Korištenje razvojnog okvira Spring Boot za implementaciju servisa SOAP i REST

Danzante, Andrea

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Andrea Danzante

Korištenje razvojnog okvira Spring Boot za implementaciju servisa SOAP i REST

DIPLOMSKI RAD

Varaždin, 2019.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Andrea Danzante

Matični broj: 46341/17–R

Studij: Informacijsko i programsko inženjerstvo

Korištenje razvojnog okvira Spring Boot za implementaciju servisa SOAP i REST

DIPLOMSKI RAD

Mentor:

Prof. dr. sc. Danijel Radošević

Varaždin, veljača 2019.

Andrea Danzante

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

Tema ovog diplomskog rada je implementacija *SOAP* i *REST* web servisa i njihova primjena u *Spring Boot* aplikaciji. Cilj ovog rada je detaljnije opisati samu implementaciju navedenih vrsta web servisa i njihove sigurnosti kroz implementaciju dviju aplikacija. Aplikacije implementirane u ovom radu su: poslužiteljska aplikacija koja omogućava sve web servise te klijentska aplikacija s intuitivnim korisničkim sučeljem za pristupanje i rad s podacima. Sva poslovna logika sustava nalazi se u poslužiteljskoj aplikaciji koja ujedno ima pristup bazi podataka, dok se sve operacije i pregledavanje podataka odvijaju preko klijentske aplikacije. Klijentska aplikacija je web mjesto s intuitivnim web stranicama koje korisnik može koristiti za praćenje vlastitih transakcija, njihov kreiranje, uređivanje ili brisanje. Kroz praktični dio rada naglasak je na samom *Spring Boot-u* i *Java-i* te poznavanje ostalih alata i programske podrške nije od velike važnosti. Sama struktura rada temeljena je na osnovnom uvodu o web servisima te opisu istih u kontekstu *Spring Boot*-a, a na samom kraju opisan je i primjer korištenja web servisa u pogledu intuitivne aplikacije.

Ključne riječi: SpringBoot; Java; SOAP; REST; Thymleaf; Facade; Service; Repository; RestRepositories;.

Sadržaj

1. Uvod

Suvremeno računalstvo napreduje iz dana u dan te traži nova rješenja kako bi ICT tehnologija bila što jednostavnija, primjenjiva u svim poslovnim okruženjima te da poboljša i olakša obavljanje poslova ljudima. Pojava novih programskih rješenja dovela je do velikog broja programskih rješenja pisanima u različitim programskim jezicima koja se izvode u različitim okruženjima. Različitosti sustava dovode do problema interoperabilnosti, nemogućnosti razmjene podataka i velikih troškova održavanja.

Servisno orijentirana arhitektura (*SOA – engl. Service-Oriented Architecture*) pristup je koji je dao odgovor na prethodno navedene probleme primjenom servisa dostupnih preko interneta i weba. Kako navodi Ed Ort u svom djelu "*Service-Oriented Architecture and Web Services: Concepts, Technologies, and Tools*" (2005), primjenom SOA strategije razvoja sustava omogućena je: ponovna iskoristivost (*engl. Reusability*), interoperabilnost (*engl. Interoperability*), skalabilnost (*engl. Scalability*), fleksibilnost (*engl. Flexibility*) i učinkovitost troškova (*engl. Cost Efficiency*). Primjena servisa omogućuje povezivanje različitih sustava primjenom dogovorenog formata poruka, smanjuje troškove održavanja jer je sada sva programska logika na jednom mjestu, servis može koristiti više korisnika odjednom, a smanjenje troškova realizirano je kroz održavanje jedne aplikacije na poslužitelju umjesto aplikacije kod svih klijenata.

Prethodno spomenuta SOA arhitektura započela je korištenje servisa te kreiranje protokola i novih arhitektura. Ovaj diplomski rad sadrži kratak uvid u dvije vrste web servisa te njihovu implementaciju u razvojnom okruženju (*engl. Framework*) *Spring Boot.* U sljedećim poglavljima prezentirani su i pojašnjeni korišteni programski alati, razvojno okruženje *Spring Boot*, konfiguracija istih te na kraju sama implementacija servisa i korisničkog sučelja.

2. Web servisi

Nastanaka web servisa usko je vezan s pozivom udaljenih metoda ili kraće RPC (*engl. Remote Procedure Call*) mehanizmom obrade. DCE/RPC (*engl. Distributed Computing Environment*) predstavlja poziv udaljene metode koji skriva strukturu i sadržaj poruka koje se prenose u binarnom zapisu. Udaljene metode pozivane su preko Opisnog jezika sučelja (*engl Interface Definition Language*) koji definira ulazne parametre i traženi rezultat. Dave Winer iz kompanije *UserLand Software* unaprijedio je razmjenu poruka u RPC-u primjenom XML-a (*engl. eXtensible Markup Language*) koji koristi oznake i ugrađene tipove podataka programskog jezika C uz dodatne *boolean* i *datetime* tipove podataka. Primjena XML-a umjesto binarno zapisanih podataka uvelike olakšava obradu, standardizirani je format dokumenta te je smanjuje broj biblioteka za obradu. Na slikama ispod prikazan je IDL poziv te XML-RPC. (Kalim, 2013)

```
/* echo.idl */
[uuid(2d6ead46-05e3-11ca-7dd1-426909beabcd), version(1.0)]
interface echo {
    const long int ECHO_SIZE = 512;
    void echo(
        [in]            handle_t h,
        [in,  string]   idl_char from_client[ ],
        [out, string]   idl_char from_server[ECHO_SIZE]
    );
}
```

Slika 1. Primjer IDL datoteke za poziv udaljene metode (preuzeto iz: (Kalim, 2013))

```
<?xml version="1.0">
<methodCall>
   <methodName>fib<methodName>
   <params>
     <param><value><i4>11</i4></value></param>
   </params>
</methodCall>
```

Slika 2. Primjer XML poziva udaljene metode (preuzeto iz: (Kalim, 2013))

Nastanak XML-RPC-a smatra se početkom web servisa kakvih danas poznajemo što je vidljivo iz samog SOAP (*engl. Simple Object Access Protocol*) koji koristi XML format za razmjenu poruka i načina komunikacije između servisa i korisnika (klijenta).

3. Korišteni alati i programska podrška

Programski primjera ovog diplomskog rada čini jednostavna aplikacija za vođenje financijskih podataka korisnika. Osnovne karakteristike vezane uz ovo poglavlje sljedeće su: korišten je *Java* programski jezik (verzija 8 ažuriranja 211), *Spring Boot* razvojni okvir, *MySQL* baza podataka na *XAMPP* poslužitelju s *Apache HTTP (engl. HyperText Transfer Protocol)* poslužiteljem i *MariaDB* bazom podataka, *Maven* alat za izgradnju programskog koda te *Tomcat* aplikacijski poslužitelj.

