparsiraju odgovor od pomenutog servisa.

Dobro dizajniran veb api bi trebalo da podržava:

1. Nezavisnost od platforme - svakom klijentu bez obzira na njegovu implementaciju je omogućeno da pozove API servisa. To podrazumeva da klijent i servis poštuje standardne protokole komunikacije i imaju usklađene formate razmene podataka.

2. Evoluciju servisa - veb API treba da podržava dodavanje funkcionalnosti nezavisno od klijentskih aplikacija. Proširivanje i razvitak API-ja ne treba da utiče na rad klijenta. Ova odlika se naziva i retroaktivna kompatibilnost (eng. backward compatibility).

U nastavku su navedeni neki od glavnih principa dizajniranja RESTful API-ja:

- REST API je koncentrisan na resurse tj bilo koji tip objekta ili podataka kome klijent pristupa.

- Resurs ima jedinstveni identifikator URI (eng. Unique Resource Identifier). Na primer URI za korisnika na platformi može biti: `https://my-platform.com/v1/users/12`

- Klijenti komuniciraju sa servisom tako što razmenjuju reprezentacije resursa. Mnogi veb API-ji koriste JSON(eng. Java Script Object Notation) format.

- API treba da koristi standardan interfejs da izvrši operacije na resursima. Najviše korišćene metode su: GET, POST, PUT, PATCH i DELETE.

- REST API koristi model zahteva bez stanja (eng. Stateless request model). HTTP zahtevi treba da budu nezavisne i atomičke operacije. To omogućava servisima da postignu veću skalabilnost. Bilo koja instanca servisa može da opsluži bilo koji zahtev od klijenta. Postoje i izuzetci od ovog pravila, to su uglavnom servisi koji služe za uskladištenje podataka.

- Linkovi su zastupljeni u reprezentaciji resursa kao u listingu 1 :

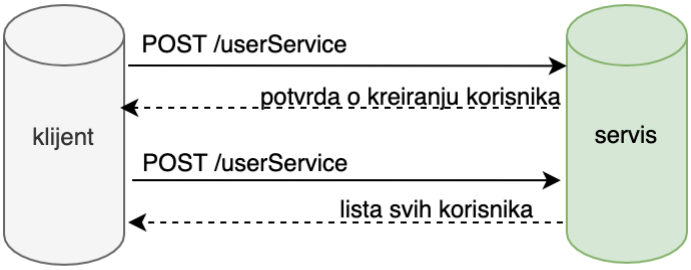
```
{
    "href":"https://my-platform.com/v1/orders/3"
    "orderValue":16.60,
    "customer_id": 123,
    "items": [
        {"href":"https://my-platform.com/v1/item/1", "cost": "1.20"  },
        {"href":"https://my-platform.com/v1/item/5",  "cost": "10"  }
    ]
}
```

Listing 1: Primer reprezentacije porudžbine jedne mušterije

3.1.1 Principi dizajniranja

Tokom 2008. Godine Leonard Richardson je predložio sledeći model za veb API-je:

- Nivo 0 - Početni nivo dizajniranja API-ja, ovde bi trebalo definisati jedan URI i sve operacije ka njemu kao tip POST. Komunikacija na nivou 0 prikazana je na slici 5.
- Nivo 1 - Uvođenje pojma resursa, treba napraviti poseban URI za svaki individualni resurs. Komunikacija na nivou 1 prikazana je na slici 6.
- Nivo 2 - Definisanje operacija nad resursima uvođenjem HTTP metoda i vracanje status kodova u odgovorima na zahteve ka servisu. Komunikacija na nivou 2 prikazana je na slici 7.
- Nivo 3 - Korisćenje hipermedija linkova u odgovorima na takav nacin da korisnik može da zaključi koji URI link i koja metoda mu je potrebna da odradi narednu logičnu akciju. Ovaj princip je poznat i pod akronimom HATEOAS (eng. Hypertext As The Engine Of Application State). Komunikacija na nivou 3 prikazana je na slici 9.

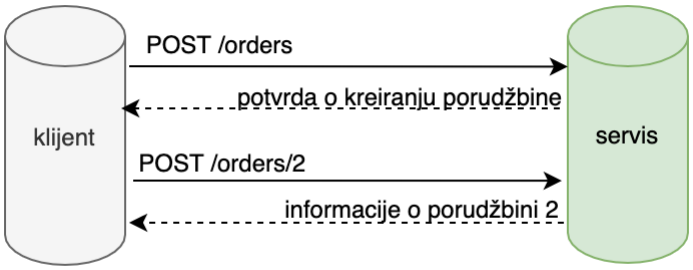


Slika 5: Primer komunikacije na Nivou 0

3.1.2 Organizacija oko resursa

Da bi organizacija API-ja bila što bolja prvo se treba fokusirati na biznis entitete koji su vidljivi klijentu. Resurs ne mora da bude zasnovan na fizičkom

podatku i ne mora tačno da preslikava entitete iz baze podataka. Obično je resurs koji se predstavlja klijentu kao jedinstven entitet, implementiran u bazi kroz nekoliko tabela. Stvaranje API-ja koji preslikava internu strukturu tabela iz baze, treba izbegavati i klijent ne bi trebalo da bude upoznat sa njom iz sigurnosnih razloga. Ovim je klijent i zaštićen od budućih promena u šemi baze podataka. Ukoliko je moguće, URI resursa treba da bude baziran na imenicama, a ne glagolima. [4]



Slika 6: Primer komunikacije na Nivou 1

Primer URI-ja na servisu internet prodavnice:

- Ispravan dizajn: `https://online-shop.com/orders`
- Pogrešan dizajn: `https://online-shop/create-order`

Entiteti su često grupisani u kolekcije, kao što su narudžbine i kupci. Kolekcija je odvojen resurs od pojedinačnog entiteta u kolekciji i treba da ima svoj URI. Sledeći URI treba da predstavlja kolekciju narudžbina:

`https://online-shop.com/orders`

Slanje HTTP GET zahteva na prethodni URI vraća sve porudžbine iz kolekcije. Svaki element kolekcije odnosno pojedinačna narudžbina ima svoj URI, a HTTP GET zahtev na njega vraća detalje o toj porudžbini.
Bitno je i usvojiti konzistentnu konvenciju imenovanja prilikom određivanja URI-ja. Generalno se koriste množine imenica za URI-je koji referenciraju na kolekcije. Takođe je dobra praksa i organizovati ih u hijerarhiju. Na primer */customers* je deo putanje ka kolekciji kupaca, a */customers/id* je deo putanje ka konkretnom kupcu. Za predstavljanje ID-a u ovom slučaju može se koristiti ili tip Integer ili tip UUID.

- URI kolekcije: `https://online-shop.com/customers`
- URI kupca: `https://online-shop.com/customers/3`

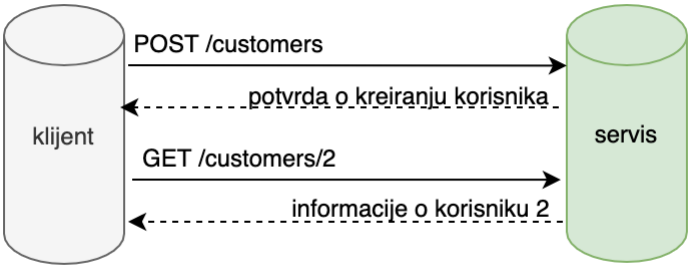
Takođe treba razmotriti veze između različitih tipova resursa i kako prikazatii te veze spoljnjem svetu. Na primer */customers/5/orders* može predstaviti sve narudžbine kupca sa identifikatorom 5, a sa druge strane slična veza može

biti predstavljena sa */orders/99/customer* koja povezuje narudžbinu sa identifikatorom 99 sa kupcem. Identifikator kupca bi našli u odgovoru na zahtev sa prethodnim delom putanje. U kompleksnijim sistemima može se izgraditi URI koji omogućava klijentu da se navigira kroz nekoliko nivoa kao */customers/1/orders/2/products*. Ipak ovakvo proširivanje modela može postati teško za implementaciju i održavanje. Bolje rešenje je korišćenje navigacionih linkova hipermedije u telu odgovora o čemu će biti reči u narednim poglavnjima.
Na kraju, ako nije moguće mapirati svaku operaciju koju implementira API na specifičan resurs, mogu se koristiti HTTP zahtevi koji startuju funkcije i vraćaju rezultat u obliku HTTP poruke. U sledećem primeru prikazan je veb API jednostavnog kalkulatora koji sabira brojeve i predstavlja operaciju sabiranja kao pseudo resurs, a parametre prima na sledeći način: */add?operand1=99&operand2=1*.

