

* **Završni rad –**

Primena mikroservisne arhitekture u Spring radnom okviru pri izradi eBanking aplikacije

Mentor: Kandidat:

Doc. Dr Nemanja Zdravković Nemanja Petrović 4688

Niš, 2021.

# Sadržaj

# Uvod

Web aplikacije su postale deo našeg svakodnevnog života, sa nekima se susrećemo preko našeg računara dok se sa nekima susrećemo preko našeg mobilnog telefona, što je danas češći slučaj.

Iako aplikacije na mobilnim telefonima nazivamo mobilnim aplikacijama, one su često samo neka vrsta prozora ili interfejsa za interakciju sa većom web aplikacijom koja se izvršava na nekom serveru i dostupna je putem internet pregledača.

Web aplikacije sa velikim brojem korisnika imaju nekoliko dodatnih zadataka:

* Moraju da podrže taj vreliki broj korsnika
* Imaju promene opterećenja u velikom rasponu, opterećenje se može promeniti u svega nekoliko sekundi, na primer ibjava neke udarne vesti
* Konstantno se uvodi novi sadržaj kako bi se privukli novi korisnici

Sve ovo mora se odvijati na finansijski isplativ način koji je isto tako održiv za programere i ostale koji rade na aplikaciji. Posedovati aplikaciju koja može da opsuluži milion korisnika istovremeno a 95% vremena je aktivno svega njih 100 hiljada nije finansijski isplativo, tu se kao rešenje nameće skalabilnost.

Cilj rada je istražiti skalabilnost web aplikacija, njihov razvoj, arhitekturu, koji se problemi javljaju i na šta treba pripaziti kod tih aplikacija. Kao rešenje skalabilnosti kod web aplikacija najviše se spominje mikroservisna arhitektura pa je ona izabrana kao temelj ovog rada.

U ovom radu objasniće se osnovna teorijska podloga web aplikacija, tehnologije koje se koriste za njihov razvoj i najčešće arhitekture aplikacija. Osim teorijske podloge web aplikacija objašnjena će biti i teorija o mikroservisima, mikroservisnoj arhitekturi, koje se problemi javljaju i kako se rešavaju. Biće navedeno i nekoliko primera diyajn paterna karakterističnih za web aplikacije i mikroservisnu arhitekturu.

Osim teorijske podloge rad će sadržati i praktični deo u kom je cilj primeniti teorijska znanja navedena u teorijskom delu rada i u uz pomoć kojih će se kreirati web aplikacija koja će na praktičnom primeru pojednostavljene aplikacije za eBanking pokazati kako se mikroservisna arhitektura koristi u praksi.

Na primeru ove aplikacije biće pokazano kako mikroservisi međusobno komuniciraju, kako klijentska aplikacija komunicira sa serverom koji se sastoji iz više mikroservisa, kako se aplikacija može skalirati po potrebi i kako se može lako nadogratiti i nastaviti razvoj u smeru ozbiljnog projekta.

# **Razvoj mikroservisne arhitekture**

## Istorija mikroservisne arhitekture

Računarska industrija se još uvek nalazi u faxi “pokušaja i greške”, gde eksperti iz ove oblasti stalno uče i pronalaze nova rešenja. Sa druge strane, u računarskoj industriji je takođe čest slučaj gde se smisli novo rešenje pa se vrati na neko prethodno rešenje. Slično je i sa pristupom arhitekture baziranoj na mikroservisima.

Većina koncepta koji se koriste u mikroservisnoj arhitekturi su uvedeni pre 10 ili čak 20 godina ali se tada mislilo da ti koncepti nemaju smisla. Slično je i sa računarstvom u oblaku, čiji su koncepti razvijeni još u doba mainframe računarstva.

Pošto ne postoji zvanični standard za mikroservise, posotji mnoštvo definicija za njih. Pojedinci često pominju servisno-orijentisanu arhitekturu (SOA) kada pokušavaju da objasne šta su mikroservisi.

SOA prethodi mikroservisima, njen osnovni princip je ideja da se organizuju aplikacije u diskretne jednice funkcionalnosti kojima se može daljinski pristupati i koje se nezavisno ažuriraju.

Iako SOA ističe da servisi treba da budu samostalni procesi, nije rečeno koji protokoli treba da se upotrebe za te procese i njihovu međusobnu interakciju i ostaje prilično nejasna u pogledu načina rasporedđivanja i organizovanja aplikacije.

SOA servisi mogu da komuniciraju pomoću IPC-a (Inter-Process Communication) koristeći istu mašinu i deljenjem memorije ili čak putem RPC-a (Remote Procedure Call). Opcije su razne i na kraju SOA može da bude sve i svašta sve dok ne pokrenete ceo kod aplikacije u jednom procesu.

Uobičajno je reći da su mikroservisi specijalizacija SOA ciljeva koji su počeli da se pojavljuju prethodnih godina. Ako želimo da damo kompletnu definiciju mikroservisa, moramo prvo da pogledamo kako je izgrađena većina softvera.

