

* **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

[Sadržaj 2](#_Toc592808313)

[1 Uvod 1](#_Toc611821919)

[2 Razvoj mikroservisne arhitekture 3](#_Toc739290797)

[Istorija mikroservisne arhitekture 3](#_Toc2067527284)

[Monolitna arhitektura 4](#_Toc498170081)

[Mikroservisna arhitektura 5](#_Toc1853295361)

[Prednosti i mane mikroservisne arhitekture 6](#_Toc1232316239)

[Paterni u mikroservisnoj arhitekturi 6](#_Toc1206737205)

[Dobre prakse u dizajniranju mikroservisa 8](#_Toc796642167)

[Mikroservisna arhitektura naspram mvc arhitekture 9](#_Toc1751845371)

[3 Primena mikroservisne arhitekture u Spring radnom okviru 11](#_Toc1264350027)

[Istorijat Spring radnog okvira 11](#_Toc580216724)

[Prikaz mikroservisne arhitekture u spring radnom okviru 12](#_Toc2126722888)

[4 Zahtevi i arhitektura sistema 14](#_Toc1113757948)

[Opis zahteva 14](#_Toc1462364784)

[Izabrane tehnologije 14](#_Toc14584773)

[Slučajevi koršćenja sistema 15](#_Toc313144053)

[Arhitektura sistema 16](#_Toc1677163621)

[Sekvencijalni dijagrami 17](#_Toc218627625)

[5 Implementacija i prikaz sistema 19](#_Toc129973358)

[Baze podataka 19](#_Toc471358907)

[Struktura Spring boot projekata 23](#_Toc61976166)

[Komunikacija između mikroservisa 30](#_Toc103853167)

[Struktura Angular projekta 31](#_Toc1703456405)

[Primenjeni dizajn paterni 33](#_Toc1887300678)

[Obrada izuzetaka 34](#_Toc1529028956)

[Swagger 36](#_Toc1600343490)

[Jedinično testiranje 39](#_Toc1887841402)

[Integraciono testiranje 41](#_Toc2027042636)

# 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

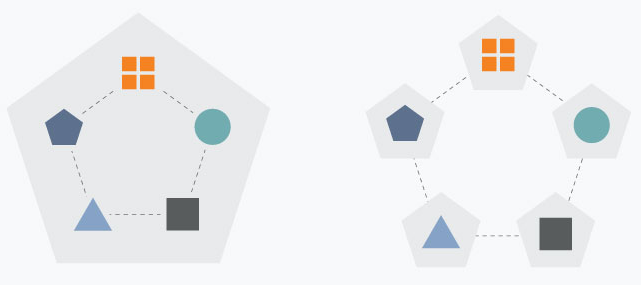
Mikroservisna arhitektura je princip razvoja softvera u obliku malih, izdvojenih servisa, pri čemu svaki servis ima svoj proces i ostvaruje komunikaciju putem jednostavnih mehanizama kao što je HTTP API.

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.

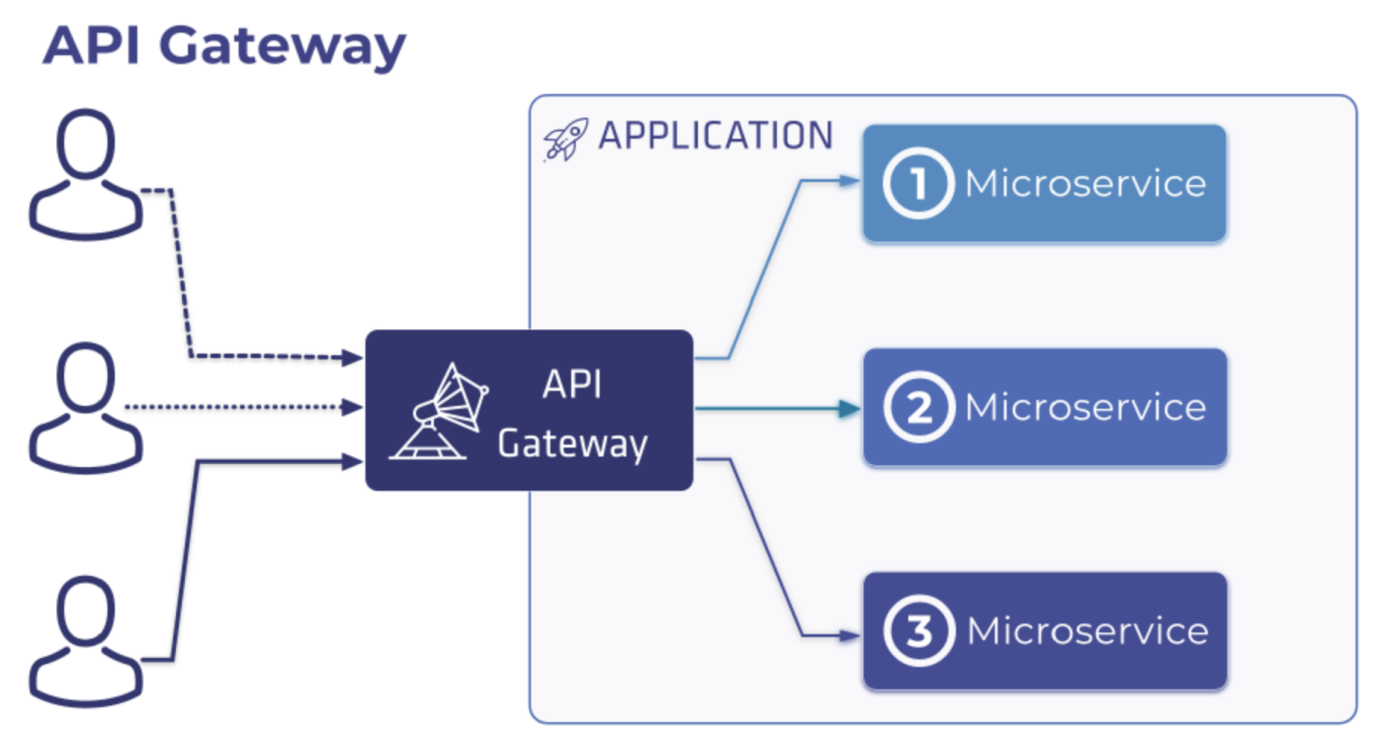
U mikroservisnoj arhitekturi primenjuje se mnogo paterna, par najbitnih biće opisani.

***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).

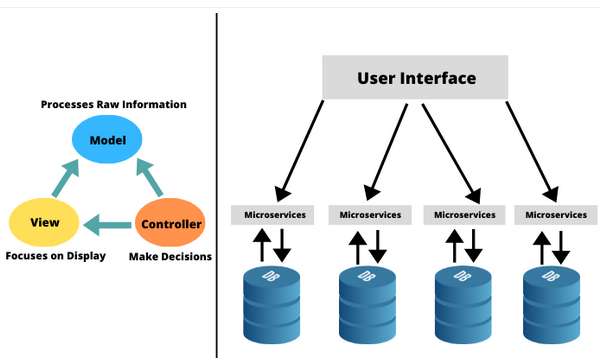
**Model** je deo koji se prine o podacima, njegov posao je da pročita sirove podatke iz baze, organizuje ih i složi tako da mogu da budu procesuirani od strane kontrolera.

**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.

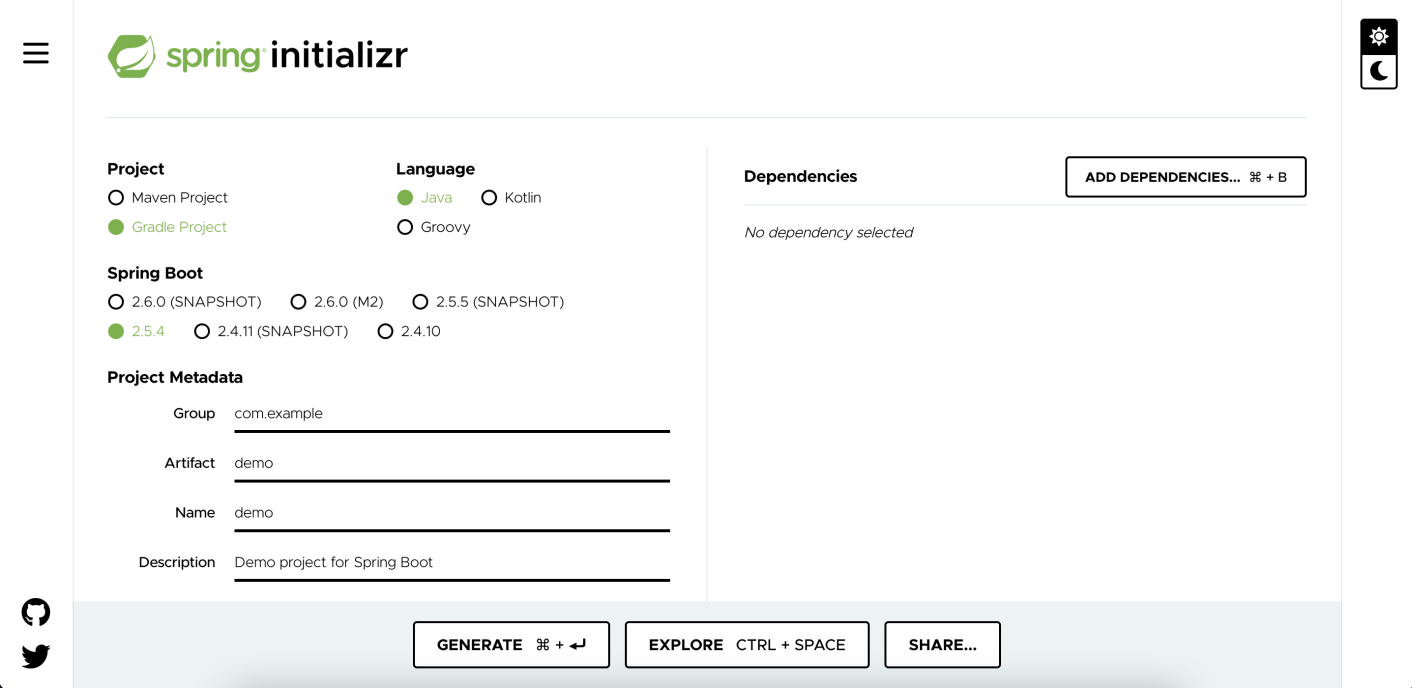
Najznačanije za razvoj mikroservisa je predstavljanje Spring boot radnog okvira koji je na neki način nastavak ili pojednostavljenje Spring radnog okvira.  
Spring boot je namenjen razvoju mikroservisa i namenjen je da olakša kreiranje projekata tako što odmah nakon kreiranja projekta nudi automatski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

