Primjena i implementacija mikroservisnih arhitektura pomoću programskog okvira Java SpringBoot

Šolja, Jura

Master's thesis / Diplomski rad

2019

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:022361

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Jura Šolja

PRIMJENA I IMPLEMENTACIJA MIKROSERVISNIH ARHITEKTURA POMOĆU PROGRAMSKOG OKVIRA JAVA SPRINGBOOT

DIPLOMSKI RAD

Varaždin, 2019.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Jura Šolja

Matični broj: 46346/17–R

Studij: Informacijsko i programsko inženjerstvo

PRIMJENA I IMPLEMENTACIJA MIKROSERVISNIH ARHITEKTURA POMOĆU PROGRAMSKOG OKVIRA JAVA SPRINGBOOT

DIPLOMSKI RAD

Mentor :

Doc. dr. sc. Andročec Darko

Varaždin, srpanj 2019.

Jura Šolja

Izjava o izvornosti

Izjavljujem da je moj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Ovaj rad obrađuje teorijske pretpostavke mikroservisne arhitekture, te opisuje i implementira praktičnu primjenu mikroservisne arhitekture u obliku web aplikacije koja je razvijena pomoću Java Spring Boot programskog okvira. U teorijskom dijelu rada su prezentirani rezultati istraživanja brojnih radova iz domene mikroservisne arhitekture, te je pronalazak prikazan u dva velika poglavlja: mikroservisi kao zasebne komponente i mikroservisna arhitektura kao cjelina. Nakon teorijskog dijela, uvod u praktični dio rada se daje upoznavanjem Spring Boot programskog okvira u programskom jeziku Java. Web aplikacija je u kontekstu osobnih financija i cilj implementacije je prikazati primjenu mikroservisnog stila arhitekture na praktičnom primjeru, te se upoznati s razvojem web aplikacije u Spring Boot programskom jeziku. Ovaj rad objedinjuje teorijski i praktični dio u jednu cjelinu i iznosi zaključke u obliku prednosti i nedostataka mikroservisne arhitekture, uzimajući u obzir mišljenja autora iz ove domene.

Ključne riječi: mikroservisi; arhitektura; servisi; soa; java; spring; boot; cloud; eureka;

Sadržaj

1. Uvod

U današnjem svijetu razvoja programskih proizvoda, vrlo je dobro poznat problem brze isporuke sigurnog programskog proizvoda. Nažalost, riječi *brza isporuka* i *siguran proizvod* često ne idu jedna uz drugu, jer je potrebno žrtvovati jedno za drugo. Kako bi timovi zaposlenika u nekom poduzeću udovoljili vrlo kratkim rokovima u kojima proizvod mora biti isporučen sa svim traženim funkcionalnostima, često su bili primorani uštedjeti na nekim drugim aspektima razvoja softvera, i tu se nažalost štedjelo na testiranju i sigurnosti. Tako je svijet dolazio do velike količine nesigurnih programskih proizvoda ispunjenih *buggovima* na čije ispravljanje se dobar dio vremena troši i nakon isporuke. Prema Zarnekow i Brenner (2005) čak 79% ukupnih troškova tijekom 5 godina razvoja programskog proizvoda čine troškovi koje stvara dodatan razvoj programskog proizvoda i održavanje programskog proizvoda u produkciji. Ipak, prije nekoliko godina pojavila su se poduzeća koja se naizgled mogu pohvaliti da isporučuju istodobno i siguran proizvod i unutar rokova. Kada su ih zapitali koja je tajna njihovog uspjeha, odgovorili su "mikroservisi".

Mikroservis (eng. *Microservice*) i dizajn arhitekture nekog sustava s naglaskom na mikroservise je danas relativno novi pojam u razvoju arhitekture softvera. U ovom radu, opisat ćemo važne teorijske značajke mikroservisa kao samostalne, neovisne komponente koje međusobno ili s okolinom komuniciraju pomoću nekog mehanizma (prema Namiot i Sneps-Sneppe, 2014, taj mehanizam je najčešće HTTP). Pokušat ćemo obrazložiti koja je optimalna veličina mikroservisa i opisati zakone koji se odnose na paralelan razvoj pojedinačnih komponenata, te navesti probleme koji se mogu javiti pri takvom razvoju. Opisat ćemo razloge zašto i kada se preporuča koristiti mikroservise, ali ćemo sagledati i negativnu stranu koja se u praksi pojavljuje. Također, dat ćemo usporedbu mikroservisne arhitekture sa servisno-orijentiranom arhitekturom (eng. *Service-Oriented Architecture*) i monolitnom arhitekturom (eng. *Monolithic Architecture*), te iznijeti najznačajnije sličnosti i razlike.

Nakon što opišemo teorijske značajke mikroservisa, krenut ćemo s dizajnom mikroservisne arhitekture. Glavne značajke mikroservisne arhitekture su servisi (na mikro razini), koji predstavljaju jednostavnu, granularnu uslugu koju obavlja neko dio sustava. Kada se servisi ispravno integriraju i povežu svojim djelokrugom djelovanja i dizajnom, oni sastavljaju kompleksni sustav ponašanja koji se sastoji od relativno jednostavnih komponenata. Stoga, osim razmišljanja o pojedinim individualnim servisima koje će sustav imati, vrlo je važno razmišljati i koji je cilj ili završno stanje kada se svaki pojedini servis integrira u jedno cijelo koje ćemo nazivati sustav. Iako se na spomen mikroservisne arhitekture misli da je pažnja usmjerena na same mikro komponente, to u suštini jest bit mikroservisne arhitekture, ali nije što se tiče cilja aplikacije koju organizacija pokušava realizirati. Pažnja mora biti usmjerena na završno stanje i programski proizvod kojeg čine međusobno izolirane, ali opet povezane, komponente. Opisat ćemo što čini servis na mikro razini i kako on komunicira s ostatkom aplikacije i okolinom. Osim servisa na mikro razini, glavne značajke mikroservisne arhitekture su još i procesi, kultura, organizacija i rješenje na makro razini arhitekture koje predstavlja funkcionirajući sustav koji objedinjuje sve preostale dijelove u jedno cijelo. (Nadareishvili, Mitra, McLarty i dr.,2016). Ove značajke arhitekture ćemo detaljnije promotriti u poglavlju *Dizajn mikroservisne arhitekture*. U tom poglavlju

također ćemo dotaknuti sigurnost mikroservisne arhitekture, te ćemo obraditi koliku važnost je potrebno pridonijeti validiranju podataka i korisničkih zahtjeva prije nego se takav zahtjev proslijedi određenoj komponenti sustava.

U poglavlju implementacije, započet ćemo s praktičnom primjenom mikroservisne arhitekture tako da se najprije upoznamo s okolinom rada. U ovom radu, koristit ćemo programski okvir (eng. *Framework*) Spring Boot za programski jezik Java. Spring Boot programski okvir je zadnjih godina postao najpopularniji programski okvir za programski jezik Java za brz razvoj jednostavnih web aplikacija, ali i složenih. Spring Boot se može koristiti za razvoj aplikacija i na strani poslužitelja i na strani korisnika, iako se u praksi češće primijenjuje neka druga tehnologija na strani korisnika (eng. *Frontend*) koja se potom spaja sa Spring Boot tehnologijama na strani poslužitelja. Tako je, na primjer, strana poslužitelja (eng. *Backend*) SAP Hybris *e-commerce* platforme razvijena pomoću Spring programskog okvira (preciznije, koriste se tehnologije Spring MVC, Spring Security, Spring EL i Spring Core).

Nakon upoznavanja sa Spring Boot programskim okvirom, započet ćemo s našim praktičnim primjerom gdje ćemo razviti web aplikaciju. Web aplikacija će imati tehničke i korisničke zahtjeve. Tehnički zahtjevi će biti zahtjevi za tehnologijom koja će se koristiti pri razvoju aplikacije, dok će korisnički zahtjevi biti određene funkcionalnosti koje naša web aplikacija mora implementirati. Neki od zahtjeva su osnovni poput operacija s korisničkim računom: registracija i prijava, promjena lozinke, pregled profila i ažuriranje podataka i sl. Dok će ostali korisnički zahtjevi biti kompleksniji: rad s transakcijama, izvještajima, uvozu i izvozu podataka i sl. Za potrebe ovih zahtjeva, implementirat ćemo tri mikroservisa:

- Korisnički mikroservis - logika operacija vezanih uz korisnički rad poput prijava, registracija, rad s lozinkama, dohvat korisničkih podataka i sl.,
- Transakcijski mikroservis - logika operacija vezanih uz obavljanje transakcija,
- E-mail mikroservis - logika operacija vezanih uz slanje e-mail poruka na korisničku e-mail adresu.

Za kraj, dat ćemo osvrt na mikroservisnu arhitekturu i pokušati pojasniti koji problem današnjice ona rješava. Također ćemo pokušati dati moguće smjerove za napredak mikroservisne arhitekture i predvidjeti što nam budućnost donosi.

2. Metode i tehnike rada

Metoda kojom ću ovaj rad napisati je metoda *od gore prema dolje* (eng. *Top-down*). Najprije ću pojasniti što je mikroservis s najviše točke gledišta, te postupno prelaziti u detalje i dati pojašnjenja što čini mikroservisnu arhitekturu i kako se, u temelje, dizajnira mikroservisna arhitektura. Kada se upoznamo s teorijskom pozadinom mikroservisne arhitekture, dati ćemo uvod u Spring i Spring Boot programske okvire. Upoznat ćemo se s alatima koji omogućuju instalaciju i korištenje Spring Boot programskog okvira, te ćemo kreirati projekt u kojem ćemo nastaviti razvijati web aplikaciju koja koristi dizajn mikroservisne arhitekture i detaljno pojasniti kako ona radi uz programski kôd. Web aplikaciju ćemo krenuti razvijati od praznog projekta, kroz svaku pojedinu funkcionalnost logičkim slijedom. To znači, najprije ćemo inicijalizirati projekt i uvesti ga u našu odabranu okolinu rada i započeti s razvojem. Programski alati koje ću koristiti pri izradi ovog rada su:

- LaTeX - programski jezik koji koristim za stvaranje ovog dokumenta, kojeg uređujem pomoću Overleaf *online* uređivača LaTeX projekta.
- Java - programski jezik u kojem ću pisati programski kôd web aplikacije.
- Spring Boot - programski okvir za programski jezik Java. Pomoću Spring Boot-a ću razviti web aplikaciju i mikroservisnu arhitekturu te aplikacije.
- Spring Initializr - *online* alat pomoću kojeg ću inicijalizirati prazan Spring Boot projekt. Pomoću Spring Initializr *online* alata ću specificirati naziv, tip, programski jezik, verziju Spring Boot programskog okvira, meta podatke (naziv paketa i artefakata) i ovisnosti projekta (eng. *Dependencies*).
- Spring Cloud Eureka - poslužitelj koji ćemo koristiti za otkrivanje servisa u našoj mikroservisnoj arhitekturi.
- Tomcat - lokalni web poslužitelj na kojem će se posluživati naša web aplikacija. Tomcat je standardno integriran u Spring Boot projekt i nije potrebna bilo kakva konfiguracija.
- IntelliJ IDEA Community Izdanje - IDE okolina (eng. *Integrated Development Environment*) koju ću koristiti za razvoj web aplikacije u praktičnom dijelu ovog rada.
- XAMPP - XAMPP (eng. *Cross-Platform (X), Apache (A), MariaDB(M), PHP (P), Perl(P)*) je alat kojim ću pokrenuti lokalni Apache web poslužitelj na kojem će se održavati MySQL baza podataka.
- phpMyAdmin - alat razvijen u programskom jeziku PHP. Koristit ćemo ga za administriranje naše MySQL baze podataka.
- Hibernate - alat objektno-relacijskog mapiranja (eng. *Object-Relational Mapping - ORM*) za Java programski jezik.
- Postman - alat koji ću koristiti za testiranje REST krajnjih točaka (eng. *Endpoint*), tj. testiranje ulaza i izlaza koje će naši mikroservisi konzumirati i proizvoditi.

3. Mikroservisi

Već dugi niz godina ljudi su pronalazili sve bolje i bolje načine razvoja sustava. Kroz te godine, učili smo i promatrali dolazak novih tehnologija i valove revolucionarnih načina na koje IT poduzeća posluju kako bi razvili sustave koji podjednako zadovoljavaju i klijente i vlasnike programskih proizvoda. U ovom poglavlju opisat ćemo što se krije iza pojma mikroservisa i zašto je mikroservisna arhitektura postala jedna od dominantnijih arhitektura dizajna programskog proizvoda.

3.1. Što su mikroservisi?

Do ovog trenutka, već smo nekoliko puta spomenuli pojam mikroservisa (kod nekih autora se može pronaći pojam i mikrousluga). U kontekstu kojem smo spomenuli taj pojam, već se na neki način da protumačiti mikroservis. Ali koja je službena definicija mikroservisa? Naime, nije moguće dati jedinstvenu definiciju mikroservisa jer aspekti variraju, već se mogu navesti karakteristike. Gledajući bez određenog konteksta, prema Thönes (2015) mikroservis može predstavljati mala aplikacija ili dio kôda koji može biti nezavisno razvijen, nezavisno skaliran i nezavisno testiran, dok Newman (2015) ukratko tumači mikroservise kao malene, autonomne servise koji međusobno surađuju.

Svi autori se slažu da mikroservis ima jednu i samo jednu odgovornost, jedan razlog da postoji i samo jedan razlog da se promijeni/zamijeni, stoga i radi samo jednu stvar koja se jednostavno može razumjeti.

Wolff (2016) raspravlja o pojmu mikroservisa s četiri različita aspekta mikroservisa:

- Veličina mikroservisa,
- Odnos između mikroservisa, arhitekture i organizacije koji se opisuje pomoću Conwayevog zakona,
- Dizajn temeljen domenama (eng. *Domain Driven Design - DDD*), te
- Korisničko sučelje (eng. *User Interface - UI*) mikroservisa.

Većina autora spominje navedena stajališta, stoga ćemo u nastavku u detalje protumačiti svake ove poglede na mikroservise i iznijeti naše, ali i mišljenje drugih autora, kako bismo dobili jasnu sliku mikroservisa kao komponente koje implementiraju logiku neke usluge (servisa).

3.1.1. Veličina mikroservisa

Već sam pojam *mikroservis* nas upućuje na predviđenu veličinu mikroservisa. Očigledno, zamišljeno je da bude malen. Ali to ne odgovara u potpunosti na pitanje veličine mikro-

servisa. Kako se mjeri veličina mikroservisa? Kako možemo znati da li je mikroservis prevelik ili idealne veličine?

Jedna očita moguća mjera veličine mikroservisa je broj linija programskog kôda (eng. *Lines of code - LOC*). Ali opet, ova mjera ne daje jasan i precizan odgovor zato što broj linija programskog kôda može značajno varirati ovisno o programskom jeziku koji se koristi za implementaciju mikroservisa, zato što programski jezici više razine češće mogu ostvariti traženi rezultat s mnogo manje linija kôda u odnosu na programske jezike niže razine. Također, broj linija kôda može varirati ovisno o stilu uvlačenja vitičastih zagrada. Na primjer, Kernighan & Ritchie (K&R) stil uvlačenja se po broju linija kôda razlikuje od broja linija kôda koji je uvučen prema Allmanovom stilu uvlačenja.

```
/**
 * Kernighan & Ritchie stil uvlake (3 linije)
 */
while(x == y) {
    radiNesto();
}

/**
 * Allman stil uvlake (4 linije)
 */
while(x == y)
{
    radiNesto();
}
```

Unatoč realnoj kritici ove mjere veličine mikroservisa, u praksi se često ipak koristi. Thönes (2015) spominje da veličina mikroservisa u linijama programskog kôda se može protezati do nekoliko tisuća linija kôda. Generalno nije važan broj linija programskog kôda, ako taj kôd radi samo jednu stvar za koju je zadužen, tj. ima jednu i samo jednu odgovornost. Po tome sudeći, ne postoji idealna veličina mikroservisa ako se mjeri brojem linija kôda, ali nije niti bitna, ako se vodimo time da mikroservis ima jednu odgovornost i jedan razlog za promjenu. Teško je zamisliti da mikroservis sadrži milijune linija kôda i da ima jednu odgovornost. Kada bi se uzele u obzir druge mjere osim broj linija programskog kôda, opet bi se vrlo teško pronašla idealna veličina mikroservisa zbog silne raznolikosti tehnologija, a i same odgovornosti koju implementira mikroservis. Sličan zaključak je donesen i u *Microservices - Flexible Software Architecture* od Wolff (2016).

3.1.2. Conwayev zakon

Conwayev zakon glasi (Herbsleb i Grinter, 1999):

> „*Bilo koja organizacija koja dizajnira sustav (u širem smislu), će proizvesti dizajn čija struktura preslikava organizacijsku komunikacijsku strukturu*“

Prije više od 30 godina, ova propozicija Melwina Conwaya je postala poznata kao Conwayev zakon. Conway je argumentirao kako je ta struktura nužna posljedica komunika-

cijskih potreba zaposlenika koji obavljaju posao u organizaciji. Važno je natuknuti da komunikacijska struktura ne mora biti identična preslika organizacijske strukture - komunikacija se često odvija na daljinu zbog geografskog razmještaja članova tima, te je posljedično i komunikacija otežana i drugačija. Na primjer, lakše je komunicirati s osobom koja sjedi uz vas (Wolff, 2016).

Ali koja je veza između Conwayeva zakona i mikroservisa? Ranije smo spomenuli da mikroservisi imaju svojstvo neovisnosti. Kada se domene timova istodobno razdvoje, timovi također postaju neovisni jedan od drugog i ne postoji potreba koordinacije i sinkronizacije posla. Tehnička koordinacija, kao i koordinacija između domena, može biti svedena na minimum. Ova činjenica olakšava paralelan rad nad zahtjevima softvera, pa čak i uvođenje softvera u produkcijski rad. Mikroservisna arhitektura ima posebno dobre predispozicije da podrži distribuciju poslova prema Conwayevom zakonu. Čak štoviše, upravo ovaj aspekt je osnovna karakteristika mikroservisne arhitekture - paralelan razvoj međusobno neovisnih komponenata koji na posljetku čine skladnu cijelinu (Wolff, 2016).

Ipak, postoji zamka koju je potrebno napomenuti. Ako je tehnička struktura uvjetovana komunikacijskoj strukturi, promjena načina komunikacije će utjecati i na promjenu strukture. Ovakva arhitekturalna promjena među mikroservisima će biti bitno kompleksnija i cjelokupan proces razvoja će biti manje fleksibilan.

U nastavku, u poglavlju *Dizajn temeljen na domenama* adresirat ćemo implementaciju distribuiranu po domenama.

3.1.3. Dizajn temeljen na domenama

Prema Pahl i Jamshidi (2016); Wolff (2016) razumijevanje dizajna temeljenog na domenama je vrlo važno za razumijevanje mikroservisa, zato što mikroservisi nalažu strukturiranje sustava većih kompleksnosti prema domenama organizacije. Zamišljeno je da svaki mikroservis predstavlja zasebnu domenu, koja je dizajnirana tako da svaka implementacija promjene ili uvođenje nove funkcionalnosti se izvodi nad samo jednim mikroservisom za koji je pojedina domena zadužena. Tek tada se postiže maksimalna korist neovisnih, razdvojenih razvojnih timova koji u paraleli vrše implementaciju bez koordinacije.

Da bi se dizajnirao model domene, Evans (2003) u svojem radu *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley) navodi sljedeće uzorke dizajna temeljenih na domenama:

- **Entitet** - entitet (eng. *Entity*) je objekt koji sadrži individualnu osobnost. Dobar primjer entiteta bi bio *klijent* na nekoj *e-commerce* platformi, i tipično su entiteti spremljeni u bazi podataka. Ovakav koncept entiteta se koristi samo kod tehničke implementacije, inače entitet suštinski pripada domeni modeliranja kao i ostali koncepti dizajniranja na temelju domenama.
- **Objekt** - ili vrijednost objekta (eng. *Object*), nemaju individualnu osobnost kao entiteti. Primjer vrijednosti objekta može biti memorijska adresa ili lokacija objekta. Objekt može utjeloviti specifičnog *klijenta* na spomenutoj *e-commerce* aplikaciji i to je suštinska razlika

objekta i entiteta.

- **Agregat** - agregat (eng. *Aggregate*) je skupina objekata iz domene koji se mogu tretirati kao jedinstvena jedinica. Na primjer, račun može biti agregat a stavke računa zasebni objekti, ali je korisno tretirati račun (zajedno sa stavkama računa) kao jedan agregat.
- **Servisi** - servisi (eng. *Services*) implementiraju poslovnu logiku u dizajnu temeljenom na domenama. Rad s računima može biti obavljen pomoću servisa koji ima pristup klijentima i artiklima, te mu je potreban entitet računa.
- **Repozitoriji** - repozitorij (eng. *Repository*) ostvaruje pristup entitetima koji su pohranjeni u nekoj perzistentnoj tehnologiji, poput baze podataka.
- **Tvornice** - tvornice (eng. *Factories*) omogućavaju generiranje kompleksnih objekata domene.

U mikroservisnoj arhitekturi, agregati su od posebne važnosti. Unutar agregata je moguće stvoriti konzistentnost, te je stoga potrebno obratiti pažnju paralelnom izvođenju rada unutar mikroservisa. Kako se konzistentnost ne bi narušila, paralelno izvršavanje zadataka unutar mikroservisa mora biti serijalizirano. Na primjer: agregat poput *košarice* sadrži ukupnu vrijednost od 130 kuna u stavkama različitih artikala. Pretpostavimo da postoji ograničenje *košarice* da kupac ne smije kupiti artikle od ukupne vrijednosti veće od 150 kuna. Kada bi kupac paralelno dodao još dva artikla vrijednosti 15 kuna, paralelne operacije bi mogle izračunati da je ukupna vrijednost košarice sada 145 kuna, ali je istinski vrijednost prekoračena za 10 kuna. Stoga, promjene nad agregatima moraju biti serijalizirane kako se ne bi dešavale navedene greške. Iz istog razloga, agregati ne mogu biti podijeljeni na više od jednog mikroservisa, jer u tom scenariju konzistentnost ne može biti osigurana (Wolff, 2016).

3.1.4. Korisničko sučelje mikroservisa

Wolff u radu *Microservices: Flexible Software Architecture* (Addison-Wesley) preporučuje korištenje korisničkog sučelja (eng. *User Interface - UI*) za pojedini mikroservis. Korisničko sučelje bi trebalo nuditi funkcionalnost mikroservisa korisniku. Jedna od prednosti korisničkog sučelja mikroservisa je ta da sva logika može biti sadržana u jednom mikroservisu - neovisno o tome da li se radi o logici korisničkog sučelja, poslovnoj logici, logici spajanja na bazu podataka i sl. Posljedično, svaka promjena tih logika će biti provedena unutar istog mikroservisa.

Ipak, ne slažu se svi autori s činjenicom da bi mikroservisi trebali sadržavati vlastito korisničko sučelje, te je li uopće potrebno. Kao što se već spominjalo, mikroservisi ne bi smjeli biti glomazni: ako se koristi mikroservis na nekoliko različitih strana klijenata, nema smisla implementirati svako korisničko sučelje za taj jedan mikroservis, te se mora razmisliti o dizajnu u kojem se mikroservis sastoji samo od čiste logike, bez korisničkog sučelja. Također je moguće razviti jedan mikroservis čija je jedina svrha pružati korisničko sučelje za neki drugi izdvojeni mikroservis, pod uvjetom da oba mikroservisa razvija isti tim kako bi se izbjegla koordinacija i ovisnost među timovima u domeni.