## Monolitna arhitektura

Monolitna arhitektura je klasičan oblik arhitekture, Aplikacija je izgrađena kao jedna celina. Tipična troslojna *Enterprise* aplikacija se najčešće sastoji iz sledećih slojeva

* Klijentski sloj (Web pregledač, HTML + Js)
* Baza podataka
* Serverska aplikacija (Java, .Net, Python....)

Serverska aplikacija obavlja funckije kao što su:

* Obrada HTTP zahteva
* Izvršavanje poslovne logike
* Čitanje i ažuriranje podataka iz baze

U ovom slučaju serverska palikacija je **monolit** - jedinstvena logička izvršna celina.

Kod monolitnih aplikacija sva logika obrade zahteva se izvršava u jednom procesu, razdelljenom i organizovanom u klase, metode i pakete. Aplikaciju je moguće razvijati i testirati čak i na laptopu, nakon čega sledi implementacija u testnom i produkcionom okruženju.

Monolitne aplikacije se horizontalno skaliraju izvršavanjem više instanci aplikacije iza uređaja za balansiranje opterećenja (load balancer).

Promene bilo kog dela aplikacije zahtevaju ponovno prevođenje i postavljanje cele aplikacije u testnom i produkcionom okruženju. Ovo dovodi do toga da su promene skupe i da je potrebno dobro planiranje.

Još jedan deo koji je skup jeste skaliranje jer se ono izvodi tako što se skalira cela aplikacija umesto samo delova koji zahtevaju više resursa.

Sa druge strane prednost ove arhitekture je što je inicijalno lakše razviti proizvod kada se sva logika i sav rad na apliakaciji odvija na jednom mestu.

## Mikroservisna arhitektura

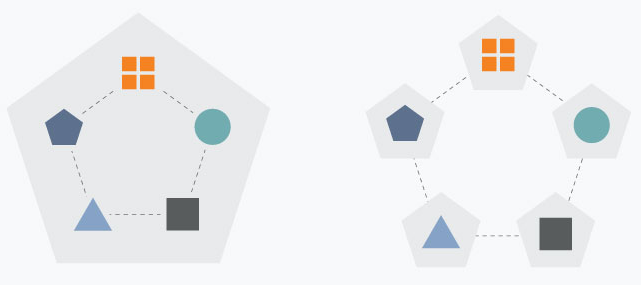
Mikroservisna arhitektura je princip razvoja softvera u obliku malih, izdvojenih servisa, pri čemu svaki servis ima svoj proces i ostvaruje komunikaciju putem jednostavnih mehanizama kao što je HTTP API.

Postoji dva načina kako se kreće sa razvojem mikroservisne arhitekture. Prvi način je da se odmah od početka krene sa razvojem koristeći mikroservise, što je najbolji način ali on može da se primeni samo kod novih aplikacija.

Mnogo češce se aplikacije koje su već napravljene koristeći monolitnu arhitekturu prebacuju na mikroservisnu. To se postiže tako što se delovi aplikacije izdvajaju u mikroservise i monolitna aplikacija s edeo po deo prebacuje na mikroservisnu arhitekturu.

U mikroservisnoj arhitekturu svaka komponenta se izvršava kao zasebna aplikacija. Ovakav način rešava jedan od glavnih problema koje monolit ima a to je skaliranje.

Pošto mikroservisna arhitektura ima više zasebnih komponenata, moguće je skalirati samo one koje imaju veliko pterećenje umesto cele aplikacije, što znantno smanjuje troškove.



*Slika 2.1.1: Monolitna arhitektura i mikroservisna arhitektura*

## Prednosti i mane mikroservisne arhitekture

Mikroservisna arhitektura ima dosta prednosti u odnosu na monolitnu arhitekturu, a samo neke od prednosit su:

* Razvojni timovi mogu nezavisno da razvijaju i da isporučuju delove aplikacije
* Komponente mikroservisa mogu da se implementiraju u različitim tehnologijama i u različitim programskim jezicima
* Moguće je korišćenje različitih baza podataka
* Komponente se mogu nezavisno isporučivati i automatski postavljati na server
* Moguće je skaliranje samo onih komponenata kojima je to potrebno

Uprkos brojnim prednostima, arhitektura bazirana na mikroservisima ima i određene nedostatke:

* Složenost međuservisne komunikacije
* Otežano testiranje
* Potreba za robusnim uravljanjem greškama i automatskim oporavkom

## Paterni u mikroservisnoj arhitekturi

Dizajn paterni u programiranju su univerzalna rešenja koja se mogu primeniti na probleme koji se često javljaju. Razlog za korišćenje paterna je taj da su oni testirani i provereni pa se time ubrzava proces razvoja. Takođe povećavaju čitljivost programskog koda i standardizuju rešavanje određenih problema.