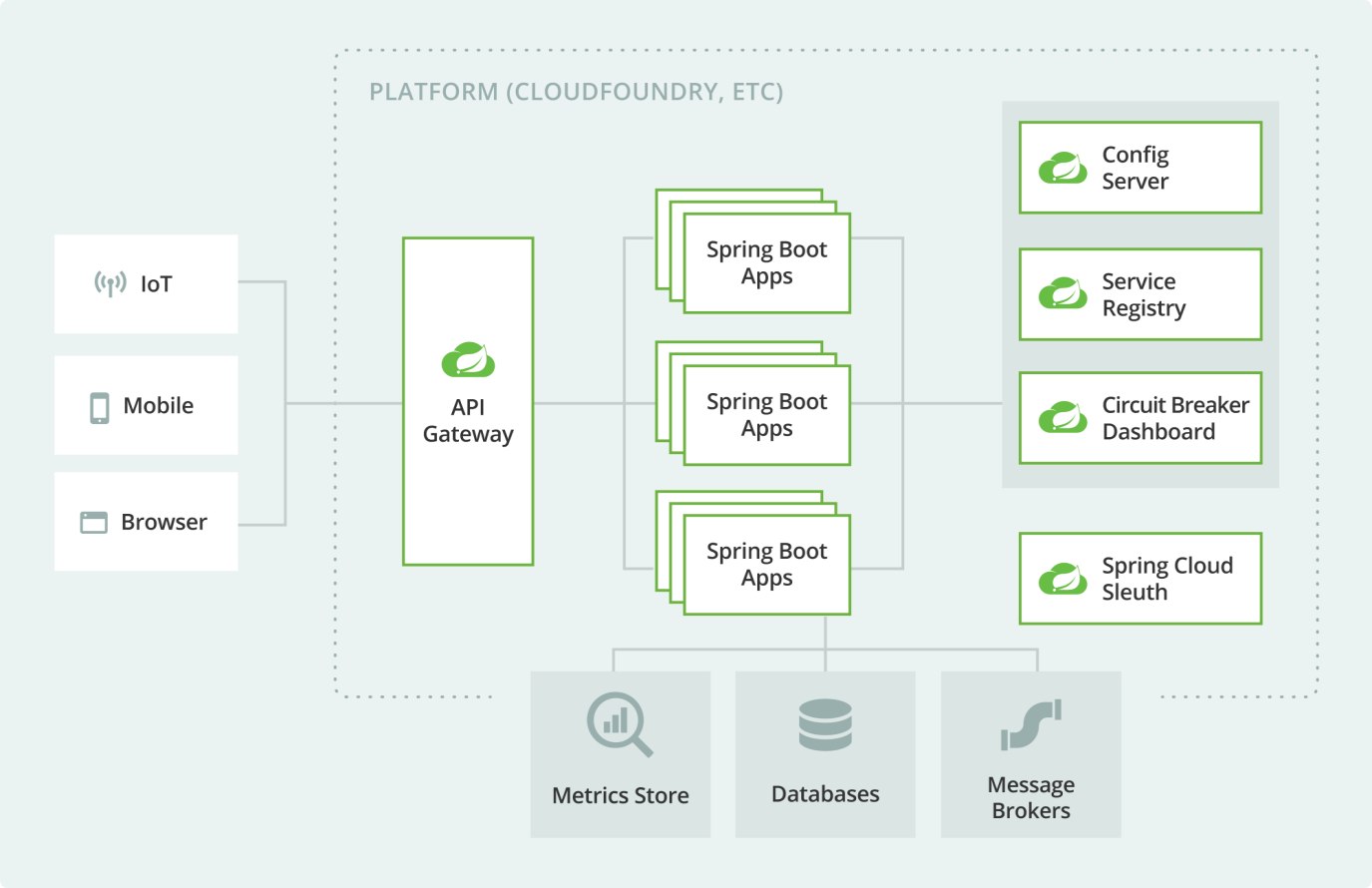


*Slika 3.2.1: Izgled forme za generisanje Spring boot servisa*

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

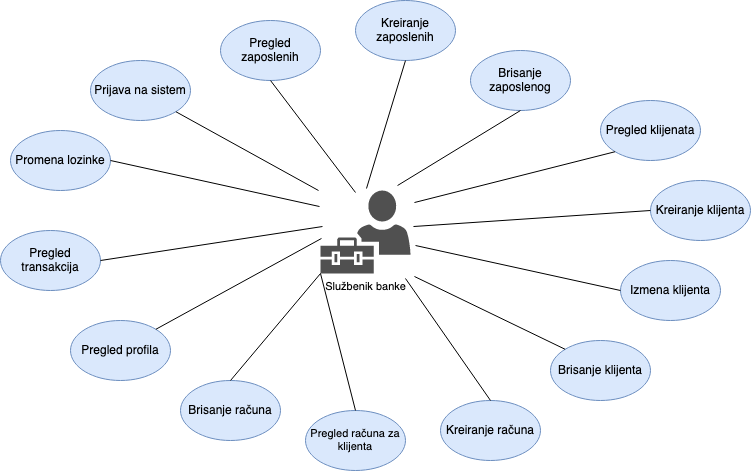
Za izradu projekta izabrane su sledeće tehnologije:

* 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.

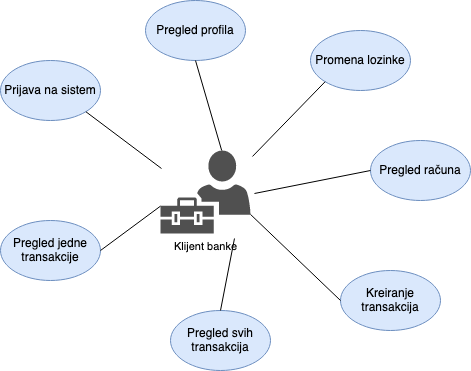
## Slučajevi koršćenja sistema

**Slučajevi korišćenja za zaposlene u banci**

****

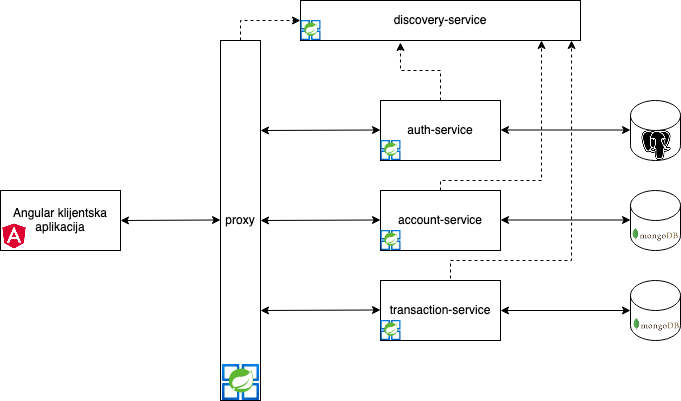
*Slika 4.2.1: Dijagram slučajeva koričćenja za zaposlene u banci*

**Slučajevi korišćenja za klijente**

****

*Slika 4.2.2: Dijagram slučajeva koričćenja za klijente*

## Arhitektura sistema



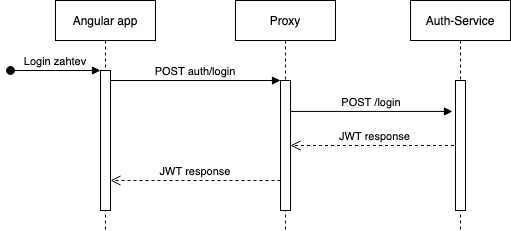
*Slika 4.3.1: Arhitektura sistema*

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:

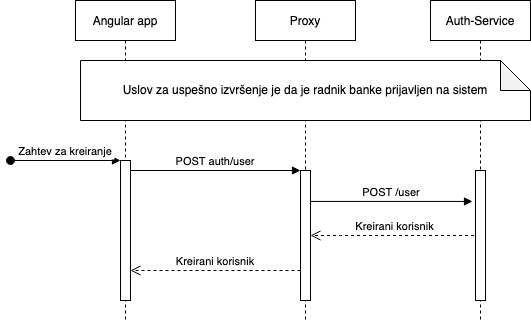
* **Angular klijentska aplikacija -** Predstavlja web aplikaciju koju putem internet pregledača koriste i klijenti i zaposleni u banci kako bi pristupili eBanking sistemu
* **proxy -** Predstavlja servis koji služi kao API Gateway koji rutira pozive sa klijentske aplikacije prema konkretnom mikroservisu
* **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

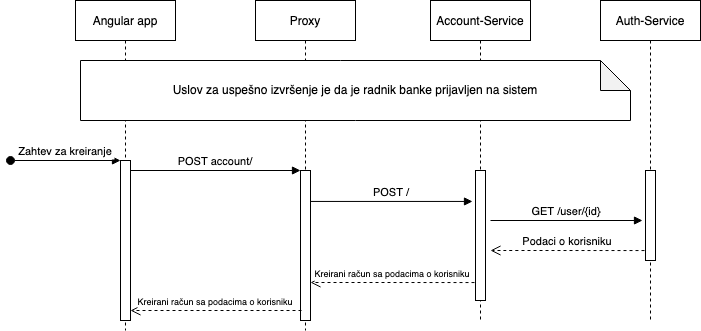
Skevencijalni dijagrami koriste kako bi korisnicima dokumentacije dali detalajn pregled nekog slučaja korišćenja uključujući i pregled svih komponenata sistema i razmenu poruka između komponenata. Ispod su dati sekvencijalni dijagrami za neke slušajeve korišćenja:



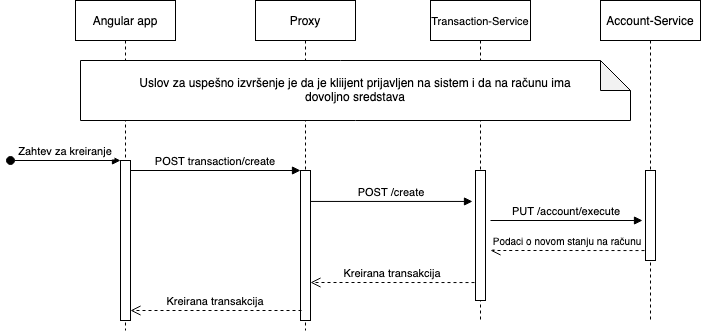
*Slika 4.4.1: Sekvencijalni dijagram za prijavu na sistem*

**

*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

Kako bi se ispoštovale dobre prakse za razvoj mikroservisne arhitekture, treba ispoštovati princip o kome je ranije bilo reči a to je princip “baza po servisu”.

Mikroservisi se razvijaju tako da svaki mikroservis koji ima potrebu da čuva neke podatke, koristi sopstvenu bazu podataka. Ovaj princip ima dosta prednosti ali i par mana.