3.1.3 Definisanje operacija preko HTTP metoda

Protokol HTTP-a definiše nekoliko metoda koje pridružuju semantičko značenje HTTP zahtevu. Najčešće korišćeni metodi u većini RESTful API-ja su:

- GET - izvlači reprezentaciju resursa sa specifičnim URI-jem. Telo odgovora sadrži informacije o resursu.
- POST - kreira novi resurs. Telo zahteva sadrži detalje novog resursa. Ova metoda se koristi i kao pokretač za određene akcije koje ne stvaraju nove resurse.
- PUT - zamenjuje postojeci resurs na zadatom URI-ju sa novim čiji su detalji prosleđeni u telu zahteva.
- PATCH - menja određene atribute resursa koji su specificirani u telu zahteva.
- DELETE - briše resurs.



Slika 7: Primer komunikacije na Nivou 2

U tabeli 2 su prikazane ustaljene prakse RESTful veb API-ja kroz primer online prodavnice:
Ponekad razlike između POST, PUT i PATCH metoda mogu biti zbunjujuće ali bitno je zapamtiti da PUT zahtev mora biti idempotentan tj da rezultalt uvek mora biti isti iako se zahtev prosledi više puta. To ne mora važiti za POST i PATCH metode. [5]

Resurs	POST	GET	PUT	PATCH	DELETE
/customers	kreiranje nove mušterije	listanje svih mušterija	grupno ažuriranje mušterija	grupno delimično ažuriranje mušterija	brisanje svih mušterija
/customers/1	greška	dohvata informacije o mušteriji 1	ažurira sve podatke mušterije 1	ažurira neke podatke mušterije 1	briše mušteriju 1
/customers/1/orders	pravi novu porudžbinu za mušteriju 1	lista sve porudžbine mušterije 1	grupno ažuriranje porudžbina mušterije 1	grupno ažuriranje delova porudžbina mušterije 1	briše sve porudžbine mušterije 1

Tabela 2: Primer upotrebe različitih http metoda

3.1.4 Formati HTTP protokola

Kao što smo ranije pomenuli, klijent i server razmenjuju reprezentacije resursa. U HTTP protokolu formati su određeni kroz media tipove. Za podatke koji nisu binarni, većina API-ja podržava tip JSON (media type = application/json) i XML tip (media type = application/xml). Content-type zaglavlje zahteva i odgovora određuje format reprezentacije. Primer :

```
POST https://online-shop.com/orders HTTP/1.1
Content-Type: application/json; charset=utf-8
Content-Length: 57

{"Id":1,"Name":"Gizmo","Category":"Widgets","Price":1.99}
```

Ako server ne podržava prosleđeni tip medije trebalo bi da vrati status kod 415(Unsupported Media Type). Klijentski zahtev može sadržati i Accept zaglavlje koje sadrži listu tipova koje klijent prihvata u odgovoru servera:

```
GET https://online-shop.com/orders/2 HTTP/1.1
Accept: application/json
```

Ako server ne podržava nijedan od navedenih tipova trebalo bi da vrati status kod 406(Not Acceptable).

GET metod - Na uspešan zahtev obično vraća status 200 (OK). Ako resurs nije pronađen vraća 404 (Not Found).

POST metod - Na kreiranje novog resursa vraća status kod 201(Created). URI novog resursa je uključen u Location zaglavlje odgovora i telo odgovora sadrži reprezentaciju resursa. Ako metod pokreće neku akciju, a ne stvaranje resursa, može se vratiti status kod 200 ili 204(No Content) ako nema tela odgovora. Ako klijent pošalje nevalidne podatke u zahtevu, server bi trebalo da

vrati 400(Bad Request) sa više informacija o grešci u telu odgovora.

PUT metod - Prilikom zamene postojećeg resursa vraća ili 200(OK) ili 204(No Content). Ukoliko zamena nije moguća iz nekog razloga vraća 409(Conflict). Dobra praksa je implementirati grupne operacije ažuriranja, u tom slučaju bi URI specificirao kolekciju, a telo zahteva bi sadržalo detalje o resursima koje treba zameniti. Ovaj pristup dosta poboljšava performanse, jer smanjuje količinu zahteva upućenih ka servisu.

PATCH metod - Menja samo delove resursa, telo zahteva sadrži samo informacije koje treba izmeniti. Najčešće korišćen format je JSON merge patch.
Primer: Ako se uzme da je ovo potpuna reprezentacija resursa:

```
{
    "name":"gizmo",
    "category":"widgets",
    "color":"blue",
    "price":10
}
```

Ovako izgleda mogući JSON merge patch, koji govori servisu da ažurira cenu, obriše boju i doda veličinu, dok ime i kategorija ostaju nepromenjeni:

```
{
    "price":12,
    "color":null,
    "size":"small"
}
```

PATCH metod vraća 200(OK) u slučaju uspeha operacije, u suprotnom jedan od sledećih kodova: 415(Unsupported Media Type), 400(Bad Request), 409(Conflict)

DELETE metod - U slučaju uspešne operacije brisanja veb servis bi trebao odgovoriti sa 204(No Content), a ukoliko resurs nije nađen pa zato ne može biti izbrisan, sa 404(Not Found).

3.1.5 Asinhrone operacije

Ponekad POST, PUT, PATCH i DELETE metode zahtevaju dosta vremena za obradu zahteva i tada ove operacije treba napraviti asinhronim. U tom slučaju treba vratiti status kod 202(Accepted) koji indicira da je zahtev prihvaćen ali nije kompletiran. Odgovor servisa treba da sadrži trenutni status zahteva i URI na kome klijent može proveravati napredak:

```
HTTP/1.1 202 Accepted
Location: /api/status/12345
```

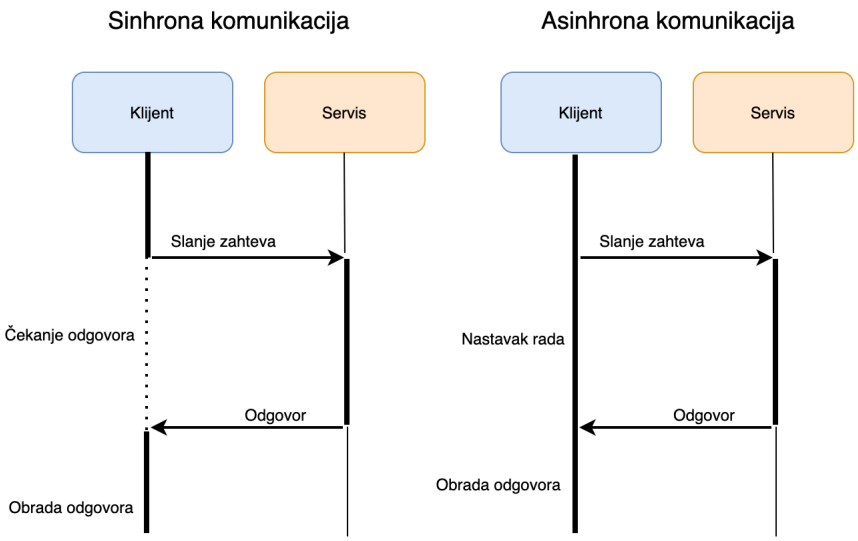
Ako klijent uputi GET zahtev ka prosleđenom URI-ju dodatno se u odgovoru može i naći link preko koga se operacija otkazuje:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "status":"In progress",
    "link": { "rel":"cancel", "method":"delete", "href":"/api/status/12345" }
}
```

Ako je asinhrona operacija napravila novi resurs, status endpoint bi vratio status kod 303(See Other) kada se operacija završi, a Location zaglavlje bi sadržalo URI novog resursa:

```
HTTP/1.1 303 See Other
Location: /api/orders/12345
```

Na slici 8 vidi se uporedni dijagram sinhrone i asinhrone komunikacije između klijenta i servisa:



Slika 8: Komunikacija klijenta i servisa u sinhronom i asinhronom načinu rada

3.1.6 Pretraga i dohvatanje podataka po stranicama

Otkrivanje kolekcije resursa kroz jedan URI povlači za sobom rizik od dohvatanja velike količine podataka iako je samo podskup podatka potreban. Ako se pretpostavi da klijentska aplikacija traži sve porudžbine preko određenog iznosa, preko URI-ja */orders* može dohvatiti sve porudžbine koje će potom filtrirati na klijentskoj strani, ali taj proces je očigledno neefikasan. Umesto toga, servisni

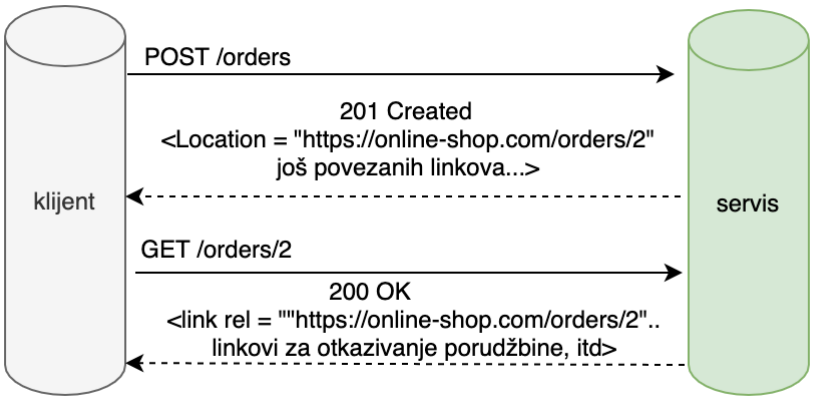
API može prihvatiti parametre za filtriranje podataka preko URI-ja kao */orders?minCost=n*. API je onda odgovoran da vrati samo resurse čija je cena veća od n.

Pošto GET operacija nad kolekcijama potencijalno može vratiti velike količine podataka, one moraju biti ograničene po jednom zahtevu. Poželjno je napraviti parametar kojim se definiše maksimalan broj elemenata za dohvatanje i startna pozicija : */orders?limit=25&offset=50* će vratiti najviše 25 elemenata posle pozicije 50 u kolekciji. Nije loše postaviti i gornju granicu za broj elemenata po stranici da bi se sprečio Denial of Service napad. Da bi olakšali posao klijentskim aplikacijama GET zahtevi sa straničenjem trebalo bi da vraćaju informaciju o ukupnom broju elemenata u kolekciji.[5] Koristan bi bio i parametar za sortiranje resursa po njihovim atributima: textit/orders?sort=ProductID.

Za kraj treba svim opcionim parametrima u URI-ju dati i podrazumevane vrednosti. Na primer prilikom implementiranja straničenja, postaviti parametar limit na vrednost 50, a offset na vrednost 0.

3.1.7 Navigacija do povezanih resursa

Glavna ideja iza HATEOAS-a (eng. Hypertext as the Engine of Application State) je mogućnost navigacije kroz ceo set resursa otkrivenih preko API-ja bez potrebnog prethodnog znanja URI šeme. Svaki GET zahtev trebalo bi da vrati hiperlinkove povezanih resursa i informacije o dostupnim operacijama nad tim resursima. Takav sistem je efektivno mašina sa konačnim stanjem jer odgovor na svaki zahtev sadrži informacije o tome kako preći u sledeće stanje. [4]



Slika 9: Primer komunikacije na Nivou 3

```
{
  "orderID":3,
  "productID":2,
  "quantity":4,
  "orderValue":16.60,
  "links":[
    {
      "rel":"customer",
      "href":"https://online-shop.com/customers/3",
      "action":"GET",
      "types":["text/xml","application/json"]
    },
    {
      "rel":"customer",
      "href":"https://online-shop.com/customers/3",
      "action":"PUT",
      "types":["application/x-www-form-urlencoded"]
    },
    {
      "rel":"customer",
      "href":"https://online-shop.com/customers/3",
      "action":"DELETE",
      "types":[]
    },
    {
      "rel":"self",
      "href":"https://online-shop.com/orders/3",
      "action":"GET",
      "types":["text/xml","application/json"]
    },
    {
      "rel":"self",
      "href":"https://online-shop.com/orders/3",
      "action":"PUT",
      "types":["application/x-www-form-urlencoded"]
    },
    {
      "rel":"self",
      "href":"https://online-shop.com/orders/3",
      "action":"DELETE",
      "types":[]
    }]
}
```

Listing 2: Odgovor servisa gde se vidi reprezentacija veze između kupca i narudžbina

U listingu 2 lista *links* ima set linkova od kojih svaki predstavlja operaciju nad povezanim entitetom i sve informacije da izvrši te operacije kao što su HTTP metod, URI entiteta i podržane tipove. Elemeti koji imaju relaciju *self* odnose se na konkretan resurs koji je dohvaćen tim zahtevom. Set linkova koji se dobija može se menjati u zavisnosti od stanja resursa i na to se misli kada se kaže da je hipertekst pokretač stanja aplikacije.

3.1.8 Verzionisanje API-ja

Veoma je verovatno da će se veb API vremenom menjati. Kako se pojavljuju novi biznis zahtevi, veze između kolekcija se mogu menjati, nove kolekcije dodavati i atributi postojećih entiteta modifikovati. Tada potencijalno postoji problem sa klijentskim aplikacijama koje koriste API. Primarno je onda obezbediti da te aplikacije nastave neometano da rade dok nove klijentske aplikacije imaju mogućnost da iskoriste dodate funkcionalnosti. Verzionisanje dozvoljava veb API-ju da otkrije neki resurs ili funkcionalnost na više načina, a klijentu da bira na koju od verzija će poslati zahtev. Svaki put kada se modifikuje API ili promeni šema resursa, novi broj verzije treba dodati na URI. Prethodne URI

verzije nastaviće da rade kao i pre vraćajući njihove verzije entiteta. [5] U listingu 3 je dat primer verzionisanja API-ja gde je promenjen objekat kupac tj tip polja adresa.

```
https://online-shop.com/v1/customers/3
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Contoso LLC","dateCreated":"2014-09-04T12:11:38.0376089Z","
    address":"1 Street Belgrade"}


https://online-shop.com/v2/customers/3
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8

{"id":3,"name":"Contoso LLC","dateCreated":"2014-09-04T12:11:38.0376089Z","
    address":{"streetAddress":"1 Street","city":"Belgrade","zipCode":11000}}
```

Listing 3: Primer verzionisanja API-ja u slučaju entiteta kupac gde je promenjen tip polja adresa

Ovakav mehanizam verzionisanja je prost ali može postati naporno održavati sve verzije tokom evolucije API-ja. Takođe komplikuje i implementaciju HATEOAS-a. Postoje i drugi načini za verzionisanje API-ja:

- Pomoću parametra u query stringu: https://online-shop.com/customers/3?version=2
- Pomoću zaglavlja:

```
GET https://online-shop.com/customers/3 HTTP/1.1
Custom-Header: api-version=1
```

- Pomoću Media tipa:

```
GET https://online-shop.com/customers/3 HTTP/1.1
Accept: application/vnd.adventure-works.v1+json
```

3.2 RPC

RPC (eng. Remote Procedure Call) je drugačiji pristup dizajniranja veb API-ja od REST-a i sličan je pozivanju funkcija u JavaScriptu. Prima ime funkcije koja se poziva i njene argumete. API je napravljen tako da otkriva javne metode ali u kontekstu HTTP-a, što podrazumeva stavljanje imena metode u URL, a argumente u query string ili telo zahteva.
API-ji zasnovani na RPC-u su dobri u slučajevima kada postoji dosta procedura koje treba izvršiti, dok su REST API-ji dobri za modelovanje resursa i CRUD operacije nad podacima. Većina RPC API-ja koristi samo GET i POST metode, GET za dohvatanje informacija, a POST za sve ostalo. Ustaljeno je videti na primer ovakvo brisanje objekta:

```
POST  /deleteObject
{ "id": 1 }
```

Umesto REST pristupa koji bi bio *DELETE /objects/1*, što i nije neka bitna razlika već je implementacione prirode. Najveća razlika između ova dva pristupa je u načinu kako se te akcije obrađuju. Za primer se može uzeti operaciju slanja poruke korisniku:

```
RPC
POST /SendUserMessage HTTP/1.1
Host: api.example.com
Content-Type: application/json
{"userId": 501, "message": "Hello!"}


REST
POST /users/51/messages HTTP/1.1
Host: api.example.com
Content-Type: application/json