3.1. *Spring Boot*

Spring Framework razvojno je okruženje otvorenog koda (*engl. Open source*) kreirano 2013. godine od strane Rod Johnsona kao poboljšanje dotad korištenog *Java Enterprise Edition (JEE)* standarda za izradu robusnih web aplikacija. Za razliku od JEE, *Spring* je pridobio naklonost korisnika dvjema karakteristikama: inverzija kontrole (*engl. Inversion of Control*) odnosno sam *framework* kontrolira izvršavanje programskog koda te injektiranje objekata o kojima ovisi rad koda (*engl. Dependency Injection*). Nastanak *Spring*-a, osim što je olakšao kreiranje robusnih *Java* aplikacija, i dalje je zahtijevao veliku količinu konfiguracijskih klasa i napora prilikom same inicijalizacije projekata. (Cornelißen, Piefel, & Sparkowsky, 2018)

Daljnji rad na samom *Spring* razvojnom okviru preuzela je kompanija *Pivotal Software* koja je poboljšanja na *Spring-u* nastavila isporučivati u obliku modula to jest nezavisnih projekata baziranih na samom *Spring*-u. *Spring Boot* jedan je od modula koji je nastao kako bi uz čim manje truda i konfiguriranja projekt bio inicijaliziran i spreman za daljnji razvoj. Neke od karakteristika *Spring Boot*-a su: unaprijed pripremljene funkcionalnosti (*eng. Out of the box functionalities*), nema generiranja klasa i koda već se koriste unaprijed definirane biblioteke te set nefunkcionalnih alata i klasa u pogledu konfiguracije i pokretanja servera, sigurnosti, metrike te ostalih pomoćnih poslova. (Cornelißen i ostali, 2018; Webb i ostali, 2018)

3.2. *Maven*

Apache Maven alat je za izgradnju i upravljanje *Java* aplikacijama, a prednosti istoga su sljedeće (Apache, 2019a):

- Olakšana izgradnja projekta tako što skriva detalje i mehanizme izgradnje i upravljanja projektom

- primjena Objektnog modela projekta (*engl. Project Object Model - POM*) koji definira sve potrebne zavisnosti projekta i način izgradnje
- pohrana dodatnih informacija o samom projektu
- podržava i provodi najbolje prakse definirajući zasebne dijelove projekta s testovima, pojednostavljuje strukturiranje projekta

3.3. *Tomcat*

Apache Tomcat aplikacijski je poslužitelj otvorenog koda namijenjen izvršavanju *Java* aplikacija. Server održava i razvija *Apache Foundation,* a isti implementira *Java Servlet*-e, *JavaServer Pages*, *Java Expression Language* i *Java WebSocket*-e koji su minimalni zahtjevi za izvršavanje *Java* web aplikacija. *Tomcat* zahtjeva konfiguracije kako bi se aplikacije mogle pokretati na istom, ali upravo korištenjem *Spring Boot*-a u samom projektu su definirane sve konfiguracije. (Apache, 2019b)

3.4. *XAMPP*

XAMPP Apache još jedna je aplikacija otvorenog koda čija je namjena pružiti mogućnost uspostavljanja stvarnih web poslužitelja za isporuku neovisnih o platformi (*engl. Deployment*). Karakteristike koje *XAMPP* navedene su u samom akronimu imena (ApacheFriends, 2019):

- X – implementiran za više platformi (*engl. cross-platform*)
- A - *Apache* HTTP poslužitelj
- M – pruža podršku za rad s *MariaDB* (odnosno *MySQL*) bazom podataka
- P – pruža podršku za PHP skriptni programski jezik
- P – podrška za *Perl* programski jezik

U ovom diplomskom radu *XAMPP* će se koristiti za pokretanje *Apache HTTP* servera na kojem će se nalaziti *MySQL* baza podataka. *XAMPP* nudi podršku za rad s bazom podataka kroz *phpMyAdmin* sustav za upravljanje bazom podataka, a s obzirom da sam *SpringBoot* pruža podršku za rad s bazom podataka, znanje *MySQL*-a skoro pa i nije potrebno za razumijevanje programskog primjera ovog diplomskog rada.

3.5. *IntelliJ*

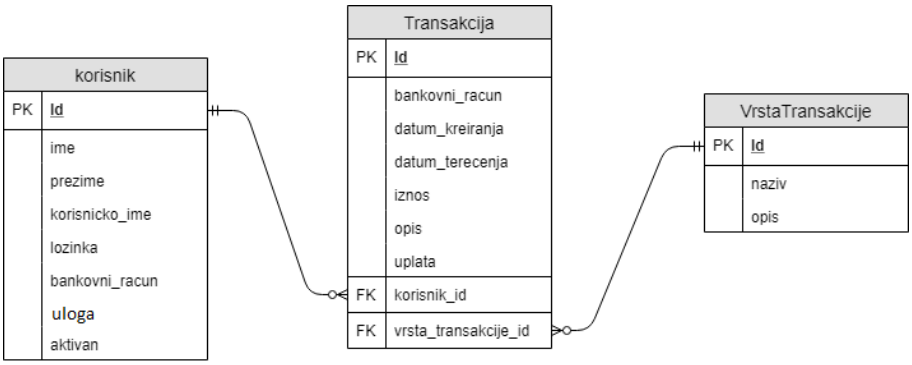
Poduzeće *JetBrains* specijalizirano je za razvoj integriranih razvojnih okruženja (*engl. Integrated Development Environment*) odnosno editora programskog koda s popratnim

funkcionalnostima za izgradnju, verzioniranje, pokretanje i ostale funkcionalnosti potrebne za rad na projektima. *IntelliJ* razvojno okruženje kreirano je primarno za razvoj *Java* aplikacija, a dostupno je u besplatnoj verziji podržanoj od strane *Apache* 2.0 licence koja podrazumijeva da je riječ o alatu otvorenog koda. (JetBrains, 2019)

Studenti FOI-a imaju pravo na licencu za *ultimate* verziju *IntelliJ* razvojnog okruženja koje podržava sve potrebne funkcionalnosti za pisanje diplomskog rada: editor *Java* koda, izgradnja aplikacije korištenjem *Maven*-a te dodatne funkcionalnosti vezane uz pisanje aplikacije u *Spring*-u odnosno *Spring Boot*-u. Dodatne mogućnosti *IntelliJ*-a su: analiza i preporuke za dovršavanje programskog koda (*engl. Code completion*), upozorenja na moguće logičke greške, podrška za *debug*-iranje te ostale funkcionalnosti koje olakšavaju pisanje *Java* koda. (JetBrains, 2019)

4. *Spring Boot* aplikacija

Tema programskog primjera diplomskog rada je sustav za vođenje financija. Ideja i cilja samog sustava je omogućiti korisniku unos transakcija (ulaznih i izlaznih) kako bi na jednom mjestu imao evidenciju osobnog novčanog tijeka.



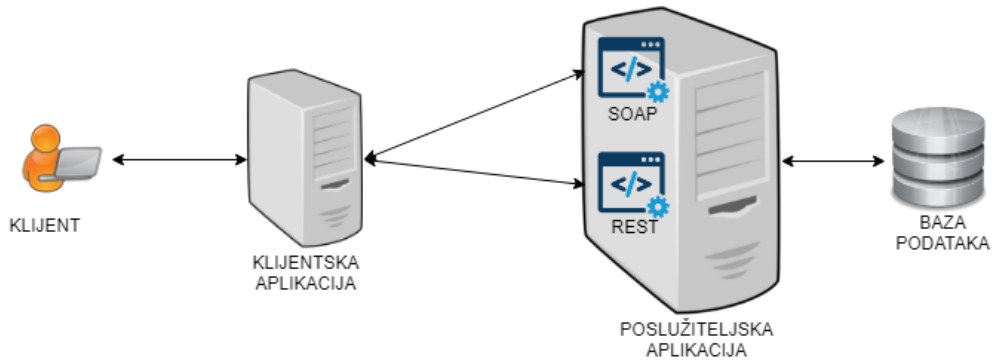
Slika 3. ERA dijagram korištene baze podataka (samstalna izrada)[1]

Slika prikazana iznad opisuje ERA model sustava za rad s transakcijama. Baza podataka korištena u sustavu sačinjena je od tri entiteta o kojim se bilježe podaci: korisnik, vrsta transakcije te transakcije. Svaki sustav zaštićen autorizacijom korisnika sadrži tablicu korisnika te je u tu svrhu kreirana ista. Korisnik sustava može obnašati jednu od dvije uloge dvije uloge odnosno biti administrator ili običan korisnik. Tablica sadrži podatke o korisnicima poput: identifikatora, imena, prezimena, jedinstvenog korisničkog imena, bankovni računa korisnika s obzirom da je riječ o praćenju transakcija te lozinka za prijavu. Korisnici dobivaju pristup sustavu registracijom, a svaki registrirani te potom prijavljeni korisnik ima pravo kreirati, pregledavati te uređivati transakcije. Običan korisnik može svoje podatke unijeti u navedenu tablicu unijeti samo registracijom ili ažuriranjem korisničkih podataka na profilu, dok administrator ima pravo promijeniti ulogu određenog korisnika.