Glavna prednost je ta što su konkretne baze manje i lakše za održavanje kao i što svaki mikroservis može da koristi različitu vrstu baze podataka to jest onu koja je najbolja za čuvanje one vrste podataka koje taj servis treba da čuva.

Mana je ta što se podaci nalaze nalaze na više mesta a ne na jednom, pa je čitanje podataka kada svi podaci nisu na istom mestu otežano, to jest mora da se čita iz više baza a onda da se sve to spoji u jedan odgovor koji se šalje kome je potreban. Iz ovog razloga prilikom dizajniranja mikroservisnih sistema potrebno je obratiti pažnju na to da je nekada bolje određene podatke duplirati na više mesta nego imati mnogo poziva između ikroservisa kako bi dobili podatke.

Kao što je prikazano u prethodnoj sekciji, za ovaj projekat izabrane su dve vrste baza podataka a ukupno sistem ima tri baze. Servisi koji imaju potrebu da čuvaju podatke su ***auth-service, account-service*** i ***transaction-service.***

**Auth-service baza**

Za ovaj servis izabrana je relaciona baza podataka iz razloga što je potrebno čuvati podatke koji su u relaciji jedni sa drugima. Konkretno potrebno je voditi računa koji korisnici imaju koju rolu, tačnije postoji relacije između korisnika i role a za to je najbolje koristiti relacionu bazu podataka.

Za konrektnu relacionu bazu izabrana je **PostgreSQL** baza podataka jer se radi o aplikaciji koja je poslovnog (enterprise) tipa i za te aplikacije se najčešće od relacionih baza bira upravo PostgreSQL. Šema baze za *auth-service* data je na slici ispod:



*Slika 5.1.1: Šema baze auth servisa*

Na prvi pogled baza deluje dosta jednostavno i ne deluje kao da je deo jedne veće aplikacije ali to je upravo i cilj kada se koristi mikroservisna arhitektura. Ideja je da se ova baza koristi isključivo za čuvanje podataka o korisnicima, na taj način postiže se da jedan mikroservis ima samo jednu ulogu.

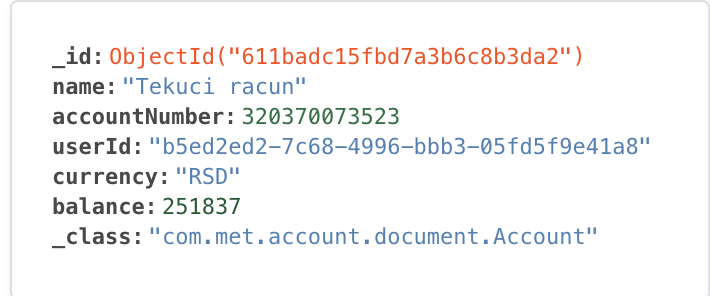
Tabele ***databasechangeloglock*** i ***databasechangelog*** služe za verzionisanje baze podataka i one su deo alata koji se zove ***liquibase*** i koji služi tome da se brine za izmene u šemi baze.

**Account-service baza**

Kao što mu sam naziv kaže, ovaj servis se bavi bankovnim računima. Za konkretno ovaj servis može da se koristi i relaciona i NoSQL baza podataka ali odabrana je NoSQL baza, tačnije odabran je **MongoDB** jer treba da se čuva samo jedan jedini entitet - račun, a u MongoDB bazi možemo da imamo fleksibilnu šemu pa ako nekada bude bilo potrebe da se za račun sačuva još neki dodatni podatak to može jednostavno da se uradi, bez izmene šeme baze i dodavanja novog polja.

MongoDB je baza podataka koja podatke čuva u dokumentima (document based database). Ova baza je jako pogodna za rad sa web aplikacijama jer web aplikacije obično koriste *JSON* dok i baza čuva podatke u *JSON* formatu pa je vrlo lako sačuvati i pročitati podatke, nije potrebna nikakva konverzija podataka.

MongoDB nema tabele kao relacione baze već koristi termin ***Kolekcija*** a u kolekciji čuva ***Dokumente*** (najpribližnije jednom redu iz termina relacionih baza). MongoDB nema fiksnu šemu kao što relacione baze imaju pa ne može da se prikaže ali na slici ispod dat je primer jednog dokumenta koji se čuva u kolekciji ***accounts***

******

*Slika 5.1.2: Primer dokumenta u account bazi*

Pošto šema nije fiksna ako je potrebno da neki račun sadrži još neki dodatni podataka, može samo direktno da se upiše u kolekciju, bez izmene postojećih dokumenata.

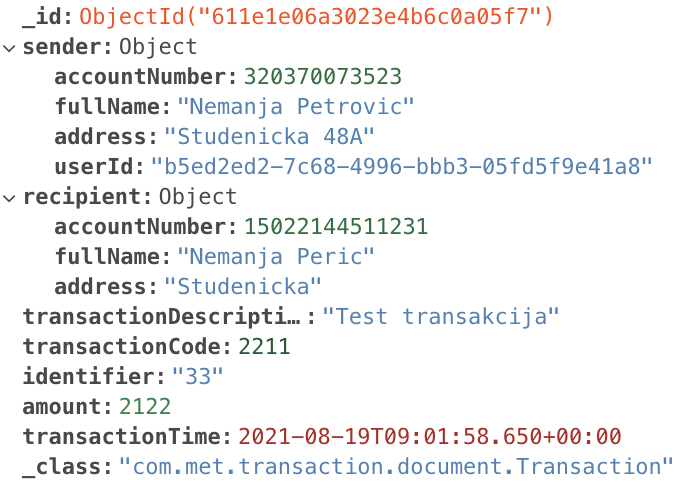
Polje ***userId*** odnosi se na jedinstveni identifikator korisnika to jest ima istu vrednost kao polje ***id***  u *auth* bazi u tabeli *users* i služi tome da se zna kome račun pripada.

**Transaction-service baza**

Ovaj servis potrebno je pamtiti sve transakcije i za njega je izabrana takođe **MongoDB** baza podataka. Pošto je o samoj bazi bilo već reči u prethodnom delu, važno je napomenuti samo da jedan dokument u kolekciji ne mora da sadrži samo primitivne tipove kao što je to slučaj u relacionim bazama već može da sadrži i kompleksne tipove tačnije objekte.

Upravo ovo je bilo od značaja pri izboru ove baze za ovaj servis. Transakcija kada se jednom izvrši više nikad ne sme da se menja, a za nju je potrebno označiti i ko su učesnici u transakciji (ime, broj računa, adresa, id korsnika ako se radi o nekom klijentu banke) a kako bi izbegli da prilikom pregleda transakcije svaki put zovemo i *account* servis i *auth* servis da dobijemo navedene podatke odlučeno je da se podaci koji su potrebni za transakciju sačuvaju u samoj transakciji. Iako je ovo malo dupliranje podataka ono je opravdano jer se time dobija brži sistem.

Pošto MongoDB nema šemu, na slici ispod je dat primer jednog dokumenta unutar kolekcije koja sadrži transakcije:



*Slika 5.1.3: Primer dokumenta u transaction bazi*

Kao što vidimo svaka transakcija sadrži dva objekata koj predstavljaju pošiljaoca i primaoca. Ako je neko od njih (ili oboje) klijent banke, biće popunjeno polje *userId.* Na taj način možemo lako da dobijemo transakcije za nekog korisnika a zavisno da li je on pošiljalac ili primalac možemo da zaključimo da li je to za njega dolazna ili odlazna transakcija.

Svaka transakcija sadrži i broj računa koji može biti interni ili eksterni. Takođe na ovaj način možemo da radimo pretragu po računu i da za račun zaključimo da li je to dolazna ili odlazna transakcija.

## Struktura Spring boot projekata

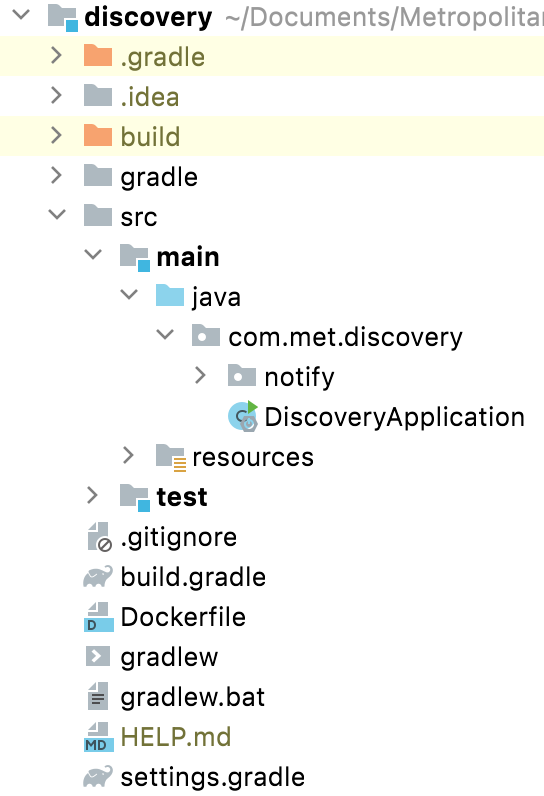
Kao što je prikazano u delu o arhitekturi, serverski deo aplikacije se sastoji od više Spring boot mikroservisa. Za generisanje svakog od njih korišćen je sajt <https://start.spring.io> i svaki od njih je generisan tako da koristi:

* Java 11
* Gradle kao alat za build projekta
* Najnoviju verziju Spring boot-a
* *Jar* format za pakovanje aplikacije

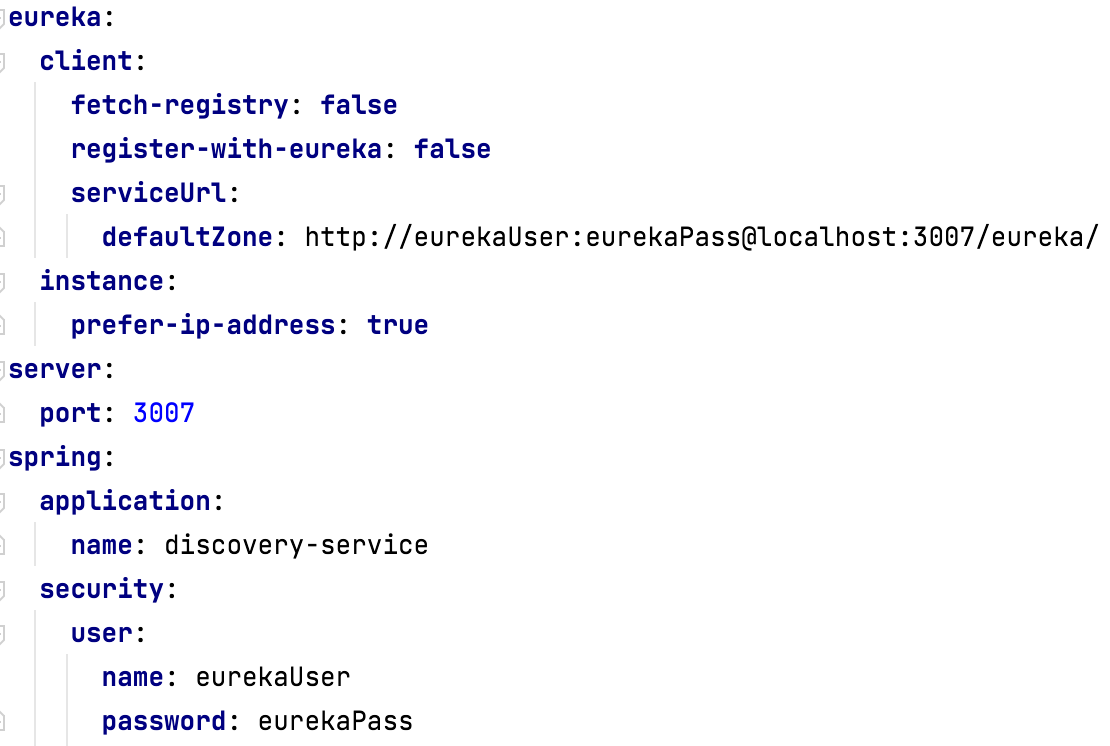
Konkretne zavisnosti za svaki od servisa se razlikuju zavisno od njegovih potreba.