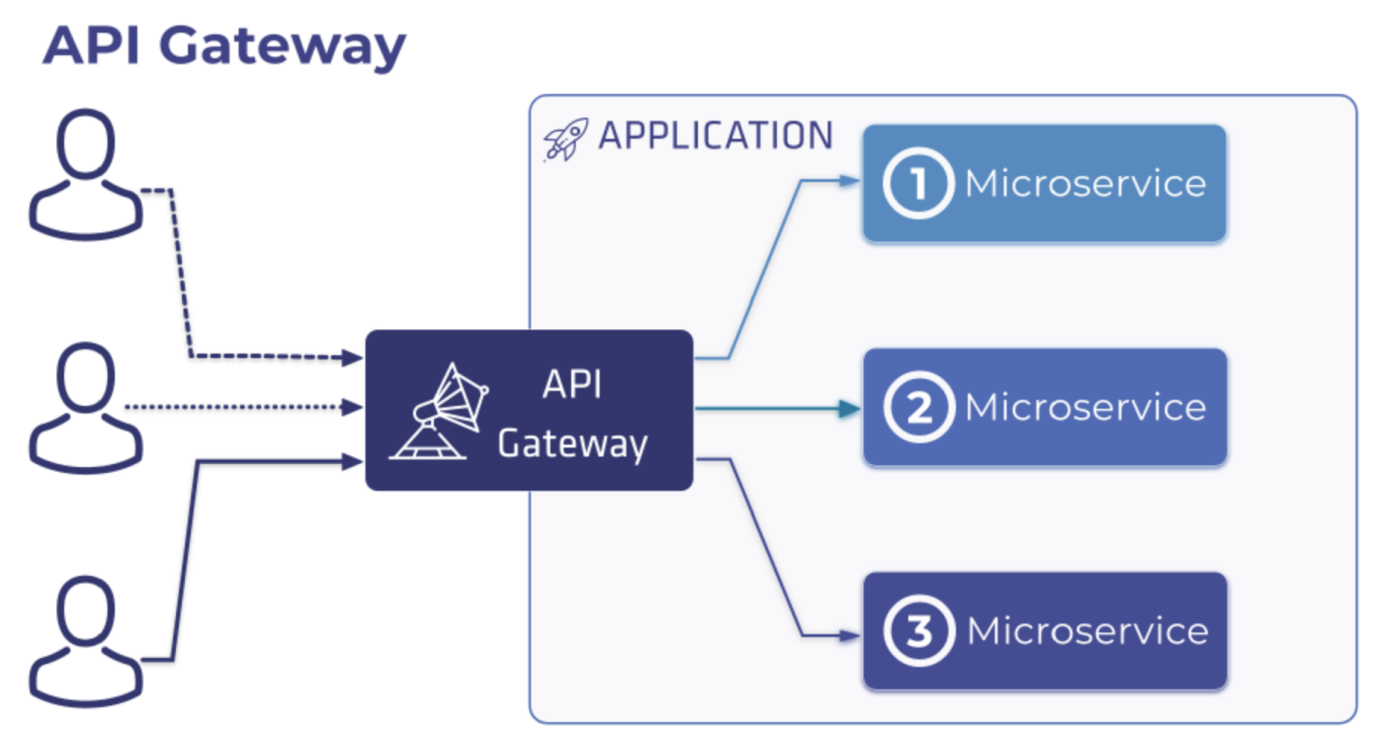
U mikroservisnoj arhitekturi primenjuje se mnogo paterna, par najbitnih biće opisani.

***Adapter***

Adapter omogućuje povezivanje dve nekompatabilne komponente, najčeće se implementira kao omotač oko jedne od postojećih klasa i daje izmenjen pristup toj klasi. Kod mikroservisne arhitekture koristi se isti princip samo što predmet nisu klase nego drugi servisi.

***API Gateway***

Ovim paternom kreira se prolaz (gateway) za sve pozive prema aplikaciji. On stoji ispred servisa u mikroservisnoj arhitekturi i služi kao ulazna tačka za sve pozive koje kreira klijent.



*Slika 2.2.1: Api gateway patern*

***Bulkhead***

Ovaj patern dobio je naziv po praksi da se na brodu pregrađuje transportni prostor, na taj način ako se deo broda poplavi onda se gubi samo deo tereta. U mikroservisnoj arhitekturi princip je sličan, cilj je da ako deo aplikacije ne radi, ostatak nastavi da radi i da pruža što vise usluga klijentima.

***Baza po servisu (Database per Service)***

U mikroservisima servisi moraju da imaju međusobnu slabu vezu (loosely coupled), baze moraju da da imaju mogućnost da se repliciraju i neki servisi imaju potrebu za raličitom vrstom baze od drugih servisa.

Da bi se ovi problemi rešili koristi se patern *baza po servisu*. Ideja je da svaki servis koji ima potrebu da čuva neke podatke, ima sopstvenu bazu podataka. Na taj način svi servisi mogu da koriste tačno onu bazu koja je najbolja za njih, nemamo zavisnost između servisa i lakse repliciramo onu bazu koja nam je potrebna da se replicira.

Pored ovih navedenih paterna, u mikroservisnoj arhitekturi se koristi jos dosta drugih, samo neki od njih koje je potrebno navesti i koji će biti korišćeni u prakticnom delu rada su:

* Health check
* Service discovery
* Circuit braker

## Dobre prakse u dizajniranju mikroservisa

Kao i kod razvoja regularnih aplikacija i u razvoju mikroservisa postoji set dobrih praksi. Poštovanje ovih praksi dovodi do toga da cela arhitektura bude dobro organizovana i laka za održavanje i nadogradnju kao i da se osigura da svi mikroservisi rade pravilno i brzo.