Spomenut ćemo dvije moguće implementacije korisničkog sučelja. Jedna alternativa je grafičko korisničko sučelje (eng. *Graphic User Interface - GUI*). U slučaju grafičkog korisnič-

kog sučelja, sučelje koristi HTML za prezentaciju sučelja korisniku. Druga alternativa nema grafičko korisničko sučelje, već nudi RESTful-HTTP (eng. *Representation State Transfer*) priključnu točku preko koje korisnik može komunicirati s mikroservisom. Podaci koji se šalju su u obliku JSON (eng. *JavaScript Object Notation*) ili XML (eng. *Extensible Markup Language*) tipa podatka. Obje alternative koriste HTTP za prijenos podataka, tj. za komunikaciju.

3.2. Razlozi korištenja mikroservisa

Nakon što smo se upoznali s mikroservisima, mogli biste se zapitati zašto koristiti mikroservisnu arhitekturu sustava. Mikroservisi nude brojne prednosti i navest ćemo ih u ovom poglavlju, ali također ćemo i dati nekoliko izazova s kojima se razvojni timovi mogu susresti. Do sada smo već spomenuli neke prednosti, ali ćemo ih ovdje dati do izražaja. Cilj detaljnog razumijevanja benefita i izazova koje mikroservisna arhitektura nudi je bolja odluka da li mikroservisna arhitektura predstavlja dobar kandidat arhitekture za sustav kojeg planiramo razviti, s obzirom na njegove zahtjeve i karakteristike.

Prednosti mikroservisa se mogu kategorizirati u nekoliko različitih domena. Pa tako prema Newman (2015); Wolff (2016) prednosti možemo podijeliti prema:

- **Tehničkoj domeni**,
- **Organizacijskoj domeni** i
- **Poslovnoj domeni**.

3.2.1. Tehničke prednosti

S pogleda tehničke domene, autori se slažu oko sljedećih benefita mikroservisa: *tehnološka heterogenost, otpornost, skalabilnost* i *neovisnost*.

Tehnološka heterogenost (eng. *Technology Heterogeneity*) ili riječima kojima Wolff (2016) spominje - sloboda odabira tehnologije zasniva se na tome da su mikroservisi međusobno razdvojene, neovisne komponente koje komuniciraju pomoću nekog komunikacijskog mehanizma, obično protokola poput HTTP-a. To znači da domena organizacije koja je zadužena za razvoj mikroservisne arhitekture nije dužna implementirati svu arhitekturu u istoj tehnologiji koja pokriva svaki kutak sustava. Mikroservise krasi ta prednost da zbog njihove međusobne neovisnosti, svaki pojedini mikroservis može biti implementiran u različitoj tehnologiji. U današnjem svijetu ne postoji tehnološki "srebrni metak" koji rješava sve moguće probleme i pruža najbolje performanse, stoga ima smisla (gdje je to moguće) pojedine komponente sustava optimizirati tako da se koristi tehnologija koja je prikladnija za određeni zadatak. Prevedeno u jezik mikroservisa, to znači da je moguće svaki mikroservis implementirati u različitoj tehnologiji koja je prikladnija za njegov zadatak s minimalnim učinkom na sustav u kojem se nalazi, jer protokol komunikacije ostaje isti. U tom slučaju mikroservis možemo promatrati kao crnu kutiju.

Mikroservisna arhitektura, zbog svoje distribuirane prirode, je teoretski manje pouzdana

arhitektura od ostalih srodnih arhitektura. Ako se arhitektura proteže kroz nekoliko različitih poslužitelja, tada postoji opasnost i od kvara u sklopovlju. U distribuiranim sustavima postoji naslijeđena opasnost od ispada u mreži pomoću koje komponente sustava komuniciraju. Glavni pojam **otpornosti i robusnosti** je *zaštitni zid* (eng. *Bulkhead*). U slučaju ispada jedne komponente distribuiranog sustava, ako je pravilno uspostavljena pregradna zaštita tada se ispad neće proširiti i potencijalno negativno utjecati na rad ostalih komponenata u sustavu. Mikroservisi čine prirodnu granicu koja bi se trebala ponašati kao zaštitni zid u slučaju ispada, ili drugim riječima kada mikroservis detektira pogrešku u radu (eng. *Malfunction*) tada bi pomoću komunikacijskog protokola trebao odaslati standardnu vrijednost koja se u sustavu tretira kao kod za pogrešku. Nakon što sustav detektira ispad servisa pomoću koda za pogrešku, u optimalnom slučaju će se na određeno vrijeme smanjiti dostupnost usluge. Ovo je značajna tehnološka prednost nad, na primjer, monolitnom arhitekturom u kojoj katastrofalna pogreška vrlo često znači ispad cijelog sustava (Newman, 2015).

Jedna očigledna prednost mikroservisa nad monolitnom arhitekturom je **skalabilnost**. S monolitnom arhitekturom, kada se ukaže potreba za skaliranjem, potrebno je skalirati sustav kao cjelinu. Mikroservisna arhitektura rješava taj problem tako da se skalira samo onaj servis koji pokazuje preopterećenost zbog porasta prometa, na primjer. Mikroservisi se kroz sustav mogu balansirati tako da se manje zahtjevni mikroservisi mogu poslužiti na sklopovlju sa slabijim performansama, dok se vremenski kritični mikroservisi za koje se predvidi da će češće biti opterećeni mogu poslužiti na sklopovlju većih performansi. Uz to, određeni mikroservisi se mogu instalirati na različite lokacije kako bi usluga lokacijski bila bliže korisnicima, čime se smanjuje vrijeme odziva.

Jedna prednost mikroservisne arhitekture, koja je u nekoliko navrata već spomenuta, koja najviše dolazi do izražaja i gotovo prikuplja sve zasluge što su spomenute prednosti uopće moguće, je **neovisnost** i **autonomnost** mikroservisa. Mikroservis može biti izoliran servis na nekoj platformi - platforma kao usluga (eng. *Platform as a Service - PAAS*), ili može biti izoliran servis kao proces na operacijskom sustavu (Newman, 2015). Izoliranost mikroservisa u arhitekturi povlači činjenicu da mikroservisi nisu međusobno ovisni, ali zajedno čine funkcionalan sustav. Međusobna komunikacija se odvija pomoću komunikacijskih kanala kojima su poruke oblikovane protokolom komunikacijskog kanala. Mikroservisi se mogu mijenjati bez utjecaja na ostatak sustava, nadograđivati ili mijenjati unutarnju logiku, te pritom samo razmišljati o tome što on nudi vanjskom svijetu, kao crna kutija. Kako bi mikroservis postao dostupan okolini, on mora implementirati neku vrstu sučelja - u praksi se vrlo često koristi API (eng. *Application Programming Interface*). API sučelje mora biti implementirano ako se mikroservisna arhitektura proteže na više međusobno udaljenih računala.

3.2.2. Organizacijske prednosti

Osim prednosti koje se manifestiraju s tehničke strane mikroservisne arhitekture, također se još naziru organizacijske prednosti. Prema Conwayevom zakonu, dizajn sustava postaje preslika komunikacijske strukture. Kako arhitektura utječe na komunikaciju među članovima tima, indirektno utječe i na organizaciju (Wolff, 2016).

Osim utjecaja kojeg opisujemo pomoću Conwayeva zakona, mikroservisna arhitektura nas uvodi u mogućnost razlamanja kompleksnog projekta u nekoliko manjih, jednostavnijih projekata. Kako su mikroservisi neovisni jedan od drugog, smanjena je potreba centralnog autoriteta koji kontrolira razvoj projekta. U teoriji, ova činjenica uklanja potrebu za centralnim upravljačkim tijelom, čime se dodatno smanjuje posao oko obavještavanja po lancu odgovornosti. Tako se nezavisni timovi se više mogu koncentrirati i ulagati trud u implementaciju zahtjeva, te upravljanje obavljati na mikro razini unutar timova.

Prema Hasite i Wojewoda (2015, Standish Group Chaos Report) uspješnost projekta ovisi i o njegovoj veličini. Pa tako samo 2% projekata glomazne veličine, 6% velike veličine, 9% srednje veličine i 21% umjerene veličine projekta završi uspješno, unutar rokova i bez prekoračenog budžeta. Dok čak 62% projekata male veličine završi uspješno, unutar rokova i bez dodatnih troškova. Prema tome, dobra je odluka razlomiti veliki projekt u nekoliko manjih. Manji projekti imaju manji obujam i djelokrug koji je potrebno realizirati, pa su i procjene rokova i budžeta češće puno preciznije i realnije. Shodno tome, značajno se smanjuje rizik od neuspjeha/prekoračenja rokova i izrađuje se bolji plan rada u fazi planiranja. Čak i ako se prekorače rokovi, obično su u puno manjoj mjeri i rijetko dovodi do napuštanja projekata. Kada se sve skupa spoji, proces donošenja odluka je kraći i lakši, a rizici su bitno smanjeni.

3.2.3. Poslovne prednosti

Tehničke i organizacijske prednosti zajedno vode do poslovnih prednosti: rizik poslovnih projekata se smanjuje, koordinacija i komunikacija između i unutar timova je manje intenzivna, te se koncentracija usmjerava prema realizaciji zahtjeva projekata. Mikroservisna arhitektura tada daje do značaja poslovnu prednost - smanjuju se rizici i omogućava se brz razvoj projektnih zahtjeva.

Paralelan rad koji omogućuje mikroservisna arhitektura direktno utječe na poslovni aspekt projekta, projekt se istodobno širi i raste u svim smjerovima. U slučaju da jedan od timova zapne ili naiđe na probleme, problemi ostaju samo na određenom mikroservisu dok se ne razriješe, dok se rad nad svim preostalim aspektima projekta nastavlja kontinuirano.

Ako je arhitektura pažljivo odabrana za potrebe projekta koji se razvija, mikroservisi također mogu podržati agilan rad (Wolff, 2016). To je svakako dobar razlog, s poslovne perspektive, da se koristi mikroservisna arhitektura.

3.2.4. Izazovi

Do sada, dali smo do izražaja samo svijetle strane mikroservisne arhitekture i naveli smo njene prednosti uvođenja. Ali ipak, mikroservisna arhitektura nije jedinstveno rješenje koje je samo potrebno realizirati i problemi će nestati, a održavanje sustava nikad više neće biti problem. Uz razvoj mikroservisne arhitekture, u praksi su se istaknuli neki problemi koje ćemo ovdje opisati.

Proces razdvajanja sustava u mikroservise uzrokuje povećanu složenost sustava, zato

što su razvojni inženjeri zaduženi za razvoj distribuiranog sustava - testiranje je znatno teže na distribuiranim sustavima, te će se najvjerojatnije morati implementirati nekakav komunikacijski mehanizam između servisa (Namiot i Sneps-Sneppe, 2014). Složenost sustava potom dovodi do izazova na tehničkoj razini poput visokog vremenskog odziva komponenata ili kvarova na individualnim mikroservisima. Razdvajanje sustava na mikroservise također stvara potencijalne probleme u integraciji i važno je dobro razumijevanje kako aplikacija komunicira s okolinom (Thönes, 2015), također javljaju se potencijalni problemi u razdvajanju funkcionalnosti na pojedini mikroservis (Wolff, 2016).

Jedan od mogućih izazova koje mikroservisna arhitektura uvodi je **nepouzdana komunikacija** (eng. *Unreliable Communication*). Mikroservisi međusobno i s okolinom komuniciraju pomoću mreže, koja je po pretpostavci nesigurna i nepouzdana. Poruke koje se šalju preko nepouzdane mreže mogu biti odaslane, ali nema garancije da će stići do namijenjenog odredišta i na vrijeme. Osim što je mreža nesigurna i nepouzdana, sami mikroservisi mogu otići u kvarno stanje. Taj problem se ne može u potpunosti otkloniti na tehničkoj strani, već se sklopovlje mikroservisa može unaprijediti kvalitetnijom opremom koja osigurava viši postotak dostupnosti. Posljedično rastu troškovi mikroservisa, ali i rizici. Rizik raste zato što i dalje postoji mogućnost ispada iz mreže, unatoč dodatnom naporu da se taj događaj spriječi, te se troškovi dodatno povećavaju u tom slučaju. Stoga, korisnost mikroservisa mora odvažiti rizike i troškove kvalitetnog sklopovlja.

Tehnološki pluralizam (eng. *Technology Pluralism*) je problem koji se potencijalno može javiti zbog tehnološke heterogenosti koju smo naveli kao prednost mikroservisne arhitekture. Mikroservisi ne moraju dijeliti razvojnu tehnologiju već se svaki zasebni mikroservis može razviti tehnologijom za kojom ima potrebe kako bi se mogla postići određena skalabilnost i performanse koje će udovoljiti zahtjevima. Ali, ako se mikroservisna arhitektura sustava sastoji od desetine ili čak stotine mikroservisa, javlja se ogromna razina kompleksnosti koja je posljedica velikog broja korištenih tehnologija. Naime, u određenom trenutku više nema razvojnog inženjera ili razvojnog tima koji u potpunosti razumije cijeli sustav. Iako zvuči problematično, u praksi se je pokazalo da zbog same karakteristike razvoja mikroservisne arhitekture vrlo rijetko se ukaže potreba da pojedinac ili skupina mora samostalno razumjeti cijeli sustav (Wolff, 2016). Rješenje koje se predlaže je uniformiranje tehnologija mikroservisa kroz sustav tako da se smanji kompleksnost ali se istodobno zadrži skalabilnost i visoke performanse.

Karakteristika mikroservisne arhitekture je ta da su organizacijska struktura i arhitektura jednake. Prema Conwayevom zakonu, ova karakteristika stvara **ovisnost između arhitekture i organizacije** (Wolff, 2016). Ova zavisnost stvara problem kada je potrebno napraviti refakturiranje arhitekture, jer to tada znači i promjene nad organizacijom. Ova pojava otežava arhitekturalne promjene i ovaj problem se javlja kod svih projekata koji su popraćeni Conwayevim zakonom.

3.3. Mikroservisna arhitektura i servisno-orijentirana arhitektura

Servisno-orijentirana arhitektura (eng. *Service-Oriented Architecture*) je vladala IT organizacijama sredinom 2000-tih godina. IT organizacije su prihvatila taj (tada novi) stil arhitekture kako bi poboljšale ponovnu iskoristivost poslovnih funkcionalnosti i unaprijedile komunikaciju i kolaboraciju između organizacije i poslovanja. U to vrijeme, nikla je nekolicina dobrih praksi implementacije servisno-orijentirane arhitekture, i još više proizvoda i alata koji se mogu iskoristiti za lakšu implementaciju servisno-orijentirane arhitekture.

Nažalost, IT organizacije su na teži način shvatile da je stil servisno-orijentirane arhitekture kompleksan, glomazan i skup u koji se ulaže previše vremena da se dizajnira i implementira - što je rezultiralo brojnim napuštenim i propalim projektima. Takav niz negativnih shvaćanja i događaja je kroz vrijeme narastao, te je servisno-orijentirana arhitektura u glavnini napuštena u IT industriji.

Danas, mikroservisna arhitektura preuzima svu pozornost IT organizacija kao što je i servisno-orijentirana arhitektura preuzimala pozornost 2000-tih. Mikroservisna arhitektura je danas najpopularniji izbor za razvoj skalabilnih i modularnih aplikacija, te se do sada pokazalo da učinkovito rješava problem velikih, kompleksnih servisno-orijentiranih arhitektura, pa i s glomaznim, prenapuhanim monolitnim aplikacijama. Ali koliko se servisno-orijentirana i mikroservisna arhitektura razlikuju? Kolike su šanse da će IT industrija također doći do bolne spoznaje s mikroservisima kao što je došla i sa servisno-orijentiranom arhitekturom? U ovom poglavlju objasnit ćemo koncepte servisno-orijentirane arhitekture i dati usporedbu s karakteristikama mikroservisne arhitekture s koncentracijom na definiciju servisa, te kako oba stila arhitekture koriste servis kao temeljnu komponentu arhitekture.

3.3.1. Karakteristike servisnih arhitektura

Mikroservisna arhitektura i servisno-orijentirana arhitektura se smatraju servisnim arhitekturama. Servisne arhitekture izražavaju važnost servisa, kao najvažnijoj komponenti arhitekture. Servis implementira i izvršava neku poslovnu funkciju organizacije, ali i druge funkcije nevezane za poslovanje. Mikroservisna arhitektura i servisno-orijentirana arhitektura su dva različita arhitekturalna stila, ali dijele neke slične karakteristike koje proizlaze iz činjenice da su oba arhitekturalna stila orijentirana na servise.

Jedna stvar koja je zajednička svim servisnih arhitekturama je ta da su generalno distribuirane arhitekture (Richards, 2015). To znači da se komponentama može pristupiti s udaljene lokacije pomoću nekog protokola pristupa: REST, SOAP (eng. *Simple Object Access Protocol*), AMQP (eng. *Advanced Message Queuing Protocol*), JMS (eng. *Java Message Service*), MSMQ (eng. *Microsoft Message Queueing Protocol*), RMI (eng. *Remote Method Invocation*), i ostalih protokola. Najčešće se u praksi pojavljuju SOAP i REST protokoli pristupa (Erl, 2016). Distribuirane arhitekture nude značajne prednosti nad monolitnim arhitekturama i arhitekturama temeljene na slojevima - skalabilnije su, neovisnije, nude veću kontrolu nad razvojem i testira-

nju. Karakteristika servisnih arhitektura je i modularnost - praksa koja opisuje enkapsulaciju dijelova aplikacije u komponente/servise koji se individualno mogu dizajnirati, razvijati, testirati i puštati u rad s nikakvom ili malo ovisnosti nad drugim komponentama/servisima u aplikaciji. Ali, ove prednosti mogu dovesti i do nekih izazova i poteškoća na koje smo upozorili u ranijem poglavlju. Razmjena koja se dobiva s ovim prednostima je povećana kompleksnost i troškovi razvoja, te se mora osigurati da distribuirana arhitektura bude pravilno implementirana kako bi se korisnost višestruko vratila u odnosu na kompleksnost i cijenu koju potencijalno stvara.

3.3.2. Što je drugačije?

U ovom poglavlju fokusirat ćemo se na pojam servisa i koliko različito mikroservisna arhitektura i servisno-orijentirana arhitektura klasificiraju servis. Arsanjani (2004) opisuje servis:

> „*Servis je softverski resurs s javnim opisom koji otkriva što servis radi. Opis servisa je moguće pretraživati, povezivati i pozivati od strane korisnika servisa. Pružatelj servisa implementira opis servisa, te također održava kvalitetu implementiranih servisnih zahtjeva za korisnika servisa*“.

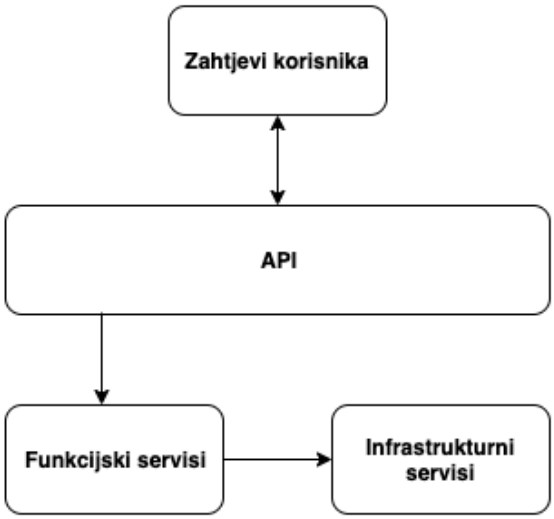
Dok OASIS (eng. *Organization for the Advancement of Structured Information Standards*) u referentnom modelu za servisno-orijentirane arhitekture (eng. *Reference Model for Service Oriented Architecture - SOA-RM*) daje službenu definiciju za servis:

> „*Servis je mehanizam koji omogućuje pristup jednoj ili nekoliko funkcionalnosti, gdje je pristup ostvaren pomoću sučelja i pristupa se u skladu s ograničenjima i pravilima koje specificira opis servisa*“.

Definicija tumači servis kao nešto što nudi poslovne mogućnosti za koje postoji dobro definirano sučelje preko kojeg je moguće zatražiti uslugu. Naime, ova definicija ipak ne daje dovoljno detaljan opis koji bi specificirao klasifikaciju, organizacijsku ovlast i granularnost. Kako je servis temeljni pojam servisnih arhitektura, dobro razumijevanje ovog pojma pomaže pobliže objasniti kontekst servisa unutar pojedinih stilova servisnih arhitektura - u ovom slučaju mikroservisa i servisno-orijentirane arhitekture. Pojam koji se odnosi na način kako su servisi klasificirani u pojedinom stilu arhitekture se prema Richards (2015) naziva **taksonomija servisa** (eng. *Service Taxonomy*).

Taksonomijom servisa servisno-orijentirane i mikroservisne arhitekture ćemo objasniti različitosti kako ova dva stila arhitekture klasificiraju servis. Na najvišoj razini, moguće je dati podjelu servisa na dvije bazične klasifikacije: *tip servisa* (eng. *Service Type*) i *poslovno područje* (eng. *Business Area*). Klasifikacija tipa servisa se odnosi na tip uloge koju servis zauzima u sveukupnoj arhitekturi. Primjerice, neki servisi mogu implementirati poslovnu funkcionalnost dok neki drugi servisi mogu implementirati funkcionalnost nevezanu uz poslovanje poput zapisivanja i bilježenja (eng. *Logging*), revizije i sigurnosti. Dok klasifikacija poslovnog područja servisa se odnosi na servis koji je implementiran da nudi uslugu u određenoj organizacijskoj jedinici. Primjer klasifikacije servisa prema poslovnom području može biti izvještavanje, izvršavanje transakcija, primanje narudžbi, itd. (Richards, 2015).

Mikroservisna arhitektura ima ograničenu taksonomiju servisa kada se ispituje klasifikacija tipa servisa - u suštini se sastoji od samo dva tipa servisa: *funkcijski servisi* (eng. *Functional Services*) koji podržavaju specifične poslovne operacije ili funkcije i infrastrukturni servisi (eng. *Infrastructure Services*) koji pružaju podršku nefunkcionalnim zadacima poput autentikacije, autorizacije, revizije, bilježenja događaja i promatranja (eng. *Monitoring*). Važna distinkcija ovih dvaju tipa klasifikacije tipa servisa je ta da infrastrukturni servisi nikad nisu javno vidljivi, već se tretiraju kao privatni, djeljivi servisi koji su dostupni samo interno drugim servisima. Funkcijski servisi su javno dostupni i vidljivi, ali generalno nisu dijeljeni među internim servisima.