**Discovery-service**

Struktura ovog mikroservisa je najjednostavnija od svih mikroservisa u ovom projektu. On sadrži samo jednu klasu koja služi kao ulaz u aplikaciju i sadrži konfiguraciju u yaml formatu. Cilj ovog mikroservisa je da podigne *Eureka server* koji vodi računa o dostupnosti ostalih mikroservisa i njihovoj lokaciji na mreži. Struktura projekta je sledeća:



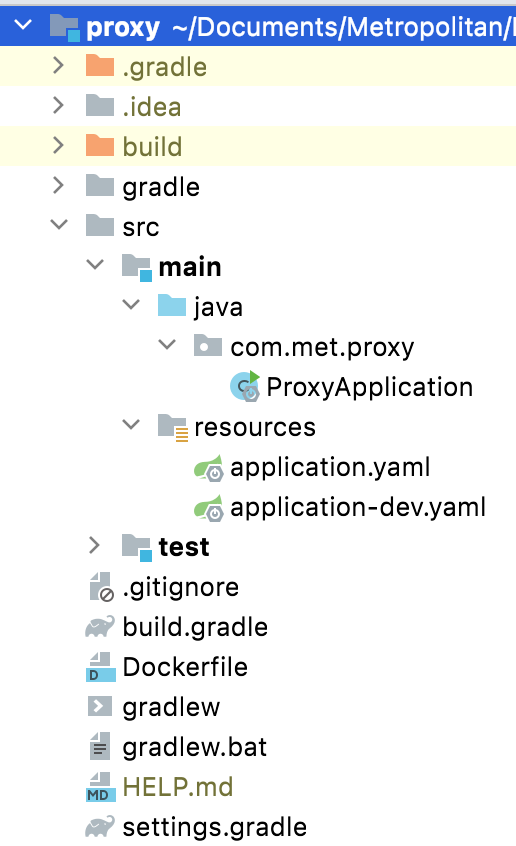
*Slika 5.2.1: Struktura discovery-service projekta*



*Slika 5.2.2: Konfiguracija discovery-service projekta*

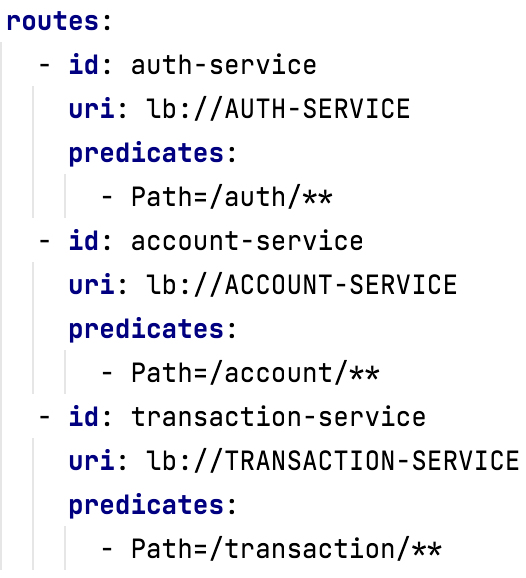
**Proxy-service**

Ovaj mikroservis služi kao API Gateway, to znači da je on ulazna tačka za sve pozive prema svim mikroservisima. Struktura mu je kao i kod discovery servisa jako jednostavna i kao i kod discovery servisa, sve u njemu se podešava kroz yaml konfiguraciju:



*Slika 5.2.3: Struktura proxy projekta*

Deo konfiguracije je sličan kao kod prethodnog servisa, ono što je najbitnije jeste konfiguracija ruta ostalih servisa koja je data na slici:

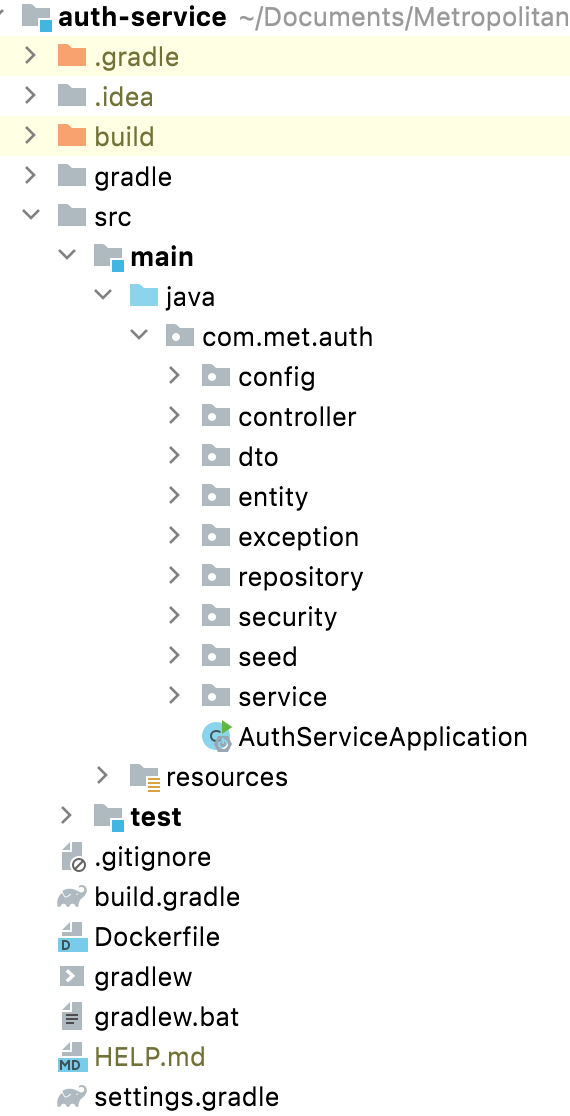


*Slika 5.2.4: Konfiguracija ruta*

Na ovom mestu se podešava šta se dešava kada klijentska aplikacija pošalje zahtev koji počinje određenom sintaksim, na primer */auth/\*\**, tada proxy taj poziv rutira na *auth-service* mikroservis.

**Auth-service**

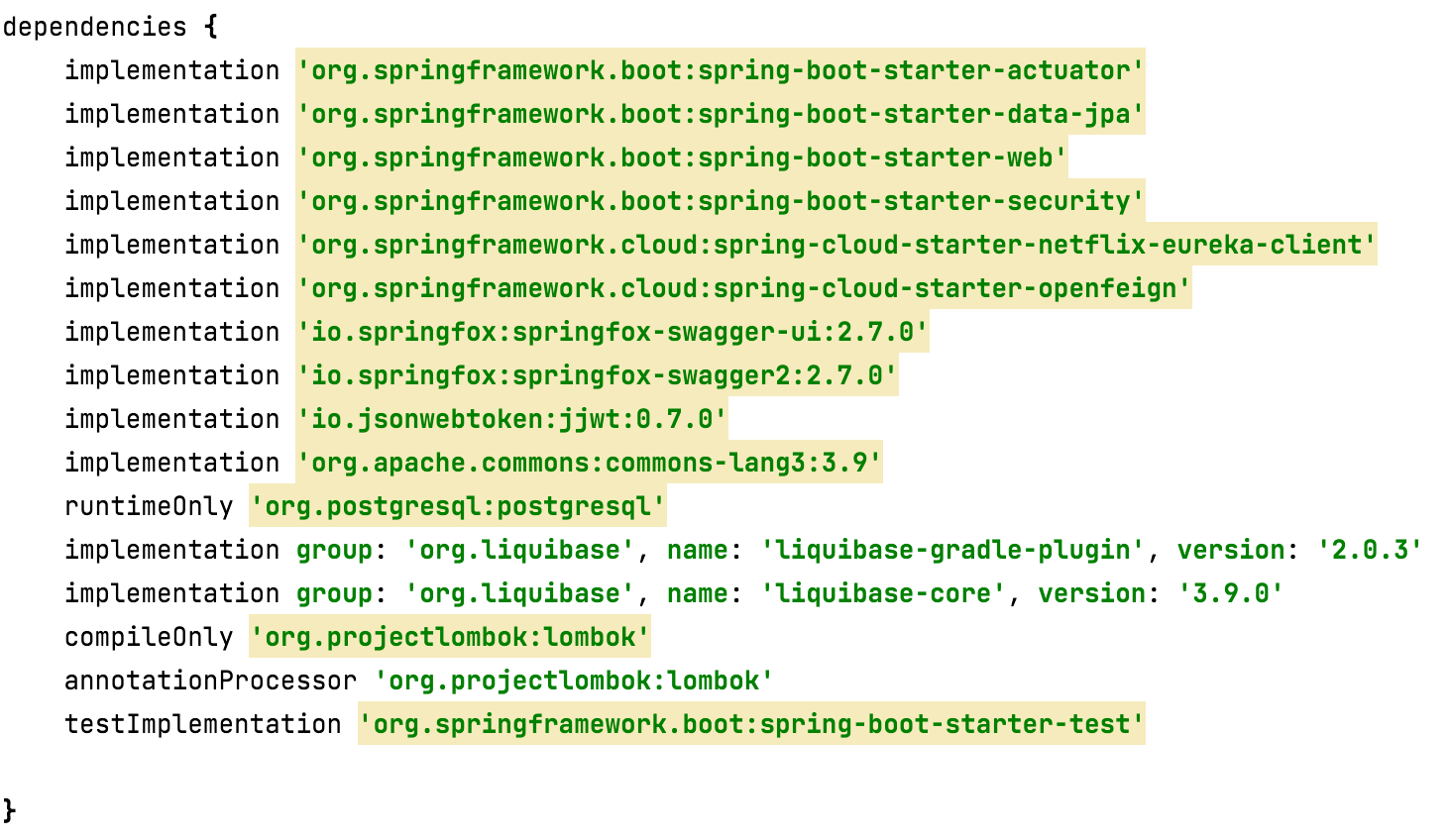
Struktura projekta data je na slici:



*Slika 5.2.5: Struktura auth-service projekta*

Struktura projekta je standardna struktura Java Spring projekata, gde su klase razdvojene na pakete po slojevima, pre nego da se objasni šta se u svakom od paketa nalazi, potrebno je pokazati šta projekat sadrži od zavisnosti (dependencies).