***Single responsibility principle***

Kao i kod objektno orijentisanog programiranja i mikroservisi treba da imaju samo jedan razlog za izmenu. Mikroservis treba radi jednu određenu stvar i treba da se izbegne da mikroservis mora da se menja iz više različitih poslovnih konteksta.

***Posebne baze za svaki servis***

Koristeći jednu bazu za sve mikroservise oni gube svrhu. Svaki servis koji čuva neke podatke treba da ima sopstvenu bazu podataka. Takođe treba koristiti za svaki servis bazu podataka koja je najpogodnija za podatke koje on lpanira da čuva, možda je za jedan mikroservis najbolja relaciona baza podataka dok je za neki drugi servis bolje baza podataka koja koristi grafove.

***Rutiranje poziva kroz API Gateway***

Klijenti ne treba da znaju koji mikroservisi postoje u sistemu, njemu treba obezbediti jadn servis koji će on da poziva a onda da taj servis poziva ostale mikroservise koji su potrebni da bi se dobio odgovor. API Gateway treba da zna koje mikroservise treba da zove za određene podatke a klijentu treba da se samo ostavi da pozove gateway.

***Otpornost na greške***

Ako dođe do kvara na nekom mikroservisu, ostatak sistema treba da nastavi da radi. Ako uzmemo za primer apkiaciju za gledanje filmova i serija preko interneta Netflix, ako dođe do kvara u servisi koji se brine o prevodima za neki film ili seriju, korisnik treba da ima mogućnost da nastavi da gleda samo bez prevoda. Treba obezbediti da ne dođe do toga da ceo sistem ne radi zato što jedan deo sistema ne radi.

***Dizajn vođen domenom (Domain driven design)***

Mikroservis treba razvijati na osnovu domena. To znači da treba odvojiti servise tako da se svaki servis brine o jednom domenu, na primeru prethodne Netflix apliakcije servis etreba odvajati tako da posotji servis za prevode, servis za serije, servis za filmove itd. Treba izbegnuti da jedan servis radi sa više različitih domena.

## Mikroservisna arhitektura naspram mvc arhitekture

MVC (Model-View-Controller) arhitektura je način na koji se deli aplikacija ili preciznije, logika aplikacije u tri dela: Model, pogled (view) i kontroler (controller).

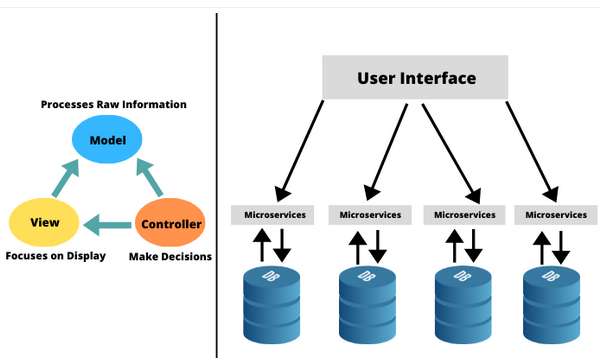
**Model** je deo koji se prine o podacima, njegov posao je da pročita sirove podatke iz baze, organizuje ih i složi tako da mogu da budu procesuirani od strane kontrolera.

**Pogled** se fokusira na prikazivanje podataka. Na ovom mestu će se podaci koje je model vratio iz baze prikazati korisniku aplikacije.

**Kontroler** je deo koji se brine o logici i o kodu koji služi za donošenje odluka. Tu se nalazi biznis logika aplikacije i određuje se šta će sledeće biti prikazano.

Između mikroservisne arhitekture i MVC arhitekture ima dosta sličnosti, obe imaju odvojenu (decoupled) arhitekturu i na komponentama u obe arhitekture može da se radi nezavisno.

Razlika između njih je u tome kako se komponente dele. U MVC arhitekturi iako imamo podelu na komponente, sve one se i dalje nalaze u okviru jednog projekta, sa izuzetkom pogleda koji može biti odvojen projekat, dok su kod mikroservisne arhitekture sve komponente posebni servisi i ima ih mnogo više od tri u MVC arhitekturi.



*Slika 2.3.1: MVC arhitektura (levo) i Mikroservisna arhitektura (desno)*

Kako se kompleksnost projekata povećava iz dana u dan potrebno je koristiti neku arhitekturu koja dozvoljava da se veliki problem podeli na manje probleme. Zavisno od toga koji problem treba rešiti i kakav je taj problem treba izabrati pravu arhitekturu.

MVC arhitektura nije pogodna za svaki problem kao ni mikroservisna arhitektura zato treba dobro iznalizirati pre početka raada koja arhitektura je bolja za određeni projekat.

Za manje projekte i manji tim MVC arhitektura je bolje jer je razvoj lakši ali za velike projekte i projekte na kojima radi više timova, mikroservisi su bolji izbor.