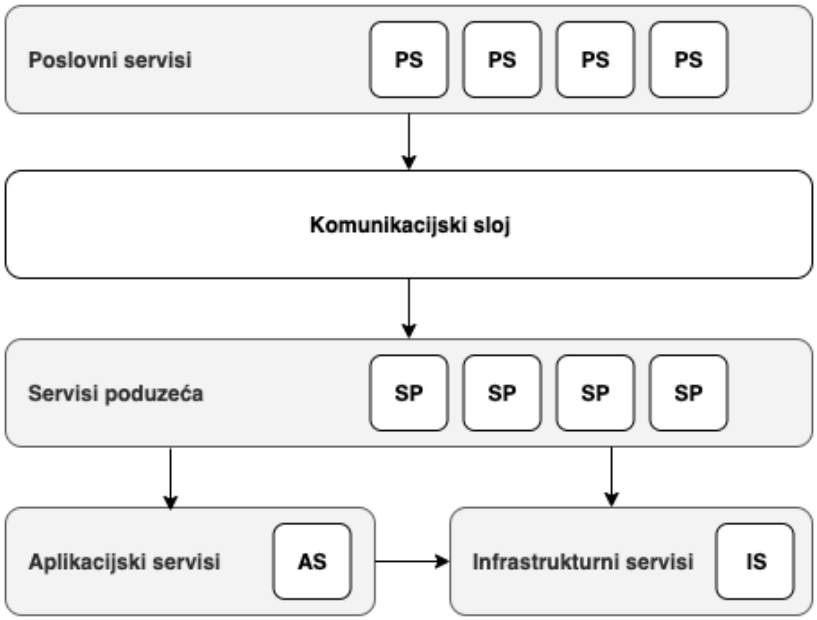


Slika 1: Taksonomija mikroservisne arhitekture (Izvor: Richards, 2015)

Servisno-orijentirana arhitektura ima vrlo različitu taksonomiju servisa - postoji distinktna i formalna servisna taksonomija kada se tumači klasifikacija tipa servisa u odnosu na mikroservisnu arhitekturu. Prema Richards (2015) servisno-orijentirana arhitektura definira četiri skupine servisa s pogleda tipa servisa i njegove uloge u sveukupnoj arhitekturi sustava. Unatoč tome, arhitekt sustava se ne mora nužno pridržavati te standardne klasifikacije servisne taksonomije. Arhitekt može implementirati vlastitu klasifikacijsku taksonomiju servisa po volji i potrebama, ali u svakom slučaju je potrebno izraditi vrlo dobru definiciju i temeljitu dokumentaciju servisne taksonomije arhitekture sustava. Standardna taksonomija uključuje četiri skupine servisa:

1. Poslovni servisi (eng. *Business Services*)
2. Servisi poduzeća (eng. *Enterprise Services*)
3. Aplikacijski servisi (eng. *Application Services*)
4. Infrastrukturni servisi (eng. *Infrastructure Services*)

Poslovni servisi su apstraktni, krupno-zrnati servisi na visokoj razini koji definiraju jezgru poslovnih operacija koje se izvršavaju na razini poduzeća. Time što su apstraktni, ne implemen-



Slika 2: Taksonomija servisa servisno-orijentirane arhitekture (Izvor: Richards, 2015)

tiraju nikakvu logiku niti protokol, već obično uključuju naziv servisa, očekivani ulaz i očekivani izlaz. Poslovni servisi za svoju reprezentaciju obično koriste XML, WSDL (eng. *Web Services Definition Language*) ili BPEL (eng. *Business Process Execution Language*).

Servisi poduzeća su konkretni, krupno-zrnati servisi na razini poduzeća koji implementiraju funkcionalnost koja je definirana nekim poslovnim procesom. Servisi na razini poduzeća imaju jedan-na-jedan ili jedan-na-više vezu i mogu biti razvijeni proizvoljnim programskim jezikom ili platformom, ili pa mogu biti gotovi komercijalni proizvodi kupljeni od nekog stranog tijela. Jedna jedinstvena karakteristika ovih servisa je da ta su generalno dijeljeni kroz cijelu organizaciju. Primjeri ovih servisa mogu biti: *DohvatiKlijenta*, *KreirajKlijenta*, *ValidirajNarudžbu* i sl. Servisi poduzeća se naslanjaju i podupiru ih aplikacijski i infrastrukturni servisi.

Aplikacijski servisi su sitno-zrnati, specifični servisi koji su vezani uz aplikacijski kontekst. Aplikacijski servisi pružaju specifičnu poslovnu funkcionalnost koja se ne nalazi na razini poduzeća. Primjerice, auto osiguravajuća kuća može javno objaviti servise koji računaju cijenu rate osiguravajuće kuće. Aplikacijski servisi mogu biti pozvani prijeko korisničkog sučelja ili kroz sučelje na razini poduzeća. Neki specifični primjeri aplikacijskih servisa mogu biti: *DodajVozača*, *DodajVozilo*, *IzračunajRatu*, i sl.

Posljednji osnovni tip servisa su *infrastrukturni servisi*. Infrastrukturni servisi su servisi koji implementiraju nefunkcionalne zadatke poput revizije, sigurnosnih zadataka, zapisivanja događaja i sl., kao i kod mikroservisne arhitekture. Ovi servisi se mogu pozivati jedino iz aplikacijskih servisa ili servisa na razini poduzeća.

Za kraj ovog poglavlja, dat ćemo zaključke i usporedbe u obliku tablice kad je prikladnije koristiti servisno-orijentiranu arhitekturu, a kada mikroservisnu arhitekturu (tablica 1).

Najznačajnija razlika servisno-orijentirane arhitekture i mikroservisne arhitekture je oči-

Tablica 1: Razlike između mikroservisne arhitekture i servisno-orijentirane arhitekture

	SOA	**Mikroservisi**
Opseg	Arhitektura na razini poduzeća	Arhitektura na razini pojedinog projekta
Fleksibilnost	Fleksibilnost ostvarena orkestracijom	Fleksibilnost ostvarena brzom isporukom i rapidnim razvojem međusobno neovisnih mikroservisa
Organizacija	Razvoj servisa po organizacijskim jedinicama	Razvoj servisa po timovima po domenama
Isporuka	Monolitna isporuka nekoliko servisa	Svaki mikroservis se može individualno isporučiti
Korisničko sučelje	Univerzalno sučelje za sve servise	Mikroservis sadrži sučelje

(Izvor: Wolff, 2016)

gledna na organizacijskoj razini. Servisno-orijentirana arhitektura stavlja naglasak na strukturu cijelog poduzeća, dok mikroservisna arhitektura pridaje pažnju individualnim projektima. Fokus rada servisno-orijentirane arhitekture je takav da se podijele razvojni timovi tako da će jedan ili više timova biti zaduženi za razvoj servisa na strani poslužitelja (eng. *Backend*), dok su neki drugi timovi zaduženi za implementaciju zahtjeva na korisničkoj strani (eng. *Frontend*). Za razliku od mikroservisne arhitekture gdje je jedan tim zadužen za implementaciju i na strani poslužitelja i na strani korisnika, s ciljem da se ostvari dobra komunikacija i ubrza razvoj funkcionalnosti - što nije cilj servisno-orijentirane arhitekture. U servisno-orijentiranoj arhitekturi, nova funkcionalnost uključuje promjene na više od jednog servisa i zahtjeva komunikaciju između velikog broja timova, dok se taj način pokušava izbjeći u razvoju mikroservisne arhitekture (Wolff, 2016).

4. Dizajn mikroservisne arhitekture

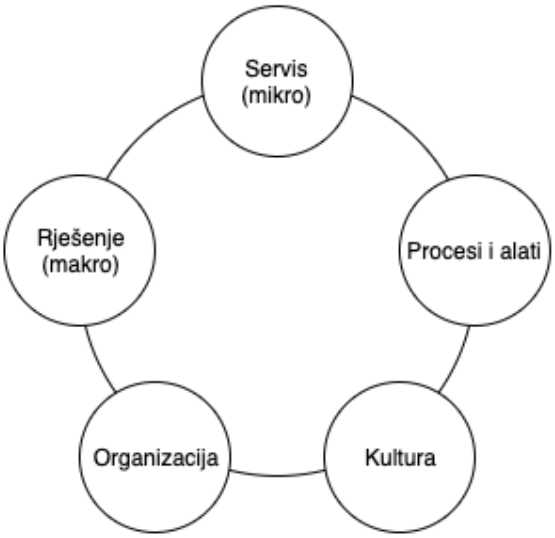
Do sada smo obrazložili da ne postoji striktna definicija mikroservisne arhitekture, već smo dali fokus karakteristikama i aspektima aplikacije koja implementira mikroservisnu arhitekturu. Ti aspekti uključuju brzinu i sigurnost pri skaliranju aplikacije pomoću koncepta modularnosti i zamjenjivosti komponenata. U ovom poglavlju ćemo ući dublje u tehničke detalje mikroservisne arhitekture. Ranije poglavlje prednosti i izazova mikroservisne arhitekture nas je upozorilo da vrlo lako može doći do zapleta i pretjerane kompleksnosti sustava ako od samog početka ne razmišljamo unaprijed, u ovom poglavlju ćemo doći do sličnog zaključka. Iako se na spomen mikroservisne arhitekture pomisli na male, nezavisne komponente, prije početka implementacije treba razmišljati o sustavu s visoke razine - potrebno je uključiti organizaciju i njenu kulturu, arhitekturu, sučelja i servise i kombinirati ih na ispravan način kako bismo dobili ravnotežu brzine i sigurnosti.

4.1. Sustavski pristup mikroservisima

Mikroservisna arhitektura traži od arhitekta i razvojnih inženjera da razmišljaju *veliko*. Kada bi se fokusirali samo na individualne servise, pronašli bi se u potencijalnim komplikacijama u kasnijim fazama razvoja. Što se misli pod razmišljanjem na *veliko*? Potrebno je konceptualizirati dizajn tako da se uzimaju u obzir svi aspekti sustava i kako oni međusobno mogu surađivati. Rezultat ovakvog sustava je ponašanje koje pridaje veću korisnost od korisnosti koja se dobije zbrajanjem korisnosti svake pojedine individualne komponente. Drugim riječima, sustav koji se sastoji od komponenata koje surađuju vrijedi više od skupina komponenata koje rade samo za sebe (Nadareishvili, Mitra, McLarty i dr., 2016).

Sustav mikroservisa obuhvaća sve aspekte organizacije pri izradi aplikacije. To znači da struktura takvog sustava uključuje organizaciju, ljude koji u njoj rade, posao koji obavljaju i rezultat koji ostvare. Osim očiglednih, vidljivih aspekata organizacije, također su vrlo važni faktori koji se tiču sustava u radu: optimizacija, koordinacija servisa, upravljanje pogreškama, operacijske prakse i sl. Kada se implementiraju svi aspekti na ispravan način, dobije se sustav koji se ponaša na predvidljiv i željen način. Ali, u organizaciji je previše pokretnih dijelova da bi se praktično proizveo dobar konceptualni dizajn. Vrlo često se u praksi susreću razne komplikacije, pogotovo kada je potrebno implementirati promjenu koja kaskadno povlači još promjena zato što su dijelovi sustava međusobno isprepleteni i povezani. Kako bi se lakše razumio kompleksan sustav, može se izraditi model kao što znanstvenici modeliraju pojave i sustave koji su prekompleksni da se zamisle. Model nam pomaže da preciznije shvatimo kako sustav se ponaša, kako lakše predvidjeti ponašanje nakon neke promjene, razaznati dijelove i konceptualizirati dizajn. Nadareishvili, Mitra, McLarty i dr. (2016) daju model dizajna mikroservisnog sustava (slika 3).

Model se sastoji od pet dijelova: *servis* na mikro razini, *rješenje* na makro razini, *procesi* i *alati*, *organizacija* i *kultura*. Za svaki od ovih elemenata ćemo opisati što je potrebno da se uspješno dizajnira mikroservisna arhitektura.



Slika 3: Model dizajna mikroservisnog sustava (Izvor: Nadareishvili, Mitra, McLarty i dr., 2016)

4.1.1. Servis

Servisi su esencijalni dio svake servisne arhitekture, pa tako i mikroservisne. U mikroservisnoj arhitekturi servisi čine gradivne blokove od kojih se konstruira cijeli sustav. Ako se servisima odredi pravilan dizajn, opseg, granularnost i svrha može se inducirati vrlo kompleksno ponašanje sustava koji se sastoji od nekoliko komponenata koje su same po sebi jednostavne. Postoje različiti uzorci dizajna servisa koji omogućavaju pravilan razvoj mikroservisa, te smo u ranijim poglavljima opisali veličinu mikroservisa, Conwayev zakon, dizajn temeljen domenama i korisničko sučelje kao moguće referentne točke koje mogu pomoći u dizajnu mikroservisa.

4.1.2. Procesi i alati

Sustav mikroservisa ne čini samo sustav međusobno povezanih servisa koji razmjenjuju informacije porukama tijekom izvođenja. Za ponašanje sustava su zaslužni procesi i alati koji se koriste i izvode u sustavu da sustav obavlja svoj posao za koji je stvoren. To uključuje procese i alate vezanih uz razvoj programskog proizvoda, isporuka programskog kôda, održavanje i upravljanje programskim proizvodom. Odabir prikladnih procesa i alata je važan faktor za proizvodnju dobrog ponašanja sustava mikroservisa. Primjerice, usvajanje DevOps i agilnih metodika razvoja programskog proizvoda i alata poput Docker-a mogu povećati promjenjivost sustava na dobro (Nadareishvili, Mitra, McLarty i dr., 2016).

4.1.3. Kultura

Od svih mikroservisnih domena koje smo spomenuli u mikroservisnom modelu, kultura se izdvaja zato što je neopipljivi faktor sustava, ali opet jedan od najvažnijih faktora. Kulturu organizacije je vrlo teško mjeriti - postoje formalne metode, anketiranja i modeliranja ali mnoga društva i voditelji instinktivno mjere kulturu organizacije i timova koji rade u njoj. Kultura organi-

zacije se uviđa u dnevnim interakcijama među člana timova, timskih rezultata i klijenata kojima teže.

Postoji nekoliko tumačenja kulture organizacije. Prema The Business Dictionary (2019) organizacijska kultura uključuje organizacijska očekivanja, iskustva, filozofija, pa i vrijednosti kojima se zaposlenici u organizaciji vode i izražena je radom, slikom o sebi, interakcijama s vanjskim svijetom i budućim očekivanjima. Kultura se zasniva na karakteru, uvjerenjima, običajima, pisanim i nepisanim pravilima koje su usvojene kroz vrijeme. Organizacijska kultura je vrlo važna zato što oblikuje odluke na najnižoj razini poslovanja. Ovako veliki utjecajni opseg na sustav predstavlja vrlo važan alat u oblikovanju sustava. Upravo zbog tog razloga da kultura oblikuje sustav od najniže razine, kultura je vrlo osjetljiv dio svakog sustava. Primjerice, ono što je u redu u Japanu, neće biti u redu u Njemačkoj i slično ono što funkcionira u osiguravajućoj kući neće funkcionirati u *e-commerce* poduzeću. Stoga, važno je pažljivo pristupiti kada se pokušava koračati koracima kojima je otišla neko poduzeće koje smatramo uzorom - ne postoje recepti ili upute koje garantiraju jednake rezultate (Nadareishvili, Mitra, McLarty i dr., 2016).

4.1.4. Organizacija

Mnoge organizacije koje su imale pozitivne rezultate nakon implementacije mikroservisne arhitekture govore kako je organizacijski dizajn ključni sastojak uspjeha. Na način rada u organizaciji utječu brojni faktori, ali najutjecajniji su ljudi s kojima radimo i način komunikacije koje imamo s njima. S pogleda mikroservisne arhitekture organizacijski dizajn predstavlja struktura, smjer kojim se krećemo pod vodstvom autoriteta, granularnosti i kompozicije timova. Kao i kultura, organizacija je vrlo osjetljiv dio sustava. Svaki dobar arhitekt mikroservisnog sustava treba razumjeti implikacije ako se izmjenjuju organizacijska svojstva, te imati na umu da je često dobar dizajn sustava nusprodukt dobrog organizacijskog dizajna. Ova pojava se može opisati Conwayevim zakonom, koji je obrazložen u ranijim poglavljima ovog rada.

4.1.5. Rješenje

U usporedbi sa servisima, rješenje (eng. *Solution*) je makro rezultat koji je kombinacija mikroservisa koji zajedno čine sustav. Kada se dizajnira specifični mikroservis, odluke koje donosimo su ograničene potrebom da se ostvari specifični izlaz tog mikroservisa, tj. ponuda usluge tog servisa. Slično, kada dizajniramo arhitekturu moramo razmišljati o potrebama te arhitekture i o svim njenim ulazima i izlazima koje proizvodi više servisa. Na toj makro razini arhitekture dizajner sustava može inducirati ponašanje sustava koje je zamišljeno.

Makro razina je finalan proizvod mikroservisne arhitekture i rezultat svih prethodnih aspekata: *servis*, *procesi* i *alati*, *organizacija* i *kultura*.

4.2. Sigurnost

Sigurnost u mikroservisnoj arhitekturi mora biti implementirana tako da servis može zatražiti samo osoba koja se je uspješno predstavila i da li ta osoba ima dovoljna prava pristupa

traženom servisu. Svaki zahtjev za mikroservisom mora biti provjeren, ali nema smisla implementirati zaštitu mikroservisa nad svakim individualnim mikroservisom. Stoga, potrebno je implementirati sigurnost na razini cijele arhitekture. To je jedini način na koji se može osigurati da se korisnički zahtjevi podjednako tretiraju kroz cijeli sustav.

Sigurnost se sastoji od dva osnovna aspekta: **autentikacije** (eng. *Authentication*) i **autorizacije** (eng. *Authorization*). Autentikacija je proces koji utvrđuje identitet korisnika, dok autorizacija provjerava da li korisnik ima dovoljna prava da pristupi određenom resursu, servisu ili pa da izvrši određene akcije. Autentikacija i autorizacija su dva međusobno različita i odvojena procesa.

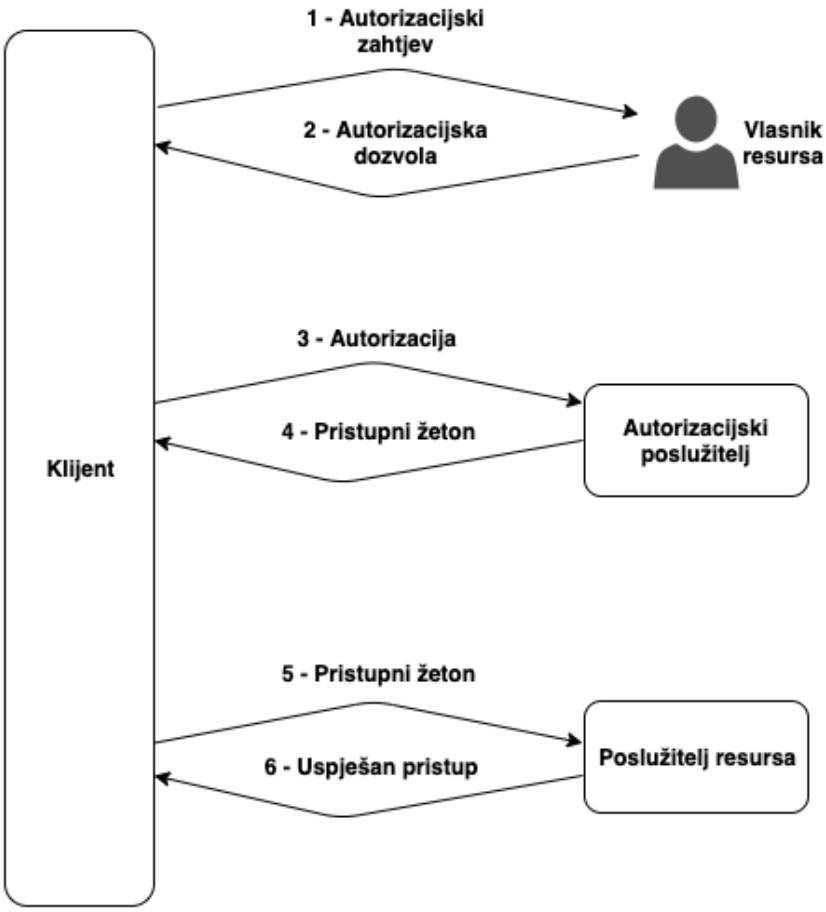
Mikroservisna arhitektura ne bi smjela izvoditi proces autentikacije korisnika; nema smisla implementirati provjeru korisničkog imena i lozinke nad svakim individualnim mikroservisom. Za taj posao potrebno je održavati centralni poslužitelj koji ima direktni pristup bazi podataka korisnika. Oko autorizacije treba promisliti malo više - najčešće je potrebno implementirati autorizaciju koja međudjeluje s centralnim poslužiteljem i samim mikroservisima zato što određeni mikroservisi sami određuju koje uloge i grupe korisnika mogu koristiti mikroservisne usluge. Grupe i uloge korisnika se pak upravljaju na centralnom serveru, stoga promjene nad autorizacijom mogu biti obavljene nad određenim mikroservisom koji primjenjuje prilagođena pravila (Nadareishvili, Mitra, McLarty i dr., 2016). U nastavku ćemo navesti neke moguće protokole kojima je moguće implementirati mehanizam autorizacije.

4.2.1. OAuth2

OAuth 2.0 je autorizacijski protokol koji omogućava aplikacijama trećih strana da ostvare limitirani pristup HTTP servisu. Pristup je moguće ostvariti u ime vlasnika resursa tako da se orkestrira interakcija odobrenja između vlasnika resursa i HTTP servisa, ili pa da se dozvoli aplikacijom treće strane da ostvari pristup na svoje ime. OAuth2 je jedno od mogućih rješenja za implementaciju autorizacije. Ovaj protokol je već široko rasprostranjen i korišten od strane mnogih velikih tehnoloških divova: *Google*, *Microsoft*, *Twitter*, *Yahoo* i dr.

Tipičan slijed događaja OAuth2 protokola je definiran standardom (IETF, 2012) (slika 4):

1. Klijent zatraži vlasnika resursa da li može izvršiti neku akciju nad njime. Primjerice, aplikacija traži zahtjev kako bi pristupio korisničkom profilu ili određenim podacima na društvenoj mreži gdje je vlasnik resursa pohranio resurs. Vlasnik resursa je obično korisnik sustava.
2. Ako vlasnik resursa dodijeli klijentu prava, klijent zaprima traženi odgovor od vlasnika resursa.
3. Klijent čita odgovor i koristi ga za slanje zahtjeva autorizacijskom poslužitelju. U gornjem primjeru, autorizacijski poslužitelj bi se nalazio u društvenoj mreži.
4. Autorizacijski poslužitelj vraća pristupni žeton (eng. *Token*).



Slika 4: OAuth2 slijed događaja (Izvor: IETF, 2012)

5. Klijent, koristeći se žetonom kojeg je dobio od autorizacijskog poslužitelja, može zatražiti poslužitelj resursa informaciju koja ga zanima. Žeton se najčešće stavlja u HTTP zaglavlje pri slanju zahtjeva.

6. Poslužitelj resursa odgovara na zahtjev.

Interakcija s autorizacijskim poslužiteljem se može odvijati na različite načine (Nadareishvili, Mitra, McLarty i dr.):

- Slučaj autorizacije pomoću lozinke - klijent prikazuje korisniku HTML formu u prvom koraku autorizacije. Vlasnik resursa (korisnik) unosi korisničko ime i lozinku. U trećem koraku tu informaciju koristi klijent kako bi dohvatio pristupni žeton s autorizacijskog poslužitelja pomoću HTTP POST metode. Ovaj pristup je nesigurniji zato što postoji mogućnost da klijent može biti neispravno/nesigurno implementiran, te se podaci mogu ugroziti.

- Implicitno odobravanje - nakon preusmjeravanja na autorizacijski poslužitelj u prvom koraku, klijent direktno pristupa pristupnom žetonu pomoću HTTP preusmjeravanja. Ovo omogućuje web preglednicima i mobilnim aplikacijama mogućnost da odmah pročitaju pristupni žeton. Treći i četvrti koraci slijede nakon. Ali, pristupni žeton nije u potpunosti zaštićen

od zlonamjernih napada zato što autorizacijski poslužitelj ne šalje pristupni žeton direktno klijentu, nego HTTP preusmjeravanjem. Ovaj način se preporučuje koristiti kada se pristupnim žetonom manipulira pomoću JavaScript programskog jezika na strani klijenta ili mobilnih aplikacija.

- Autentikacija klijenta - klijent se autenticira u prvom koraku kako bi se dohvatio pristupni žeton s autorizacijskog poslužitelja. Klijent na ovaj način može pristupiti podacima bez dodatnih informacija od vlasnika resursa. Na ovaj način, primjerice, statistički softver može iščitati i analizirati podatke klijenata.

Pomoću pristupnog žetona klijent ostvaruje prava pristupa zaštićenim resursima. Iz tog razloga, pristupni žeton mora biti zaštićen; kada bi neautorizirane osobe ostvarile prava pristupa nad resursima mogle bi se pretvarati da su vlasnici tih resursa i izvršavati akcije nad njima, što je potencijalno vrlo opasno.