Zavisnosti su definisane u fajlu ***build.gradle:***

******

*Slika 5.2.6: Zavisnosti auth-service projekta*

Servis sadrži dosta zavisnoti iz Spring Boot radnog okvira. Tu su:

* Zavisnost za rad sa bazom - **data-jpa**
* Zavisnost za bezbednost - **security**
* Zavisnost za konekciju na discovery servis - **eureka-klijent**

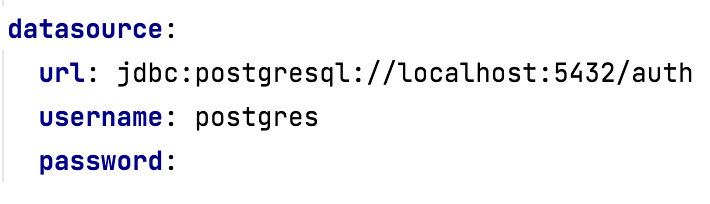
Pored toga sadrži dodatne zavisnosti koje nisu vezane za Spring a najbitnije su:

* Zavisnost za jwt tokene - **jsonwebtoken:jwt**
* Zavisnost za dokumentaciju API-ja - **swagger**
* Zavisnost za pristup PostgreSQL bazi

Sadržaj paketa dat na slici strukture projekata je sledeći:

* **config -** klase vezane za konfiguraciju
* **controller** - REST kontroleri
* **dto** - Klase za prenos podataka
* **entity** - Klase koje predstavljaju entitete koji se čuvaju u bazi
* **exception** - Klase za obradu izuzetaka
* **repository** - Interfejsi za rad sa bazom podataka
* **security** - Klase vezane za bezbednost
* **seed** - Klasa koja ubacuje inicajlne podatke u bazu
* **service** - Interfejsi i njihova implementacija koji sadrže poslovnu logiku

Konfiguracija je slična kao i kod ostalih, ono što je u konfiguraciji ovog projekta bitno je konekcija na PostgreSQL bazu podataka:



*Slika 5.2.7: Konfiguracija za konekciju na bazu podataka*

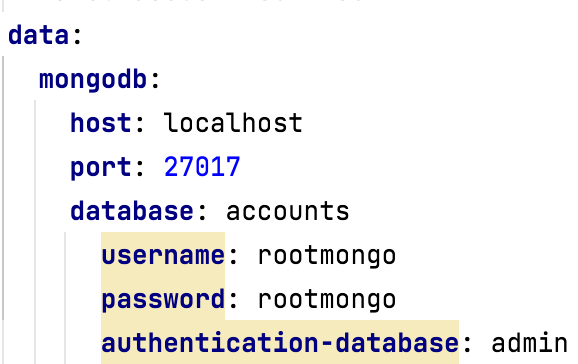
U samu implementaciju ovih klasa nije potrebno duboko zalaziti, kao prilog radu dat je kod a kod se takođe nalazi na javnom github repozitorijumu čija je lokacija data na kraju rada. Ono što je bitno napomenuti za ovaj konkretno mikroservis jeste da se on bavi svime što se tiče samih korisnika aplikacije (njihovo kreiranje, izmena, čitanje, brisanje) i da se pored toga bavi autentifikacijom korisnika koji žele da pristupe sistemu.

Autentifikacija se radi koristeći JWT (Json web token). Nakon uspešne prijave na sistem, korisnik dobija token koji koristi za pristup stranicama. Token sadrži osnovne podatke o korisniku kao i njegovu rolu pa se na osnovu njega proverava da li korisnik ima pravo prisutpa određenom resursu ili ne. Pored toga token ima vreme trajanja, nakon isteka vremena potrebno je dobiti novi token kako bi korisnik mogao da nastavi da koristi aplikaciju.

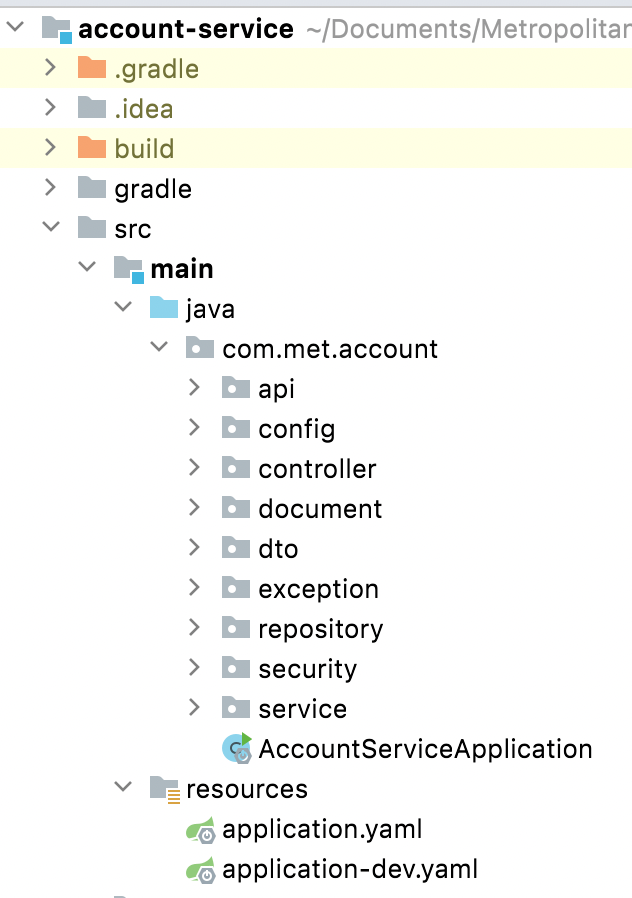
**Account-service**

Jedna od glavnih stvari unutar eBanking sistema je mogućnost da se poveže tekući račun sa korisnikom ili da se korisniku otvori novi tekući račun. Većina banaka podržava račune u različitim valutama kao i mogućnost da korisnik poseduje više racuna pa i sama aplikacija mora da podrži te mogućnosti.

Servis ima konfiguraciju kao i zavisnosti kao i prethodni projekti, ono što se razlikuje jeste zavisnost za rad sa MongoDB bazom kao i konfiguracija za konekciju na MongoDB bazu.



*Slika 5.2.8: Konfiguracija za konekciju na mongo db bazu*

**

*Slika 5.2.9: Struktura account servisa*

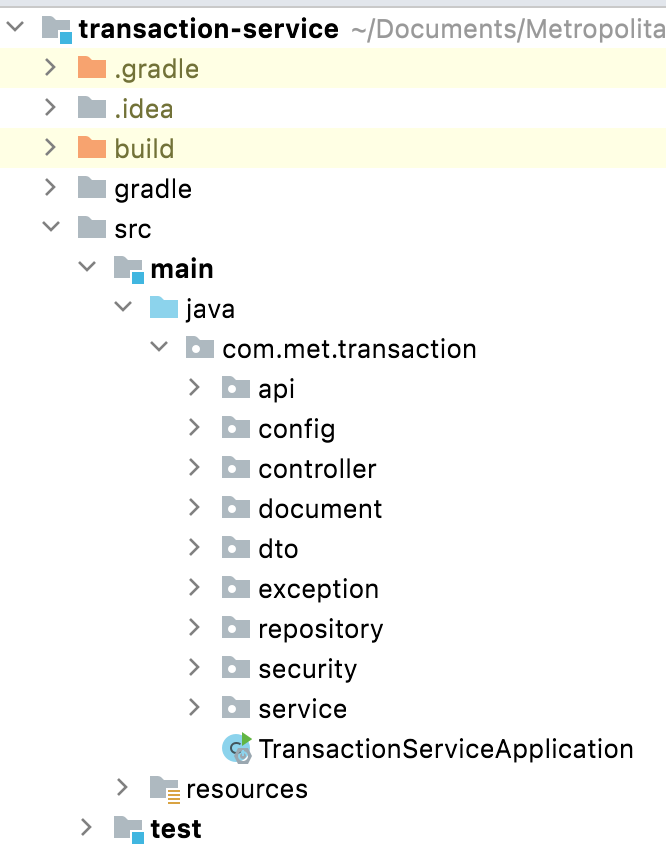
Na slici strukture vidimo razliku izmđu *account* servisa i *auth* servisa a to je da *account* servis nema paket ***entity*** već ima paket ***document***. *Account* servis koristi MongoDB bazu pa je praksa da se klase koje predstavljaju elemente koji se čuvaju u bazi čuvaju u ***document*** paketu jer u samoj bazi ta klasa predstavlja jedan dokument.

**Transaction-service**

Ako se sagleda ceo eBanking sistem najčeće korišćen deo je svkako deo koji se bavi transakcijama. Uplate i isplate se izvršavaju stalno pa je jako bitno da deo sistema koji se bavi transakcijama bude jako brz kao i da ima mogućnost da se skalira po potrebi jer će “trpeti” dosta više saobraćaja nego drugi delovi sistema.

Kako bi problem brzine bio rešen, izabrana je MongoDB baza jer ima jako brzo upisvanje a s obzirom da nemamo relacije unutar transakcije i brzina čitanja će biti jako brza.

Drugi problem, problem skaliranja je rešen samim izborom arhitekture. Pošto je arhitektura mikroservisna, samim tim sve što ima veze sa transakcijama se nalazi u jednom mikroservisu, taj mikroservis može da se skalira nezavisno od ostatka sistema.