Osnovni entitet sustava je transakcija u kojoj korisnik definira sve potrebne podatke za praćenje vlastitog novčanog tijeka. Prilikom kreiranja transakcije, korisnik zadaje datum terećenja odnosno datum kad je transakcija službeno izvršena, unosi bankovni račun na koji je izvršena isplata ili s kojeg je došla uplata, definira je li riječ o uplati (stupac uplata je *true* ili 1 odnosno *false* ili 0) te uz iznos transakcije i opis iste definira vrstu transakcije. „Zastavica“ uplata u transakciji definira je li riječ u pritoku novčanih sredstava ili isplati s korisničkog računa. Stupci „datum_kreiranja“ i „korisnik_id“ zadaju se automatski i to s podacima o trenutnom vremenu kreiranja i korisniku koji je poslao zahtjev na poslužiteljsku aplikaciju.

[1] Kreirano draw.io online alatom – www.draw.io

Posljednji stupac transakcije koji je ujedno i posljednji entitet o kojem se zapisuju podaci, vrsta transakcije. Administrator sustava ima uz prava pregledavanja još i pravo kreirati, uređivati te brisati vrste transakcije. Svaka vrsta transakcije opisana je identifikatorom, nazivom te opisom što znači svaka vrsta.



Slika 4. Osnovna arhitektura aplikacija (samstalna izrada)[2]

Sama arhitektura sustava vrlo je jednostavna, a sastoji se od dvije manje aplikacije: klijenta i poslužitelja. Poslužiteljska aplikacija srž je sustava iz razloga što ista sadrži poslovnu logiku i ima pristup bazi podataka. Klijentska aplikacija osnovna je implementacija klijenta za *SOAP* i *REST* (*engl. Representational state transfer*) web servise na poslužitelju, a ista dohvaća podatke te ih prezentira na korisniku razumljiv način. Obije aplikacije zaštićene su od pristupa neautoriziranog korisnika i to primjenom dva načina prijave odnosno autorizacije korisnika. Prilikom prijave u klijentsku aplikaciju, ovisno njegovoj ulozi, korisnik dobiva prava pristupa određenim stranicama te na taj način su svi podaci prikazani u korisničkoj aplikaciji zaštićeni i dostupni samo prijavljenom korisniku. Poslužiteljska aplikacija je samostalna te joj je moguće pristupiti pomoću alata za kreiranje zahtjeva za *SOAP* i *REST* ili čak preglednikom stoga svaka metoda mora biti zaštićena. Zaštita na poslužiteljskoj strani realizirana je autorizacijom korisnika prilikom poziva metode, na navedeni način poslužitelj ne sadrži određeno stanje u pogledu sesije i praćenju podataka korisnika već se prilikom poziva metode u *HTTP* zahtjevu šalju korisničko ime i lozinka.

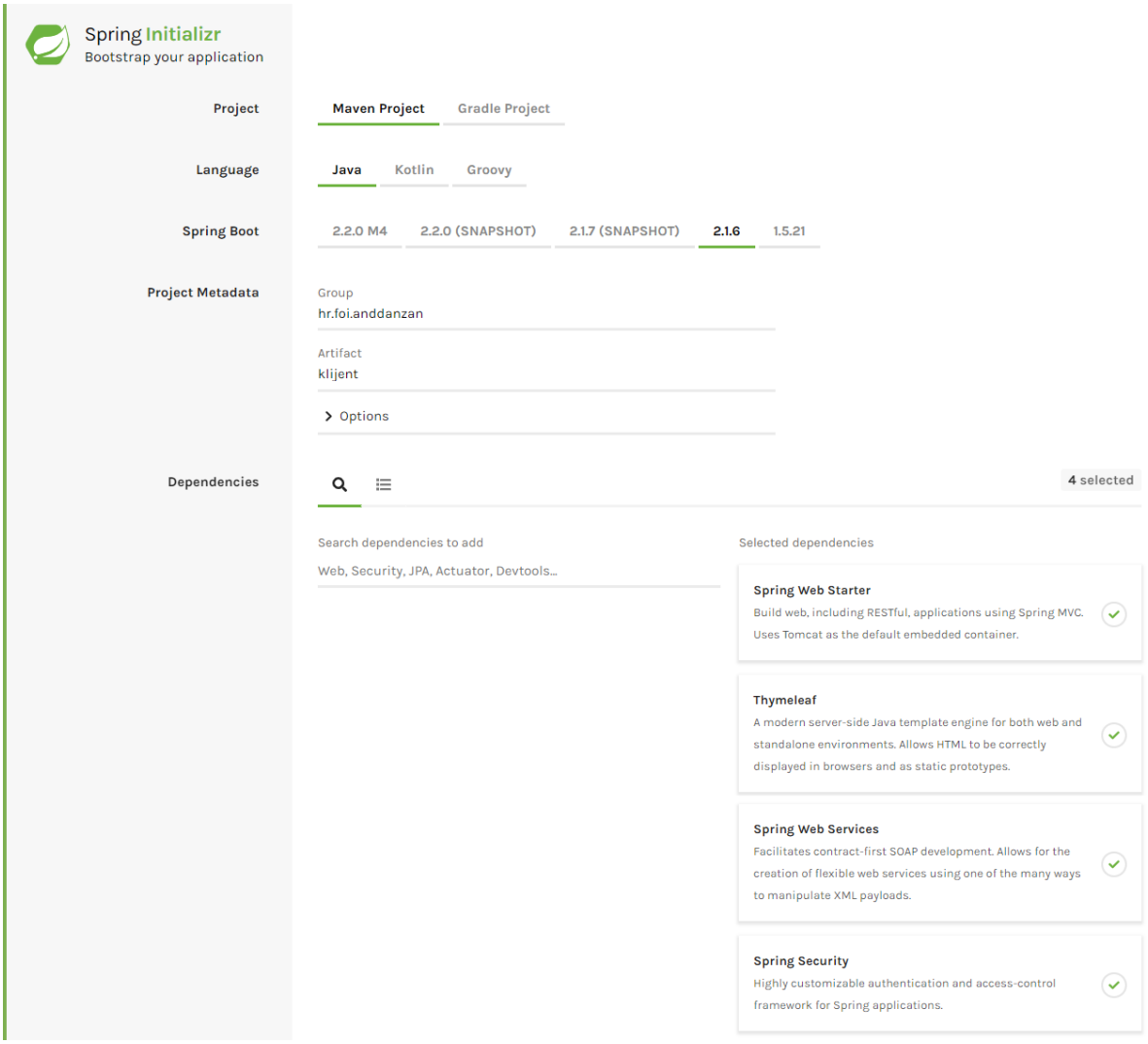
[2] Kreirano draw.io online alatom – www.draw.io

5. *Spring Boot* inicijalizacija projekata

Implementacija *SOAP* i *REST* servisa nalazi se u samoj aplikaciji poslužitelja, ali kako bi korisnik mogao na jednostavan način koristiti sustav potrebna je i klijentska aplikacija. Obije aplikacije kreirane su pomoću *Spring Boot* razvojnog okvira, a inicijalizirane putem online servisa za inicijalizaciju *SpringInitializr* koji nudi tvrtka *Pivotal Software.* U nastavku poglavlja opisane su inicijalizacije obiju aplikacija pomoću spomenutog servisa.