Pristupni žeton može sadržavati neke dodatne enkodirane informacije. Primjerice, mogu biti sadržane informacije poput punog imena vlasnika resursa, te razinu prava koja vlasnik resursa ima. Razina prava nad resursima određuje koje akcije se mogu izvršavati nad kojim resursima, i mogu biti pridodana korisniku specifično ili cijeloj grupi korisnika.

4.2.2. JSON Web Žeton

Prema IETF (2015) JSON Web Žeton (eng. *JSON Web Token - JWT*) je kompaktan i siguran način prenošenja informacija između dvije strane. Informacije unutar JSON Web Token-a su enkodirane kao JSON objekt koji se koristi kao teret od komponente JSON Web Potpisa (eng. *JSON Web Signature*) ili kao čitljiv tekst JSON Web Enkripcije (eng. *JSON Web Encryption*). Integritet tereta se može osigurati pomoću implementacije digitalnog potpisa i MAC-a (eng. *Message Authentication Code*).

U mikroservisnoj arhitekturi korisnik se može autenticirati pomoću jednim od OAuth2 pristupa. Nakon toga, korisnik ima pristup korisničkom sučelju određenog mikroservisa ili pak može poslati REST zahtjev. Svaki zahtjev korisnika može proći kroz nekoliko mikroservisa koji, koristeći se pristupnim žetonom korisnika, određuju da li korisnik ima dovoljna prava pristupa. U slučaju JSON Web Žetona, žeton se najprije mora dekriptirati i zatim provjeriti da li je ispravan digitalni potpis autorizacijskog poslužitelja. Posljedično, odluka da li korisnik može ili ne može koristiti usluge mikroservisa na način na koji želi ovisi o informacijama koje nosi pristupni žeton.

Važni je napomenuti da pristupni žeton ne smije sadržavati informacije o tome kojem mikroservisu korisnik smije pristupiti, a kojem ne smije. Pristupni žeton, prema OAuth2 standardnoj specifikaciji, se izdaje od strane autorizacijskog poslužitelja. Kada bi se prava pristupa mikroservisima određivala na autorizacijskom poslužitelju, tada bi postajala opasnost od napada na centralno mjesto koje je zaduženo za određivanja prava pristupa nad cijelim sustavom, te bi se tada prirodno urođena sigurnost distribuiranih sustava poput mikroservisa umanjila ili čak poništila. Stoga, autorizacijski poslužitelj jedino mora upravljati korisničkim grupama, dok pojedinačni mikroservisi odlučuju pravo pristupa na temelju informacija iz pristupnog žetona (Nadareishvili, Mitra, McLarty i dr., 2016).

5. Uvod u Spring Boot programski okvir

Nakon što smo završili s teorijskim dijelom mikroservisa kao neovisnih komponenata koje nude neku specifičnu uslugu korisniku i dizajnom mikroservisne arhitekture kao distribuirani skup mikroservisa kojima se može pristupiti pomoću implementiranog sučelja, ovim poglavljem započet ćemo s praktičnim dijelom rada. U nastavku ćemo se upoznati s *Spring* i *Spring Boot* programskim okvirom kojeg ćemo kasnije koristiti za razvoj web aplikacije koja koristi mikroservisnu arhitekturu.

5.1. Spring

Spring je iz početka predstavljao laganiju (eng. *Lightweight*) alternativu za Java Enterprise Edition - omogućavao je razvoj inače velikih Enterprise Java Bean-ova na jednostavniji način koristeći se mehanizmima injekcija ovisnosti (eng. *Dependency Injection*) i aspektno-orijentiranog programiranja (eng. *Aspect-Oriented Programming*) kako bi se postigli jednaki kapaciteti Enterprise Java Bean-ova koristeći se samo jednostavnim Java klasama (eng. *Plain Old Java Object - POJO*) (Walls, 2016; Gutierrez, 2016; Carnell, 2017).

Ali, Spring je imao jedan veliki nedostatak - svaki projekt je morao biti konfiguriran pomoću XML-a. Mnogo XML-a. Uključivanje dodatnih Spring biblioteka poput Spring transakcija i Spring MVC (eng. *Model-View-Controller*) je zahtijevalo dodatne eksplicitne XML konfiguracije. Dodavanje Thymeleaf biblioteke za manipuliranje HTML pogledima je zahtijevalo dodatnu eksplicitnu konfiguraciju. Konfiguriranje servleta i filtera je zahtijevalo eksplicitnu konfiguraciju u *web.xml* datoteci. Od Spring 2.5 verzije, nadodane su anotacije i Java bazirane anotacije koje bitno umanjuju XML konfiguracije, ali ni tada nije bilo bijega od silne konfiguracije. Sve navedeno je stvaralo trenje u razvoju programskog proizvoda, svako vrijeme koje je iskorišteno za konfiguriranje je vrijeme koje nije iskorišteno za implementaciju funkcionalnosti aplikacije. Ovakav način rada nije pomagao Spring programskom okviru u rješavanju poslovnih izazova (Walls, 2016). *Spring Boot* mijenja taj način rada. Spring Boot jest dio Spring programskog okvira, ali uvodi nove načine rada koji omogućuju brži i lakši način podizanja web aplikacija i uz gotovo nikakvu konfiguraciju.

5.2. Spring Boot

Prema Webb, Syer, Long i dr. (2013) Spring Boot omogućuje lako kreiranje samostalnih web aplikacija spremnih za produkcijsko okruženje temeljenih na Spring programskom okviru. Spring Boot preuzima samo potrebno od Spring platforme i drugih biblioteka, da bi se web aplikacija mogla započeti razvijati s minimalno gnjavaže. Spring Boot također omogućuje kreiranje Java aplikacija pomoću sučelja naredbenog retka (eng. *Command Line Interface*). Primarni ciljevi Spring Boot programskog okvira su:

- Pružiti radikalno brže i široko dostupno iskustvo u kreiranju početničkih aplikacija za sav

Spring razvoj,

- Pružati funkcionalnosti koje je moguće iskoristiti bez da se ta funkcionalnost razvija lokalno (eng. *Out of the Box - OOTB*), ali omogućiti neometan razvoj vlastitih funkcionalnosti čim Spring Boot OOTB funkcionalnosti više nisu adekvatne,
- Pružati niz nefunkcionalnih dodataka koji su uobičajeni za neki projekt (ugrađeni poslužitelji, sigurnost, metrike, provjere ispravnosti aplikacije, vanjska konfiguracija),
- Potpuno uklanjanje automatskog generiranja kôda i zahtjeva za XML konfiguracijom.

Kako Spring Boot to postiže? Walls, 2016, spominje ova četiri esencijalna pojma koji svaki na svoj način pojednostavljuju razvoj Spring Boot aplikacije:

- **Automatska konfiguracija** - Spring Boot automatski pruža konfiguracije koje su u praksi vrlo često korištene, te se u aplikacije mogu integrirati uz minimalno podešavanja,
- **Početne biblioteke** - Spring Boot osigurava da će biti uključene biblioteke koje su potrebne za aplikacijske funkcionalnosti koje razvojni inženjer namjerava razvijati,
- **Sučelje naredbenog retka** - Omogućuje razvijanje i pokretanje kompletnih aplikacija samo uz pomoć kôda, bez tradicionalnog kompiliranja projekta,
- **Pokretač** (eng. *Actuator*) - Pruža uvid u proces izvršavanja Spring Boot aplikacije.

Ali, bitno je i napomenuti stvari koje Spring Boot ne ostvaruje, kako bismo dobili sveukupnu sliku mogućnosti Spring Boot programskog okvira. Prva stvar koju je važno spomenuti je ta da Spring Boot nije poslužitelj aplikacija. Ovo je često pogrešno mišljenje koje proizlazi iz činjenice da je moguće pokrenuti Spring Boot web aplikacije pomoću sučelja naredbenog retka bez tradicionalnog poslužitelja. Spring Boot omogućuje tu funkcionalnost zato što sadrži ugrađeni, lokalni poslužitelj (Tomcat, Jetty ili Undertow) unutar web aplikacije. Spring Boot također ne implementira Java specifikacije poput Java Persistence API (JPA) ili Java Message Service (JMS). Spring Boot podržava Java specifikacije, ali je potrebno uključiti potrebne anotacije koje automatski konfiguriraju specifična zrna (eng. *Beans*) koja će biti zadužena za implementaciju Java specifikacije.

Za kraj je još važno podsjetiti da je Spring Boot iznutra Spring - sadrži iste konfiguracije koje bismo morali konfigurirati da Spring Boot to ne učini za nas, kao u starijem Spring programskom okviru.

5.3. Spring Cloud

Jedan od temeljnih koncepata mikroservisne arhitekture je taj da se svaki mikroservis upakirava i isporučuje kao samostalni i neovisni artefakt. Instance servisa bi se trebale podizati relativno brzo i svaka instanca se mora razlikovati od druge. To znači da ćemo, prije ili kasnije, morati odlučiti gdje ćemo isporučiti i održavati servis (Carnell, 2017):

- **Fizički server** koristi sve manje organizacija kao preferirani način pružanja usluga. Fizički server je potrebno održavati, teško je i skupo povećati kapacitete u slučaju da potreba naiđe, te može biti vrlo skupo horizontalno skalirati mikroservise na više fizičkih servera.
- **Virtualne mašine** (eng. *Virtual Machine Images*) - ključna prednost virtualnih slika je njihova mogućnost da se mogu u vrlo kratkom roku pokrenuti i zaustaviti instance mikroservisa kao odgovor na potrebe skalabilnosti i događaje ispada iz mreže. Virtualne mašine su najvažnija komponenta svih većih pružatelja usluga u oblaku (eng. *Cloud Services*). Mikroservis se može upakirati u virtualnu mašinu i tako se može kreirati nekoliko instanca servisa koje će se brzo i efikasno pokrenuti.
- **Virtualni kontejner** (eng. *Virtual Container*) je rezultat isporuke mikroservisa u virtualnim mašinama. Umjesto da se servis isporuči kao cijela virtualna mašina, mnogi to čine u kontejnerskim tehnologijama poput Dockera. Virtualni kontejneri se pokreću unutar virtualne mašine, te omogućuju segregaciju pojedinih virtualnih mašina u niz procesa koji dijele tu sliku virtualne mašine na makro razini.

Prednost mikroservisne arhitekture koja se temelji na tehnologijama u oblaku je elastičnost. Poslužitelji usluga u oblaku dozvoljavaju brzinsko pokretanje novih virtualnih mašina i kontejnera. Ako pak kapaciteti dozvoljavaju da ugasimo pojedine virtualne mašine, to se može učiniti bez ikakvih troškova. Korištenjem poslužitelja usluga u oblaku postiže se izrazita horizontalna skalabilnost pomoću tehnika koje smo opisali virtualnim mašinama i kontejnerima. Ovakva elastičnost dovodi do povećane otpornosti. Ako se jedan od mikroservisa ruši uslijed pogrešaka iznutra, možemo pokrenuti nekoliko instanca servisa da se aplikacija održi na životu dovoljno dugo da se isprave pogreške.

Spring Cloud projekt je pod kišobranom (eng. *Umbrella Project*) Spring razvojnog tima. Spring Cloud implementira skup standardnih uzoraka dizajna koje zahtijevaju distribuirani sustavi, u obliku skupa Java Spring biblioteka. Unatoč imenu koje sugerira da je Spring Cloud rješenje u oblaku, ono to nije - Spring Cloud zato pruža mogućnosti koje su neizbježne pri razvoju aplikacija koje se razvijaju s ciljem da se isporuče na oblaku. Takve aplikacije podliježu principu pod nazivom *Dvanaest-Faktorna Aplikacija* (Carnell, 2017; RV, 2017):

- *Repozitorij programskog kôda* (eng. *Codebase*) - sav aplikacijski programski kôd i informacije poslužitelja moraju biti verzionirane. Svaki mikroservis mora imati nezavisan repozitorij programskog koda na nekom od sustava za verzioniranje izvornog kôda.
- *Ovisnosti* - eksplicitna deklaracija ovisnosti aplikacije putem nekih od alata razvoja poput Java *Maven*. JAR (eng. *Java Archive*) biblioteke se moraju deklarirati brojem verzije biblioteke. Time se osigurava da se mikroservis uvijek razvija istim verzijama biblioteka.
- *Konfiguracija* - konfiguracija aplikacije se mora spremati izdvojeno od izvornog kôda aplikacije. Idealno, konfiguracija se ne bi smjela nalaziti u istom repozitoriju projekta kao i izvorni kôd.
- *Prateće usluge* (eng. *Backing Services*) - potrebno je osigurati da je u bilo kojem vremenu moguće jednostavno zamijeniti implementaciju pristupa bazi podataka u internom servisu

(eng. *In-House Manages Service*) s implementacijom servisa treće strane. Primjerice, zamjena naše MySQL baze podataka s bazom podataka kojom upravlja Amazon.

- *Razvoj, isporuka, produkcija* - načelo koje se odnosi na princip razdvajanja razvoja aplikacije, isporuke i puštanje u produkciju aplikacije. Jednom kada je programski kôd razvijen, razvojni inženjer nikad ne smije raditi promjene za vrijeme pokretanja aplikacije. Svaka promjena se mora učiniti u razvojnom procesu, te se tada radi ponovna isporuka.

- *Procesi* - mikroservisi moraju uvijek biti bez stanja (eng. *Stateless*). Mikroservise možemo zaustaviti i zamijeniti u bilo kojem trenutku bez straha od gubitka podataka.

- *Vezivanje na mrežna vrata* (eng. *Port binding*) - mikroservis je potrebno moguće pokrenuti bez ovisnosti o nekoj drugoj aplikaciji ili web poslužitelja. Servis bi se trebao pokrenuti samostalno na komandnom sučelju i odmah mora biti dostupan preko HTTP mrežnih vratiju.

- *Konkurentnost* (eng. Concurrency) - skalabilnost je potrebno postići horizontalnim širenjem u obliku stvaranja novih instanca mikroservisa, umjesto razvijanja višedretvenog modela unutar jednog servisa. To ne znači da je loše razviti višedretveni rad unutar mikroservisa, nego ovaj princip govori o tome da višedretveni rad ne smije biti jedini mehanizam skalabilnosti.

- *Upotrebljivost* (eng. *Disposability*) - Mikroservisi moraju omogućiti takvu upotrebljivost da se mogu pokrenuti i zaustaviti prema zahtjevu. Vrijeme pokretanja mora biti minimalnog trajanja, a procesi se moraju u miru zaustaviti prije nego se mikroservis ugasi.

- *Paritet razvojnih i produkcijskih okolina* - potrebno je minimizirati razliku između svih postojećih okolina rada u kojima se servis pokreće. Razvojni inženjer mora koristiti identičnu infrastrukturu na lokalnom računalu koja se koristi i za isporuku servisa, gdje će se servis i koristiti u produkciji. Ovaj princip također nalaže da vrijeme isporuke servisa mora biti u satima, svakako ne u danima ili tjednima. U idealnom scenariju, čim se programski kôd završi, mora biti testiran i poslan u produkcijsku okolinu u što kraćem roku.

- *Dnevnik* (eng. *Logging*) - svi događaji moraju biti zabilježeni u dnevnik. Dnevnik je potrebno upravljati nekakvim alatima koji omogućuju grupiranje podataka i slanja na centralnu lokaciju. Mikroservisi se ne smiju baviti s logikom ili mehanikom kako se ovaj proces odvija, te bi razvojni inženjer mora vizualno proučiti događaje na nekom zaslonu.

- *Administratorski procesi* - razvojni inženjeri će često morati izvršavati administratorske zadatke nad mikroservisima, primjerice migracija i konverzije podataka. Ovaj princip nalaže da se ovakvi zadaci moraju izvršavati pomoću skripata koje su pripremljene na razini repozitorija mikroservisa. Izvorni kôd skripata mora biti identičan u svim okolinama rada, te se ovi zadaci nikad ne izvršavaju ad-hoc.

Koristeći se Spring Cloud programskim okvirom, razvojni inženjeri se samo moraju fokusirati na razvoj poslovnih funkcionalnosti koristeći se Spring Boot programskim okvirom. Sve ostale pogodnosti poput distribuiranosti, otpornost na pogreške i samo-oporavljanje pruža Spring Cloud. Spring Cloud je razvijen pridržavajući se pristupa kojeg se drži i Spring Boot:

konvencija iznad konfiguracije, što znači da Spring Cloud omogućuje razvojnim inženjerima brz start s minimalno posla oko konfiguriranja aplikacija. Također, sakriva kompleksnosti i pruža jednostavno deklarativno sučelje konfiguracije da se izgrade sustavi. Spring Cloud nudi široki izbor rješenja za razvijanje registra servisa, te se može odabrati ono rješenje koje najbliže odgovara potrebama. Primjerice, postoje popularne opcije pod nazivima Eureka, Consul ili Zookeeper (RV, 2017). U praktičnom dijelu ovog rada koristit ćemo se Eureka registrom servisa.

5.3.1. Otkrivanje servisa pomoću Spring Cloud Eureka poslužitelja

Prije upoznavanja s Eureka poslužiteljem, moramo se upoznati s pojmovima *dinamična registracija* (eng. *Dynamic registration*) i *dinamično otkrivanje* (eng. *Dynamic discovery*) (RV, 2017):

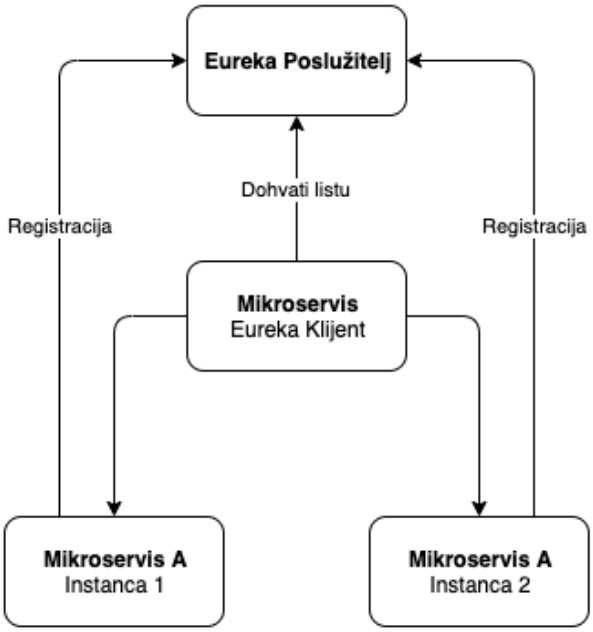
Dinamična registracija se primarno odnosi na poslužitelja usluga. S dinamičnom registracijom kada se pokrene novi servis, ono automatski objavi svoju dostupnost u centralnom servisnom registru. Vrijedi i suprotno - kada se servis ugasi, automatski se uklanja s popisa servisa na centralnom servisnom registru. Registar u svakom trenutno održava točnu informaciju koji servisi su dostupni sa svim njihovim podacima.

Dinamično otkrivanje se pak odnosi na korisnika servisa. U ovoj situaciji korisnik pretražuje servisni registar kako bi saznao kako može doći do traženog servisa. Ovdje se podrazumijeva situacija gdje korisnik ne poznaje i ne zna gdje se nalazi potrebni servis, pa zato pretražuje centralni registar servisa. Jednom kada korisnik dozna gdje se nalazi servis, poveznicu sprema u međuspremnik kako ne bi iznova pretraživao registar servisa.

Eureka poslužitelj ćemo primarno koristiti za dinamično otkrivanje i balansiranje opterećenja (eng. *Load Balancing*). Eureka se sastoji od komponente poslužitelja i komponenata klijenata: komponenta poslužitelja igra ulogu servisnog registra u kojem svi mikroservisi objavljuju svoju dostupnost, dok komponente klijenata su sami mikroservisi koji se tijekom pokretanja objave komponenti poslužitelja. Spomenuta registracija komponenta klijenata se odvija tako da tijekom registracije, komponenta klijenta uključuje svoj identitet i poveznicu (eng. *Uniform Resource Locator - URL*) preko koje se može dohvatiti. Jednom kada se mikroservis registrira kod centralnog registra servisa, registar servisa (komponenta poslužitelja) svakih 30 sekundi provjerava dostupnost svih registriranih mikroservisa. Taj događaj se još naziva *otkucaji srca* (eng. *Hearbeat*). Ako komponenta poslužitelja u nekoliko navrata ne može dohvatiti mikroservis, ono uklanja pristupnu točku dotičnog mikroservisa s popisa registriranih servisa.

Komponente klijenata minimalno jednom moraju kontaktirati registar. U tom trenutku se informacija koju drži registar replicira na klijenta i on ju sprema u lokalni među spremnik, kako bi se smanjio broj poziva prema registru. Otkucajima srca registra, ova informacija se osvježava periodično svakih 30 sekundi.

U nadi da smo postavili dobre temelje za razumijevanje Spring Boot i Spring Cloud programskih okvira, u nastavku ćemo proći kroz instalaciju Spring Boot programskog okvira i inicijalizaciju projekata u kojima ćemo nastaviti razvijati web aplikaciju.



Slika 5: Spring Cloud Eureka (Izvor: RV, 2017)

5.4. Instalacija i korištenje Spring Boot programskog okvira

Spomenut ćemo dva različita načina instalacije i korištenja Spring Boot programskog okvira:

1. Spring Boot CLI
2. Spring Initializr

Spring Boot CLI je pristup korištenju Spring Boot programskog okvira pomoću sučelja naredbenog retka. Da bi se Spring Boot CLI koristio, najprije ga je potrebno instalirati. Postoji nekoliko različita načina za to: iz preuzete Spring Boot distribucije, koristeći se Groovy Environment Manager-om, pomoću Homebrew za OS X operacijski sustav, itd. Nakon što se je Spring Boot CLI uspješno instalirao, sve naredbe se upisuju u sučelje za naredbeni redak poput Command Prompt alata za Windows OS. Ključna riječ kojom započinje svaka naredba je *spring*. Pa tako primjerice, da bismo provjerili koja verzija Spring Boot programskog okvira je instalirana potrebno upisati:

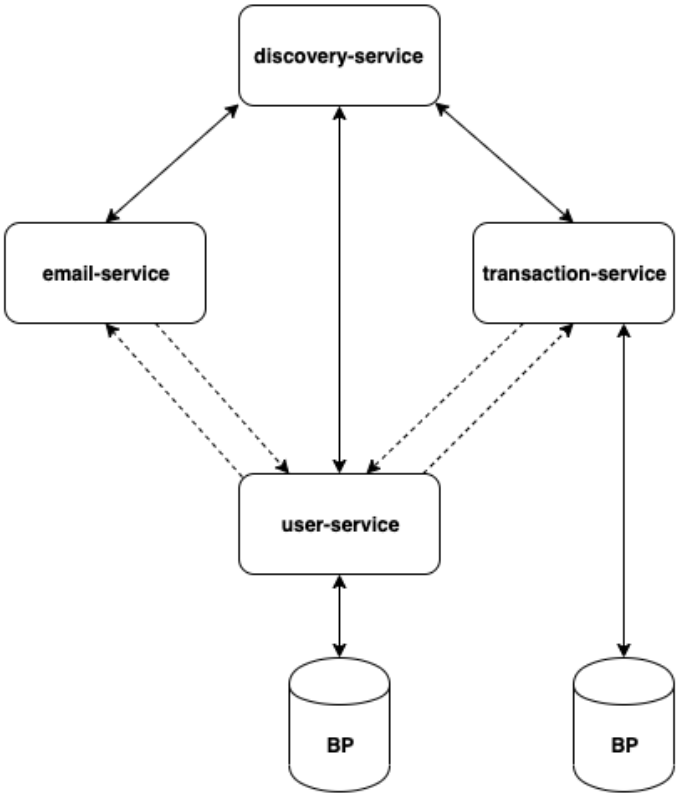
```
spring --version
```

Ako je Spring Boot CLI ispravno instaliran, naredba će se izvršiti i ispisat će se verzija Spring Boot programskog okvira koja je instalirana na računalu.