*Slika 5.2.10: Struktura transaction servisa*

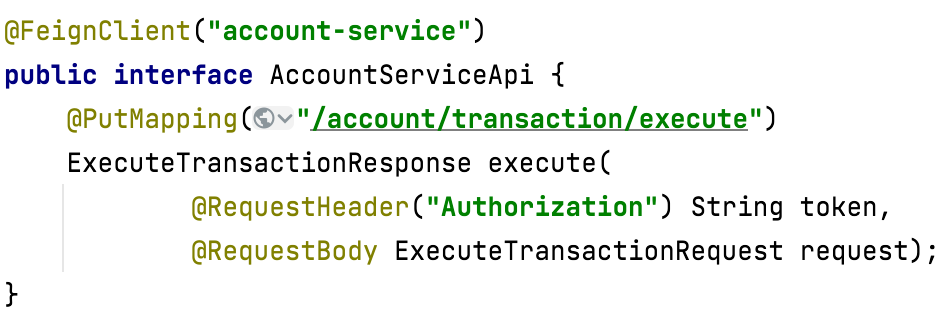
## Komunikacija između mikroservisa

Mikroservisi mogu da komuniciraju na dva načina:

* Sinhrono
* Asinhrono

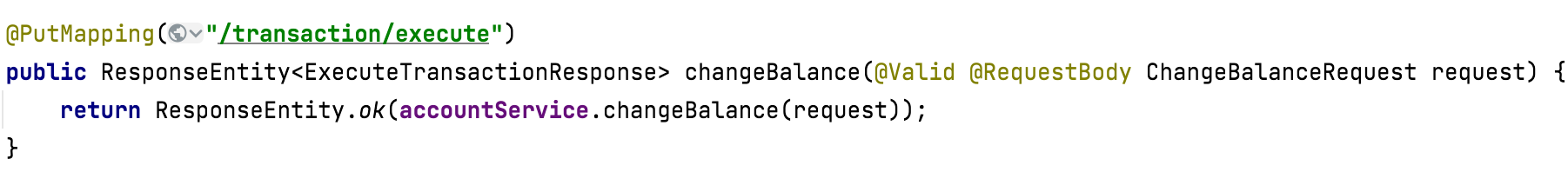
S obzirom da dati sistem nema potrebu za asinhronom komunikacijom, svi mikroservisi komuniciraju sinhrono, koristeći **REST** pozive. Kako se ne bi ručno pravila konekcija i nakon dobijanja odgovora ručno mapirao *json* odgovor na Java klasu, Spring poseduje biblioteku pod nazivom ***feign-client.***

***Feign*** podržava pisanje HTTP poziva tako što se definiše interfejs gde se opiše koji mikroservis se zove, zatim se definišu metode u interfejsu kojima se definiše URI, definišu se parametri za poziv kao i klasa koja se očekuje kao odgovor dok samu implementaciju interfejsa Spring automatski napravi u pozadini.



*Slika 5.3.1: Feign klijent*

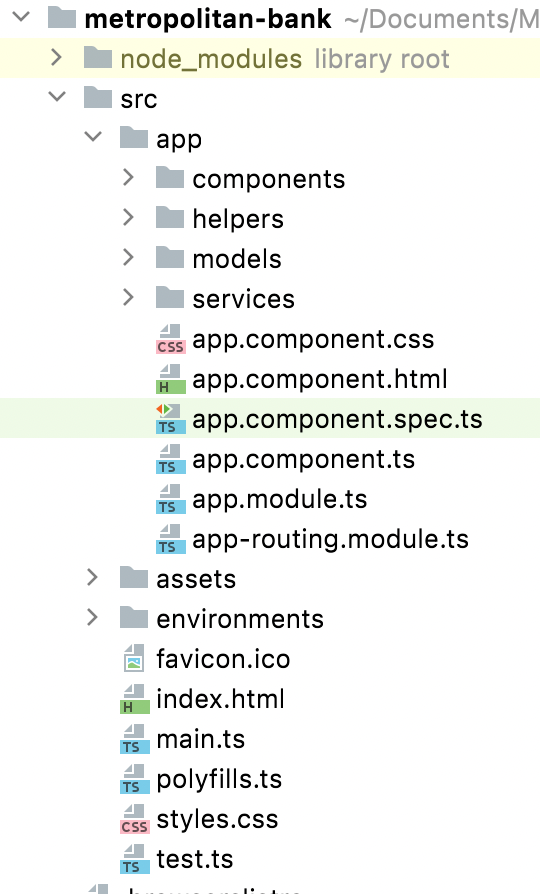
Na slici iznad dat je primer interfejsa iz *transaction* servisa koji služi za komuniciranje sa *account* servisom kao i metoda koja služi izvršavanje transakcije na računu. Metoda poziva URI koji je deo account servisa i kao parametre šalje jwt token u hederu kao i telo koje sadrži objekat klase koja dalje sadrži podatke o transakciji. Kao odgovor očekuje se objekat klase koja sadrži podatke o tome da li je transakcija uspešno izvršena.



*Slika 5.3.2: Izgled resursa na account servisu*

## Struktura Angular projekta

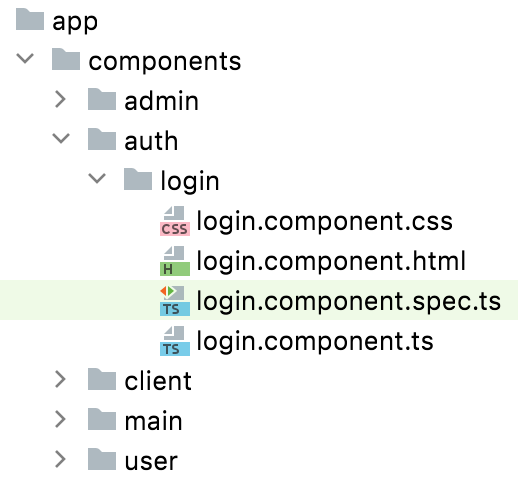
Za izradu klijentskog dela aplikacije izabran je radni okvir Angular, verzija 12. Angular je radni okvir za Javascript razvijen od strane Gugla i verzije 2 i novije, uključujući i verziju 12 koriste TypeScript za pisanje koda što ga čini dosta privlačnim programerima koji već znaju Javu. Pored toga, privlačnim ga čini i sama strutkura projekata, jer za razliku od drugnih Javascript radnih okvira, Angular ima striktiniju strukturu ali samim tim je i lakša za nove članove tima koji imaju iskustva sa Angularom kao i za programere koji imaju iskustva sa Javom, jer sturktura podseća na strukturu Java projekata.



*Slika 5.4.1: Struktura Angular projekta*

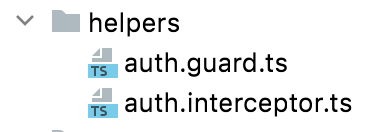
Kao što se na slici vidi, aplikacija se sastoji iz više foldera po kojima su logički razdvojeni delovi aplikacije i to na sledeći način:

***components -*** U ovom folderu nalaze se komponente koje se koriste za prikazivanje unutar web pregledača. Komponente su dodatno podeljene po folderima zavisno kom delu aplikacije pripadaju. Svaka komponenta sastoji se od četiri fajla, jedan *html* u kom je definisan izgled komponente, jedan *css* fajl gde se kompoeneta stilizuje, jedan *javascript* fajl gde je definisana logika i jedan *javascript* fajl koji služi za pisanje jediničnih testova za tu komponentu.



*Slika 5.4.2: Komponente u Angular projektu*

***helpers -*** U ovom folderu nalaze se klase koje služe kao “pomoć” drugim klasa, konkretno u ovom projketu tu se se nalaze klase koje se brinu o tome da li korisnik ima pravo pristupa određenom resursu.



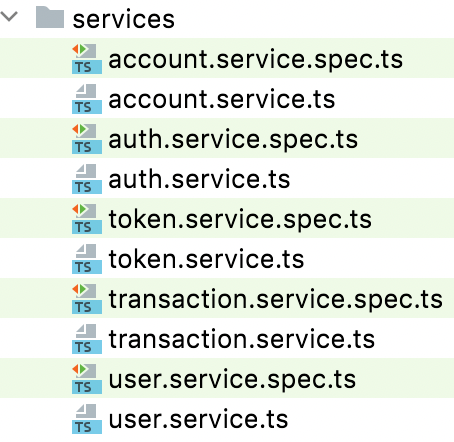
*Slika 5.4.3: Helpers klase u Angular projektu*

***models -*** Ovaj folder sadrži jednostavne klase koje služe za predstavljanje podataka, ako se uzme analogija iz Java sveta, ove klase su isto kao i POJO (Plain Old Java Object) klase.



*Slika 5.4.4: Models klase u Angular projektu*

***services -*** Ovaj folder sadrži klase koje služe za komunikaciju sa serverskim delom aplikacije, tačnije za pozivanje *REST* servisa kao i za mapiranje odgovora sa tih servisa u modele definisane u prethidnom folderu. Pored same klase servisa folder sadrzi i klase za jedinično testiranje.



*Slika 5.4.5: Service klase u Angular projektu*

Pored samog Angulara, projekat sadrži još dosta zavisnosti a one najbitnije su:

* Bootsrap
* rxjs
* express

## Primenjeni dizajn paterni

O dizajn paternima je već bilo reči u ovom radu, opisano je koji se dizajn paterni najčešće koriste u razvoju mikroservisnih aplikacija, u ovom delu biće fokus na sve dizajn paterne koji su iskorišćeni u aplikaciji.

***Api gateway -*** Ovaj dizajn patern iskorišćen je kako bi se napravilo jedno mesto koje će služiti kao ulaz za sv epozive prema svim mikroservisima a mikroservis ***proxy*** predstavlja implementaciju ovog paterna.

***Baza po servisu -*** Kao što smo videli svi mirkoservisi u aplikaciji koji čuvaju neke podatke imaju sopstvenu bazu podataka, samim tim iskorišćen je ovaj dizajn patern.

***Singleton -*** Singleton predstavlja klasu koja ima samo jednu instancu. Zrna (bean) u Springu su Singleton klase, samim tim ovaj patern je iskorišćen u svim mikroservisima u svim klasama koje koriste Spring anotacije: *@RestController, @Service, @Conmponent, @Configuration...*

***Observable -*** Ovaj patern iskoričćen je na klijentskoj strani, u Angular projektu, i to preko *rxjs* biblioteke. Patern služi da se tok (stream) obrađuje koristeći operacije koje se koriste za nizove.

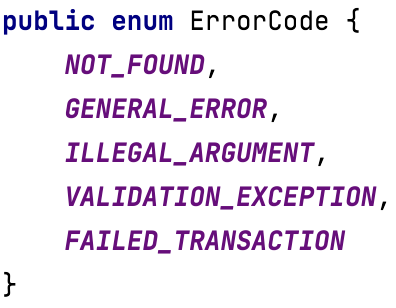
***Factory -*** Ovaj patern nije direknto iskorišćen ali se Spring jako oslanja na njega. Spring koristi ovaj patern kako bi napravio instance za sva zrna. Ovaj patern se još koristi i za učitavanje konfiguracije iz fajlova bez potrebe da se aplikacija ponovo kompajlira.

## Obrada izuzetaka

Izuzeci (exceptions) su neuobičajna stanja (greške), koja se javljaju prilikom izvršavanja programa. O samoj obradi izuzetaka u Javi ovde neće biti puno reči jer to nije tema rada, već će fokus biti na obradi izuzetaka u Springu i Angularu.