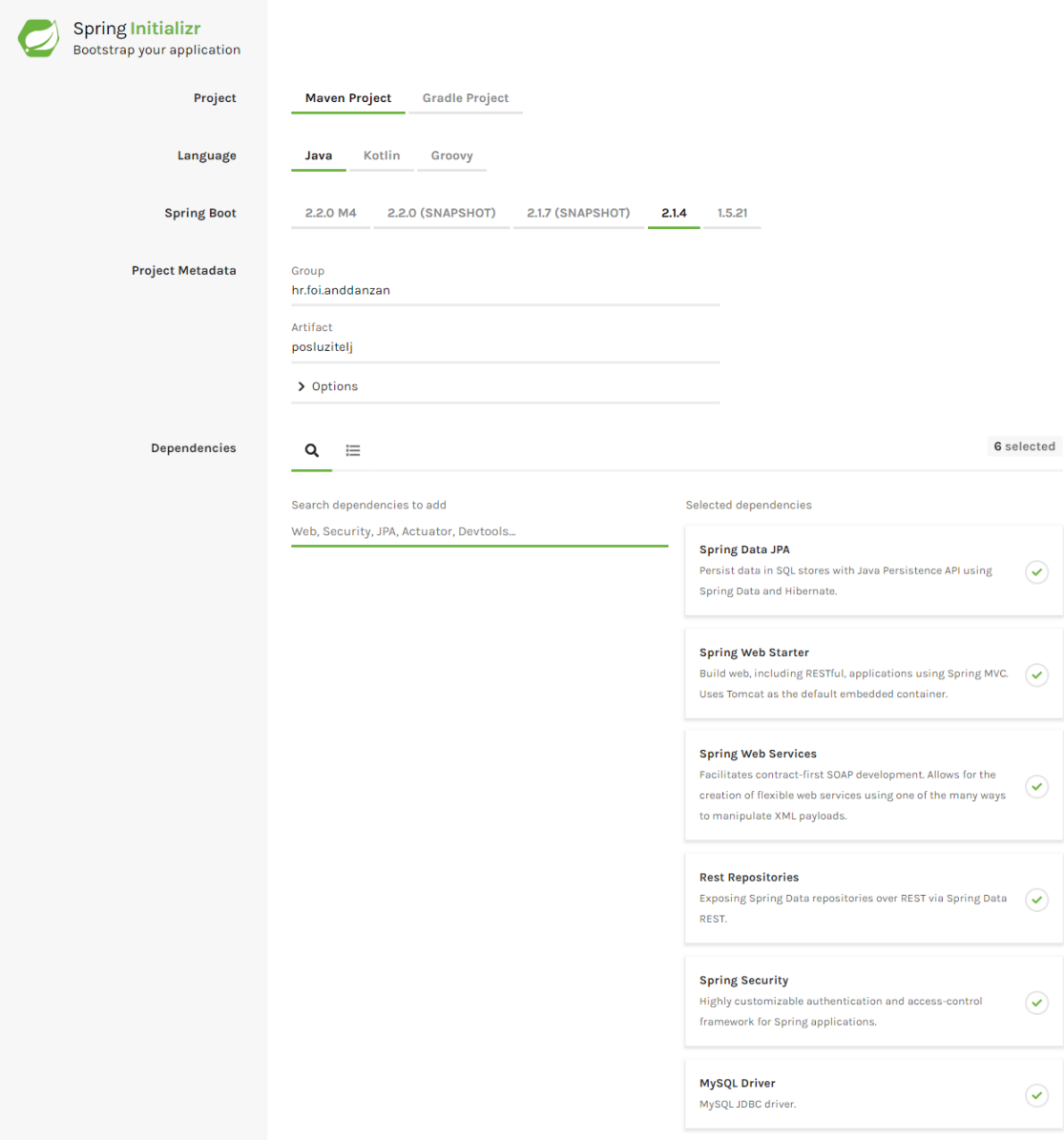
5.1. Postavke inicijalizacije projekta

Prilikom kreiranja Spring Boot projekta, odabir zavisnosti definira samu namjenu i mogućnosti projekta. Jednostavnost inicijalizacije Spring Boot projekta u najvećoj mjeri manifestira se odabirom *starter*-a točnije prikladnim opisnikom zavisnosti. (Webb i ostali, 2018)



Slika 5. Inicijalizacija klijentske aplikacije korištenjem online servisa Spring Initializr (PivotalSoftware, 2019f)

Slika iznad odlomka prikazuje početne postavke klijentske aplikacije inicijalizirane u *Spring Boot* verziji 2.1.5. (iz razloga što je to najnovija stabilna verzija dostupna na dan inicijalizacije) te *Java* programskom jeziku. Osnovni podaci koji se moraju zadati su: nazivi grupa kojoj pripada programski sadržaj (*engl. Group*) te naziv samog artefakta (*engl. Artefact*) koji zajedno čine strukturu paketa projekta. Najznačajnije postavke projekta odabrane su pod opcijom zavisnosti (*engl. Dependency*), a iste definiraju sam projekt i njegovu namjenu. U prikazanoj klijentskoj aplikaciji odabrane su sljedeće zavisnosti koje su detaljnije opisane u nastavku poglavlja: *Spring Web Starter* (zavisnost za izgradnju web aplikacije), *Thymleaf* (zavisnost *Java XML/XHTML/HTML5* jezika predložaka korišten za izgradnju grafičkih sučelja), *Spring Web Services* (zavisnost potrebna za izgradnju *SOAP* klijenta) te *Spring Security* (zavisnost za zaštitu aplikacije).



Slika 6. Inicijalizacija poslužiteljske aplikacije korištenjem online servisa Spring Initializr (PivotalSoftware, 2019g)

Drugu aplikaciju sustava čini poslužiteljska aplikacija koja sadrži svu polovnu logiku i pristup podacima u bazi. Navedena aplikacija kreirana je na isti način kao i klijentska, samo s različitim zavisnostima. Na dan kreiranja najnovija stabilna verzija *Spring Boot*-a bila verzija 2.1.4. razlika inicijalizacije aplikacije poslužitelja također je u samom imenu artefakta koja je u ovom slučaju „poslužitelj“. Poslužiteljska aplikacija također koristi zavisnosti: *Spring Web Starter, Spring Security* te *Spring Web Services*, ali kako bi aplikacija imala dodatne mogućnosti pristupa bazi podataka te kreiranju web servisa potrebno je dodati još zavisnosti: *Spring Data JPA* (*zavisnost JPA* iz *Jave* poboljšana primjenom *Spring* razvojnog okvira), *MySQL Driver* (upravljački program za pristup bazi podataka temeljenoj na *MySQL* upitnom jeziku) te *Rest Repositories* (zavisnost za objelodanjivanje strukture baze podataka preko *REST* servisa).

5.2. Korištene *Spring Boot* zavisnosti

5.2.1. *Spring Web Starter* zavisnost

Category/License		Group / Artifact
Apache 2.0		org.hibernate.validator » hibernate-validator
Web Framework Apache 2.0		org.springframework » spring-web
Web Framework Apache 2.0		org.springframework » spring-webmvc
Apache 2.0		org.springframework.boot » spring-boot-starter
Apache 2.0		org.springframework.boot » spring-boot-starter-json
Apache 2.0		org.springframework.boot » spring-boot-starter-tomcat

Slika 7. Objedinjene zavisnosti unutar Spring Boot Web Starter zavisnosti (Preuzeto iz: (PivotalSoftware, 2019f)*)*

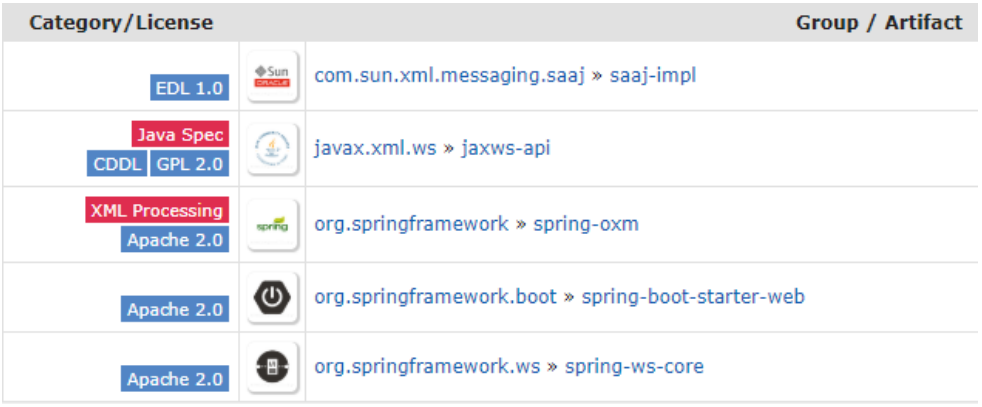
Osnovna zavisnost za izgradnju web aplikacija temeljene na *Spring Boot* razvojnom okviru je *Spring Web Starter* zavisnost. Slika iznad prikazuje sve zavisnosti koje su objedinjuje unutar iste, a omogućuju okruženje za izvođenje i izgradnju web aplikacije. Prikazane zavisnosti i njihove važnosti su sljedeće (PivotalSoftware, 2019f):