Spring Initializr (https://start.spring.io/) je web aplikacija koja omogućuje generiranje strukture Spring Boot projekta. Ne generira nikakav aplikacijski kod, ali daje osnovnu strukturu ovisno o nazivu projekta kojeg smo specificirali, nazivima projektnih artefakata, bibliotekama koje smo uključili, specifikacija tehnologije pomoću koje ćemo kompilirati kod (Maven ili Gradle),

itd. Spring Initializr se također može kombinirati sa Spring Boot CLI načinom korištenja Spring Boot programskog okvira. Pomoću Spring Initializr web aplikacije sada ćemo kreirati četiri različita projekta:

- **discovery-service** - obavljat će ulogu komponente poslužitelja kao Eureka registar servisa. Ovisnost koju moramo dodati ovom projektu je *Eureka Server*.
- **user-service** - obavljat će ulogu korisničkog mikroservisa, kao Eureka klijent. Ovaj mikroservis će jedini implementirati i korisničko sučelje, te će imati pristup bazi podataka. Stoga, ovisnosti koje moramo dodati su:
  - *Eureka Discovery* - uključuje Eureka klijent i dinamičnu registraciju servisa.
  - *Session* - uključuje API i implementacije koje će se koristiti za upravljanje informacijama u korisničkoj sesiji,
  - *Web* - uključuje servlet web aplikaciju sa Spring MVC i Tomcat ugrađenim poslužiteljem,
  - *Thymeleaf* - uključuje XML/XHTML/HTML5 sustav predložaka,
  - *Security* - uključuje spring-security konfiguraciju pomoću koje ćemo osigurati web aplikaciju,
  - *JPA* - uključuje Spring Data i Hibernate JPA pomoću kojih ćemo komunicirati s bazom podataka,
  - *MySQL* - MySQL JDBC upravljački program (eng. *Driver*)
- **email-service** - email mikroservis koji će pozivati samo korisnički mikroservis, neće imati pristup bazi podataka niti korisničko sučelje. Sadržavat će metode za slanje email poruka, stoga je potrebno dodati ovisnost:
  - *Eureka Discovery* - uključuje Eureka klijent i dinamičnu registraciju servisa.
  - *Mail* - uključuje skup ovisnosti za rad s email porukama
- **transaction-service** - transakcijski mikroservis koji poziva korisnički mikroservis. Transakcijski mikroservis će imati pristup bazi podataka, ali neće imati korisničko sučelje. Ovisnosti koje je potrebno dodati:
  - *Eureka Discovery* - uključuje Eureka klijent i dinamičnu registraciju servisa.
  - *JPA* - uključuje Spring Data i Hibernate JPA pomoću kojih ćemo komunicirati s bazom podataka,
  - *MySQL* - MySQL JDBC upravljački program (eng. *Driver*)



Slika 6: Arhitektura mikroservisne aplikacije

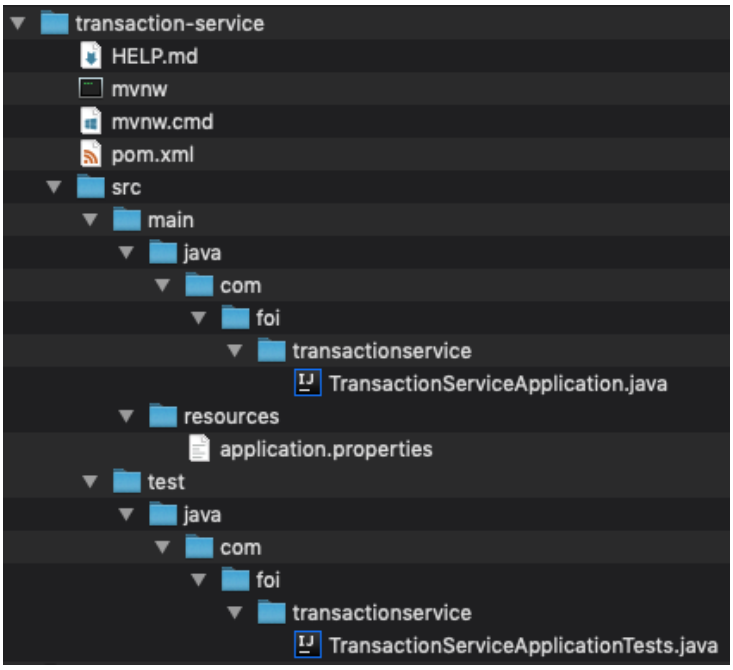
Navedene ovisnosti će biti potrebne za početni razvoj web aplikacije, te nije važno ako se na početku ne uključe sve ovisnosti. Maven (i Gradle) projekt omogućuje dodavanje ovisnosti po potrebi tijekom i u svim fazama razvoja aplikacije.

Klikom na *Generate Project* se kreira projekt sa specificiranom strukturom, te se spremi u sažetom obliku u .zip datoteci. Kada raspakiramo projekt, možemo primijetiti nekoliko datoteka, od kojih bismo izdvojili *pom.xml*.

pom.xml (eng. *Project Object Model*) je fundamentalna jedinica rada u Mavenu, te ona čuva u XML obliku informacije o projektu i konfiguracijskim detaljima koje Maven koristi kako bi izgradio projekt (ASF, 2019). Nama će *pom.xml* biti od važnosti zato što ćemo u ovu datoteku nadodavati ovisnosti po potrebi kroz razvoj aplikacije. Kada se doda ovisnost, potrebno je pričekati da Maven izgradi projekt i tada se klase iz te ovisnosti tada mogu koristiti u aplikaciji.

Sljedeća važna datoteka jest *src* direktorij. *src* direktorij sadrži još dva direktorija: *main* i *test*. Direktorij *test* sadrži pakete u kojima se nalaze testovi aplikacije, poput jediničnih i integracijskih testova. Direktorij *main* pak sadrži pakete aplikacije, gdje će se nalaziti sve naše buduće klase koje ćemo razvrstati po paketima ovisno o njihovoj namijeni. Direktorij *main* sadrži i direktorij *resources* koji sadrži dodatne direktorije u kojima ćemo kreirati poglede, dizajn stranice i sav programski kôd koji će se izvršavati na korisničkoj strani, primjerice JavaScript. I zadnja važna datoteka koja će biti od koristi je *application.properties* u *resources* direktoriju. Datoteka *application.properties* će sadržavati sve postavke koje se odnose na globalnu aplikaciju, poput poveznice do baze podataka, pristupni podaci za bazu podataka, konfiguracijski parametri za

email poslužitelj, itd.



Slika 7: Struktura transaction-service projekta

Nakon što smo se upoznali sa strukturom naših projekata, započet ćemo s razvojem naše web aplikacije koja se sastoji od komponenata mikroservisne arhitekture: discovery-servis, user-service, transaction-service, email-service. U sljedećem poglavlju napravit ćemo uvod u aplikaciju i navesti zahtjeve koje mora implementirati.

6. Razvoj mikroservisne arhitekture u Spring Boot programskom okviru

Od ovog poglavlja krenut ćemo s razvojem web aplikacije. Aplikaciju ćemo razvijati u projektima koje smo kreirali u prethodnom poglavlju. U ovom poglavlju naučit ćemo osnovne i naprednije elemente programiranja pomoću Java Spring Boot programskog okvira, te naučiti koristiti alate koji će pomoći u uspostavljanju baze podataka, u testiranju REST krajnjih točaka, i sl. Najprije ćemo se detaljno proći kroz tehničke zahtjeve gdje ćemo detaljno pojasniti tehničke zahtjeve koje web aplikacija mora realizirati, u opsegu jedne web aplikacije. U tehničkim zahtjevima razradit ćemo i konceptualni dizajn baze podataka, te ćemo pojasniti alat kojim ćemo izraditi bazu podataka.

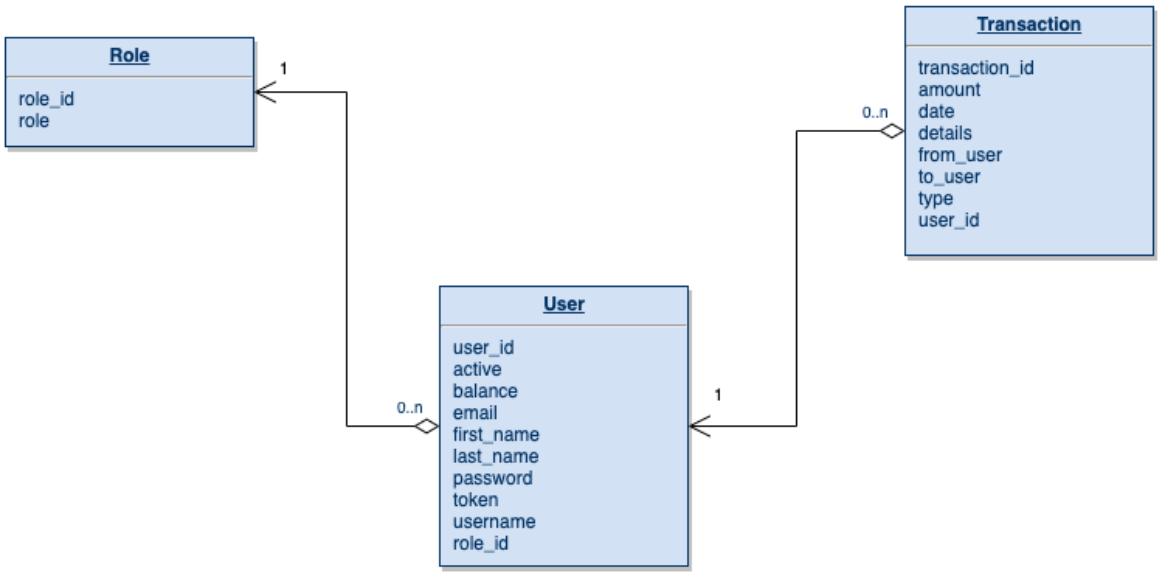
6.1. Tehnički zahtjevi web aplikacije

Pod tehničkim zahtjevima web aplikacije podrazumijevamo informacije koje se tiču tehničkog dizajna, razvoja i procedura koje se odnose na zacrtane zahtjeve našeg programskog proizvoda. U većim, poslovnim sustavima tehnička dokumentacija u detalje opisuje poslovne zahtjeve, standarde i najbolje prakse, te specificira zahtjeve sustava - funkcionalne, sučelja i dizajn. Osim zahtjeva, može se pronaći i popis pojmova, referenca na druge dokumente, rizika, ovisnosti i pretpostavka.

U ovom poglavlju nećemo opisivati opsežnu tehničku dokumentaciju, već ćemo se ukratko upoznati s alatima i procedurama kojih ćemo se pridržavati, te će biti naglasak na način kako ćemo razviti našu aplikaciju i uspješno realizirati korisničke zahtjeve. Također ćemo se upoznati i s alatima koji će pružati podršku razvoju naše web aplikacije, kao što je poslužitelj baze podataka i alat za testiranje REST krajnjih točaka naših mikroservisa.

Osim spremnih Spring Boot projekata (mikroservisa) koje smo kreirali pomoću Spring Initializr-a, najvažniji alat koji ćemo koristiti u svakom trenutku je XAMPP. Pomoću XAMPP alata pokrenut ćemo lokalni Apache Web Server, MySQL Database i ProFTPD poslužitelje koje ćemo koristiti za upravljanje bazom podataka. Kada su svi poslužitelji uspješno pokrenuti, pomoću web preglednika pristupamo *phpMyAdmin*-u koji se nalazi na *localhost* ili *127.0.0.1* lokalnoj adresi. PhpMyAdmin ćemo koristiti za kreiranje i administriranje naše MySQL baze podataka.

Za svrhe ove web aplikacije kreirat ćemo novu bazu podataka pod nazivom *myfinance*. Baza podataka će sadržavati sljedeće tablice: *Uloga* (eng. *Role*), *Korisnik* (eng. *User*) i *Transakcija* (eng. *Transaction*). Struktura baze podataka je prikazana ERA dijagramom (slika 8). Prikazana struktura će biti dovoljna da pohranjuje sve relevantne informacije koje ćemo koristiti za spremanje korisnika, uloga i transakcija provedenih u aplikaciji. Umjesto direktnog pisanja SQL-a za kreiranje tablica, koristit ćemo objektno-relacijsko mapiranje (eng. *Object-Relational Mapping - ORM*) za kreiranje i upravljanje podacima u bazi podataka. Točnije, koristit ćemo *Hibernate ORM* alat za Java programski jezik. U nastavku slijede detaljni opisi tablica i njihovih atributa (tablica 2, tablica 3, tablica 4).



Slika 8: ERA dijagram

Tablica 2: Tablica Uloga

Atribut	**Opis**	**Tip podatka**	**Ograničenje**
role_id	Jedinstveni identifikator	INT(11)	Primarni ključ
role	Tekstualni opis uloge	VARCHAR(255)	Obavezan upis

Tablica 3: Tablica Korisnik

Atribut	**Opis**	**Tip podatka**	**Ograničenje**
user_id	Jedinstveni identifikator	INT(11)	Primarni ključ
active	Aktiviran korisnički račun	BIT(1)	Nema
balance	Saldo računa	DOUBLE	Nema
email	Email adresa	VARCHAR(255)	Obavezan upis, jedinstvena
first_name	Ime	VARCHAR(255)	Obavezan upis
last_name	Prezime	VARCHAR(255)	Obavezan upis
password	Lozinka	VARCHAR(255)	Obavezan upis
token	Žeton pomoću kojeg će se mijenjati lozinka i aktivirati korisnički račun	VARCHAR(255)	Nema
username	Korisničko ime	VARCHAR(255)	Obavezan upis, jedinstveno
role_id	Uloga	INT(11)	Vanjski ključ

Tablica 4: Tablica Transakcija

Atribut	Opis	Tip podatka	Ograničenje
transaction_id	Jedinstveni identifikator	INT(11)	Primarni ključ
amount	Iznos	DOUBLE	Obavezan upis
date	Datum	DATE	Obavezan upis
details	Detalji transakcije	VARCHAR(255)	Nema
from_user	Pošiljatelj	VARCHAR(255)	Obavezan upis
to_user	Primatelj	VARCHAR(255)	Obavezan upis
type	Tip transakcije	VARCHAR(255)	Obavezan upis
user_id	Korisnik koji je kreirao transakciju	INT(11)	Vanjski ključ

Tablice *Uloga* i *Korisnik* su povezane vezom jedan-na-više (eng. *One-to-Many*), što znači da jedan slog iz tablice *Korisnik* može biti povezan s jednim i samo jednim slogom iz tablice *Uloga*, ili drugim riječima jedan slog iz tablice *Uloga* može imati više slogova u tablici *Korisnik*. Uloga može biti ili "običan korisnik" ili "administrator". S druge strane, jedan slog iz tablice *Korisnik* može imati više slogova u tablici *Transakcija*, ili drugim riječima jedan slog iz tablice *Transakcija* može biti povezan s jednim i samo jednim slogom u tablici *Korisnik*. Ukratko, jedan korisnik može imati jednu ulogu i može kreirati više transakcija, dok jednu ulogu može imati više korisnika i jednu transakciju može napraviti samo jedan korisnik.

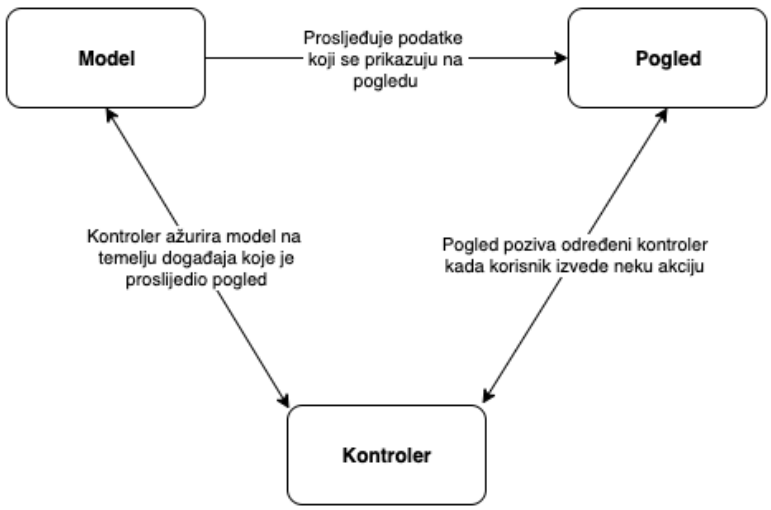
Ove tablice ćemo zapisati u obliku programskog koda tako da za svaku tablicu kreiramo klasu i anotiramo ju s anotacijom *@Entity*. Hibernate alat za objektno-relacijsko mapiranje će pretvoriti navedene klase u oblik relacijske tablice. Stupci entitetnih klasa se anotiraju s anotacijom *@Column*. Dodatna ograničenja poput obaveznog upisa se anotiraju s *@NotNull* - postoji mnogo anotacija, te ih mi nećemo sve prolaziti, već ćemo pojasniti one važnije i često korištene. Vanjski ključevi se mogu identificirati pomoću anotacija kardinalnosti i ograničenja poput *@ManyToOne* i *@OnDelete*. Još je važno spomenuti kada Hibernate kreira navedene tablice - u *resources/application.properties* datoteci možemo opisati neka svojstva aplikacije koja vrijede globalno za pojedinu aplikaciju. Za Hibernate postoji svojstvo *spring.jpa.hibernate.ddl-auto* na koje možemo pridružiti NONE, VALIDATE, UPDATE, CREATE ili CREATE-DROP. Mi ćemo koristiti UPDATE i CREATE-DROP. UPDATE svojstvo će pri pokretanju aplikacije provjeriti razlike između baze podataka i entitetnih klasa, te ažurirati bazu podataka s novostima entitetnih klasa, bez gubitka podataka. CREATE-DROP pak pri pokretanju aplikacije najprije obriše sve tablice i stupce s njihovim podacima, te se zatim kreiraju iznova prazne tablice. U početku kada se rade brojne izmjene nad bazom podataka i nema funkcionalnosti prijave i registracije, možemo držati svojstvo CREATE-DROP dok nećemo biti sigurni da su tablice u bazi podataka kompletne. Nakon što implementiramo funkcionalnost registracije, važno je promijeniti svojstvo na UPDATE da se svaka promjena nad tablicama ažurira bez gubitka podataka. Osim Hibernate svojstva, u *application.properties* ćemo staviti poveznicu na našu bazu podataka, te validno korisničko ime i lozinku kojom će se aplikacija autenticirati prema bazi podataka. Svaki mikroservis ćemo pokrenuti kao zasebnu Spring Boot aplikaciju s kojom se može komunicirati preko mrežnim vratiju. Kako bismo to postigli, u spomenutoj *application.properties* datoteci

moramo upisati *server.port* parametar. Eureka poslužitelj će biti dostupan na mrežnim vratima pod brojem 8761, korisnički mikroservis će biti dostupan pod brojem 8081 pod nazivom *user-service* (parametar *spring.application.name*), email mikroservis pod brojem 8082 i pod nazivom *email-service*, te transakcijski mikroservis pod brojem 8083 pod nazivom *transaction-service*. Spring Cloud Eureka poslužitelj će voditi evidenciju naših mikroservisa i možemo ih provjeriti putem putanje *http://localhost:8761/*.

Tako će primjerice user-service mikroservis sadržavati sljedeće podatke u *application.properties*:

```
# ===============================
# APPLICATION
# ===============================
# Application configuration
server.port=8081
spring.application.name=user-service
# ===============================
# DATA SOURCE
# ===============================
# Database connection configurations
spring.datasource.url=jdbc:mysql://localhost:3306/myfinance?useUnicode=true&
    useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=
    UTC&useSSL=false
spring.datasource.username=root
spring.datasource.password=root
# ===============================
# JPA - HIBERNATE
# ===============================
# Database Initialization - none, validate, update, create or create-drop
spring.jpa.hibernate.ddl-auto=update
```

Dizajn razvoja web aplikacije će se temeljtiti na MVC (eng. *Model-View-Controller*) uzorku dizajna. MVC suštinski odvaja web aplikaciju na tri komponente (Wood, Loy i Eckstein, 1998):

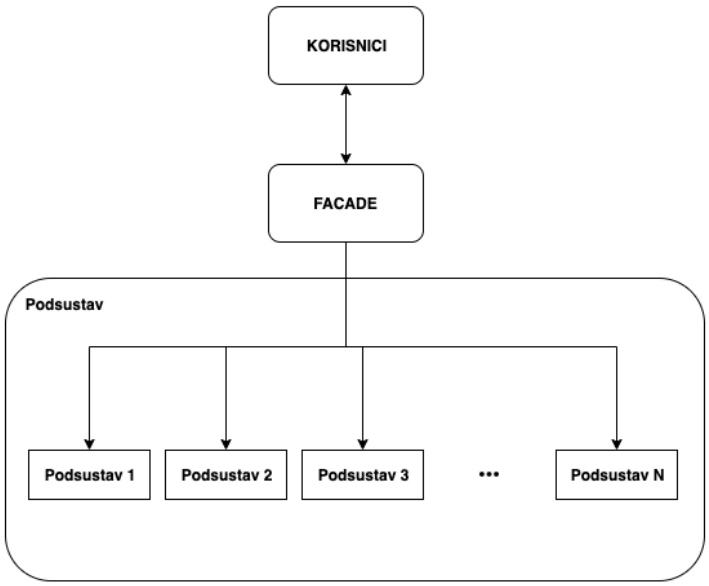


Slika 9: Arhitektura MVC uzorka dizajna (Izvor: Wood, Loy i Eckstein, 1998)

- **Model** (eng. *Model*) - centralna komponenta MVC uzorka dizajna. Model predstavlja dinamičku strukturu aplikacije, potpuno neovisnost od korisničkog sučelja, te upravlja podacima, logikom i pravilima aplikacije. Model prihvaća ulazne podatke od pogleda preko kontrolera.
- **Pogled** (eng, *View*)- bilo koja reprezentacija informacija u nekom obliku (graf, dijagram, tablica). Pogled se odnosi na način na koji prikazujemo informaciju korisniku.
- **Kontroler** (eng. *Controller*) - prihvaća korisničku akciju i pretvara ga u naredbu za model ili pak natrag za pogled. Kontroler prima ulazne podatke od pogleda, najčešće ih validira i tada se šalje modelu, ili da se daje odgovor pogledu. Dobra praksa je da kontroler ne implementira nikakvu kompleksnu logiku, već obavlja interakcije između pogleda i modela, te poziva operacije iz nižih slojeva aplikacije ako je potrebno. U mikroservisnoj arhitekturi implementirat ćemo i REST kontroler koji će pružati sučelje za rad sa servisima.

Naša struktura web aplikacije također će se temeljiti na MVC uzorku dizajna, te kako bismo lakše upravljali klasama kreirat ćemo pakete u kojima će se grupirati srodni elementi web aplikacije. Na slici 8 ukratko je grafički prikazana interakcija između modela, kontrolera i pogleda.

Osim MVC uzorka dizajna, koristit ćemo i Facade uzorak dizajna. Facade je strukurni uzorak dizajna koji pruža sučelje više razine putem kojeg se pristupa sučeljima i operacijama na nižim razinama sustava, tj. omogućava lakše i jednostavnije pozivanje servisa podsustava. U našoj web aplikaciji koristit ćemo Facade da uvedemo slojevitost sustava - Facade će predstavljati sloj između kontrolera i naših mikroservisa. Kontroler će injektirati ovisnost Facade, koje će potom na nižem sloju pozivati sloj mikroservisa, dok će se rezultat obrade vraćati kontroleru pomoću Facade.



Slika 10: Facade uzorak dizajna

Ovime završavamo tehničke zahtjeve web aplikacije. U sljedećem poglavlju započinjemo s korisničkim zahtjevima, te uz opise postupka kako specifični zahtjev implementirati,

prilagat ćemo programski kôd po potrebi kako bi se Spring Boot programski okvir s mikroservisnom arhitekturom prikazao na praktičnom primjeru.