Svakako, ako na početku projekta radi mali tim i krene sa MVC arhitekturom pa projekat i tim kasnije narastu, moguće je odraditi izmenu i prebaciti projekat na mikroservisnu arhitekturu.

# Primena mikroservisne arhitekture u Spring radnom okviru

## Istorijat Spring radnog okvira

Spring radni okvir postoji već dugi niz godina. Predstavljen je od strane Rod Džonsona (Rod Johnson) u oktobru 2002. godine i to u njegovoj knjizi “*Expert one-on-One J2EE Design and Development*”.

Zvanično je prva verzija postala dostupna u junu 2003. godine i to pod *Apache 2.0* licencom što znaci da je radni okvir besplatan za korišćenje i modifikaciju u bilo koje svrhe. Iako je prva verzija izašla 2003. godine, prva produkciona verzija pojavila se skoro godinu dana kasnije u martu 2004. godine.

Nakon prve verzije usledile su verzije:

* 2.0 u oktobru 2006.
* 2.5 u novembru 2007.
* 3.0 u decembru 2009.
* 3.2.5 u novembru 2013.
* 4.0 u decembru 2013.

Trenutna verzija Springa je verzija 5.3.6. Verzija 5.0 je donela značajne novine a jedna od njih je podrška za reaktivno programiranje.

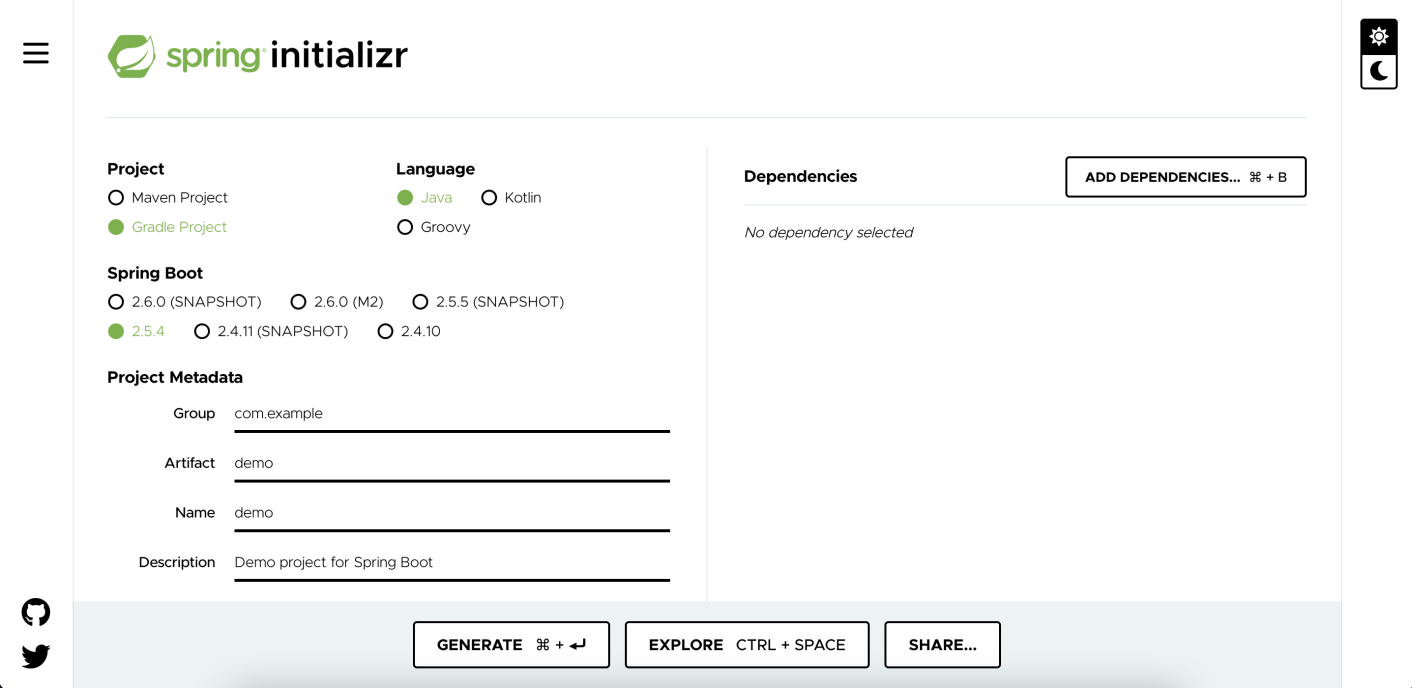
Najznačanije za razvoj mikroservisa je predstavljanje Spring boot radnog okvira koji je na neki način nastavak ili pojednostavljenje Spring radnog okvira.  
Spring boot je namenjen razvoju mikroservisa i namenjen je da olakša kreiranje projekata tako što odmah nakon kreiranja projekta nudi automatski konfigurisanu i “production ready” aplikaciju.

U ovom radu glavni fokus će biti na kreiranju mikroservisa koristeći Spring boot radni okvir.