- `spring-boot-starter` - osnovna zavisnost je koja je ujedno i jezgra samog *Spring Boot* razvojnog okvira
- `spring-web` - s obzirom da se *Spring Boot* temelji na *Spring*-u tako je i temelj web zavisnosti sama *Spring* web zavisnost
- `spring-boot-starter-json` - omogućava obradu i rad s *JavaScript Object Notation* (JSON) objektima koji su česti format za razmjenu podataka u *REST* servisu,

- `spring-webmvc` - podržava izradu web aplikacije u *Model-View-Controller* arhitekturi
- `spring-boot-starter-tomcat` - sadrži svu potrebnu konfiguraciju, biblioteke i sam *Tomcat* aplikacijski server
- `hibernate-validator` - omogućava primjenu anotacija za validaciju aplikacijskih ograničenja (varijabli) pomoću *XML*-a

Spring Web Starter zavisnost obavezna je u obije aplikacije iz razloga što ista omogućava razvoj web aplikacije i pruža sve potrebne zavisnosti i *Tomcat* aplikacijski server za izvršavanje aplikacija. Prilikom pokretanja aplikacija omogućena je konfiguracija *Tomcat* aplikacijskog poslužitelja pomoću *application.properties* datoteke.

5.2.2. *Spring Web Services* zavisnost



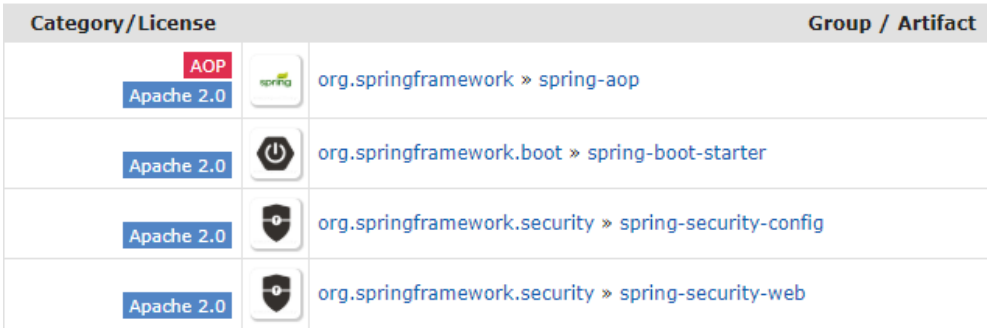
Slika 8. Objedinjene zavisnosti unutar Spring Boot Web Services zavisnosti (Preuzeto iz: (PivotalSoftware, 2019e))

Web *services* zavisnost prikazana na slici iznad osnovna je zavisnost za implementaciju *SOAP* web servisa u *Spring Boot* razvojnom okruženju. Sama zavisnost također objedinjuje nekoliko zavisnosti pod sobom poput: osnovne zavisnosti `spring-boot-starter-web` iz razloga što je web servise moguće implementirati samo u web aplikaciji. Uz samu web zavisnost srž ovog *dependecy*-a čini i `spring-ws-core` koji sadrži sve potrebne zavisnosti vezane uz web servise *Spring*-a. Ostale zavisnosti prikazane na slici su (PivotalSoftware, 2019e):

- `saaj-impl` - odnosno *SOAP with Attachments API (engl. Application programming interface) for Java* za rad s SOAP porukama
- `jaxws-api` – *Java EE API* za rad s *XML* dokumentima web servisa i samim web servisima te
- `spring-oxm` – mapiranje *XML* dokumenta u ili iz *Spring* objekta.

Opisana zavisnost korištena je u oba projekta iz razloga što na poslužiteljskoj strani moramo implementirati pristupne točke servisa, dok na klijentskoj strani potrebno je kreirati klijenta za slanje zahtjeva i zaprimanje odgovora servisa.

5.2.3. *Spring Security* zavisnost

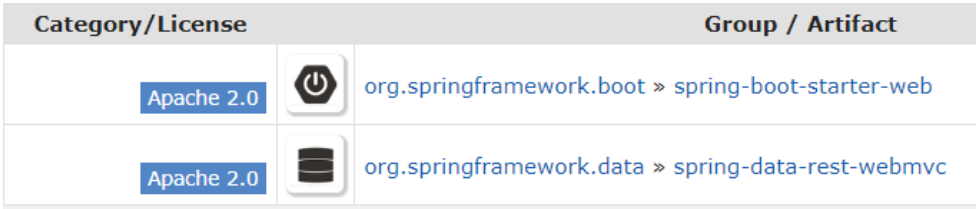


Slika 9. Objedinjene zavisnosti unutar Spring Security zavisnosti (Preuzeto iz: (PivotalSoftware, 2019c)*)*

Slika iznad prikazuje *Spring Security* zavisnost odnosno sigurnosna zavisnost, kao i web zavisnost, temelji se na *Spring* zavisnosti, u ovom slučaju `spring-security-web` zavisnosti. Kako bi se zavisnost iz *Spring* razvojnog okruženja mogla primijeniti na *Spring Boot* tu je `spring-boot-starter` s osnovnim zavisnostima i konfiguracijama vezanim uz *Spring Boot* razvojno okruženje. Preostale zavisnosti koje se koriste unutar samog *security*-a su: `spring-security-config` – sadrži osnovne postavke sigurnosti i `spring-aop` - *Aspect-Oriented Programming* (*AOP*). (PivotalSoftware, 2019c)

Navedena i opisana zavisnost korištena je, kao i u prethodnim slučajevima, u obije aplikacije kako bi se postigla zadovoljavajuća razina sigurnosti sustava. Sama implementacija sigurnosti opisana je u potpoglavljima „Konfiguracija autorizacije metoda servisa“ u poslužiteljskoj aplikaciji te „Prijava, registracija i sigurnosna konfiguracija“ na klijentskoj strani, ali zasad je dovoljno reći da navedena zavisnost prilikom uključivanja u projekt omogućuje konfiguraciju dozvola pristupa nad određenim dijelovima sustava. Navedeni način rada *Spring Security* zavisnosti temeljen je na principu davanja dozvola (*engl. Whitelisting*) pri kojem eksplicitno moramo zadati dozvole nad sustavom.

5.2.4. *Rest Repositories* zavisnost



Slika 10. Objedinjene zavisnosti unutar Spring Boot Rest Repositories zavisnosti (Preuzeto iz: (PivotalSoftware, 2019b)*)*

Rest repositories zavisnost prikazana na slici iznad vrlo je jednostavna što se tiče zavisnosti koje se nalaze pod njom, a one su sljedeće (PivotalSoftware, 2019b):

- `spring-boot-starter-web` – osnovna web zavisnost koja je pojašnjena na početku ovog poglavlja
- `spring-dana-rest-webmvc` – zavisnost temeljena na *Spring* razvojnom okviru i njegovoj podršci za rad s bazom podataka. Unutar zavisnosti nalaze se podrške za *Java Persistance API*, Hibernate te *Spring Object/Relational Mapping* koje uz pomoć `spring-data-rest-core` kreiraju osnovni *REST* servis prema samoj bazi podataka.

Navedena zavisnost koristi se samo na poslužiteljskoj strani, a ista automatski omogućava pristup podacima iz baze podataka. U *Spring Boot* projektu pristup bazi podataka moguć je preko repozitorij sučelja, a primjenom *rest repositories* sam razvojni okvir navedena sučelja „pretvara“ u *REST* servise i čini ih dostupnima bez potrebe za dodatnim implementacijama.