{"message": "Hello!"}
```

Iako izgledaju slično, postoji velika konceptualna razlika u ovim pristupima. RPC ima zadatak da pošalje poruku ali nije definisano šta će se posle desiti sa tim podacima, da li će u bazi završiti cela poruka ili ne. Sa druge strane REST će napraviti novi resurs tipa Poruka povezan sa korisnikom pod id-jem 51. U svakom trenutku poruka se može dohvatiti sa GET pozivom na isti URI, dok to nije slučaj sa RPC putanjom.

U realnim situacijama nije uvek moguće pratiti čist REST dizajn, pa aplikacije kombinuju ova dva pristupa, što im omogućava da dobiju fleksibilniji API koji se lako može nadograditi.

4 Baza podataka

Kod mikroservisne arhitekture najpoželjniji princip integracije sa bazom podataka je da svaki mikroservis ima svoju bazu koju će prilagodoti svojim potrebama. Svaki mikroservis bi trebao da ima direktan pristup samo svojoj bazi, a podatke iz baza drugih servisa bi trebao da dobija preko njihovih API-ja.
Ovaj pristup ima dosta prednosti:

- Domenski podaci su enkapsulirani unutar servisa
- Šema može da se razvija bez direktnog uticaja na druge servise
- Svako čuvanje podataka može individualno da se skalira
- Neuspeh prilikom operacija nad podacima neće direktno uticati na druge servise

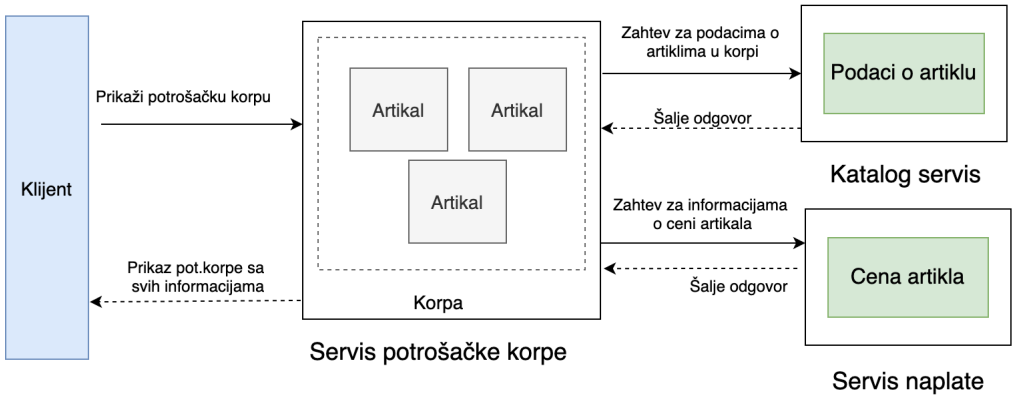
Kod jednostavnih web aplikacija najčešći odabir su SQL relacijske baze podataka, mada se to može i promeniti u toku razvoja i alternativno izabrati neka NoSQL baza. Dok su relacione baze podataka najbolji izbor kod podataka sa kompleksnijom strukturom, NoSQL baze su dobile na popularnosti zbog svoje jednostavnisti i mogućnosti skaliranja.

4.1 Pretraživanje podataka između servisa i tipovi međuservisne komunikacije

Mikroservisi često zahtevaju integraciju sa drugim mikroservisima i ta integracija najčešće uključuje pretragu podataka. Pogledajmo primer sa slike 10 gde je prikazan servis koji dodaje artikle u potrošačku korpu korisnika. Baza podataka ovog mikroservisa sadrži entitet korpe i listu artikala ali ne i detaje o artiklima kao što je cena, tip, itd. Ti podaci su smešteni u katalog servisu i servisu naplate.

Jedna od opcija da se ovaj problem reši je da servis potrošačke korpe napravi HTTP zahteve ka katalog servisu i servisu naplate i dobavi podatke koji mu nedostaju. Takvo rešenje je relativno jednostavno za implementaciju ali nije prikladno za sve situacije. Ti pozivi su uvek sinhroni i blokiraće izvršavanje cele operacije sve dok se odgovor ne vrati ili dok ne istekne čekanje na odgovor od drugog servisa. Postoji rizik da servisi koji su bili nezavisni postanu vezani između sebe, što može poništiti dobrobiti mikroservisne arhitekture. Međutim taj rizik je izražen samo ako scenario uključuje veliki broj poziva ka drugim servisima.

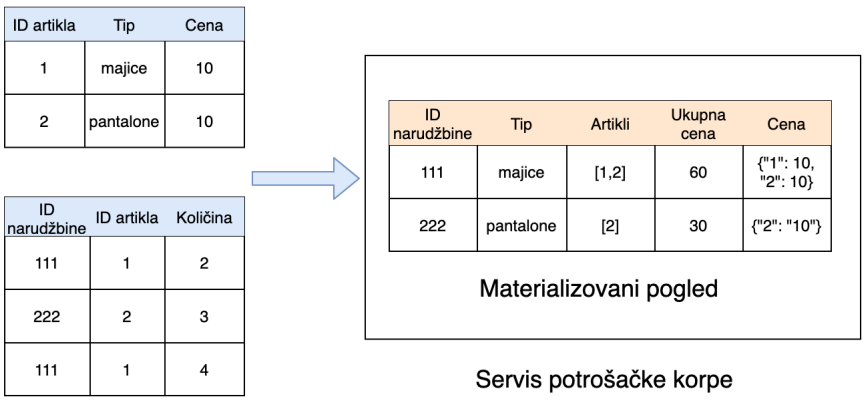
Drugo moguće rešenje je korišćenje materializovanih pogleda. Kod ovog pristupa, servis u lokalu čuva denormalizovanu kopiju podataka sa drugog servisa. Umesto da servis potrošačke korpe pretražuje katolg servis i servis naplate, on ima svoju kopiju tih podataka u lokalu. To elimiše stvaranje nepotrebne međuzavisnosti između servisa (eng. coupling) i poboljšava vreme odziva servisa.



Slika 10: Pretraživanje podataka između servisa - HTTP metoda

Cela operacija se izvršava u okviru jednog procesa.

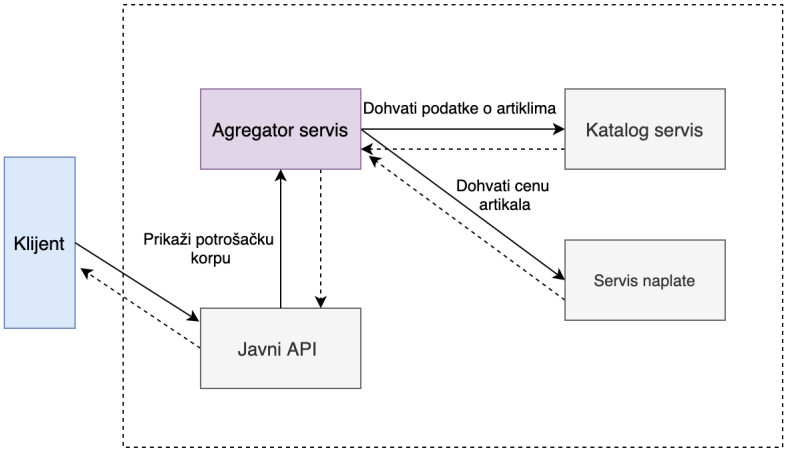
Materializovani pogledi sadrže samo one podatke sa drugih servisa koji su potrebni da bi se određeni upiti izvršili, ponekada je to i samo jedan upit. Oni takođe mogu uključivati i neke prekalkulisane podatke u cilju optimizacije upita. U slučaju da se izvorni podaci promene, pogled se mora ponovo napraviti. Praćenje promena na izvornim podacima može se pratiti automatski u regularnim intervalima ili kada sistem detektuje promenu. Iz tabele koja predstavlja materijalizovani pogled moguće je samo čitanje podataka. Na slici 11 se nalazi primer stvaranja materijalizovanog pogleda.



Slika 11: Primer stvaranja materializovanog pogleda

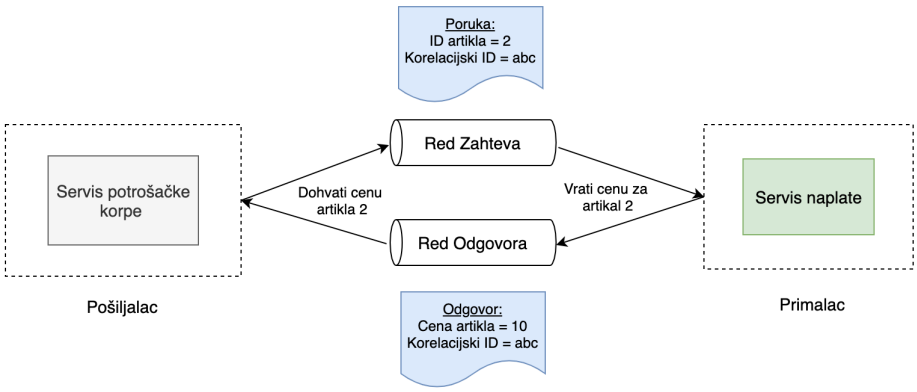
Treća opcija za sprečavanje međuzavisnosti servisa je Agregator servis metoda sa slike 12. Ona funkcioniše tako što izoluje celokupnu operaciju i pozive ka

drugim mikroservisima i centralizuje tu logiku u okviru novog specijalizovanog mikroservisa koji se naziva agregator. Agregator servis orkestrira sve HTTP pozive u sekvencijalnom redu, prikuplja rezultate i vraća ih pozivaocu operacije. Iako koristi HTTP pozive ka ostalim mikroservisima, argegator smanjuje direktnu međuzavisnost komponenti uključenih u izvršavanje operacije.



Slika 12: Pretraživanje podataka između servisa - Agregator servis metoda

U nastavku će biti prikazana još jedna popularna opcija, a to je komunikacija preko redova poruka (eng. message queueing). Razmena poruka preko reda je asinhrona, gde Pošiljalac (eng. Producer) stavlja poruku u red, a Primalac (eng. Consumer) je čita iz reda. Na slici 13 prikazan je tip komunikacije gde Pošiljalac stavlja u red poruku koja sadrži upit o ceni artikla i korelacijski ID. Server Primalac čita tu poruku sa reda izvršava upit i stavlja odgovor u novi red sa istim korelacijskim ID-jem odakle će ga čitati servis Pošiljalac.



Slika 13: Pretraživanje podataka između servisa - Metoda red poruka

5 Programski jezik Python

Python je sveopšti, interpreterski jezik visokog nivoa. Prvi put je objavljen 1991. godine od strane Guido van Rossum-a sa namerom da baci akcenat na čitljivost koristeći dosta karaktera beline za struktuiranje koda. Jezički konstrukti jezika i njegova objektno orjentisanost pomažu pri pisanju koda kako na malim tako i na velikim projektima. Prva značajna revizija jezika desila se 2008. godine pojavljivanjem Python 3.0 verzije koja nije bila potpuno kompatibilna sa dotadašnjom 2.7 verzijom, kojoj je zvanično istekla podrška 2020. godine.

Python je dinamički tipiziran jezik što znači da neka ponašanja ispoljava tek u vreme rada programa, a ne prilikom kompilacije što je slučaj kod statičkih jezika. Postoji više implementacija interpretera za ovaj jezik kao što su:

- CPython - Podrazumevana implementacija u C programskom jeziku.
- Jython - Implementacija Pythona za Java platformu. Neke Java biblioteke se mogu koristi u kombinaciji sa Python klasama.
- IronPython - Implementacija Pythona za .NET platforme, omogućava korišćenje .NET biblioteka.
- PyPy - Implementacija Pythona u Pythonu.

Pored objektno orjentisane i stukturalne paradigme koje su potpuno podržane, Python delimično podržava i funkcionalnu i aspektno orjentisanu paradigmu. On takođe poseduje i automatsko čišćenje memorije (eng. garbage collection) koje se oslanja na brojanje referenci objekata i detekciju ciklusa kao i obimnu standardnu biblioteku koja sadrži dosta paketa za naučna izračunavanja, obradu podataka, itd. Python je dizajniran da bude proširiv, pre nego da sve funkcionalnosti budu u jezgru jezika. [6] To omogućava lako dodavanje modula u postojećim aplikacijama.
Glavna filozofija jezika je opisana u dokumentu Zen of Python prikazanom ispod:

```
Lepo je bolje nego ružno
Eksplicitno je bolje nego implicitno
Jednostavno je bolje nego složeno
Složeno je bolje nego komplikovano
Ravno je bolje nego ugnježdeno
Retko je bolje nego gusto
Čitljivost je najvažnija
Posebni slučajevi nisu dovoljno posebni da krše pravila
Ipak je praktičnost važnija od čistoće
Greške nikad ne treba prećutati ...osim ako se eksplicitno ne prećuti
Kad se pojavi dvosmislenost, ne treba pogađati
Trebalo bi da postoji jedan - poželjno i samo jedan - očigledan način da se nešto uradi
.. mada taj način ne mora da bude vidljiv na prvi pogled, osim ako niste Holanđanin
Sada je bolje nego nikada... mada je nikada često bolje nego upravo sada
Ako je implementaciju teško objasniti, ideja je loša
Ako je implementaciju lako objasniti, mora da je ideja dobra
Prostori imena su sjajna ideja - hajde da ih napravimo još!
```

Već godinama unazad Python je jedan od najpopularnijih programskih jezika. Neki od razloga za to su :

- Brz razvoj aplikacija - Dinamičko tipiziranje i vezivanje omogućavaju brz razvoj - RAD (eng. Rapid Application Development) pristup u agilnom razvoju softvera gde je cilj plasirati MVP (eng. Minimum Viable Product) što je pre moguće, a kasnije praviti poboljšanja na osnovu kritika korisnika.
- Laka integracija - Integracija sa drugim komponentama je olakšana zbog modularne prirode jezika.
- Dostupnost biblioteka - Python ima jako veliku standardnu biblioteku sa preko 250 000 paketa.
- Udoban ciklus razvoja - Ciklus pisanja i testiranja koda je olakšan sa interpreterom.
- Velika produktivnost - Jednostavna sintaksa jezika čini ga lakim za učenje, rad i razumevanje koda.

Međutim Python nije najadekvatniji jezik u svim situacijama. Ukoliko je aplikacija dosta vezana za procesor i većinu vremena troši na računanje stvari onda Python nije dobar izbor. Isto važi i za sve aplikacije gde je brzina izvršavanja presudan faktor. Interpreterski jezici su generalno sporiji od kompajliranih jezika i imaju manje sposobnosti paralelizacije poslova.
Razvoj i održavanje jezika je pod nadzorom neprofitne organizacije Python Software Foundation koja poseduje prava intelektualne svojine. Zvanična dokumentacija može se pronaći ovde [7].

5.1 Flask okruženje

Flask je mikro okruženje (eng. microframework) za programski jezik Python koje u svom jezgru nudi bazične funkcionalnosti veb aplikacije. Lak je za učenje i nadograđivanje pomoću ekstenzija i mnoštva Python biblioteka. Zbog njegove minimalističke filozofije obično je potrebno koristiti dodatne alate kako bi izgradili punu funkcionalnost veb aplikacije. Fleksibilnost je glavna odlika ovog mikro okruženja.

Slika 14: Logo okruženja Flask

Flask je dobar izbor u situacijama kada inicijalno nije jasna arhitektura buduće aplikacije i kada se zna da je potrebno doneti još tehničkih odluka, a nije poželjno ograničiti se odlukama koje bi morale da se donesu ranije sa nekim težim okruženjem kao što je Django.[8] Flask nudi dosta opcija i može se koristiti:

- Prilikom eksperimentisanja sa arhitekturom, bibliotekama i najnovijim tehnologijama.
- Kada nije poznat broj budućih mikroservisa.
- U slučaju da postoji mnoštvo malih i različitih funkcionalnosti u aplikaciji.
- Kada je poznato da će postojeća aplikacija u bliskoj budućnosti morati da se deli na više mikroservisa.
- Kada inicijalno nije poznato koje okruženje i biblioteke su potrebne.

Jedna od velikih prednosti ovog okruženja je jednostavnost razvojnog procesa aplikacije. Flask takođe ima bolje performanse od nekih težih okruženja jer je kompaktniji i ima manje slojeva. Kod kompleksnijih aplikacija koje se dosta oslanjaju na biblioteke i kojima je bitna stabilnost, Flask možda i nije najbolji izbor. Ovakvi projekti mogu postati skupi za održavanje jer Flask ne garantuje podršku za biblioteke koje se koriste, a koje se u međuvremenu mogu ugasiti. Nasuprot tome, Django na primer, održava svoje ugrađene biblioteke (kao što su šabloni i ORM (eng. Object Relational Mapping)) i ima alate za tranziciju na novije verzije, što dosta smanjuje rizik održavanja aplikacije.

Prednosti korišćenja sistema zasnovanih na Flask okruženju:

- Visoka fleksibilnost.
- Kompatibilnost sa novim tehnologijama.
- Bolje skaliranje kod jednostavnijih aplikacija.
- Više opcija za podešavanje.
- Malo bolje performanse okruženja nego kod težih okruženja.
- Lakoća upotrebe.
- Manja baza koda aplikacije.

Mane sistema zasnovanih na Flask okruženju:

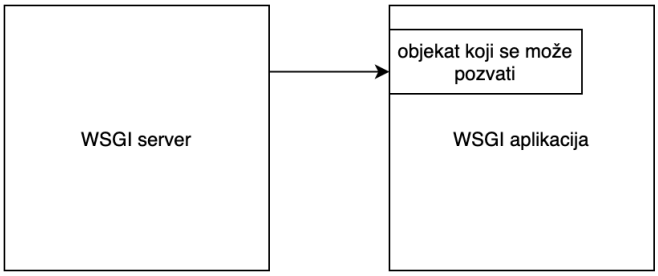
- Sklonost sigurnosnim rizicima.
- Sporiji MVP proces kod kompleksnih aplikacija.
- Veći troškovi održavanja kod kompleksnih aplikacija.

Konkretne funkcionalnosti okruženja sa više tehničkih detalja i primera biće navedeni u opisu Praktičnog dela rada. Logo biblioteke Flask je prikazan na slici 14, a zvanična stranica biblioteke može se pronaći ovde [9].

5.2 WSGI

WSGI (eng. veb Server Gateway Interface) je specifikacija interfejsa preko koga veb server i aplikacija komuniciraju. Obe strane interfejsa i serverska i aplikacijska su specificirane u PEP 3333 standardu. Ako su aplikacija ili okruženje pisani po WSGI specifikaciji, moći će da se pokrenu na bilo kom serveru koji implementira istu specifikaciju. [10]

Tradicionalni veb serveri nisu imali mogućnosti da pokreću Python aplikacije. U kasnim 1990-tim programer Grisha Trubetskoy napisao je Apache modul koji je izvršavao izvorni kod u Python-u i nazvao ga *mod_python*. On se koristio sve do 2000-tih, ali nije bio standardna specifikacija. Zbog raznih propusta u tom modulu, Python zajednica je napravila WSGI standard koji je sada i zvanično prihvaćen. Po standardu WSGI server mora imati objekat koji može pozvati na strani WSGI aplikacije, kao što je prikazano na slici 15.



Slika 15: WSGI server i aplikacija

Prednosti korišćenja WSGI interfejsa:

- Fleksibilnost - Mogućnost promene servera koju aplikacija koristi bez modifikacija na samoj aplikaciji ili okruženju. To dosta olakšava posao razvojnom timu jer ne moraju unapred da odaberu veb server za aplikaciju.
- Bolje skaliranje - Opsluživanje hiljada zahteva za dinamičkim sadržajem u istom trenutku je domen WSGI servera, ne programskog okruženja.

Konfiguracija veb servera definiše koje zahteve treba proslediti WSGI serveru na obradu. Kada je zahtev obrađen i odgovor generisan od strane WSGI servera, prosleđuje se nazad do veb servera i do pretraživača.
Na primer, sledeća konfiguracija 4 Nginx veb servera obrađuje statičke objekte kao što su slike, JavaScript i CSS fajlovi, u */static* direktorijumu, a sve ostale zahteve prosleđuje WSGI serveru na portu 8000.

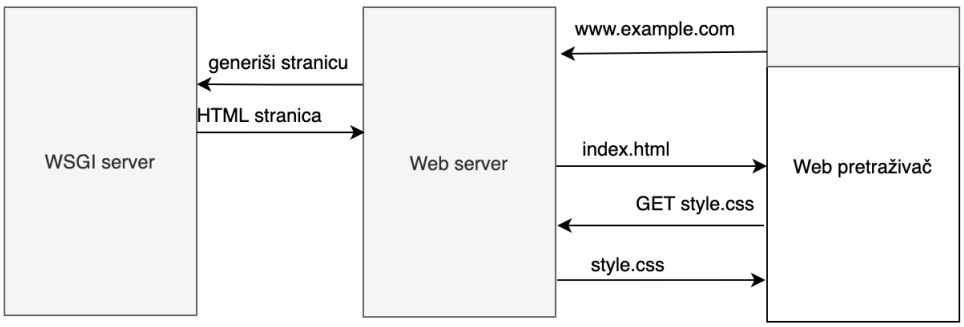
```
# this specifies that there is a WSGI server running on port 8000
upstream app_server_djangoapp {
    server localhost:8000 fail_timeout=0;
}