## Prikaz mikroservisne arhitekture u spring radnom okviru

Kao što je već navedeno, za razvoj mikroservisne arhitekture u ovom radu koristiće se Spring boot. Novi Spring boot projekat je najlakše kreirati na sajtu <https://start.spring.io>.

Na ovom sajtu potrebno je uneti podatke o projektu, izabrati jezik u kome želimo da radimo i podesiti koje zavisnosti želimo da budu podešene u generisanom projektu.

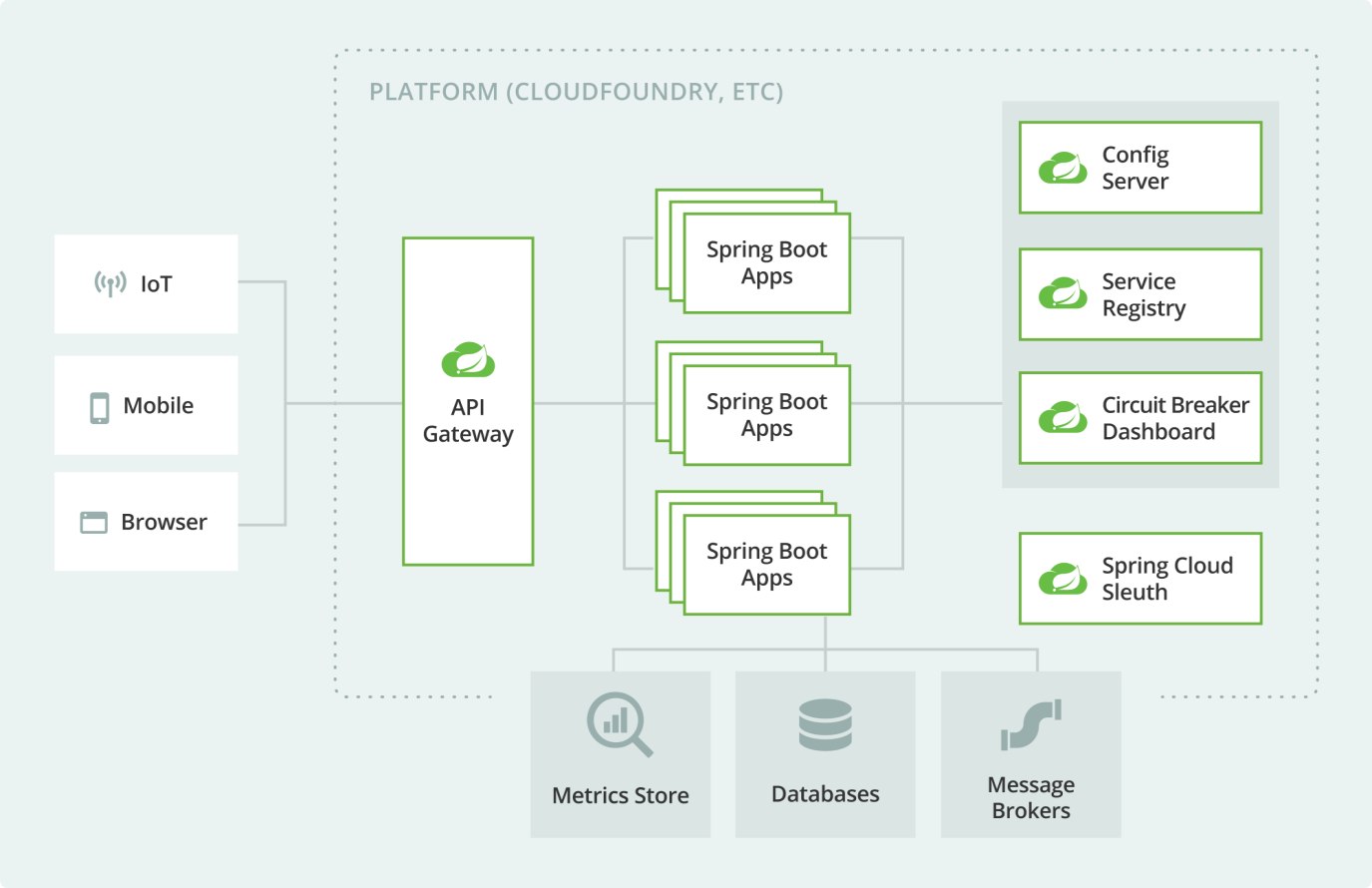


*Slika 3.2.1: Izgled forme za generisanje Spring boot servisa*

Spring boot sadrži već neke postojeće mikroservise koji služe u mikroservisnoj arhitekturi, na primer *Gateway (proxy)* servis ili *Discovery* servis, potrebno je samo uključiti ih u generisani servis kao zavisnost i uraditi konfiguraciju.

Većina mikroservis projekata koji koriste Spring boot radni okvir za razvoj ima sličnu arhitekturu, na taj način je omogućeno da programeri koji su radili na nekom projektu koji koristi Spring boot vrlo lako rade i na nekom drugom koji korisit istu tehnologiju.