5.2.5. *Spring Data JPA (Java Persistance API)* i *MySQL Driver* zavisnost

Category/License	Group / Artifact
Transactions CDDL GPL 2.0	javax.transaction » javax.transaction-api
XML Processing CDDL 1.1	javax.xml.bind » jaxb-api
O/R Mapping LGPL 2.1	org.hibernate » hibernate-core
AOP Apache 2.0	org.springframework » spring-aspects
Apache 2.0	org.springframework.boot » spring-boot-starter-aop
Apache 2.0	org.springframework.boot » spring-boot-starter-jdbc
Apache 2.0	org.springframework.data » spring-data-jpa

Slika 11. Objedinjene zavisnosti unutar Spring Dana JPA zavisnosti (Preuzeto iz: (PivotalSoftware, 2019a))

Kreiranje i pristup bazi podataka moguć je pomoću zavisnosti prikazane na slici iznad točnije pomoću *Spring Dana JPA* zavisnosti. Poslužiteljska aplikacija komunicira s *MySQL* bazom podataka stoga su u projektu potrebni i upravljački programi koji se dodaju preko *MySQL Driver* zavisnosti. *Driver* omogućava *Java* spajanje na bazu podataka (*engl. Java*

Database Connectivity - JDBC), sadrži definicije i potrebne upravljačke programe za komunikaciju s bazom. Obije zavisnosti koriste se samo u poslužiteljskoj aplikaciji iz razloga što samo ona ima pristup bazi podataka.

Sa slike *Spring Data JPA* zavisnosti možemo vidjeti sve potrebne zavisnosti za rad s bazom, a one su sljedeće (PivotalSoftware, 2019a):

- `javax.transaction-api` – korišten za kreiranje *SQL* transakcija pomoću anotacija
- `jaxb-api` – programsko sučelje za rad s *XML* dokumentima, njihovo tumačenje te preslikavanje u *Java* objekte
- `hibernate-core` – *Spring Boot*-ov *ORM (engl. Object-relational mapping)* alat za preslikavanje baze u objekte i obratno
- `spring-aspects` i `spring-boot-starter-aop` – zavisnosti za aspektno orijentirano programiranje
- `spring-boot-starter-jdbc` – zavisnost za ostvarivanje i održavanje veze s bazom podataka
- `spring-dana-jpa` – srž *JPA* zavisnosti iz *Spring*-a koje je nadograđeno svim prethodno opisanim zavisnostima

5.2.6. *Thymleaf* zavisnost

Slika 12. Objedinjene zavisnosti unutar Spring Boot Thymeleaf zavisnosti (Preuzeto iz: (PivotalSoftware, 2019d)*)*

Posljednja zavisnost koja je ujedno korištena samo u klijentskoj aplikaciji prikazana je na slici iznad, a riječ je o *Thymleaf* zavisnosti. *Thymleaf* je moderni *Java template-engine* namijenjen izvođenju na poslužiteljskoj strani, a omogućuje izgradnju grafičkog sučelja u *Spring Boot* aplikacijama. U samom kreiranju stranice pomoću *Thymleafa* omogućava primjenu jednostavne programske logike i pristupati *Java* objektima što je uvelike slično s *Java Server Pages* tehnologijom. (PivotalSoftware, 2019d)

Zavisnosti koje objedinjuje *Thymleaf* starter sljedeće su (PivotalSoftware, 2019d): `spring-boot-starter` (već spomenuta osnovna zavisnost svake *Spring Boot* aplikacije), `thymleaf-spring-5` (osnovna zavisnost za primjenu *Thymleafa*) te `thymeleaf-extras-java8time` (zavisnost za komunikaciju *Java* programa s *Thymeleafom*).

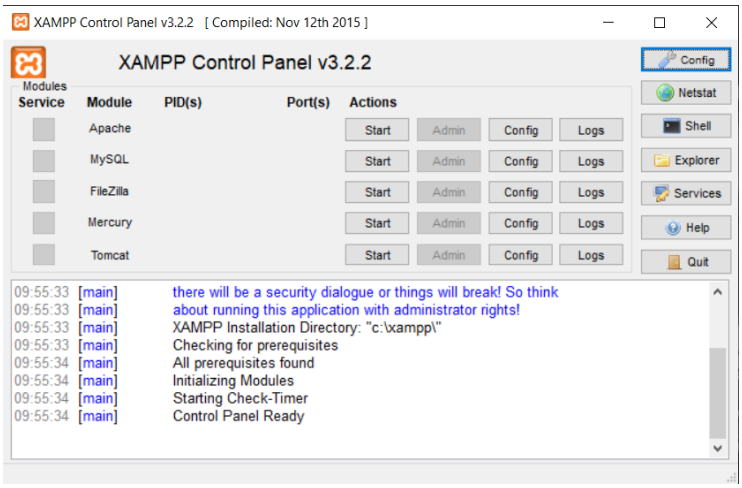
6. Praktični rad

Praktični dio diplomskog rada uključuje implementaciju aplikacije za praćenje korisnikovog novčanog tijeka pomoću *SOAP* i *REST* web servisa. Implementacija je realizirana pomoću poslužiteljske aplikacije s poslovnom logikom i bazom podataka te klijentske aplikacije za pristup servisima. Poslužiteljska aplikacija pruža web servise koji provode osnovne operacije kreiranja, čitanja, ažuriranja i brisanja (*engl. Create, Read, Update and Delete - CRUD*) na *MySQL* bazi podataka, dok je klijentska aplikacija implementirana

kao *SOAP* i *REST* klijent koji korisniku prezentiraju podatke i omogućavaju provođenje opisanih operacija kroz intuitivno sučelje.

6.1. Poslužitelj za bazu podataka

Prije samog početka razvoja poslužiteljske aplikacije potrebno je konfigurirati i pokrenuti *Apache* poslužitelj s programskom podrškom za *MySQL*. Prethodno opisana *XAMPP* aplikacija koristi se u većini slučajeva kao aplikacijski poslužitelj za pokretanje *PHP* ili *Perl* programskih jezika no u ovom slučaju ista je korištena kao poslužitelj s bazom podataka.



Slika 13. Slika početno zaslona XAMPP aplikacije

Slika iznad prikazuje početni zaslon *XAMPP* aplikacije. Prije samog pokretanja poslužitelja za bazu podataka potrebno je postaviti jednu konfiguraciju na samom *MySQL* modulu aplikacije. *Spring Boot* razvojni okvir zahtjeva postavljanje početne vrijednosti vremenske zone (*engl. default time zone*) sustava baze podatka. Postavljanje vremenske zone za *MySQL* vrši se odabirom opcije *Config* koja potom ponudi odabir *.ini* datoteke koja sadrži početne postavke. Klikom na ponuđenu datoteku otvara se ista u tekstualnom editoru te uz postojeće postavke samo moram zadati željenu vrijednost za parametar `default_time_zone`. Nakon što smo postavili vremensku zonu potrebno je pokrenuti

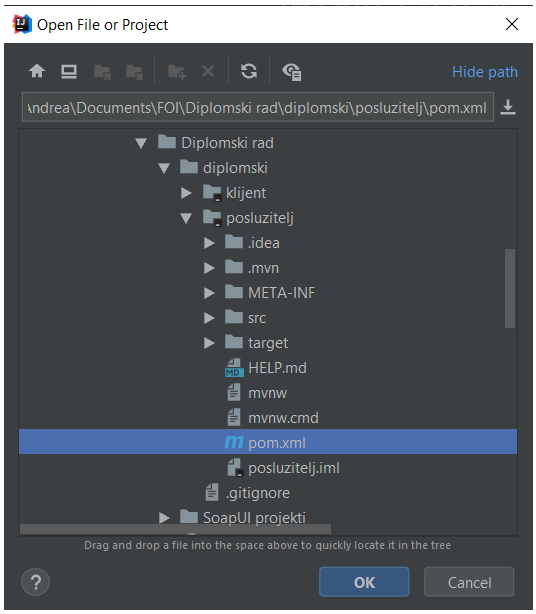
module *Apache* koji pruža lokalni *Apache* poslužitelj te pokrenuti *MySQL* programsku podršku za bazu podataka.