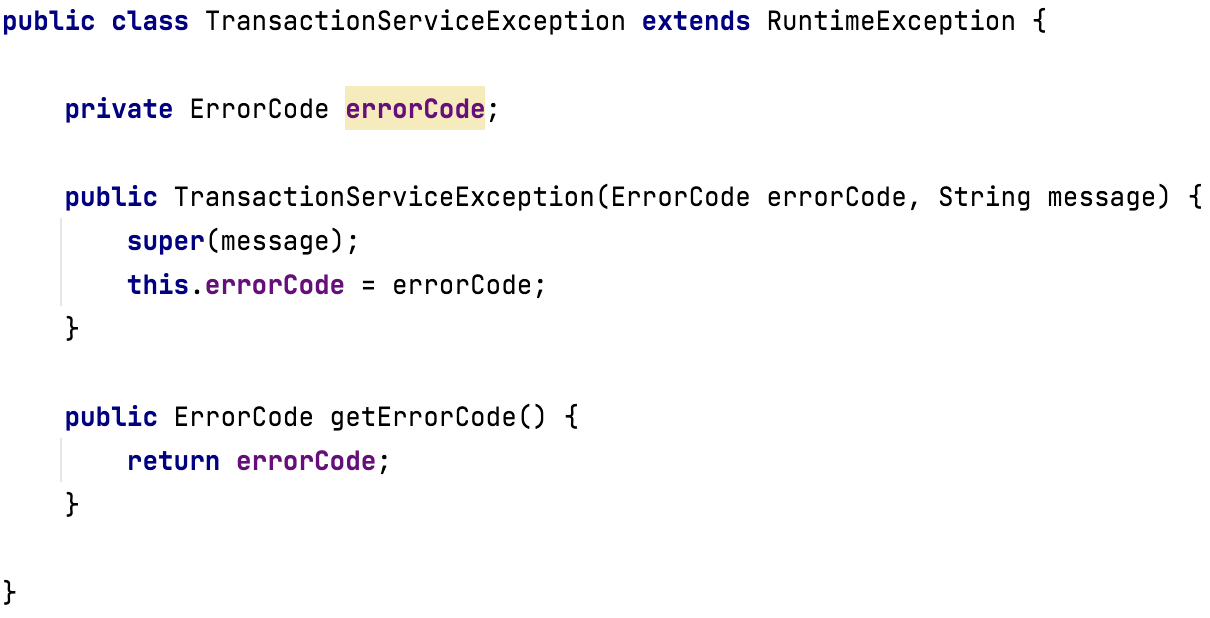
Pored standardnih izuzetaka koji se javljaju u Javi, na primer *NullPointerException* u Spring aplikacijama mogu da se jave i drugi izuzeci. Neki od njih su izuzeci koji se javljaju prilikom validacije parametara nekog zahteva, izuzeci koji se javljaju kada se pokuša upis u bazu nekog podatka koji ne sme da bude duplikat itd. Pored ovih izuzetaka, neke izuzeci su specifični za svaku aplikaciju to jest svaki servis i njih “baca” sam programer po potrebi.

Svaki mikroservis poseduje jednu klasu koja najsleđuje klasu *RuntimeException* i koja je proširuje za još jednu informaciju pored poruke greške a ta informacija je kod greške. Dobra praksa je da se koriste kodovi za greške pored same poruke jer klijent koji koristi mikroservis može na osnovu koda koji je konstantan da prikaže poruku koja njemu najviše odgovara (na primer na različitim jezicima).



*Slika 5.5.1: Kodovi grešaka u servisu transaction-service*

Na prethodnoj slici dati su kodovi grešaka koje *transaction* servis može da vrati. Tu su između ostalog kod da transakcija nije uspela ili da neki entitet nije pronađen.



*Slika 5.5.2: TransactionServiceException klasa*

Za sve greške koje programer sam javlja, koristi se klasa sa prethodne slike. Klasa nasleđuje *RuntimeException* kako se ne bi svuda postavljao ***try/catch*** blok već se na jednom mestu obrađuju sve greške. To mesto je klasa koja nasleđuje Spring klasu *ResponseEntityExceptionHandler* i koja je anotirana Spring anotacijom *@ControllerAdvice*.

Ova klasa služi kao presretač svih grešaka u aplikaciji i u zavisnosti od toga koja je greška u pitanju generiše odgovor koji se vraća iz REST kontrolera.

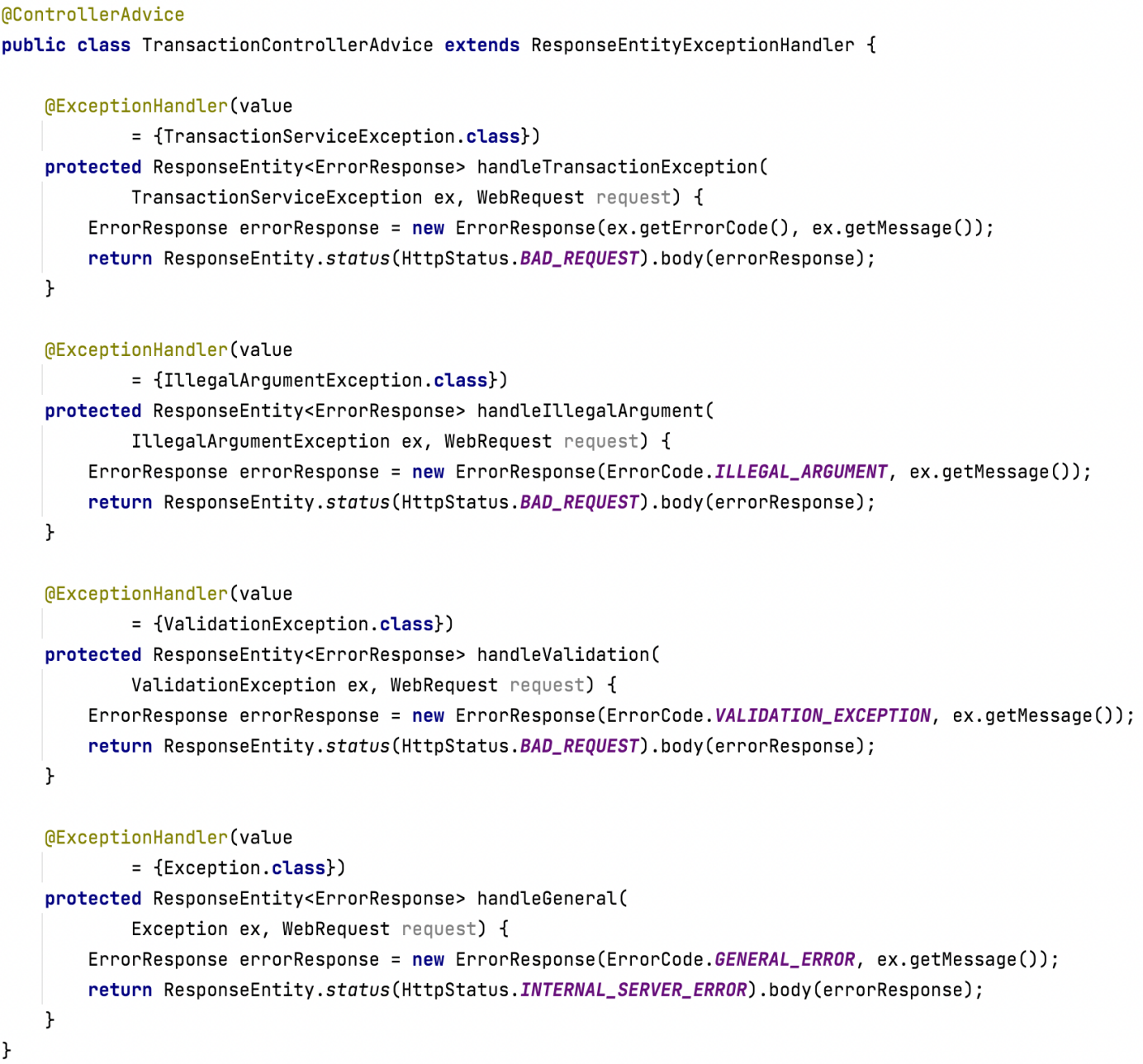
Na slici isopd dat je izgled te klase unutar *transaction* servisa. Kao što se vidi klasa posebno prepoznaje više vrsta izuzetaka.

Prvi od njih je ***TransactionServiceException*** izuzetak o kome je bilo reči u prethodnom pausu, kada se taj izuzetak dogodi, klijentu se vraća odgovor koji sadrži kod i poruku greške koje je programer postavio kada je kreirao izuzetak.

Drugi izuzetak je ***IlegalArgumentException***, on se dešava kada se nekom REST pozivu proslede argumenti koje taj poziv ne očekuje.

Treći izuzetak je ***ValicationException*** koji se dešava kada se u telu zahteva nalaze podaci koji zahtevaju validaciju a ta validacija nije uspešna, na primer očekuje se polje koje za vrednost treba da ima email ali tu je neki običan string.

Poslednji izuzetak je ***Exception*** a to je bilo koji drugi izuzetak jer svi izuzeci nasleđuju ovaj. Tada sistem nije tačno siguran šta se desilo pa klijentu vraća kod da se dogodila neka generička greška i poruku iz te greške.



*Slika 5.5.3: Klasa za obradu izuzetaka*

## Swagger

Swagger je softver za automatsko generisanje dokumentacije za REST API-je koji kao odgovor daju JSON dokumente. Swagger omogućuje da se na jednom mestu vide svi dostupni API pozivi kao i da se vidi koji poziv očekuje koje parametre, koji su opcioni a koji ne kao i koji JSON objekat se očekuje kao odgovor za taj poziv. U Spring projektima, kada uključimo swagger kao zavisnost, dobijamo automatsko generisanje dokumentacije za sve metode u svim REST kontrolerima.