Kada se pokreće novi projekat u Spring boot-u obično se podigne jedan servis koji služi kao Gateway, u Springu za to se koristi zavisnost pod nazivom *Zuul* kao i servis koji služi kao *Discovery* server, u Springu se za to koristi zavisnost pod nazivom Eureka. Tipična arhitektura jednog Spring boot mikroservis projekta data je na slici ispod:



*Slika 3.2.2: Izgled tipične Spring boot mikroservisne arhitekture*

Kao što vidimo na slici, tipična arhitektura se sastoji iz više delova a najvažniji za ovaj rad su:

* Api Gateway-a koji služi kao proxy server koji rutira zahteve sa klijenta
* Service Registry-ja koji služi da vodi računa koji mikroservisi su trenutno podignuti i gde se nalaze
* Config Server-a koji je opcioni i služi za konfigurisanje mikroservisa
* Više mikroservisa koji se bave poslovnom logikom

Pored navedenih stvari na slici vidimo i elemente kao što su baza podataka, klijentske aplikacije itd. ali one nisu deo Spring radnog okvira.

# Zahtevi i arhitektura sistema

## Opis zahteva

Za potrebe ovog rada, kako bi pokazali kako se razvija aplikacija sa mirkoservisnom arhitekturom potrebno je kreirati eBanking apliakciju koja zadovoljava sledeće uslove:

* Zaposlenima u banci potrebno je dozovliti kreiranje naloga za klijente
* Zapolsenima u banci potrebno je dozvoliti kreiranje naloga za druge zaposlene
* Zaposlenima u banci potrebno je dovoliti kreiranje računa za klijente
* Zapolsenima u banci potrebno je obezbediti pregled svih klijenata
* Zaposlenima u banci potrebno je obezbetiti pregled svih računa
* Zaposlenima je potrebno dozvoliti brisanje korsnika
* Zaposlenima je potrebno dozovliti brisanje računa
* Zaposlenima u banci potrebno je obezbediti pregled svih transakcija za određeni račun
* Klijentima je potrebno dozvoliti pregled sopstvenih računa
* Klijentima je potrebno dozvoliti kreiranje nove transakcije
* Klijentima je potrebno dozvoliti pregled transakcije

## Izabrane tehnologije

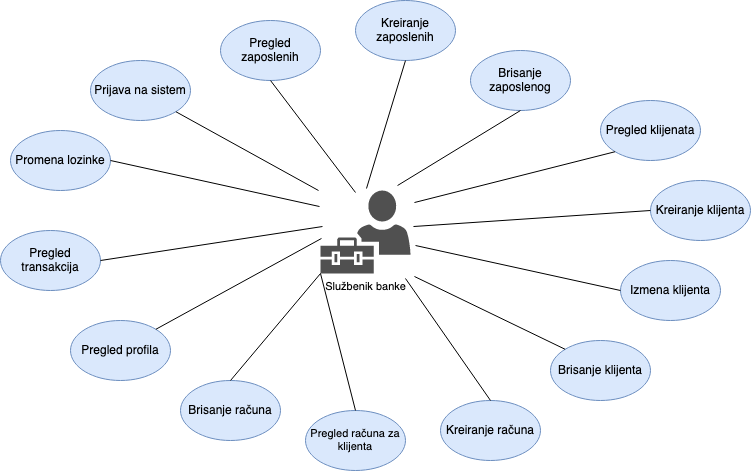
Za izradu projekta izabrane su sledeće tehnologije:

* Za serverski deo (backend) izabrana je Java 11 sa Spring radnim okvirom
* Za klijentski deo (frontend) izabrane je Angular 12
* Za baze podataka izabrani su PostgreSQL i MongoDB

O svakoj tehnologiji i razlogu za njen izbor biće reči u nastavku rada.

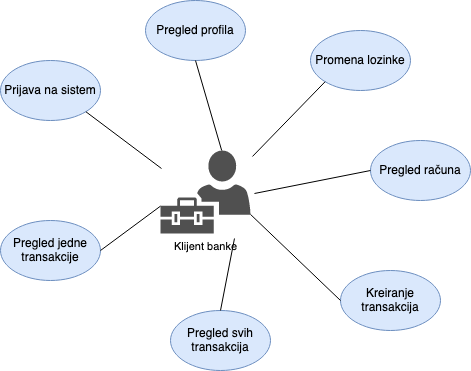
## Slučajevi koršćenja sistema

**Slučajevi korišćenja za zaposlene u banci**

****

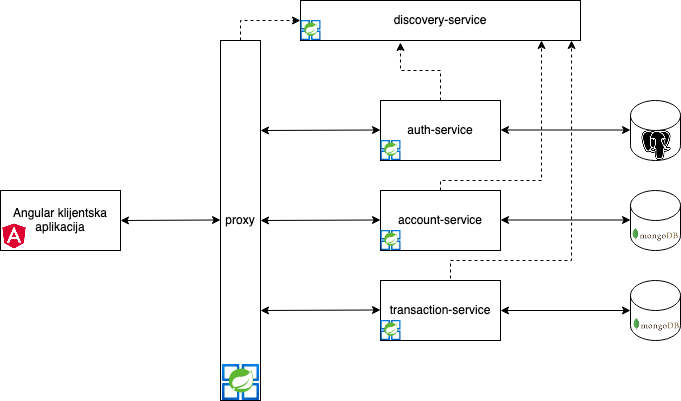
*Slika 4.2.1: Dijagram slučajeva koričćenja za zaposlene u banci*