# Nginx is set up to run on the standard HTTP port and listen for requests
server {
  listen 80;

  # nginx should serve up static files and never send to the WSGI server
  location /static {
    autoindex on;
    alias /srv/www/assets;
  }

  # requests that do not fall under /static are passed on to the WSGI
  # server that was specified above running on port 8000
  location / {
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;
    proxy_redirect off;

    if (!-f $request_filename) {
      proxy_pass http://app_server_djangoapp;
      break;
    }
  }
}
```

Listing 4: Konfiguracija Nginx veb servera

Neke implemetacije WSGI servera:

- Gunicorn - najpopularniji server za UNIX operativne sisteme.
- uWSGI - sve više se koristi zbog visoko performantne implementacije.
- mod_wsgi - Apache modul koji prati WSGI specifikaciju.

Na slici 16 prikazana je komunikacija između klijenta i WSGI aplikacije.



Slika 16: Primer podele odgovornosti između komponenti

6 Praktični deo rada

6.1 Opis projekta

Ideja koja stoji iza projekta Seven Tweets je da se napravi mreža mikroservisa koja funkcioniše slično kao Twitter. U toj mreži, svaki čvor (eng. node) je zaseban servis sa sopstvenom bazom podataka. Takođe mu se dodeljuje i korisničko ime (eng. username) koje je jedinstveni identifikator tog servisa. Svaki servis u mreži zna IP adrese drugih servisa (ili korisnika) i komunicira sa njima preko HTTP protokola. Svi čvorovi poseduju identičan API ali imaju različite startne konfiguracije.

Servisi mogu raditi i pojedinačno bez priključivanja mreži. Da bi se priključili postojećoj mreži potrebno je da pošalju zahtev na URL putanju /**register** uz koju je potrebno znati i adresu jednog od čvorova u mreži. Zahtev inicira protokol za pristupanje mreži. Novi servis šalje željeno korisničko ime u zahtevu, a u odgovoru dobija mapu ostalih servisa sa njihovim adresama. Nakon toga prolazi sve čvorove iz mape i njima šalje zahteve za registraciju kako bi ga oni dodali u svoje lokalne mape čvorova. Prilikom gašenja sevisa poslaće se zahtev svim članovima mreže na putanju /**unregister** i on će biti uklonjen iz lokalnih mapa ostalih servisa.

Osnovne funkcionalnosti svakog servisa su:

- Unos novog tvita u bazu.
- Stvaranje retvita sa referencom na postojeći tvit drugog korisnika (dobavljanje sadržaja se vrši dinamički prilikom zahteva).
- Pregled, brisanje i ažuriranje pojedinačnih tvitova.
- Pretraga tvitova jednog korisnika sa filter parametrima.
- Distribuirana pretraga tvitova svih korisnika sa filter parametrima.
- Paginacija rezultata pretrage.

Aplikacija Seven Tweets napisana je u Python3 programskom jeziku i okruženju Flask. Dodatne informacije o Python-u mogu se pročitati u sekciji 5, a o Flasku u podsekciji 5.1. Izvorni kod projekta objavljen je na Github-u i može se pronaći ovde [11].

Prilikom dizajniranja API-ja praćena su pravila REST API-ja iz podsekcije 3.1 dok su izuzetak putanje za registraciju i odjavu servisa koje prate RPC dizajn iz podsekcije 3.2.

6.1.1 Flask šabloni

Zarad veće modularnosti aplikacije korišćen je koncept Flask šablona (eng. blueprints). Šabloni mogu dosta pojednostaviti rad većih aplikacija i njihovo

buduće proširivanje i održavanje. Flask šablon je entitet zavistan od entiteta Flask aplikacija i može se posmatrati kao set operacija koje se mogu regitrovati pod klasom Flask aplikacija. Prednosti korišćenja šablona:

- Podela aplikacije na više manjih celina.
- Registrovanje šablona na API-ju aplikacije sa određenim URL prefiksom ili pod domenom.
- Isti šablon se može registrovati više puta, sa različitim URL prefiksima, subdomenima i dodatnim opcijama.
- Kroz šablone se mogu proslediti statički fajlovi, filteri i drugi parametri.
- Jednostavno se mogu dodati nove funkcionalnosti aplikacije.

Seven Tweets aplikacija ima dva registrovana šablona:

1. Šablon koji sadrži API putanje za registraciju i odjavu servisa.
2. Šablon koji sadrži API putanje za sve operacije nad tvitovima.

Na listingu 5 prikazan je način na koji se metoda iz http sloja dodaje u šablon. To se postiže upotrebom dekorator funkcije *register_ blueprint* izvedenog iz klase *Blueprint*. Dekoratori su po definiciji funkcije koje za ulazni parametar imaju drugu funkciju. U ovom slučaju potrebno je samo dekorisati metodu koja treba da postane deo šablona.

```
1000 import logging
from flask import Blueprint, request, jsonify
1002 from seventweets.config import g
from seventweets.exceptions import except_handler
1004
register_blueprint = Blueprint('register', __name__)
1006 logger = logging.getLogger(__name__)


1008
@register_blueprint.route('/register', methods=['POST'])
1010 @except_handler
def register():
1012     """
     Handles register request
1014     :return: servers dict
     """
1016     address = '{}:{}'.format(request.remote_addr, request.get_json()['
     port'])
     name = request.get_json()['name']
1018     response = g.registry.register(name=name, address=address)
     logger.info('registered: {}'.format(g.registry.servers[name]))
1020     return response


1022
@register_blueprint.route('/unregister/<string:name>', methods=['DELETE'
     ])
1024 @except_handler
def unregister(name):
1026     """
     Handles unregister request
1028     :param name: Server which to remove from servers dict
     :return: json with message
1030     """
     ip = request.remote_addr
1032     g.registry.unregister(name, ip)
     logger.info('unregistered: {}'.format(name))
1034     return jsonify({'status': 'removal complete'})
```

Listing 5: Primer šablona za registraciju i odjavu servisa

Registrovanje šablona u aplikaciji može se videti na listingu 6 i postignuto je korišćenjem metode *register_blueprint* iz samog objekta *application*. Potrebno je pozvati navedenu metodu za svaki od postojećih šablona.