6.2. Korisnički zahtjevi web aplikacije

Korisnički zahtjevi će biti specifične akcije koje korisnik može izvršiti u web aplikaciji. U ovom poglavlju prolazit ćemo logičkim slijedom i implementirati ih u web aplikaciji. Korisničke zahtjeve smo podijelili na zahtjeve za korisnički račun - specifične implementacije koje se odnose na rad s korisničkim računima i informacijama vezanih uz njih, te zahtjeve za funkcionalnost aplikacije - specifične implementacije vezane uz kontekst web aplikacije a odnose se na radnje koje mogu izvršiti autenticirani i potvrđeni korisnici.

Zahtjevi za korisničkim računom su:

- **Prijava** (eng. *Sign in*) - odnosi se na implementaciju pogleda koji sadrži polje za unos korisničkog imena i lozinke. Kada korisnik pritisne gumb, potrebno je provjeriti s bazom podataka da li navedeni par korisničkog imena i unesene lozinke postoji u bazi podataka. Ako ne postoji ili je korisnički račun neaktivan, vrati povratnu informaciju o pogrešci, inače propustiti korisnika i zapamtiti ga u aplikacijskoj sesiji.
- **Registracija** (eng. *Sign up*) - odnosi se na implementaciju pogleda koji sadrži polja za unos imena, prezimena, email adrese, korisničkog imena, lozinke i ponovljene lozinke. Na pritisak gumba, potrebno je validirati unos korisnika. Sva polja moraju biti ispunjena, određene minimalne i maksimalne duljine znakova, email adresa i korisničko ime moraju biti slobodni u bazi podataka, te se lozinka i ponovljena lozinka moraju poklapati. Tada registrirati novog korisnika, ili vratiti poruku o pogrešci.
- **Promjena lozinke** (eng. *Change Password*) - odnosi se na implementaciju pogleda za unos korisničke email adrese, slanje email poruke s jedinstvenim žetonom pomoću kojeg će korisnik izmijeniti lozinku, te pogleda u kojem korisnik unosi novu lozinku. Potrebno je provjeriti upisanu email adresu u bazi podataka, te ako ne postoji vratiti poruku o pogrešci.
- **Aktivacija korisničkog računa** (eng. *User Activation*) - kada se korisnik uspješno registrira, potrebno je poslati email poruku s poveznicom koja aktivira korisnički račun. Sve dok korisnik ne potvrdi legitimnost svoje email adrese klikom na poveznicu za aktivaciju korisničkog računa, mora mu se onemogućiti prijava u aplikaciju.
- **Pregled profila** - odnosi se na implementaciju pogleda koji sadrži formu s imenom, prezimenom, email adresom, korisničkim imenom i praznim poljima za promjenu lozinke. Ovaj pogled može vidjeti samo autenticirani korisnik, te su polja automatski popunjena s korisničkim podacima. Korisniku mora biti omogućeno da ažurira prikazane podatke.
- **Ažuriranje podataka** (eng. *User Data Update*) - omogućava korisniku izmjenu popunjenih podataka na pogledu pregleda profila. Pritiskom na gumb podaci se ažuriraju i spremaju u bazu podataka za dotičnog korisnika koji je izvršio tu akciju.

Zahtjevi za funkcionalnost aplikacije su:

- **Unos transakcija prihoda i rashoda** (eng. *Transaction Entry*) - odnosi se na implementaciju pogleda koji sadrži formu za unos nove transakcije koja može biti prihod (eng. *Income*) ili rashod (eng. *Outcome*). Forma sadrži polja za unos pošiljatelja, primatelja, detaljima transakcije, datumom i iznosom transakcije. Potrebno je popuniti sva polja, te realizirati ograničenje koje provjerava tip transakcije - ako je transakcija rashoda, tada pošiljatelj mora biti trenutni korisnik kako bi se iznos transakcije oduzeo od trenutnog korisnika. Ako je transakcija prihoda, tada primatelj transakcije mora biti trenutni korisnik. Tako će aplikacija voditi računa o trenutnom iznosu financija za svaki korisnički račun.
- **Kreiranje mjesečnih izvještaja** (eng. *Monthly Report*) - odnosi se na implementaciju pogleda na kojem je moguće odabrati određeni mjesec i pritiskom na gumb kreirat će izvještaj koji će se prikazati u obliku tablice na ekranu korisnika za odabrani mjesec. Izvještaj će sadržavati sve transakcije koje su izvršene taj mjesec, s prikazanim svim popratnim informacijama vezanih uz transakcije - pošiljatelj, primatelj, detalji transakcije, iznos, datum i tip transakcije.
- **Kreiranje godišnjih izvještaja** (eng. *Yearly Report*) - identična funkcionalnost kao i kod kreiranja mjesečnih izvještaja, ali omogućuje odabir godine za koju se želi kreirati izvještaj. Aplikacija u tom trenutku pretražuje sve transakcije koje su se izvršile tu godinu.
- **Prikaz svih transakcija** (eng. *All Transactions*) - odnosi se na implementaciju pogleda koji sadrži tablicu svih izvršenih transakcija za trenutnog korisnika aplikacije. Tablica će sadržavati sve popratne informacije vezanih uz transakcije - pošiljatelj, primatelj, detalji transakcije, iznos, datum i tip transakcije.
- **Uvoz i izvoz transakcija u PDF i CSV obliku** (eng. *Transaction Import and Export*) - odnosi se na implementaciju gumba iznad tablice svih transakcija koji omogućuje izvoz u PDF i CSV obliku, te uvoz transakcija u CSV obliku. Uvoz transakcija mora najprije validirati točnost učitanih podataka, te ih tada spremiti u bazu podataka.

6.3. Zahtjevi za korisnički račun

6.3.1. Prijava i registracija

Kako nam je za većinu aplikacije, pogotovo za zahtjeve funkcionalnosti aplikacije, potrebna autentikacija korisnika, smisleno je krenuti s implementacijom prijave i registracije korisnika. Spring Boot inicijalno omogućava prijavu u aplikaciji pomoću unutarnjih paketa, bez da se zadužuje razvojnog inženjera da razmišlja o tome. Ali, to zadovoljava potrebe tako dugo dok aplikacija ne traži autentikaciju korisnika pomoću vlastite baze podataka. Tada je potrebno učiniti nekoliko stvari. Mi ćemo krenuti od nižih slojeva, te razviti svu potrebnu logiku i zatim kreirati poglede i kontrolere koji će koristiti implementiranu logiku.

Krenimo s razvojem našeg prvog mikroservisa user-service. Unutar njega kreirat ćemo novi servis pod imenom *UserService*. Prisjetimo se da je dobra praksa kreirati srodne klase u

svoje pakete - prije kreiranja mikroservisa, kreirat ćemo paket *service* i unutar njega novu klasu sučelje (eng. *Interface*) s dogovorenim nazivom. *UserService* sučelje će sadržavati potpise metoda koje će mikroservis implementirati, s navedenim tipom podatka i parametrima. Viši slojevi aplikacije, poput kontrolera, nikad neće direktno injektirati ovisnost servisa, već sučelja pomoću kojeg će pozivati potrebne operacije. *UserService* sučelje sadrži sljedeće operacije:

```
public interface UserService
{
    Optional findByUsername(String username);

    Optional findByEmail(String email);

    UserEntity createUser(UserModel userModel);
}
```

Metoda *findByUsername(String username)* će vraćati *UserEntity* objekt, te će se za sada koristiti za provjeru korisničkog imena u bazi podataka kako bismo znali da li je uneseno korisničko ime kod registracije zauzeto ili slobodno. Metoda *findByEmail(String email)* također vraća *UserEntity* objekt ako se u bazi podataka pronađe korisnik s proslijeđenom email adresom. Metoda *createUser(UserModel userModel)* stvara novi UserEntity objekt te ga sprema u bazu podataka nakon uspješne registracije korisnika.

Sada je potrebno kreirati novi paket unutar *service* paketa u kojem ćemo kreirati klasu implementacije sučelja *UserService*. Kreirajmo stoga novi paket pod nazivom *impl* i unutar njega novu klasu *UserServiceImpl* koju ćemo anotirati s anotacijom *@Service* kojom jednostavno označavamo da se klasa nalazi na servisnom sloju. Potrebno je također navesti da *UserServiceImpl* implementira *UserService* sučelje, te i *UserDetailsService* Spring Boot sučelje koje nam je potrebno za implementaciju prijave u web aplikaciji. Kompletna implementacija *UserServiceImpl* servisa sada izgleda ovako:

```
@Service
public class UserServiceImpl
        implements UserService, UserDetailsService
{
    @Override
    public Optional findByUsername(final String username)
    {
        return userRepository.findByUsername(username);
    }

    @Override
    public Optional findByEmail(final String email)
    {
        return userRepository.findByEmail(email);
    }

    @Override
    public UserEntity createUser(final UserModel userModel)
    {
        final UserEntity newUser = new UserEntity();
        newUser.setActive(true);
```

```
        newUser.setEmail(userModel.getEmail());
        newUser.setFirstName(userModel.getFirstName());
        newUser.setLastName(userModel.getLastName());
        newUser.setPassword(passwordEncoder.encode(userModel.getPassword()));
        newUser.setUsername(userModel.getUsername());
        final Optional roleEntity = roleRepository.findByRole(ROLE_USER);
        roleEntity.ifPresent(newUser::setRole);
        return userRepository.save(newUser);
    }

    @Override
    public UserDetails loadUserByUsername(final String username)
    {
        final Optional optionalUserEntity = userRepository.findByUsername(username);
        if (!optionalUserEntity.isPresent())
        {
            throw new UsernameNotFoundException(USERNAME_NOT_FOUND_EXCEPTION_MESSAGE
                );
        }
        return optionalUserEntity.get();
    }
}
```

U svim metodama možemo primijetiti da koristimo *UserRepository* i *RoleRepository* - ovakva sučelja nasljeđuju *JpaRepository* sučelje koje nam olakšava pristup do naše baze podataka. Ovakva sučelja se anotiraju s anotacijom *@Repository* da specificiramo da se sučelje nalazi na perzistentnom sloju i nije ih potrebno implementirati, već samo pozivati. Spring Boot programski okvir automatski može povezati stupce s traženim podatkom, samo je ključno započeti naziv metode s *findBy* i dodati naziv stupca, poput *Email* ili *Username*. Hibernate ovakvu metodu pretvara u SQL upit primjerice:

```
SELECT * FROM user WHERE user.email = email;
```

Naše sučelje za pristup tablici korisnika u bazi podataka će izgledati ovako:

```
@Repository
public interface UserRepository extends JpaRepository
{
    Optional findByUsername(String username);

    Optional findByEmail(String email);
}
```

Sada kada smo implementirali osnovni dio korisničkog mikroservisa i sučelja za pristup bazi podataka, sljedeće što je potrebno jest kreirati konfiguracijsku klasu sigurnosti koja Spring Boot aplikaciji govori da koristimo vlastitu implementaciju prijave. Takvu klasu ćemo nazvati *SecurityConfiguration* i smjestiti ju u paket *security*. Klasu ćemo anotirati s anotacijom *@EnableWebSecurity* i *@Configuration* koje označavaju konfiguracijsku klasu sigurnosti. Klasa također mora naslijediti Spring Boot apstraktnu klasu *WebSecurityConfigurerAdapter* i moramo nadjačati metode *configure(AuthenticationManagerBuilder auth)* i *configure(HttpSecurity http)* pomoću anotacije *@Override*.

Prva metoda prihvaća parametar *AuthenticationManagerBuilder* i pomoću tog objekta specificiramo implementaciju *UserDetailsService* sučelja i zrna *PasswordEncoder*. *UserDetailsService* smo već implementirali u našem *UserService* servisu, a *PasswordManager* možemo definirati u klasi konfiguracije pomoću *@Bean* anotacije koja označava da se radi o zrnu i omogućuje injektiranje ovisnosti.

Druga metoda prihvaća parametar *HttpSecurity* i pomoću tog objekta specificiramo prava pristupa do određenih pogleda, stranica prijave, odjave, postavke oko CORS (eng. *Cross-Origin Resource Sharing*), postavke oko CSRF (eng. *Cross-Site Request Forgery*) i dr. Mi ćemo pomoću tog objekta specificirati putanju do našeg vlastitog pogleda koji sadrži prijavu, te putanju na koju će se usmjeriti korisnik kada se odjavi iz aplikacije, te isključiti podršku CORS-a i CSRF-a. *SecurityConfiguration* će potom izgledati ovako:

```
@EnableWebSecurity
@Configuration
public class SecurityConfiguration extends WebSecurityConfigurerAdapter
{
    @Override
    protected void configure(final AuthenticationManagerBuilder auth) throws
        Exception
    {
        auth.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder
            ());
    }

    @Override
    protected void configure(final HttpSecurity http) throws Exception
    {
        http.cors().and().csrf().disable();
        http//
            .authorizeRequests()//
            .antMatchers(URL_USER_SECURITY)//
            .authenticated()//
            .anyRequest()//
            .permitAll()//
            .and()//
            .formLogin()//
            .loginPage(URL_LOGIN)//
            .defaultSuccessUrl(URL_HOME)//
            .permitAll()//
            .and()//
            .logout()//
            .logoutUrl(URL_LOGOUT)//
            .logoutSuccessUrl(URL_LOGOUT_SUCCESS);
    }

    @Bean
    public PasswordEncoder passwordEncoder()
    {
        return new BCryptPasswordEncoder();
    }
}
```

Možemo primijetiti da sve stranice koje sadrže u poveznici */user/*** traže autenticirani pristup, tj. anonimni korisnici ne mogu pristupiti tom sadržaju. Sve ostale stranice poput prijave, registracije i promjene lozinke ne traže autenticirani pristup.

Sada je samo još potrebno kreirati poglede i kontrolere. Počnimo s pogledom registracije. Kreirajmo HTML datoteku *registration.html* u *resources/templates* direktoriju. Ovaj pogled će sadržavati formu i polja za unos imena, prezimena, email adrese, korisničkog imena, lozinke i ponovljene lozinke. Sva polja su obavezna, pa je potrebno postaviti atribut *required="required"*. Duljinu znakova možemo odrediti s *minlength* i *maxlength*. Ova forma, nakon što korisnik pritisne gumb potvrde registracije, će pokušati preslikati navedena polja na klasu s navedenim atributima. Pa je stoga potrebno kreirati model u paketu *model* i nazovimo ga *UserModel*.

Navedeni model, i općenito sve klase, također moraju sadržavati metode za dohvaćanje i spremanje (eng. *Getter and Setter*) vrijednosti u ove varijable, ali ih nećemo prikazivati zbog preglednosti. Na pogledu se pomoću *Thymeleaf* sustava predložaka preslikava pojedino polje na atribut klase - primjerice polju za unos korisničkog imena je potrebno dodati atribut *th:field="*{username}"*. Još je potrebno sada specificirati na razini forme koji objekt ona preslikava, i to se radi pomoću atributa *th:object="${userModel}"*. Pomoću atributa *th:action="@{/registration}"* specificiramo na koju putanju se šalju podaci, POST metodom.

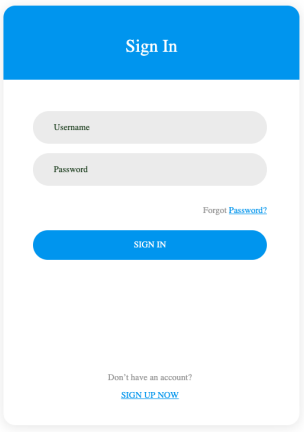
Sada je potrebno kreirati kontrolera koji će moći primiti te podatke koji se šalju s pogleda. Kreirajmo paket *controller* i novu klasu kontrolera *RegistrationController* koju ćemo anotirati s *@Controller*. Sada je potrebno implementirati dvije metode - jedna koja će upravljati GET zahtjevima, i druga koja će upravljati POST zahtjevima. Metoda koja upravlja GET zahtjevima je samo zadužena za specificiranje kojeg pogleda je potrebno prikazati, i s obzirom na to da se radi o formi registracije koja traži *UserModel* objekt za preslikavanje polja forme, moramo joj proslijediti taj objekt. Metoda koja upravlja POST zahtjevima ima malo veće odgovornosti: ona prima popunjeni *UserModel* objekt s podacima s forme, te ih validira. Kako smo spomenuli, potrebno je validirati email adresu, korisničko ime i poklapanje lozinka. U Spring Boot programskom okviru validaciju provodimo novom klasom koja je zadužena samo za provjeru podataka. Stoga, kreirajmo novi paket *validation* i kreirajmo novu klasu *UserFieldsValidator*, anotirajmo ju anotacijom *@Component* i kažemo da implementira *Validator* Spring Boot sučelje. U ovom trenutku potrebno je provjeriti da li su email adresa i korisničko ime zauzeti, stoga moramo pozvati mikroservis *UserService* koji dohvaća korisnike koji imaju navedene email adrese ili korisnička imena. Ali, prije nego pozovemo mikroservis, moramo implementirati Facade uzorak dizajna zato što smo se složili da je loša praksa pozivati niže slojeve aplikacije iz najviših. Stoga, kreirajmo paket *facade* i u njemu kreiramo sučelje *UserFacade*. Također kreiramo *impl* paket i u njemu kreiramo *UserFacadeImpl* koja implementira *UserFacade* sučelje. Anotiramo ju s *@Component* anotacijom i realiziramo metode koje pozivaju mikroservisne operacije.

U tom trenutku možemo injektirati ovisnost *UserFacade* u *UserFieldsValidator* klasu, te pozvati operacije *findByUsername* i *findByEmail*. Ako operacije vrate neprazan objekt, to će značiti da je email adresa ili korisničko ime zauzeto u bazi podataka i potrebno je vratiti poruku o pogrešci korisniku.

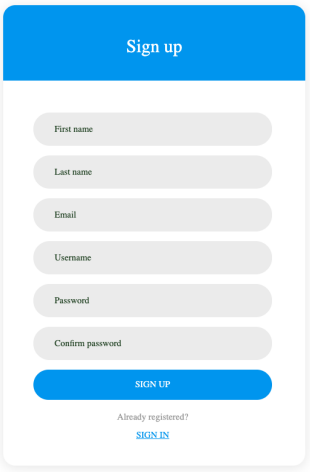
```
@Component
public class UserFieldsValidator implements Validator
{
    @Override
    public boolean supports(final Class aClass)
    {
        return UserModel.class.equals(aClass);
    }

    @Override
    public void validate(final Object o, final Errors errors)
    {
        final UserModel userModel = (UserModel) o;
        if (!ObjectUtils.isEmpty(userFacade.findByEmail(userModel.getEmail())))
        {
            errors.rejectValue(EXISTING_EMAIL_FIELD, EXISTING_EMAIL_CODE);
        }
        if (!ObjectUtils.isEmpty(userFacade.findByUsername(userModel.getUsername()))
            )
        {
            errors.rejectValue(EXISTING_USERNAME_FIELD, EXISTING_USERNAME_CODE);
        }
        if (!StringUtils.equals(userModel.getPassword(), userModel.
            getMatchingPassword()))
        {
            errors.rejectValue(MATCHING_PASSWORD_FIELD, MATCHING_PASSWORD_CODE);
        }
    }
}
```

Registracija je ovime implementirana. Na sličan način je za prijavu potrebno kreirati pogled *login.html* u *templates* direktoriju. Pogled prijave samo sadrži dva polja za unos korisničkog imena i lozinke. Forma se šalje POST metodom na poveznicu */login*, za koju je potrebno kreirati kontroler *LoginController*. Kontroler prijave je vrlo jednostavan jer ne mora voditi računa o objektima na pogledu, već samo upravlja GET zahtjevima koji usmjeruju na HTML stranicu. POST metodom prijave upravlja Spring Boot pomoću konfiguracije sigurnosti koju smo implementirali ranije. Stoga, *LoginController* je u ovom obliku potpuno dovoljan.

I ovime završava i prijava korisnika. Kada se korisnik uspješno prijavi, tada ga se preusmjerava na početnu stranicu aplikacije koju vide samo autenticirani korisnici. Kada bi anoniman korisnik pokušao pristupiti početnoj stranici, tada bi ga sigurnost aplikacije preusmjerila natrag na pogled prijave. Za taj dio je zadužena *SecurityConfiguration* klasa, opisana ranije. Dizajn aplikacije se može izraditi tako da se CSS datoteka spremi u *resources/static/css* direktorij. Općenito, sav statičan sadržaj se sprema u *static* direktorij: JavaScript kod, fontovi, biblioteke i dr. Spring Boot automatski učitava datoteke u *static* direktoriju, te na pogledu nije potrebno specificirati punu putanju, već putanju unutar *static* direktorija. Na sljedećim slikama se može vidjeti pogled prijave (slika 11) i registracije (slika 12):



Slika 11: Pogled prijave



Slika 12: Pogled registracije

6.3.2. Promjena lozinke

Kako smo već pojasnili, promjena lozinke će se odvijati tako da korisnik najprije upiše svoju email adresu u predviđeno polje. U tom trenutku će aplikacija provjeriti da li upisana email adresa postoji. Ako ne postoji, vraća se poruka o pogrešci, inače se kreira novi UUID (eng. *Universal Unique Identifier*) žeton, sprema se u bazu podataka za korisnika koji ima upisanu email adresu, te se taj UUID žeton šalje na upisanu email adresu u obliku poveznice na koju korisnik može kliknuti. Kada korisnik klikne na poveznicu, aplikacija provjerava validnost žetona tako da pretraži sve korisnike da li ikoji korisnik ima pridružen taj žeton. Ako je žeton validan, korisnik može upisati novu lozinku, inače se ispisuje pogreška.

Za ovu korisničku funkcionalnost, morat ćemo kreirati metode u email mikroservisu koje mogu raditi s email porukama. Stoga, u email-service mikroservisu najprije je potrebno dodati biblioteku za rad s email porukama u *pom.xml* datoteku:

```
    org.springframework.boot
    spring-boot-starter-mail
```

Nakon toga možemo započeti s implementacijom mikroservisa. Kreirajmo sučelje *EmailService* s metodom *sendForgottenPasswordEmail(final UserEntity userEntity)*. Kreirajmo zatim *EmailServiceImpl* servis koji će implementirati *EmailService* sučelje:

```
@Service
public class EmailServiceImpl implements EmailService
{
    @Async
    @Override
    public boolean sendForgottenPasswordEmail(final UserEntity userEntity)
    {
        ServiceInstance userServiceInstance = discoveryClient.getInstances(
            USER_SERVICE).get(0);
        final SimpleMailMessage passwordResetEmail = new SimpleMailMessage();
        passwordResetEmail.setFrom(fromEmail);
```

```
        passwordResetEmail.setTo(userEntity.getEmail());
        passwordResetEmail.setSubject(emailPasswordResetSubject);
        passwordResetEmail.setText(emailTextPasswordReset + userServiceInstance.
            getUri()
                + URL_RESET_PASSWORD + QUESTION_MARK + TOKEN + EQUALS + userEntity.
                    getToken());
        try
        {
            javaMailSender.send(passwordResetEmail);
            return true;
        }
        catch (MailException ex)
        {
            return false;
        }
    }
```

Obratimo pozornost na objekt klase *DiscoveryClient*. Ova instanca će nam koristiti za otkrivanje korisničkog mikroservisa i njegove lokacije pomoću našeg discovery-service Spring Cloud Eureka poslužitelja, kako bismo mogli generirati poveznicu koja sadrži pogled koji je implementiran na korisničkom mikroservisu.