**Slučajevi korišćenja za klijente**

****

*Slika 4.2.2: Dijagram slučajeva koričćenja za klijente*

## Arhitektura sistema



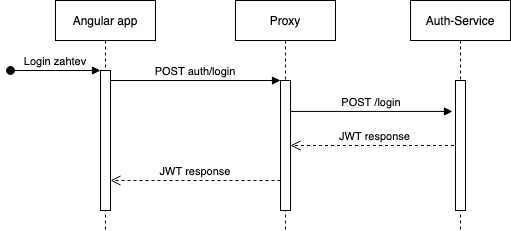
*Slika 4.3.1: Arhitektura sistema*

Na slici iznad dat je izgled izabrane arhitekture za implementaciju sistema. Kao što je navedeno u listi izabranih tehnologija, na serverskoj strani koriste se mikroservisi razvijeni u Spring boot radnom okviru i dve različite baze podataka dok se na klijentskoj strani koristi Angular radni okvir. Sledi opis svake komponente:

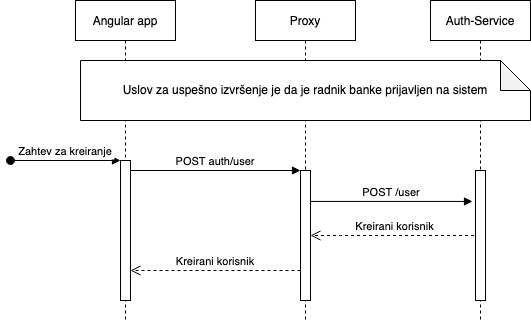
* **Angular klijentska aplikacija -** Predstavlja web aplikaciju koju putem internet pregledača koriste i klijenti i zaposleni u banci kako bi pristupili eBanking sistemu
* **proxy -** Predstavlja servis koji služi kao API Gateway koji rutira pozive sa klijentske aplikacije prema konkretnom mikroservisu
* **discovery-service -** Predstavlja servis kome se svi ostali mikroservisi javljaju (isprekidana linija na dijagramu) i koji vodi računa o tome koji servisi su dostupni i na kojoj adresi se oni nalaze
* **auth-service -** Predstavlja mikroservis koji je zadužen za autentifikaciju korisnika kao i za sve operacije nad korisnicima
* **account-service -** Predstavlja servis koji je zadužen za sve operacije nad bankovnim računima
* **transaction-service -** Predstavlja servis koji je zadužen za operacije nad transakcijama

## Sekvencijalni dijagrami

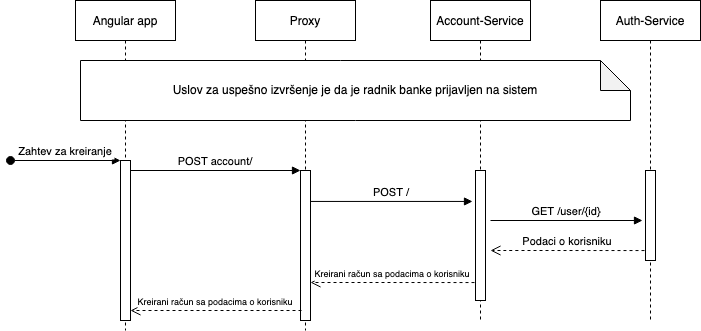
Skevencijalni dijagrami koriste kako bi korisnicima dokumentacije dali detalajn pregled nekog slučaja korišćenja uključujući i pregled svih komponenata sistema i razmenu poruka između komponenata. Ispod su dati sekvencijalni dijagrami za neke slušajeve korišćenja:



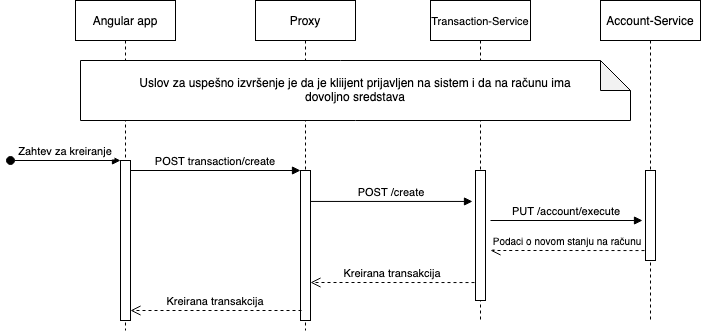
*Slika 4.4.1: Sekvencijalni dijagram za prijavu na sistem*

**

*Slika 4.4.2: Sekvencijalni dijagram za kreiranje klijenta*

**

*Slika 4.4.3: Sekvencijalni dijagram za kreiranje računa*

**

*Slika 4.4.4: Sekvencijalni dijagram za kreiranje transakcije*

# Implementacija i prikaz sistema

## Baze podataka