6.2. Otvaranje i pokretanje projekta u *IntelliJ*-u

Implementacija obije aplikacije izvršena je u razvojnom okruženju *IntelliJ* iz razloga što je isto namijenjeno razvoju *Java* aplikacija te pruža svu porubnu programsku i korisničku podršku za rad s *Java* web aplikacijama i *Spring Boot* razvojnim okvirom.

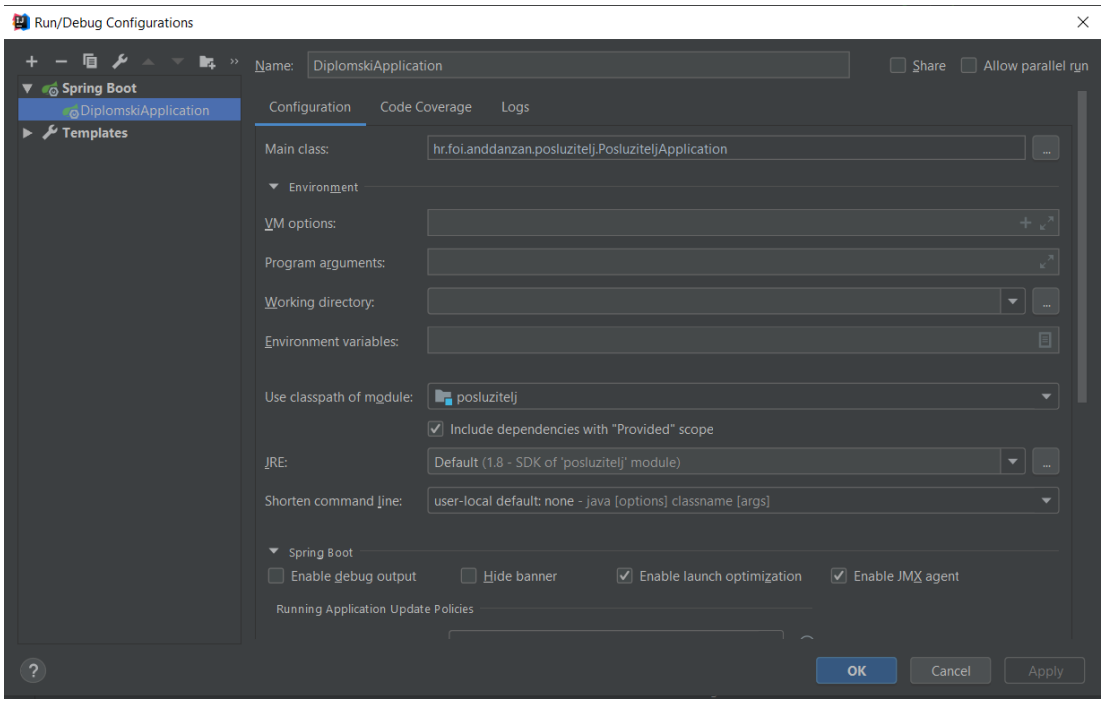
Slika 14. Početni prozor IntelliJ razvojnog okruženja

Pokretanjem samog *IntelliJ*-a otvara se prozor na slici iznad pomoću kojeg otvaramo, kreiramo ili učitavamo projekt sa sustava za verzioniranje koda. Projekti klijenta i poslužitelja kreirani su već pomoću online alata *Spring Initializr* stoga odabiremo opciju *Open* kako bi otvorili generirani projekt.



Slika 15. Odabir osnovne pom.xml datoteke projekta

Odabrana opcija otvara skočni prozor prikazan na slici iznad pomoću kojeg odabiremo *pom.xml* datoteku projekta te kliknemo *OK.* Novootvoreni skočni prozori nudi nekoliko opcija kako bi *IntelliJ* znao što želimo s odabranom datotekom. Odabiremo *Open as Project* kako bi odabranu datoteku „označili“ kao korijensku *pom* datoteku *Maven* projekta. Prilikom prvog otvaranja projekta editoru je potrebno duže vrijeme jer mora indeksirati sve datoteke i preuzeti zavisnosti na računalo. *Maven* alat za izgradnju radi na principu preuzimanja svih zavisnosti u obliku *jar* (*engl. Java ARchive*) biblioteka u lokalni *.m2* direktorij.



Slika 16. Postavka konfiguracije pokretanja SpringBoot projekta

Učitani projekt u ima spremnu *Spring Boot* konfiguraciju za pokretanje te ostale postavke projekta. Postavke konfiguracije za pokretanje i debugiranje aplikacije možemo pronaći u gornjem desnom kutu *IntelliJ*-a odabirom na opciju *Edit configurations.* Slika prikazana iznad odlomka primjer je osnovnih postavki za pokretanje aplikacije, a one su: postavljanje korištene verzije *Java* okruženja za izvršavanje aplikacije (*Java Runtime Enviromenta - JRE*) te glavne klase aplikacije koja ujedno i pokreće istu.

6.3. Implementacija poslužiteljske aplikacije

6.3.1. Konfiguracija aplikacije, kreiranje i povezivanje s bazom podataka

Baza podataka na koju se poslužiteljska aplikacija povezuje nalazi se na *Apache* poslužitelju s *MySQL* programskom podrškom koji pruža *XAMPP* aplikacija. Prvi korak u

kreiranju baze podataka je kreiranje korisnika s pravima nad bazom te same baze u kojoj će se nalaziti entiteti sustava. Kreiranje same baze podataka vrši se korištenjem *phpMyAdmin* sučelja pomoću kojega kreiramo korisnika i bazu za pripadajućeg korisnika. Za ovaj diplomski rad, baza podataka i korisnik kreirani su na sljedeći način:

1. Odabir opcije *User accounts* u izborniku na početnoj stranici
2. Klik na opciju *Add user account*
3. Uneseni potrebni podaci za prijavu (*engl. Login information*)
4. Selektirana opcija *Create database with same name and grant all privileges* – odabrana opcija kreirati će bazu podataka istog imena kao i ime korisnika koji ima sva prava nad bazom.
5. Pohrana unosa kikom na opciju *Go* na dnu stranice.
6. Nakon kreiranja baze podataka potrebno je provjeriti pravilo uspoređivanja znakova (*engl. Collation*) baze podataka i tablica koje će se nalaziti u njoj te postaviti isti na *utf8_unicode_ci* kako bi u bazu mogli pohranjivati znakove iz hrvatske abecede.

Povezivanje na bazu podataka u *Spring Boot*-u vrši se pomoću *Spring Data JPA* odnosno JPA zavisnosti. Za povezivanje dovoljno je postaviti obvezne parametre u *application.properties* datoteci s prefiksom `spring.datasource` te pozadinski servisi kreiraju vezu s bazom podataka.

```
spring.jpa.hibernate.ddl-auto=create
spring.datasource.url=jdbc:mysql://localhost:3306/anddanzan
                          ?useUnicode=yes&characterEncoding=UTF-8
spring.datasource.username=anddanzan
spring.datasource.password=anddanzan
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.data.rest.basePath=/restrepositories
```

Datoteka prikazana iznad predstavlja *application.properties* datoteku korištenu u poslužiteljskoj aplikaciji. Prva opcija u primjeru definira da se baza podataka kreira svakim pokretanjem projekta, a isto je primijenjeno iz razloga što se testni podaci unose prilikom pokretanja, pa da ne dođe do nagomilavanja podataka. Kao što je prethodno spomenuto, opcije s prefiksom `spring.datasource` osnovne su za povezivanje projekta na bazu, a parametri koji moraju biti postavljeni sljedeći su:

- `url` - definira link poslužitelj na kojem se nalazi baza podatka
- `username` i `password` - za prijavu korisnika s pravima nad bazom
- `driver` - definira upravljački program *JDBC*-a za rad s bazom.

Posljednja postavka dokumenta vezana je uz prethodno spomenutu i opisanu zavisnost *rest repositories*, a navedena postavka definira korijensku putanju automatski kreiranih *REST* metoda.