Za slanje email poruka poslužit ćemo se besplatnim Gmail SMTP (eng. *Simple Mail Transfer Protocol*) protokolom. Svojstva Gmail SMTP protokola ćemo navesti u *application.properties* datoteci, ispod ostalih svojstava:

```
# ================================
# EMAIL
# ================================
spring.mail.host=smtp.gmail.com
spring.mail.port=587
spring.mail.username=myfinance.application@gmail.com
spring.mail.password=MyFinance0321
spring.mail.properties.from=myfinance-support@ecx.com
spring.mail.properties.subject.password.reset=Password Reset Request
spring.mail.properties.text.password.reset=To reset your password, click the link:
spring.mail.properties.mail.smtp.starttls.enable=true
spring.mail.properties.mail.smtp.starttls.required=true
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.connectiontimeout=5000
spring.mail.properties.mail.smtp.timeout=5000
spring.mail.properties.mail.smtp.writetimeout=5000
```

U ovim svojstvima potrebno je specificirati poslužitelj, mrežna vrata, korisničko ime i lozinku. Mi smo ujedno i specificirali svojstva email poruka poput naslova email poruke i teksta email poruke. Kada smo završili s konfiguracijom slanja email poruka, kreirajmo *EmailFacade* sučelje i *EmailFacadeImpl* klasu koja poziva operacije *EmailService* mikroservisa, na način na koji smo kreirali i *UserFacade* Facade.

Kada smo završili s implementacijom email mikroservisa, potrebno je nadopuniti korisnički mikroservis s nekoliko novih radnji. Prije nego se email poruka pošalje, potrebno je kreirati UUID žeton i spremiti ga u bazu podataka. Stoga moramo implementirati metodu *cre-*

ateToken(UserEntity userEntity):

```
@Override
public void createToken(final UserEntity userEntity)
{
    userEntity.setToken(UUID.randomUUID().toString());
    userRepository.save(userEntity);
}
```

Kada smo implementirati metodu kreiranja i spremanja žetona, potrebno je implementirati još metode koje mogu pronaći korisnika po njegovom žetonu, te ga na posljetku prebrisati kada se postavi nova lozinka. Za pronalaženje korisnika po njegovom žetonu koristit ćemo perzistentno sučelje *UserRepository*. Stoga, dodajmo novu metodu za pretraživanje korisnika po žetonu u *UserRepository* sučelje:

```
Optional findByToken(String token);
```

I sada novokreiranu metodu možemo pozvati u korisničkom servisu:

```
@Override
public Optional findByResetToken(final String token)
{
    return userRepository.findByResetToken(token);
}
```

Te i metoda za postavljanje nove lozinke i brisanje žetona s korisničkog računa:

```
@Override
public void resetUserPassword(final UserEntity userEntity, final String newPassword)
{
    userEntity.setPassword(passwordEncoder.encode(newPassword));
    userEntity.setToken(null);
    userRepository.save(userEntity);
}
```

U ovom trenutku preostalo je kreirati dva pogleda i kontroler. Stoga, kreirajmo *forgotten-password.html* pogled koji sadrži jedno polje za unos email adrese, koje ćemo dohvaćati u kontroleru kao parametra. Kreirajmo sada i pogled *reset-password.html*. Ovaj pogled sadrži jedno vidljivo polje za unos nove lozinke, i jedno nevidljivo polje koje služi za spremanje vrijednosti žetona. Kada korisnik pokuša postaviti novu lozinku, kontroleru se šalje i žeton i lozinka.

Kada su pogledi kreirani, preostalo je još samo kreirati kontroler. Nazovimo ga *PasswordController*. Potrebno je implementirati metode koje će upravljati GET zahtjevima kada korisnik otvori *forgotten-password.html* i *reset-password.html* poglede, te metode koje će upravljati POST metodama kada korisnik pošalje popunjena polja s pogleda. Na kontroleru bi istaknuli metodu koja poziva email-service mikroservis kojem se šalje UserEntity objekt u HttpEntity zaglavlju.

```
@Controller
public class PasswordController
{
    @RequestMapping(value = VIEW_FORGOTTEN_PASSWORD, method = RequestMethod.POST)
    public String postViewForgottenPassword(
```

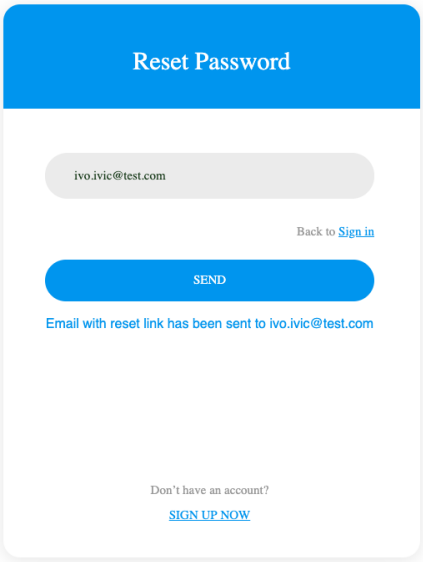
```
            @RequestParam(PARAMETER_EMAIL)
            final
            String email,
            final Model model,
            final HttpServletRequest request)
    {
        final Optional optionalUserEntity = userFacade.findByEmail(email);

        if (optionalUserEntity.isPresent())
        {
            userFacade.createToken(optionalUserEntity.get());
            HttpHeaders headers = new HttpHeaders();
            headers.setContentType(MediaType.APPLICATION_JSON);
            HttpEntity entity = new HttpEntity(optionalUserEntity.get(), headers);
            ResponseEntity result = restTemplate.exchange(
                    EMAIL_SERVICE_URL,
                    HttpMethod.POST,
                    entity,
                    Boolean.class
            );
            if (!ObjectUtils.isEmpty(result.getBody()) && result.getBody())
            {
                model.addAttribute(
                        MODEL_ATTRIBUTE_SUCCESS,
                        MODEL_ATTRIBUTE_SUCCESS_MESSAGE + email
                );
            }
            else
            {
                model.addAttribute(MODEL_ATTRIBUTE_ERROR,
                    MODEL_ATTRIBUTE_ERROR_MESSAGE);
            }
        }
        else
        {
            model.addAttribute(MODEL_ATTRIBUTE_ERROR, MODEL_ATTRIBUTE_EMAIL_MESSAGE)
                ;
        }
        return VIEW_FORGOTTEN_PASSWORD;
    }
}
```

U trenutku kada korisnik upiše email adresu, aplikacija validira da li upisana email adresa postoji, i ako postoji radi se zahtjev prema email mikroservisu s parametrom korisničkog entiteta u JSON obliku, koji odrađuje slanje email poruke. Ako ne postoji, ispisuje se poruka o pogrešci. Email mikroservis zahtjev prima na svojem REST kontroleru koji na zadanoj putanji prima parametar korisničkog entiteta:

```
@RestController
public class EmailController
{
    @RequestMapping(value = MAPPING_SEND_FORGOTTEN_PASSWORD_EMAIL)
    public Boolean sendForgottenPasswordEmail(
```

```
            @RequestBody
                    UserEntity userEntity)
    {
        return emailFacade.sendForgottenPasswordEmail(userEntity);
    }
}
```

Kada korisniku stigne email poruka u email poštanski sandučič, poveznica na koju može kliknuti izgleda primjerice ovako: *http://192.168.22.110:8081/reset-password?token=5708972c-c643-4e12-8c7f-1ac79cbeb28f*. Parametar koji se šalje uz poveznicu je *token* i njegov pridružen UUID. Kada korisnik klikne na poveznicu, preusmjerava ga na *reset-password.html* pogled i u tom trenutku aplikacija validira URL parametar - žeton. Ako je žeton prisutan u bazi podataka, smatra se validnim i korisnik može upisati novu lozinku. Kada se upiše nova lozinka, upisana lozinka i žeton se zajedno šalju kontroleru koji poziva *findByToken(token)* pomoću *userFacade* objekta. Ako se korisnik s tim žetonom pronađe, njegov žeton se u bazi podataka prebriše, postavlja se nova lozinka i korisnika se preusmjerava na pogled prijave s porukom o uspješnoj radnji. Ako žeton nije pronađen u bazi podataka, tj. mikroservis vrati prazan objekt, kontroler vraća poruku na pogled da je žeton nevažeći.



Slika 13: Pogled unosa email adrese nepristupačnog korisničkog računa

Na slici 13 možemo vidjeti realizirani pogled za upis email adrese za čiji korisnički račun se želi promijeniti lozinka. Ako se korisnik prisjeti lozinke, može napustiti pogled promjene lozinke pomoću poveznice do pogleda za prijavu, te do pogleda za registraciju ako želi stvoriti novi račun.

6.3.3. Aktivacija korisničkog računa

Do sada, svaki registrirani korisnički račun je bio odmah i aktivan. Ova web aplikacija mora implementirati mehanizam aktiviranja korisničkog računa putem email adrese, pomoću

žetona kao i kod promjene lozinke. Temelje slanja email poruka smo postavili u prethodnoj implementaciji u email mikroservisu, te je sada samo potrebno kreirati novu metodu za slanje email poruke s poveznicom koja aktivira korisnički račun, čim se korisnik registrira. Tako dugo dok korisnik ne aktivira korisnički račun ne može se autenticirati u web aplikaciji. Stoga, u email mikroservisu u klasi *EmailService* kreirajmo novu metodu:

```
@Override
public boolean sendActivationEmail(final UserEntity userEntity)
{
    ServiceInstance userServiceInstance = discoveryClient.getInstances(USER_SERVICE)
        .get(0);
    final SimpleMailMessage activationEmail = new SimpleMailMessage();
    activationEmail.setFrom(fromEmail);
    activationEmail.setTo(userEntity.getEmail());
    activationEmail.setSubject(emailActivationSubject);
    activationEmail.setText(emailActivationText + userServiceInstance.getUri()
            + URL_VERIFY_ACCOUNT + QUESTION_MARK + TOKEN + EQUALS + userEntity.
                getToken());
    try
    {
        javaMailSender.send(activationEmail);
        return true;
    }
    catch (MailException ex)
    {
        return false;
    }
}
```

Kada se korisnik uspješno registrira, korisnički mikroservis će pozvati REST metodu od email kontrolera koji će pozvati operaciju servisa za slanje aktivacijske email poruke. Stoga, potrebno je nadopuniti korisnički mikroservis *UserService* s metodom koja aktivira korisnički račun u bazi podataka kada korisnik klikne na poveznicu koja je stigla na njegovu email adresu:

```
@Override
public void activateUser(final UserEntity userEntity)
{
    userEntity.setActive(true);
    userEntity.setToken(null);
    userRepository.save(userEntity);
}
```

Za ovu implementaciju nije potrebno kreirati novi pogled. Korisnik će dobiti nakon uspješne registracije da je na njegovu email adresu poslana poveznica koju mora otvoriti kako bi aktivirao korisnički račun. Nakon što klikne, pozivat će se *RegistrationController* koji sluša događaj aktivacije sa žetonom kao parametar u putanji, primjerice: *http://192.168.22.110:8081/verify-account?token=37916136-f4fc-4c5c-b21d-f853c2112a31*. Metoda koja uhvati događaj najprije provjerava da li je žeton validan u bazi podataka. Ako jest, aktivira korisnički račun na koji je pridružen žeton i ispisuje poruku uspjeha, inače se ispisuje poruka o nevažećem žetonu.

6.3.4. Pregled profila i ažuriranje podataka

Zadnja korisnička funkcionalnost je pregled profila i mogućnost promjene i ažuriranja korisničkih podataka poput imena, prezimena, email adrese i promjena lozinke. Ovaj pogled mora biti sakriveni od anonimnih korisnika, što znači da samo autenticirani korisnici mogu pristupiti pogledu profila. Što je i logički jasno, jer ovaj pogled će sadržavati formu koja će sadržavati polja koja se automatski popunjavaju korisničkim podacima kada korisnik pristupi pogledu. Za ovu funkcionalnost bit će potrebno kreirati novi pogled i klasu koja će popuniti *UserModel* na temelju podataka koji se dohvate iz baze podataka, implementirati nove metode u korisničkom servisu koje će dohvaćati trenutnog korisnika iz sesije, kreirati kontroler koji će slušati GET i POST zahtjeve, te kreirati klasu validacije koja će validirati polja prije ažuriranja podataka.

Najprije ćemo proširiti naš korisnički mikroservis s metodama dohvaćanja korisnika iz aplikacijske sesije i ažuriranja korisničkog računa s promijenjenim podacima. Metoda *getUserEntity()* će pomoću *SecurityContextHolder* klase dohvatiti kontekst i objekt autentikacije koji sadrži sve podatke korisnika koji je trenutno prijavljen. Ova metoda će vraćati objekt *UserEntity* klase.

```
@Override
public UserEntity getUserEntity()
{
    final Authentication authentication = SecurityContextHolder.getContext().
        getAuthentication();
    return findByUsername(authentication.getName()).get();
}
```

Nakon toga ćemo implementirati metodu *updateUser(UserModel userModel)* koja prima objekt popunjen s korisničkim podacima. Nakon toga potrebno je preslikati podatke s modela na objekt klase *UserEntity* i zatim spremiti podatke u bazu podataka.

Sljedeće što je potrebno je kreirati klasu *UserPopulator* koja će populirati objekt klase *UserModel*, obratnom operacijom kao u operaciji korisničkog servisa *updateUser()*. *UserPopulator* će primiti objekt klase *UserEntity* i preslikati podatke na objekt klase *UserModel*, kojeg ćemo tada slati preko kontrolera na pogled. Pogled će u tom trenutku preslikati podatke objekta na odgovarajuća polja u formi. Klasa *UserPopulator* sadrži samo jednu metodu *populate(UserEntity userEntity)*.

UserPopulator klasu će koristiti *UserFacadeImpl*. Kada korisnik pristupi pogledu profila, poziva se GET metoda kontrolera gdje se tada poziva metoda koja populira objekt *userModel* s podacima iz baze podataka pomoću klase populatora. Stoga implementirajmo metodu *getUserModel()* u klasi *UserFacadeImpl*:

```
@Override
public UserModel getUserModel()
{
    final UserEntity userEntity = getUserEntity();
    return userPopulator.populate(userEntity);
}
```

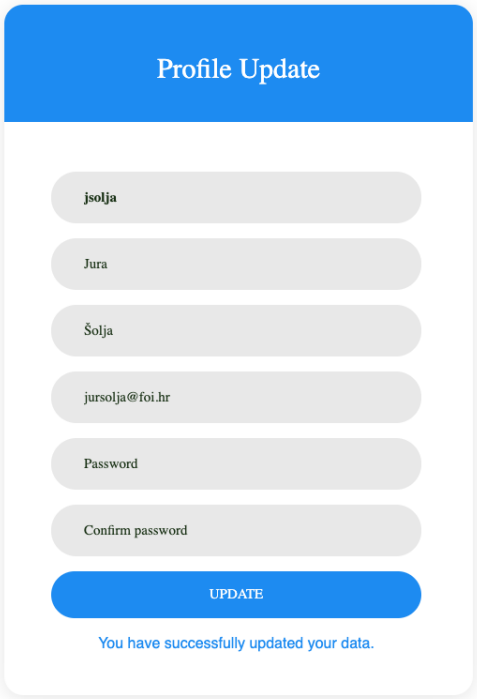
Kreirajmo sada kontroler *ProfileController* koji će implementirati spomenute GET i POST

metode. GET metoda kontrolera će pozivati *userFacade.getUserUserModel()* operaciju kako bi dohvatio popunjeni objekt i predstavio ga pogledu koji će preslikati podatke na polja forme. POST metoda će primiti podatke koje je korisnik poslao, validirat će ih i osvježiti korisnički račun s novim podacima korisnika.

Nakon što su kontroler i pogled implementirani, korisnik može promijeniti podatke, ali pritom mora upisati lozinku. Ako korisnik upiše staru lozinku, ostat će nepromijenjena, inače mora ponoviti lozinku i tada će se postaviti nova lozinka za taj korisnički račun.

Korisniku je također onemogućena promjena korisničkog imena. Kada korisnik promijeni email adresu, aplikacija uspoređuje email adresu s ostalim email adresama u bazi podataka. Ako pronađe par, tada uspoređuje korisnička imena vlasnika tih email adresa. Ako se korisnička imena tih korisnika poklapaju, to znači da korisnik nije mijenjao svoju email adresu, već neke druge podatke. Ako se korisnička imena ne poklapaju, to znači da je korisnik pokušao promijeniti email adresu na zauzetu email adresu i tada se ispisuje pogreška.

Na slici 14 je vidljiva stranica za pregled i ažuriranje korisničkog profila:



Slika 14: Pregled i ažuriranje korisničkog profila

6.4. Zahtjevi za funkcionalnost aplikacije

6.4.1. Unos transakcija prihoda i rashoda

U ovom poglavlju konačno ćemo dotaknuti implementaciju dijela aplikacije koji se tiče transakcija. Kako smo u ranijim poglavljima već dali objašnjenje ove funkcionalnosti, odmah

ćemo krenuti na implementaciju.

Za potrebe ove funkcionalnosti kreirat ćemo novi servis *TransactionService* u transakcijskom mikroservisu koji će obavljati operacije vezane uz transakcije. Za početak, implementirajmo metodu koja će kreirati novu transakciju:

```
@Service
public class TransactionServiceImpl implements TransactionService
{
    @Override
    public TransactionEntity makeTransaction(final TransactionModel transactionModel
        )
    {
        final TransactionEntity newTransaction = new TransactionEntity();
        newTransaction.setAmount(transactionModel.getAmount());
        newTransaction.setDate(transactionModel.getDate());
        newTransaction.setDetails(transactionModel.getDetails());
        newTransaction.setFromUser(transactionModel.getFromUser());
        newTransaction.setToUser(transactionModel.getToUser());
        newTransaction.setType(transactionModel.getType());
        final Optional userEntity = userRepository.findByUsername(userService
                .getUserEntity()
                .getUsername());
        userEntity.ifPresent(newTransaction::setUserEntity);
        updateBalance(newTransaction);
        return transactionRepository.save(newTransaction);
    }

    private void updateBalance(final TransactionEntity transactionEntity)
    {
        final UserEntity userEntity = userRepository
                .findByUsername(userService.getUserEntity().getUsername())
                .get();
        final double currentBalance = userEntity.getBalance();
        if (transactionEntity.getType().equals(INCOME))
        {
            userEntity.setBalance((float) (currentBalance + transactionEntity.
                getAmount()));
        }
        else
        {
            userEntity.setBalance((float) (currentBalance - transactionEntity.
                getAmount()));
        }
        userRepository.save(userEntity);
    }
}
```

Ova metoda će kreirati novi entitetni objekt *newTransaction* na kojeg će se pridružiti podaci iz objekta *transactionModel* koji se šalje iz kontrolera. Nakon toga se ažurira novčano stanje korisničkog računa i vraća se ažuran objekt natrag na kontroler. Kao i za prethodne mikroservise, kreirat ćemo Facade *TransactionFacade*.

Kako bi mogli primati zahtjeve korisnika, potrebno je kreirati i realizirati operacije REST kontrolera - GET metoda za prikaz pogled i POST metoda za primanje zahtjeva kada korisnik kreira novu transakciju. POST metoda će prije pozivanja mikroservisa putem *TransactionFacade* validirati podatke.

1. Iznos transakcije mora biti strogo veći od nule;
2. Tip transakcije mora biti prihod ili rashod;
3. Ako je tip transakcije prihod, primatelj transakcije mora biti trenutno prijavljeni korisnik;
4. Ako je tip transakcije rashod, pošiljatelj transakcije mora biti trenutno prijavljeni korisnik;
5. Ako je tip transakcije rashod, pošiljatelj transakcije mora imati dovoljno sredstva na računu.

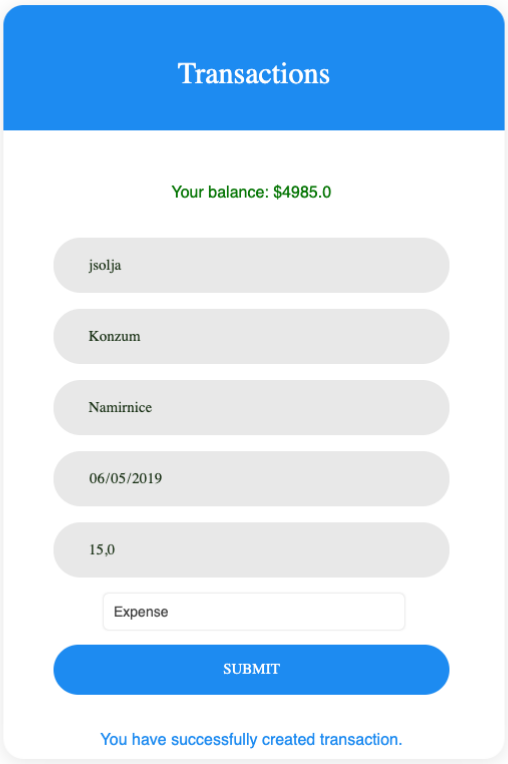
Metoda koja validira navedene podatke implementirana je u *TransactionFieldsValidator* klasi u korisničkom mikroservisu. *TransactionFieldsValidator* nasljeđuje Spring Boot sučelje *Validator*, u metodi *validate(Object o, Errors erros)*.

Nakon što smo implementirali validaciju podataka, još je jedino preostalo realizirati operacije kontrolera koji će objediniti sve što smo implementirali do sada. Kako smo naveli, kontroler sadrži GET i POST metode za primanje korisničkih zahtjeva. Izdvojili bismo POST metodu koja poziva transakcijski mikroservis i prosljeđuje mu TransactionModel objekt putem HttpEntity zaglavlja.

```
@Controller
public class TransactionController
{
    @RequestMapping(value = URL_TRANSACTION, method = RequestMethod.POST)
    public String postViewTransaction(
            @ModelAttribute(MODEL_ATTRIBUTE_TRANSACTION_MODEL)
            @Valid
            final TransactionModel transactionModel, final Model model, final
                BindingResult bindingResult)
    {
        ValidationUtils.invokeValidator(transactionFieldsValidator, transactionModel
            , bindingResult);
        if (!bindingResult.hasErrors())
        {
            transactionModel.setUserEntity(userFacade.getUserEntity());
            HttpHeaders headers = new HttpHeaders();
            headers.setContentType(MediaType.APPLICATION_JSON);
            HttpEntity httpEntity = new HttpEntity(transactionModel, headers);
            ResponseEntity result = restTemplate.exchange(
                    TRANSACTION_SERVICE_URL + MAKE_TRANSACTION,
                    HttpMethod.POST,
                    httpEntity,
                    TransactionEntity.class
            );
            if (ObjectUtils.isEmpty(result.getBody()))
            {
```

```
                model.addAttribute(MODEL_ATTRIBUTE_ERROR,
                    MODEL_ATTRIBUTE_ERROR_MESSAGE);
            }
            model.addAttribute(MODEL_ATTRIBUTE_SUCCESS,
                MODEL_ATTRIBUTE_SUCCESS_MESSAGE);
        }
        model.addAttribute(MODEL_ATTRIBUTE_USER_BALANCE, userFacade.getUserModel().
            getBalance());
        return VIEW_TRANSACTION;
    }
}
```

Sada korisnik može kreirati transakcije - svaka validna transakcija se sprema u bazu podataka kako bi se kasnije mogle prikazati na pogledu transakcija za pojedinog korisnika. Pogled kreiranja transakcija izgleda kao na slici:



Slika 15: Pogled transakcije

6.4.2. Prikaz svih transakcija

Kako bi korisnik mogao pregledati sve transakcije koje je izvršio, potrebno je kreirati pogled koji će sadržavati tablicu sa svim stupcima koji sadržavaju atribute transakcije: pošiljatelj, primatelj, detalji, iznos, tip i datum transakcije. Pogled koji prikazuje sve transakcije će biti i ujedno početna stranica kada se korisnik prijavi (eng. *Homepage*). Za ovo, jedino nam je potrebno kreirati metode za dohvaćanje svih transakcija određenog korisnika iz baze podataka. Stoga, kreirajmo metodu dohvaćanja svih transakcija u *TransactionRepository* sučelju u

transakcijskom mikroservisu:

```
@Repository
public interface TransactionRepository extends JpaRepository
{
    List findByUser(UserEntity userEntity);
}
```

Ova metoda će vraćati listu objekata *TransactionEntity*, a prima objekt *UserEntity* klase. Ovu metodu će pozivati servis *TransactionService*, koju će pak pozivati metoda iz *TransactionFacade* kada korisnik zatraži prikaz svih transakcija putem pogleda i kontrolera. Stoga, potrebno je još samo kreirati kontrolera koji će pozivati operacije dohvaćanja transakcija iz nižih slojeva aplikacije i prikazati ih na pogledu. U REST kontroleru transakcijskog mikroservisa implementirat ćemo metodu koja poziva servis i vraća listu transakcija po korisniku:

```
@RestController
public class TransactionController
{
@RequestMapping(value = FIND_BY_USER)
public TransactionListModel findByUser(
        @RequestBody
                UserEntity userEntity)
{
    TransactionListModel transactionListModel = new TransactionListModel();
    transactionListModel.setTransactionEntityList(transactionFacade.findByUser(
        userEntity));
    return transactionListModel;
}
}
```

Ovu metodu poziva korisnički mikroservis iz *HomeController* kontrolera:

```
@Controller
public class HomeController
{
    @RequestMapping(value = URL_HOME, method = RequestMethod.GET)
    public String home(final Model model)
    {
        ResponseEntity result = findTransactionsByUser();
        model.addAttribute(MODEL_ATTRIBUTE_TRANSACTIONS, result.getBody().
            getTransactionEntityList());
        return VIEW_HOME;
    }
}
```

Tablica transakcija je prikazana korisniku kao na slici 16. Korisnik može sortirati stupce transakcija uzlazno i silazno, odabrati broj redaka koji se prikazuje po stranici (10, 25, 50, 100), može listati stranice i pretraživati bilo koji podatak u tablici.