```
import yaml
import logging
from flask import Flask

from seventweets.server import Server
from seventweets.tweets_blueprint import tweets_blueprint
from seventweets.register_blueprint import register_blueprint

logger = logging.getLogger(__name__)


class MyFlask(Flask):
    """
    Custom Flask class. Instantiates and runs Server before app.run()
    When loading different server configs, path to config needs to be
    changed
    """
    def __init__(self, *args, **kwargs):
        with open("config/server1.yaml", 'r') as ymlfile:
            cfg = yaml.load(ymlfile)
        server_name = cfg['server']['server_name']
        database = cfg['server']['database']
        my_address = cfg['server']['my_address']
        serving_at_port = cfg['server']['serving_at_port']
        connect_to = cfg['server']['connect_to']
        server = Server(
            server_name=server_name,
            database=database,
            my_address=my_address,
            serving_at_port=serving_at_port,
            connect_to=connect_to
        )
        server.run(production=True)
        super(MyFlask, self).__init__(*args, **kwargs)


application = MyFlask(__name__)
application.register_blueprint(register_blueprint)
application.register_blueprint(tweets_blueprint)


if __name__ == "__main__":
    application.run()
```

Listing 6: WSGI fajl

6.1.2 Arhitektura aplikacije

Seven Tweets aplikacija ima troslojnu arhitekturu, odnosno :

1. **Prezentacijski (HTTP) sloj** - Sadrži funkcije koje obrađuju zahteve sa API putanja. Parsiraju zahtev klijenta, pozivaju odgovarajuću funkciju iz biznis sloja i vraćaju odgovor klijentu.

```
1000 @tweets_blueprint.route('/tweets/<int:id>', methods=['GET'])
     @except_handler
1002 def get_tweet(id):
         """
1004     Handles get request for specific tweet
         :param id: id of the tweet
1006     :return: Response with acquired tweet
         """
1008     result = operations.get_tweet(id)
         return jsonify(result)
```

Listing 7: Primer funkcije iz http sloja koja prikazuje specifičan tvit

2. **Aplikativni (biznis) sloj** - Logika izvršavanja operacija se nalazi u ovom sloju.

```
1000 def get_tweet(id):
         """
1002     Gets tweet from local database with supplied id
         :param id: id of the tweet
1004     :return: one tweet as dict
         """
1006     result = None
         for row in db.get_tweet_by_id(id):
1008         result = get_tweet_content(row)
         if not result:
1010         raise NotFound('id not found')
         return result
```

Listing 8: Primer funkcije iz biznis sloja za prikazivanje specifičnog tvita

3. **Sloj baze podataka** - Sva komunikacija sa bazom podataka dešava se kroz ovaj sloj.

```
1000     def get_tweet_by_id(id):
         """
1002     Returns tweet with supplied id
         :param id: id of the tweet
1004     :return: tweet as row
         """
1006     query = '''SELECT id,name,tweet,creation_time,type,reference
                    FROM tweets WHERE id = %s'''
1008     return g.db.query_execute(query, (id,))
```

Listing 9: Primer funkcije iz sloja baze podataka koja dohvata tvit sa prosleđenim identifikatorom

Na listingu 7 prikazana je metoda iz http sloja koja dohvata pojedinačne tvitove pozivom metode iz biznis sloja aplikacije, parsiranjem njenog odgovora i vraćanjem rezultata u JSON formatu. Navedena metoda biznis sloja prikazana je na listingu 8. Ona komunicira direktno sa slojem baze i proverava vraćeni

rezultat. U ovom slučaju ako sloj baze nije vratio objekat u biznis sloju će se kreirati greška sa brojem 404 i to će se propagirati do http sloja. Funkcija iz db sloja prikazana je na listingu 9. Dohvatanje tvita iz baze vrši se direktno preko SQL upita koji je ispisan u kodu metode db sloja i koji nalazi tvit preko njegovog identifikatora.

6.1.3 Obrada grešaka

Funkcije iz http sloja imaju jedinstvenu obradu grešaka, tako da će korisnici uvek dobijati uniformne odgovore prilikom neuspeha operacija. Obrada grešaka je implementirana u vidu dekorator funkcije prikazane na listingu 10. Dekorator funkcija je napravljena uz pomoć alata *wraps* iz standarne biblioteke *functools*. U telu dekorator funkcije može se videti da se sve greške formatiraju u JSON strukturu sa poljem *message* koje sadrži tekst greške i polje *code* koje sadrži http status kod odgovora. Za greške koje nisu tipa *HttpException* status kod se postavlja ručno. Metode iz http sloja aplikacije samo treba dekorisati ovom funkcijom i obrada grešaka će biti aktivirana.

```
1000 def except_handler(f):
         """
1002     Handles exceptions caught in http layer (server.py)
         :param f: wrapped function
1004     :return: function or appropriate exception message
         """
1006     @wraps(f)
         def wrapper(*args, **kwargs):
1008         try:
                 return f(*args, **kwargs)
1010         except HttpException as e:
                 body = {
1012                 'message': str(e),
                     'code': e.CODE,
1014             }
                 logger.warning(body['message'])
1016             return jsonify(body), e.CODE
             except TypeError:
1018             body = {
                     'message': 'bad request',
1020                 'code': 400,
                 }
1022             logger.exception(body['message'])
                 return jsonify(body), 400
1024         except KeyError:
                 body = {
1026                 'message': 'bad request json does not have required
     fields',
                     'code': 400,
1028             }
                 logger.exception(body['message'])
1030             return jsonify(body), 400
             except Exception as e:
1032             body = {
                     'message': str(e),
1034                 'code': 500,
                 }
1036             logger.exception(body['message'])
                 return jsonify(body), 500
1038     return wrapper
```

Listing 10: Dekorator funkcija za obradu grešaka

6.2 Instalacija

Preduslovi: Potrebno je instalirati Python verziju 3.4.3 ili noviju [12] i Postgresql bazu [13]. Potrebno je zatim kreirati korisnike i njihove baze na jednoj mašini ili više mašina ako je cilj da servisi budu fizički odvojeni. To se može uraditi:

1. Ručno preko Postgres aplikacije.
2. Ili preko skripte **/bin/create_db.sh** pisane za operativni sistem Linux.

```
# daje se pravo egzekucije skripti
$ chmod +x create_db.sh
# kreira se korisnik i baza
$ ./create_db.sh username db_name
```

Preostali potreban softver za izvršavanje aplikacije nalazi se u fajlu **requirements.txt**, a dodatni softver za pokretanje testova u fajlu **requirements-dev.txt**.
Preporučeno je napraviti virtuelno okruženje prema uputsvima odavde [14]. Instalirati sav potreban softver u okruženju na ovaj način:

```
$ source path-to-env/bin/activate
$ pip install -r requirements.txt
```

Pre pokretanja aplikacije treba dodati modul seventweets u sistemsku promenljivu PYTHONPATH:

```
$ export PYTHONPATH="Lokalna_putanja_do_/seventweets"
```

Za inicijalizaciju baze i pokretanje servisa koristiće se skripta **/start.py**.

Za inicijalizacija baze sa testnim podacima potrebno je izvršiti sledeću komandu:

```
$ python start.py --dbname=yourdatabase --sname=username
  init_db
```

Prvi servis može se startovati komandom:

```
$ python start.py --dbname=yourdatabase --sname=username
  run_server --port=num-of-port
```

Za pokretanje svakog novog servisa potrebno je izvršiti skriptu sa sledećim parametrima:

```
$ python start.py --dbname=yourdatabase --sname=username
  run_server --port=num-of-port --con=ip:port
```

gde je **–con** parametar adresa nekog servisa koji je već registrovan u mreži.

6.3 Servisni API

API Seven Tweets aplikacije je interno podeljen u dva šablona i tako će biti prikazan u nastavku zbog bolje preglednosti. To naravno nije primetno sa klijentske strane koja vidi jedinstven API.

Putanje koje su u šablonu za registraciju i odjavu sa mreže servisa:

POST **/register**

Dodaje novi servis u postojeću listu servisa

Parameter

no parameter

Body:

```
{"port" : <int: port na kome je aktivan servis>,
 "name" : <string: ime servisa>}
```

Response application/json

200 OK
400 Bad Request

DELETE **/register/{name}**

Briše servis iz liste servisa

Parameter

<string: name> ime servisa

Response application/json

200 OK
401 Forbidden

Putanje koje su u šablonu za operacije nad tvitovima:

POST **/tweets**
Kreira novu poruku u bazi korisnika

Parameter

no parameter

Body:

```
{"tweet" : <string: poruka>}
```

Response application/json

201 Created

GET **/tweets**
Dohvata sve poruke korisnika

Parameter

no parameter

Response application/json

200 OK

GET **/tweets/{id}**
Dohvata poruku sa zadatim identifikatorom

Parameter

<int: id> id poruke

Response application/json

200 OK
404 Not Found

PUT **/tweets/{id}**

Ažurira poruku sa zadatim identifikatorom

Parameter

<int: id> id poruke

Body:

```
{"tweet" : <string: nova poruka> }
```

Response application/json

201 Created
404 Not Found

DELETE **/tweets/{id}**

Briše poruku sa zadatim identifikatorom

Parameter

<int: id> id poruke

Response application/json

204 No Content
404 Not Found

POST **/retweet**

Kreira novu poruku u bazi korisnika sa referencom ka originalnoj poruci

Parameter

no parameter

Body:

```
{ "name" : <string: ime originalnog servisa>,
  "id" : <int: id originalne poruke>}
```

Response application/json

201 Created
400 Bad Request

GET	**/search_me**
	Pretraga poruka lokalnog korisnika sa zadatim parametrima

Parameter	
<string: content>	deo poruke
<string: from_time>	pretraga poruka nastalih posle datog vremena
<string: to_time>	pretraga poruka nastalih pre datog vremena
<int: per_page>	broj poruka po stranici
<string: last_creation_time>	datum nastanka zadnje poruke na prethodnoj stranici

Response	application/json
200 OK	

Putanja preko koje je moguće uraditi distribuiranu pretragu tvitova:

GET	**/search**
	Pretraga poruka svih korisnika sa zadatim parametrima

Parameter	
<string: content>	deo poruke
<string: name>	pretraga poruka samo jednog korisnika
<string: from_time>	pretraga poruka nastalih posle datog vremena
<string: to_time>	pretraga poruka nastalih pre datog vremena
<int: per_page>	broj poruka po stranici
<string: last_creation_time>	datum nastanka zadnje poruke na prethodnoj stranici

Response	application/json
200 OK	

6.4 Testiranje softvera

U cilju postizanja veće produktivnosti, pouzdanosti i bržih isporuka novih verzija softvera, testiranje je fundamentalni deo razvojnog ciklusa aplikacije. Treba sagledati sva očekivana korišćenja softvera i napisati testove koji potvrđuju uspeh ili neuspeh u zavisnosti od situacije koja se analizira. Takođe je bitno predvideti što više problematičnih slučajeva korišćenja (eng. edge case) i napisati testove koji analiziraju ponašanje aplikacije prilikom takvih situacija.

Slika 17: Logo biblioteke Pytest

Automatizacija testiranja je najbolji način da se obezbedi izvršavanje testova. U projektu se koriste dve vrste automatizovanih testova:

1. Jedinični testovi
2. Testovi aplikativnog interfejsa

Jedinični testovi testiraju pojedinačne jedinice koda, najčešće na nivou metoda i idealno bi trebali pokrivati 100% izvornog koda. Oni mogu verifikovati sintaksnu ispravnost koda i semantičku ispravnost na nivou funkcije koja se testira, tj očekivano ponašanje funkcije prilikom različitih ulaznih parametara. Nakon provere pojedinačnih delova koda mora se testirati i kako te komponente rade zajedno i da li se predviđene operacije izvršavaju kako je očekivano. Za to se koriste **Testovi aplikativnog interfejsa** koji podrazumevaju slanje zahteva na svaku od API putanja servisa sa različitim parametrima koji su pažljivo odabrani za svrhe testiranja. Za potrebe testiranja ovog projekta odabrana je biblioteka Pytest čiji je logo prikazan na slici 17, a može se preuzeti ovde [15] . Tokom testiranja koristi se "lažni"(eng Mock) servis i baza podataka tako da ne postoji prava http komunikacija između klijenta i servera, a prava baza podataka ostaje netaknuta. Pre izvršavanja testa nova prazna baza se popunjava testnim podacima, a nakon završetka testa se gasi, time se postiže maksimalna izolacija i podiže se pouzdanost svakog testa.

Testovi za aplikaciju se mogu pokrenuti komandom:

```
$ python -m pytest <lokalna_putanja_do/tests>
```

putanja do direktorijuma /tests se može izostaviti ako se komanda izvršava iz korenog direktorijuma projekta.

6.5 Produkcijski režim rada

Kao i svaka WSGI aplikacija Seven Tweets može biti pokrenut na bilo kom WSGI serveru. Podrazumeni server koji koristi Flask okruženje je Werkzeug ali on nije preporučen za upotrebu pri velikim opterećenjima aplikacije kao što je uglavnom slučaj za produkcijsko okruženje. Za te potrebe koristi se drugi WSGI server Gunicorn koga odlikuje stabilnost i mala potrošnja resursa. [16] On se može instalirati komandom:

```
$ pip install gunicorn
```

Konfiguracija za njega podešava se u fajlu **config/gunicorn.py** i u nastavku sledi primer te konfiguracije:

```

bind = "127.0.0.1:8001"
workers = 1
worker_class = 'gthread'
threads = 10
```

Ovim je serveru zadato da aplikaciju startuje na lokalnoj mašini, portu 8001 i da pokrene jedan proces sa 10 niti koji će obrađivati http zahteve korisnika. Parametri se mogu uskladiti sa trenutnim potrebama aplikacije za skaliranjem. Aplikacija na Gunicorn serveru pokreće se komandom:

```
$ gunicorn --config=config/gunicorn.py wsgi
```

Server će pokrenuti fajl **wsgi.py** iz korenog direktorijuma projekta koji sadrži objekat pozivanja (eng. Callable App). Za svaki sledeći servis koji se pokreće mora se dostaviti nova konfiguracija i promeniti wsgi fajl ili napraviti novi. Primer konfiguracije za konekciju sa lokalnom bazom dat je u nastavku:

```

user: seventweets1
password: seventweets1
host: localhost
port: 5432
unix_sock:
database: seventweets1
ssl_context:
timeout:
```

Preporučeno je i izolovanje baze na sopstvenu mašinu i komunikacija putem SSL protokola što može biti podešeno preko fajla **config/db_conf.yaml**.

Napomena: Servis je testiran na operativnim sistemima Linux i macOS dok je za pokretanje na Windows sistemima poželjna upotreba Dockera.

7 Zaključak

U ovom radu dat je prikaz osnovnih koncepata mikroservisne arhitekture i opisane su tehnike razvoja REST veb servisa u programskom jeziku Python i okruženju Flask.

U sklopu rada, dizajnirana je i razvijena aplikacija Seven Tweets. Aplikacija je kao softver otvorenog koda, dostupna na adresi [11]. Tokom razvoja aplikacije vodilo se računa da budu ispoštovana pravila troslojne arhitekture i da se postigne što veći stepen modularnosti. U radu je prikazana realizacija pretrage podataka u sistemu koji ima više odvojenih baza podataka kao i pokretanje servisa na produkcionom serveru Gunicorn. Pokazano je kako se opisana aplikacija može pokrenuti sa različitim konfiguracijama i kako može formirati mrežu servisa koji funkcionišu slično kao platforma Twitter. U navedenoj mreži svaki servis ima sopstvenu bazu podataka i komunicira sa ostalim servisima preko HTTP protokola.

Dalji razvoj aplikacije podrazumevao bi unapređenja u procesu registracije i odjave servisa iz mreže, kao i paginacije rezultata na putanjama sa distribuiranom pretragom. Unapređenje bi uključivalo i brisanje direkntog slanja SQL upita ka bazi i korišćenje ORM(eng. Object-relational mapping) biblioteke kao što je SQLAlchemy. Servis bi bio dodatno poboljšan uvođenjem nekog alata za kontinuiranu integraciju softvera.

Literatura

[1] Irakli Nadareishvili, Ronnie Mitra, Matt Mclarty, and Mike Amundsen. *Microservice Architecture*. O'Reilly Media, Inc., 2016.

[2] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc., 2014.

[3] Joseph Ingeno. *Software Architect's Handbook : Become a successful software architect by implementing effective architecture concepts*. Packt Publishing Limited, Birmingham, United Kingdom, 2018.

[4] Leonard Richardson, Mike Amundsen, and Sam Ruby. *RESTful Web APIs: Services for a Changing World*. O'Reilly Media, Inc., 2013.

[5] Mark Masse. *REST API: Design Rulebook*. O'Reilly Media, Inc., 2012.

[6] Luciano Ramalho. *Fluent Python*. O'Reilly Media, Inc., 2015.

[7] Dokumentacija Python programskog jezika. https://www.python.org/.

[8] Miguel Grinberg. *Flask web development: Developing web applications with Python*. O'Reilly Media, Inc., 2014.

[9] Dokumentacija biblioteke Flask. https://flask.palletsprojects.com/en/1.1.x/.

[10] Tarek Ziade. *Python Microservices Development*. Packt Publishing Limited, Birmingham, United Kingdom, 2017.

[11] Github stranica sa izvornim kodom projekta. https://github.com/gingercookiemage/seventweets.

[12] Stranica za preuzimanje Python programskog jezika. https://www.python.org/downloads/.

[13] Stranica za preuzimanje Postgresql baze podataka. https://www.postgresql.org/download/.

[14] Dokumentacija biblioteke Venv. https://docs.python.org/3/library/venv.html.

[15] Dokumentacija biblioteke Pytest. https://docs.pytest.org/en/stable/contents.html.

[16] Dokumentacija Gunicorn servera. https://gunicorn.org/.