Screenshot 2021-08-31 at 16.30.23

*Slika 5.6.1. Swagger zavisnost*

Nije dovoljno samo podesiti zavisnost, potrebno je izvršiti i konfiguraciju swagger-a. To se radi tako što se kreira klasa koja je anotirana Spring anotacijom *@Configuration* a zatim se u toj klasi izvrši konfiguracija swagger-a kao na slici:



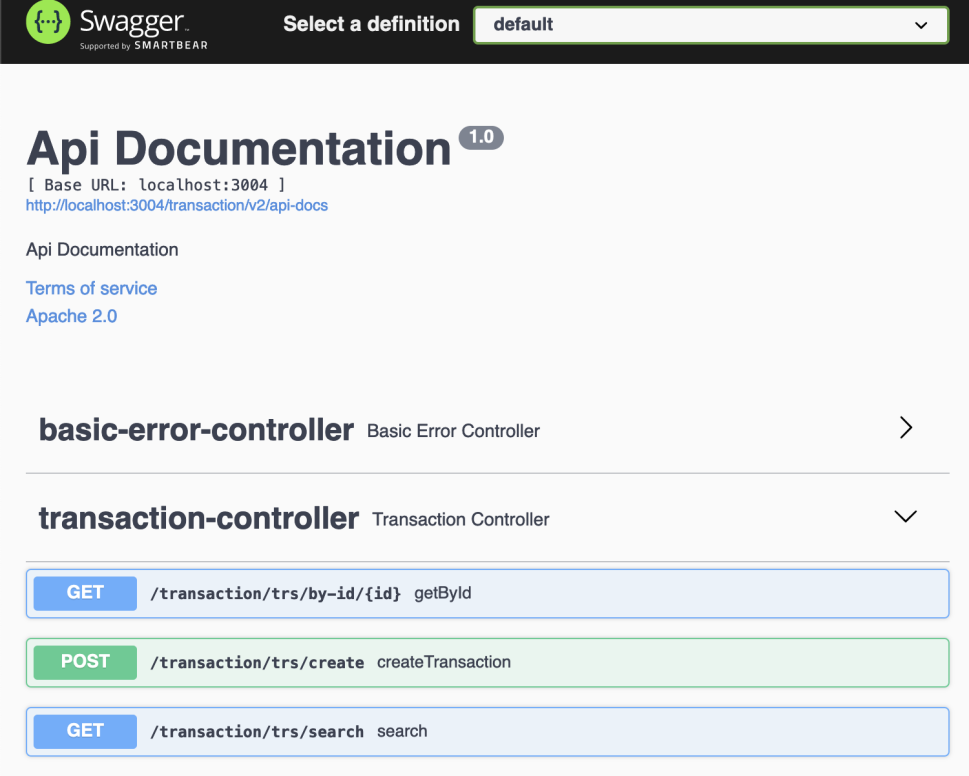
*Slika 5.6.2. Swagger konfiguracija*

Poslednji korak je podesiti bezbednost za url na kom je swagger dostupan, jer ako se koristi Spring Security, taj url će biti zaključan.

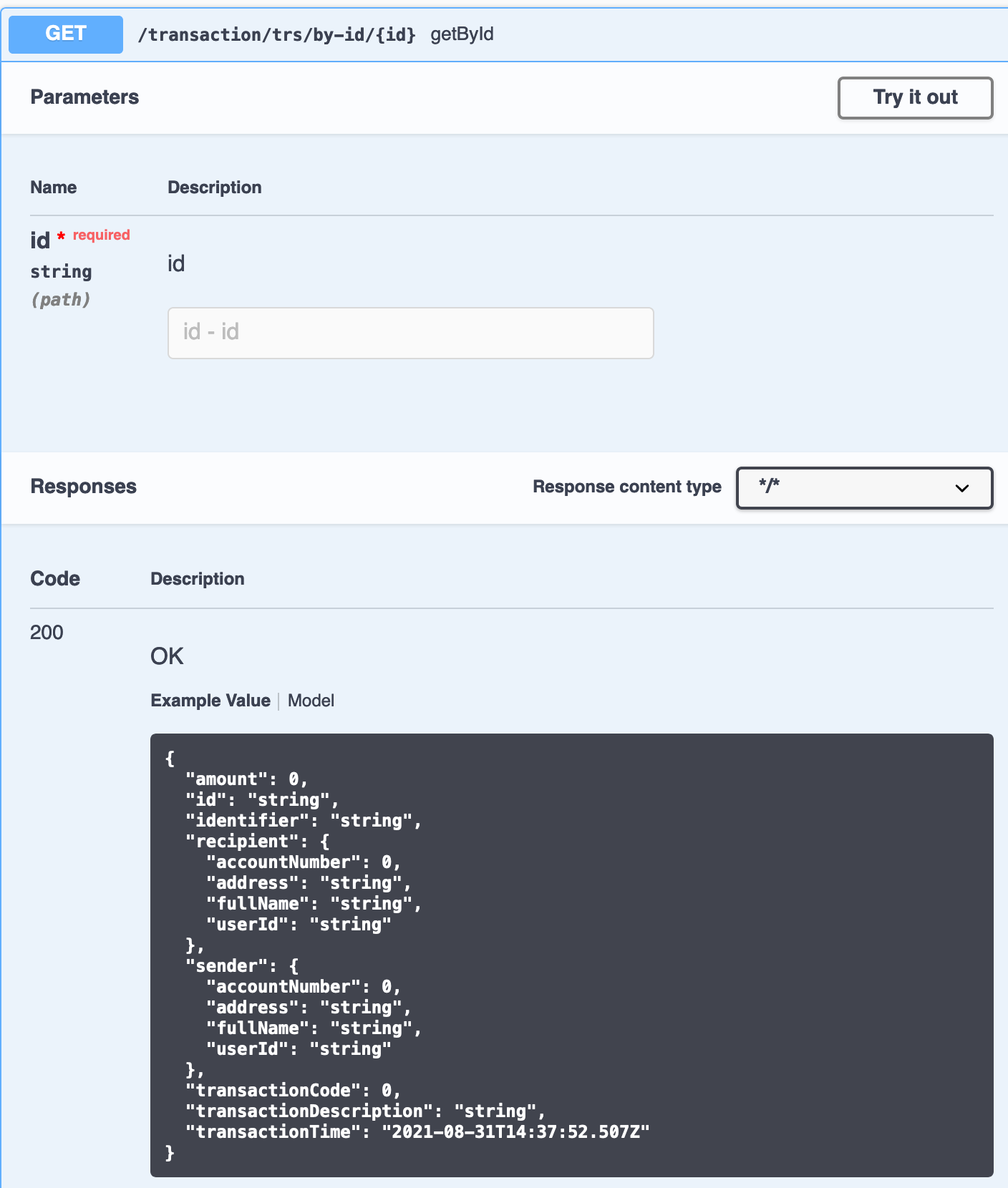


*Slika 5.6.3. Swagger bezbedonosna konfiguracija*

Nakon ovih koraka, Swagger dokumentacija će biti dostupna na url: {urlServisa}/swagger-ui (na primer [http://localhost:3004/transaction/swagger-ui )](http://localhost:3004/transaction/swagger-ui))



*Slika 5.6.4. Swagger dokumentacija za transaction servis*



*Slika 5.6.5. Swagger dokumentacija za jedan poziv*

## Jedinično testiranje

Mnogi programeri posmatraju testiranje kao dokaz da njihovi programi ispravno rade. Ali, zapravo, programi se testiraju da bi se dokazalo postojanje grešaka. Pošto je cilj da se greška otkrije, test se smatra uspešnim samo ako se greška otkrije ili ako dođe do otkaza u toku testiranja.

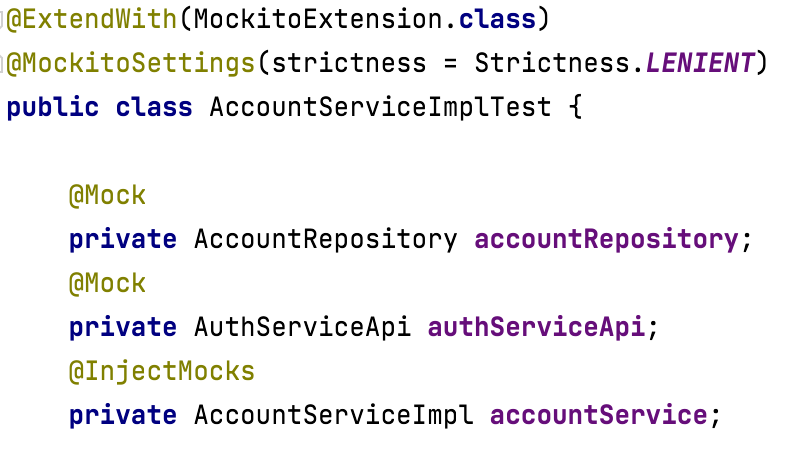
Prvo se zasebno testira svaka programska komponenta, nezavisno od ostalih delova sistema. Takvo testiranje modula (**jedinično testiranje,unit test**) proverava da li pojedinačne komponente ispravno funkcionišu sa svim očekivanim tipovima ulaza, u skladu sa dizajnom komponente. Jedinično testiranje treba vršiti u kontrolisanom okruženju tako da tim za testiranje može komponenti koja se testira da predaje unapred definisan skup podataka, i da posmatra izlazne akcijei rezultate. Takođe, tim za testiranje proverava unutrašnju strukturu podataka, logiku i granične  
uslove za ulazne i izlazne podatke.

Java programi se u najčešće testiraju koristeći bibloteku koja se zove ***Junit*** a kako se ne bi kreirale sve zavisnosti koje neka klasa poseduje, koristi se biblioteka za lažiranje (**mock**) koja se zove ***Mockito***. Spring radni okvir pruža mogućnost da se kroz jednu zavisnost u projektu, dobiju obe navedene biblioteke.

Screenshot 2021-09-03 at 10.38.54

*Slika 5.7.1. Spring zavisnost za testiranje*

Za klasu koju treba testirati se pravi klasa koja predstavlja test. S obzirom da se radi o jediničnom testiranju, sve zavisnosti treba lažirati, praktično sve zavisnosti koje su u klasu uvežene anotacijom ***@Autowired*** se uvoze i u test klasu ali koristeći anotaciju ***@Mock.***



*Slika 5.7.2. Test klasa za jedan servis*

Na prethodnoj slici dat je izgled jedne klase za jedan servis u projektu *account-service.* Sve zavisnosti su lažirane koristeći ***@Mock*** anotaciju dok se klasa koja se testira anotira sa ***@InjectMocks.*** S obzirom da klasa koristi lažiranje zavisnosti, cela klasa je anotirana anotacijom koja govori da se klasa proširuje dodatnom klasom iz ***Mockito*** biblioteke.

Kod jediničnog testiranja potrebno je testirati i usepšne i neuspešne slučajeve neke metode. Na slici ispod dat je primer testa za kreiranje računa koji predstavlja uspešan slučaj a na sledećoj slici dat je primer testa za čitanje jednog računa po id-ju koji predstavlja neuspešan slučaj.



*Slika 5.7.3. Test za uspešan slučaj*

Svi pozivi ka klasama koje su zavisnosit u klasi koje se testiraju, se lažiraju to jest putem ***Mockito*** biblioteke se kaže šta će se desiti kada dodje do izvršenja tog poziva. Na taj način se mogu testirati i ivični slučajevi (edge cases) koje je u regularnom izvršavanju programa teško izazvati.



*Slika 5.7.4. Test za neuspešan slučaj*

## Integraciono testiranje

Kada se završi jedinično testiranje skupa komponenti, sledi provera da li su interfejsi između komponenti pravilno definisani i realizovani. **Integraciono testiranje** je postupak provere da li sistemske komponente sarađuju kao što je opisano u specifikacijama dizajna sistema i programa.