Slika 16: Pogled početne stranice i transakcija

6.4.3. Kreiranje mjesečnih i godišnjih izvještaja

Slično kao i kod pregleda svih transakcija koje je korisnik izvršio, ova funkcionalnost implementira dodatne mogućnosti odabira specifičnog mjeseca/godinu u kojem intervalu želi ispis izvršenih transakcija. Za ovu funkcionalnost ćemo kreirati dva nova pogleda: jedan za mjesečni izvještaj i jedan za godišnji izvještaj. Oba izvještaja ćemo unaprijediti dodatnom funkcionalnošću izvoza u PDF i CSV obliku u sljedećem poglavlju.

Započnimo s mjesečnim izvještajem. Na pogledu je potrebno kreirati jedno polje unosa datuma - korisniku ćemo omogućiti ugrađeni odabir datuma (eng. *Datepicker*) kako bismo bili sigurni da će biti proslijeđen datum ispravnog formata. Transakcije ćemo dohvaćati iz baze podataka pomoću nove operacije *TransactioRepository* sučelja u transakcijskom mikroservisu:

```
List findByUserAndDateBetween(UserEntity userEntity, Date beginDate, Date endDate);
```

Operacija omogućuje dohvat transakcija u određenom intervalu. Ovakav oblik nam omogućava višestruku iskoristivost i za sljedeće funkcionalnosti poput godišnjeg izvještaja, kako se ne bismo ograničili samo na mjesečni interval. To znači da je potrebno realizirati logiku oko određivanja prvog i zadnjeg dana u mjesecu kojeg korisnik odabere. Ove radnje će biti implementirane u *TransactionService* servisu. Određivanja datuma prvog dana je trivijalno, ali određivanja datuma zadnjeg dana u mjesecu nije zato što se mjeseci međusobno razlikuju po broju dana. Za ove potrebe iskoristit ćemo klasu *LocalDate* iz paketa *java.time*. Kada odredimo interval prvog i zadnjeg datuma odabranog mjeseca, pozivamo operaciju iz *TransactionRepository* koja nam vraća listu transakcija za trenutnog korisnika i odabranog mjeseca. Implementirana operacija mikroservisa izgleda ovako:

```
@Override
public List findByUserAndChosenMonth(final UserEntity userEntity, final String
    beginDate)
{
    final DateTimeFormatter dateFormat = DateTimeFormatter.ofPattern(yyyy_MM_dd);
    final LocalDate localDate = LocalDate.parse(beginDate + FIRST_DAY_OF_MONTH,
        dateFormat);
    final LocalDate endDate = localDate.withDayOfMonth(localDate.lengthOfMonth());
    return transactionRepository.findByUserAndDateBetween(
```

```
            userEntity,
            java.sql.Date.valueOf(beginDate + FIRST_DAY_OF_MONTH),
            java.sql.Date.valueOf(endDate)
    );
}
```

Korisnički mikroservis će sadržavati *ReportController* kontroler koji će igrati ulogu kao i prethodni kontroleri. Primat će GET i POST zahtjeve korisnika, te izvršiti potrebne radnje. *ReportController* će hvatati zahtjeve i za mjesečne i godišnje izvještaje, u zasebnim GET i POST metodama. Kada korisnik odabere mjesec, kontroler poziva transakcijski mikroservis koji poziva *TransactionService* koji poziva *TransactionRepository* i vraća se lista transakcija. Ako je lista transakcija prazna ispisuje se poruka o pogrešci, inače prikazuje tablica transakcija u odabranom intervalu. Metoda kontrolera koja poziva transakcijski servis izgleda ovako:

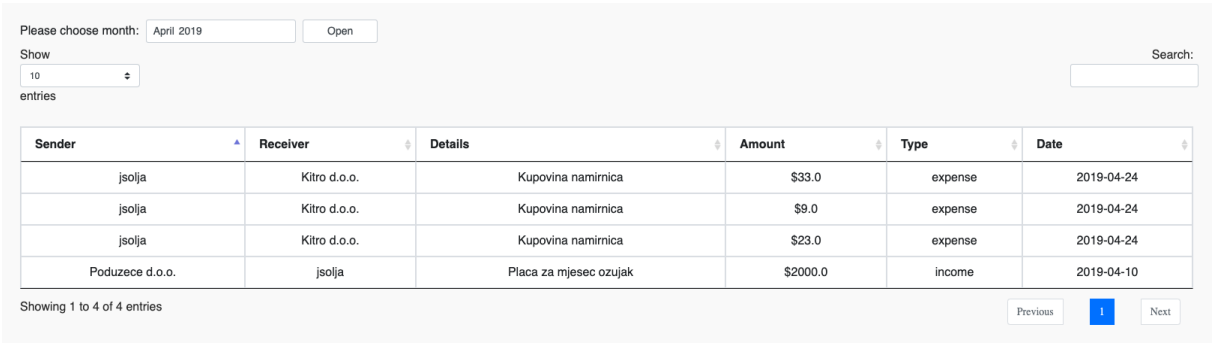
```
@Controller
public class ReportController
{
    @RequestMapping(value = URL_MONTHLY_REPORT, method = RequestMethod.POST)
    public String postViewMonthlyReport(
            @RequestParam(DATE)
            final String date, final Model model)
    {
        UserWithDateModel userWithDateModel = new UserWithDateModel();
        userWithDateModel.setDate(date);
        userWithDateModel.setUserEntity(userFacade.getUserEntity());
        Gson gson = new Gson();
        String request = gson.toJson(userWithDateModel);
        HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_JSON);
        HttpEntity entity = new HttpEntity<>(request, headers);
        ResponseEntity result = restTemplate.exchange(
                TRANSACTION_SERVICE_URL + FIND_BY_USER_AND_CHOSEN_MONTH,
                HttpMethod.POST,
                entity,
                TransactionListModel.class
        );

        ...

        return VIEW_MONTHLY_REPORT;
    }
}
```

Programski kôd koji je implementiran za mjesečni izvještaj je već vrlo dobra podloga za godišnji izvještaj. Potrebno je samo implementirani novu operaciju u *TransactionService* mikroservisu koja prima odabranu godinu kao parametar, te se zatim uzima trivijalni prvi i zadnji datum za tu godinu. Nakon toga se poziva ista operacija za dohvat transakcija u određenom intervalu.

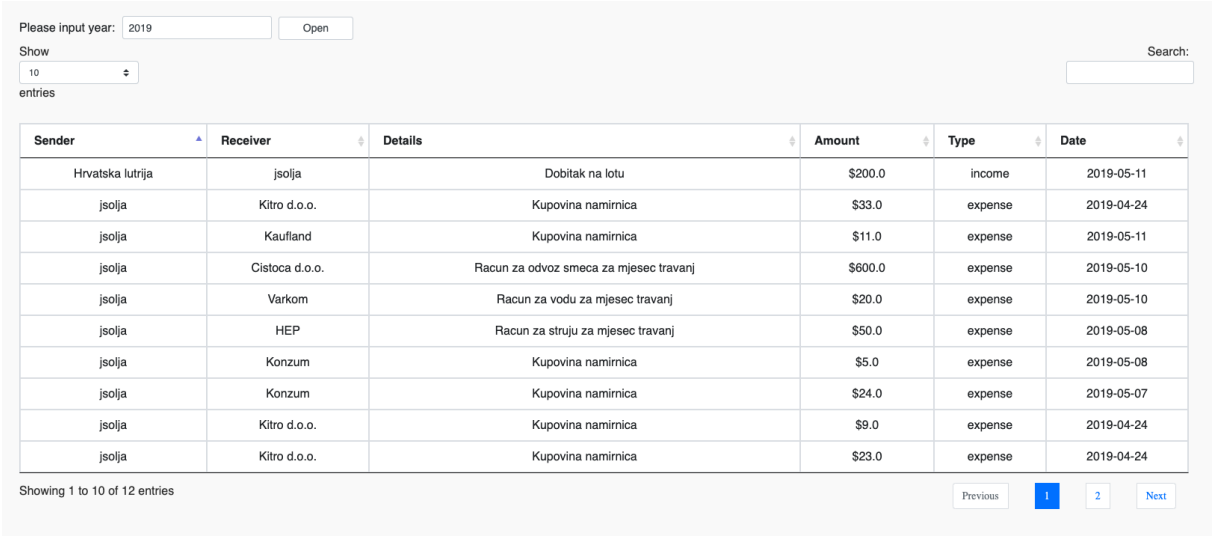
```
@Override
public List findByUserAndChosenYear(final UserEntity userEntity, final String year)
{
```

```
    return transactionRepository.findByUserAndDateBetween(
            userEntity,
            java.sql.Date.valueOf(year + FIRST_DAY_OF_YEAR),
            java.sql.Date.valueOf(year + LAST_DAY_OF_YEAR)
    );
}
```

Te je sada samo potrebno proširiti *ReportController* korisnički kontroler s metodama koje primaju GET i POST zahtjeve za godišnji izvještaj, te identično za transakcijski mikroservis. Kada korisnik upiše godinu, poziva se transakcijski mikroservis koji vraća listu transakcija, kao i kod kreiranja mjesečnih izvještaja. Pogled mjesečnog i godišnjeg izvještaja je realiziran kao na slikama 17 i 18:



Slika 17: Pogled mjesečnog izvještaja



Slika 18: Pogled godišnjeg izvještaja

6.4.4. Uvoz i izvoz transakcija u PDF i CSV obliku

Zadnja funkcionalnost koju ćemo implementirati je generiranje PDF i CSV izvještaja nad svim pogledima na kojima se nalazi tablični prikaz transakcija, te uvoz transakcija u obliku CSV zapisa. Za generiranje PDF i CSV izvještaja koristit ćemo *DataTables* dodatak jQuery JavaScript biblioteke. Biblioteka automatski kreira kontrole u sklopu tablice, te ćemo još dodatno

specificirati CSS (eng. *Cascading Style Sheets*) klasu za primjenu dizajna i mjesto na kojem će se kontrole ugraditi na pogledu. Integracija PDF i CSV kontrola će biti implementirana u *main.js* JavaScript datoteci:

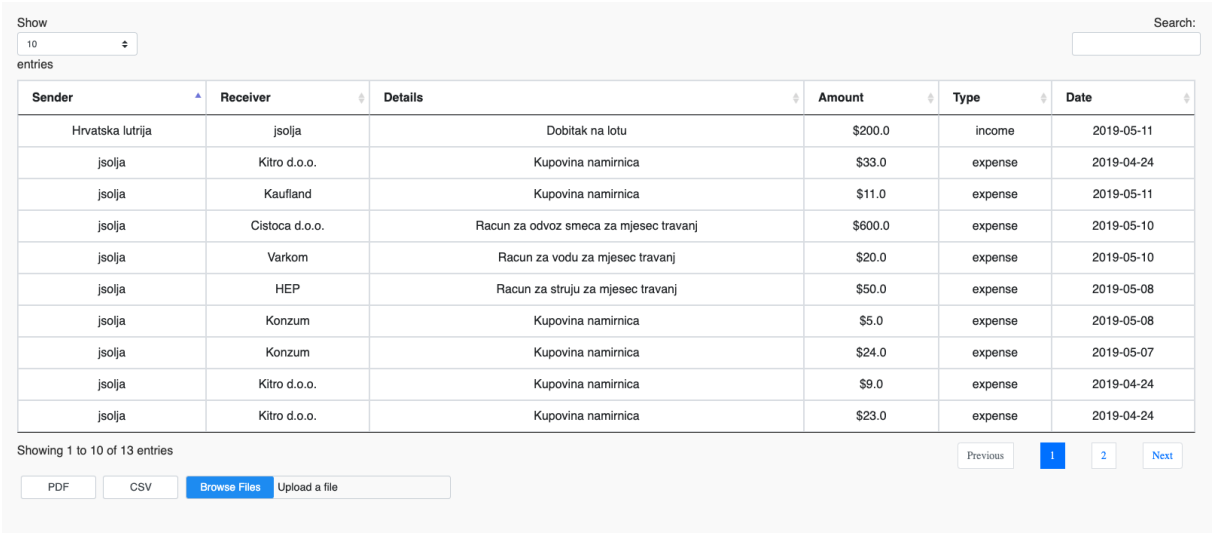
```
var table = $('#transactionTable').DataTable();
var buttons = new $.fn.dataTable.Buttons(table, {
    buttons: [{
        extend: 'pdf',
        className: 'form-control form-control-sm page-custom resize'
    },
    {
        extend: 'csv',
        className: 'form-control form-control-sm page-custom resize'
    }]
}).container().appendTo($('#table-actions'));
```

Uvoz transakcija će biti nešto kompleksniji. Bit će potrebno korisniku prikazati element koji na klik korisnika otvara dijalog koji omogućuje odabir CSV datoteke. Kada korisnik proslijedi naziv datoteke, kontroler će transakcijskom mikroservisu prosljediti naziv datoteke *TransactionService* koji će tu datoteku otvoriti i pokušati pročitati retke. Ako format CSV zapisa nije pravilan, mikroservis će vratiti pogrešku. Za čitanje CSV zapisa, koristit ćemo *OpenCSV* biblioteku koju je prije korištenja potrebno dodati u *pom.xml* konfiguracijsku datoteku. Zatim je potrebno implementirati spomenutu operaciju čitanja CSV datoteka u transakcijskom mikroservisu:

```
@Override
public boolean importCsvTransactions(final String csvFilePath)
{
    final String filePath = new File(csvFilePath).getAbsolutePath();
    try (
        final CSVReader csvReader = new CSVReader(new FileReader(filePath))
    )
    {
        String[] data;
        while ((data = csvReader.readNext()) != null)
        {
            final TransactionEntity newTransaction = new TransactionEntity();
            newTransaction.setFromUser(data[0]);
            newTransaction.setToUser(data[1]);
            newTransaction.setDetails(data[2]);
            newTransaction.setAmount(Integer.parseInt(data[3]));
            newTransaction.setType(data[4]);
            newTransaction.setDate(java.sql.Date.valueOf(data[5]));
            final Optional<UserEntity> optionalUserEntity = userRepository.
                findByUsername(data[6]);
            optionalUserEntity.ifPresent(newTransaction::setUserEntity);
            updateBalance(newTransaction);
            if (!ObjectUtils.isEmpty(transactionRepository.save(newTransaction)))
            {
                return false;
            }
        }
    }
```

```
    catch (final Exception ex)
    {
        Logger.getLogger(TransactionServiceImpl.class.getName()).log(Level.SEVERE,
            null, ex);
    }
    return true;
}
```

Korisniku će biti omogućen uvoz CSV datoteke s transakcijama na početnoj stranici - kada korisnik nema prijašnjih transakcija, aplikacija će ponuditi poveznicu na kojoj može kreirati novu transakciju ili učitati datoteku s transakcijama, a za korisnike koji imaju povijest transakcija oni će moći uvesti transakcije na početnoj stranici ispod tabličnog prikaza prijašnjih transakcija. Na sljedećoj slici je prikazan pogled s početnom stranicom na kojoj su vidljive kontrole za izvoz u PDF i CSV obliku, te kontrola za uvoz transakcija u CSV obliku:



Slika 19: Pogled početne stranice s PDF i CSV uvozom i izvozom transakcija

Ovom funkcionalnošću završavamo s primjenom i implementacijom Spring Boot web aplikacije. U ovom poglavlju prikazali smo osnovne i napredne tehnike razvoja pomoću Spring Boot Java programskog okvira, te smo pokazali razvoj mikroservisne arhitekture u jednoj web aplikaciji. Sav programski kod koji je prikazan u ovom radu će biti dostupan na GitHub online stranici putem poveznice: https://github.com/jsolja/MyFinance.

7. Zaključak

U ovom radu pokrili smo teorijske osnove mikroservisne arhitekture, te opisali i implementirali praktičnu primjenu mikroservisne arhitekture u obliku web aplikacije koja je razvijena pomoću Java Spring Boot programskog okvira. U implementaciji mikroservisne arhitekture pridržavali smo se teorijskih pretpostavki koje smo iznijeli u uvodnim poglavljima ovog rada: prikladna veličina i dizajn mikroservisa, te način korištenja i sučelje mikoservisa. Naveli smo razloge korištenja mikroservisne arhitekture i opisali prednosti koje nudi u tehničkoj, organizacijskoj i poslovnoj domeni. Ali, u skladu s ostalim autorima, došli smo do otkrića da mikroservisna arhitektura nije srebrni metak koja garantira uspjeh. Čak štoviše, neke od prednosti mikroservisne arhitekture mogu dovesti mnoge do zamke iz koje se je teško izvući, poput primjerice tehnološkog pluralizma. Osim općenitih prednosti i izazova, da li smo i jasnu, objektivnu usporedbu mikroservisne arhitekture sa servisno-orijentiranom arhitekturom na razini servisa i opsega projekta koji razvija dotičnu arhitekturu. Nakon detaljnog upoznavanja, opisali smo pristup dizajnu mikroservisne arhitekture koji bi idealno pomogao u implementaciji mikroservise arhitekture i povećao šanse uspješnosti projekta. Takav pristup smo nazvali sustavskim pristupim dizajnu mikroservisa i daje model koji sadrži pet ključnih elemenata: servis, procesi i alati, kultura, organizacija i rješenje. Uz dizajn mikroservisne arhitekture, dotaknuli smo i standarde sigurnosti koje mikroservisi i sustav u kojem djeluju moraju zadovoljavati. Dotaknuli smo OAuth2 i JSON Web Žeton sustave sigurnosti kao jednih od mogućih integracija sigurnosnih provjera u sustavu. Nakon toga upoznali smo se s Java Spring Boot programskim okvirom i dali uvod u drugu, praktičnu cjelinu ovog rada koja se sastoji od opisa web aplikacije razvijene u Java Spring Boot programskim okviru i koja koristi mikroservisnu arhitekturu. Web aplikacija implementira tehničke i korisničke zahtjeve, u kontekstu osobnih financija. Drugim riječima, razvili smo web aplikaciju koja omogućuje korisničke zahtjeve poput prijave, registracije, pregleda i ažuriranja vlastitih podataka, te zahtjeve za funkcionalnostima aplikacije koja omogućuje kreiranje transakcija, prikaza povijesti transakcija, kreiranja izvještaja u obliku tabličnog prikaza, PDF prikaza i CSV prikaza, te uvoz transakcija u CSV obliku. Sve funkcionalnosti su u jezgri implementirane pomoću tri mikroservisa: korisnički mikroservis, email mikroservis i transakcijski mikroservis.

Cilj ovog rada bio je dati teorijsku i praktičnu pozadinu mikroservisnih arhitektura, te se upoznati s Java Spring Boot programskim okvirom, s primjenom znanja stečenih u teorijskom dijelu rada. Mikroservisna arhitektura je još uvijek vrlo aktualna tema i nije moguće jasno predvidjeti nastavak razvoja tehnologija koje će dati značajnije prednosti i posljedično istisnuti mikroservisnu arhitekturu iz popisa modernih arhitektura. Kada se osvrnemo na prolaznost arhitekturalnih stilova u prošlosti, možemo znati da ništa ne traje zauvijek.

Popis literature

[1] R. Zarnekow i W. Brenner, „Distribution of Cost over the Application Lifecycle - a Multi-case Study.”, siječanj 2005, str. 68–79.

[2] D. Namiot i M. Sneps-Sneppe, „On micro-services architecture”, *International Journal of Open Information Technologies*, sv. 2, br. 9, str. 24–27, 2014.

[3] I. Nadareishvili, R. Mitra, M. McLarty i M. Amundsen, *Microservice architecture: aligning principles, practices, and culture*. " O'Reilly Media, Inc.", 2016.

[4] J. Thönes, „Microservices”, *IEEE software*, sv. 32, br. 1, str. 116–116, 2015.

[5] S. Newman, *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.

[6] E. Wolff, *Microservices: Flexible Software Architecture*. Addison-Wesley Professional, 2016.

[7] J. D. Herbsleb i R. E. Grinter, „Architectures, coordination, and distance: Conway's law and beyond”, *IEEE software*, sv. 16, br. 5, str. 63–70, 1999.

[8] C. Pahl i P. Jamshidi, „Microservices: A Systematic Mapping Study.”, *CLOSER (1)*, 2016, str. 137–146.

[9] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston: Addison-Wesley, 2003.

[10] S. Hasite i S. Wojewoda, „Standish Group 2015 Chaos Report - Q&A with Jennifer Lynch”, 2015. adresa: https://www.infoq.com/articles/standish-chaos-2015.

[11] M. Richards, *Microservices vs. service-oriented architecture*. O'Reilly Media, 2015.

[12] T. Erl, *Service-oriented architecture: analysis and design for services and microservices*. Prentice Hall Press, 2016.

[13] A. Arsanjani, „Service-oriented modeling and architecture”, *IBM developer works*, sv. 1, str. 15, 2004.

[14] OASIS, „Reference Model for Service Oriented Architecture”, 2006. adresa: https://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf.

[15] T. B. Dictionary, „Organizational Culture”, 2019. adresa: http://www.businessdictionary.com/definition/organizational-culture.html.

[16] I. E. T. F. IETF, *The OAuth 2.0 Authorization Framework*. 2012. adresa: https://tools.ietf.org/html/rfc6749.

[17] ——, *JSON Web Token (JWT)*. 2015. adresa: https://tools.ietf.org/html/rfc7519.

[18] C. Walls, *Spring Boot in action*. Manning Publications, 2016.

[19] F. Gutierrez, *Pro Spring Boot*. Springer, 2016.

[20] J. Carnell, *Spring Microservices in Action*. Manning Publications Co., 2017.

[21] P. Webb, D. Syer, J. Long, S. Nicoll, R. Winch, A. Wilkinson, M. Overdijk, C. Dupuis i S. Deleuze, „Spring boot reference guide", *Part IV. Spring Boot features*, sv. 24, 2013.

[22] R. RV, *Spring 5.0 Microservices*. Packt Publishing, 2017.

[23] T. A. S. F. ASF, *Introduction to the POM*. 2019. adresa: https://maven.apache.org/guides/introduction/introduction-to-the-pom.html.

[24] D. Wood, M. Loy i R. Eckstein, *Java Swing*. 1998. adresa: https://www.oreilly.com/library/view/java-swing/156592455X/ch01s04.html.

Popis slika

Popis tablica