6.3.2. Kreiranje entiteta baze podataka

Nakon što su korisnik i baza podataka kreirani potrebno je kreirati entitete o kojima će se bilježiti podaci u sustavu. Kreiranje samih entiteta vrši se pomoću *Java* klasa i *Spring Boot* razvojnog okvira. Klase entiteta definiraju se korištenjem `@Entity` anotacije što razliku od *Java EE* aplikacija više ne zahtjeva kreiranje i uređivanje *persistance.xml* datoteke (Webb i ostali, 2018). *Spring Boot* pozadinski mikroservisi skeniraju sve klase koje koriste spomenutu anotaciju te iste definira kao klase entiteta, a uz navedenu anotaciju, u svrhu ovog diplomskog rada koriste se i sljedeće anotacije unutar samih klasa entiteta (Oracle, 2015):

- `@Id` – definira primarni ključ klase entiteta te ista mora biti primitivnog tipa podataka
- `@GeneratedValue` – definira strategiju za generiranje primarnog ključa entiteta. U ovom slučaju koristi se `GenerationType.IDENTITY` koji definira da se za primarni ključ koristi *identity* stupac.
- `@ManyToOne` – klase `Korisnik` i `VrstaTransakcije` povezane u s klasom `Transakcija` pomoću vanjskog ključa, a spomenuta anotacija definira kardinalnost veze (više naprema jedan u ovom slučaju).
- `@Table` – koristi se za povezivanje tablice s *Java* klasom koja ima ovu anotaciju. Na ovaj način moguće je već kreirane tablice povezati na klasu ili ukoliko tablica ne postoji, koristiti će se vrijednost iz anotacije za ime tablice u bazi podataka.
- `@JoinColumn` – anotacija za preimenovanje stupca koji je ujedno i vanjski ključ.

Potpoglavlje „*Spring Boot* aplikacija“ sadrži ERA dijagram baze podataka i pripadajućih entiteta, a isti su realizirani pomoću spomenutih *entity* klasa koje su prikazane u nastavku ovog poglavlja.

```
@Entity
@Tabel(name = "korisnik")
public class KorisnikEntitet
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String ime;
    private String prezime;
    private String korisnickoIme;
    private String lozinka;
    private String bankovniRacun;
    private boolean aktivan;
    ...
```

Prikazani *Java* kod definira klasu korisnika koja će biti korištena za kreiranje entiteta i pohranu pročitanih podataka iz baze. Tema ove aplikacije je sustav za praćenje transakcije korisnika, a klasa `KorisnikEntitet` sadrži podatke o korisniku sustava o kojem bilježimo transakcije.

```
@Entity
@Table(name = "vrsta_transakcije")
public class VrstaTransakcijeEntitet
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String naziv;
    private String opis;
    ...
```

Svaka transakcija o kojoj bilježimo podatke može imati jednu od unaprijed ponuđenih vrsta transakcija, kao što su na primjer: stanarina, režije, uplata po dogovoru … Klasa `VrstaTransakcijeEntitet` definira osnovne stupce tablice `vrsta_transacije` u bazi podataka te se ista koristi za rad s podacima.

```
@Entity
@Table(name = "transakcija")
public class TransakcijaEntitet
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String bankovniRacun;
    private String opis;
    private Date datumKreiranja;
    private Date datumTerecenja;
    private BigDecimal iznos;
    private boolean uplata;

    @ManyToOne
    @JoinColumn(name = "korisnik_id")
    private KorisnikEntitet korisnikEntitet;

    @ManyToOne
    @JoinColumn(name = "vrsta_transakcije_id")
    private VrstaTransakcijeEntitet vrstaTransakcijeEntitet;
    ...
```

Osnovni entitet ove aplikacije su transakcije koje su pohranjene u tablicu `transakcija`, a podaci iste čitaju se u *Java* klasu imena `TransakcijaEntitet` prikazana iznad. Transakcija je povezana s korisnikom i vrstom transakcija na način da korisnik može imat više transakcija, a također vrsta transakcije može se nalaziti na više transakcija odnosno riječ je o vezi jedan naprema više koja je implementirana spomenutom anotacijom `@ManyToOne` iznad varijable vanjskog ključa. *Spring Boot* razvojni okvir vanjske ključeve iz klasa entiteta u bazu podataka zapisuje tako da se u stupcu vanjskog ključa nalazi identifikator definiran u klasi vanjskog ključa (anotacijom `@Id`).

6.3.3. Pristupanje podacima u bazi podataka

Klase entiteta u *Spring Boot*-u koriste se za kreiranje entiteta u samoj bazi podataka te u konačnici za čitanje podataka u sustav i pohranu podataka iz sustava u bazu. Kako bi u samom sustavu mogli provoditi operacije nad *entity* objektima te u konačnici nad bazom podataka, moramo implementirati repozitorij (*engl. Repository*) sučelje. Svaka *entity* klasa mora imati svoje *repository* sučelje s anotacijom `@Repository` koja kreira mikroservis za rad s bazom podataka. *Spring Boot* projekt koristi *Spring Data JPA* zavisnost koja pruža `JpaRepository<T,ID>` sučelje prošireno s nekoliko internih sučelja *Spring Data*-e što u konačnici pruža veliki broj mogućnosti za rad s bazom podataka poput: paginacije, sortiranja, dohvaćanje prema identifikatoru (`findById`), brisanje (`remove`) i pohranu (`save`) objekata. (Oracle, 2015)

Prednost korištenja *JPA* zavisnosti u *Spring Boot* aplikaciji je kreiranje upita na temelju imena metode unutar repozitorij sučelja i to na način da metoda mora započeti s `get` ili `find` ukoliko dohvaća podatke te potom korištenje ključne riječi `By` i naziv stupca prema kojem vršimo pretragu. Metode za dohvaćanje podataka mogu sadržavati `And` i `Or` operator za spajanje više stupaca u pretrazi te u konačnici možemo primijeniti `OrderBy` ili `GroupBy` operatore. (Oracle, 2015; Webb i ostali, 2018)

```
@Repository
public interface KorisnikRepozitorij
          extends JpaRepository<KorisnikEntitet, Integer>
{
    Optional<KorisnikEntitet> findByKorisnickoIme(String ime);
}
```

Klasa `KorisnikRepozitorij` osim osnovnih metoda sadrži dodatnu metodu za pretragu prema korisničkom imenu. Metode `findByKorisnickoIme` osnovni je primjer spomenutog preslikavanja upita u *MySQL* upit za pretragu nad bazom podataka. Prefiks `findBy` definira da je riječ o dohvaćanju podataka, stupac `KorisnickoIme` definira naziv stupca koji pretražujemo te s obzirom da se pretražuje samo prema jednom stupcu metoda ima samo jedan parametar.

```
@Repository
public interface VrstaTransakcijeRepozitorij
          extends JpaRepository<VrstaTransakcijeEntitet, Integer>
{
    Optional<VrstaTransakcijeEntitet> findByNaziv(String naziv);
}
```

Klasa repozitorija `VrstaTransakcijeRepozitorij`, kao i prethodno spomenuti repozitorij za pristupanje podacima o korisnicima, posjeduje samo jednu metodu. Metoda `findByNaziv` koristi se kod kreiranja vrste transakcije za pretragu postoji li već vrsta transakcije s danim nazivom kako ne bi postojale dvije vrste jednakog naziva.

```
@Repository
public interface TransakcijaRepozitorij
          extends JpaRepository<TransakcijaEntitet, Integer>
{
    Optional<Page<TransakcijaEntitet>> findByKorisnikEntitet_KorisnickoIme(
                            String korisnickoIme, Pageable stranicenje);

    int countByVrstaTransakcijeEntitet_Id(int id);
}
```

Repozitorij sučelje za transakcije specifično je po tome što sadrži metodu za dohvaćanje transakcija koja pretražuje objekt korisnika, koji je ujedno vanjski ključ, prema korisničkom imenu. Navedeni upit morao bi se izvršiti spajanjem dviju tablica u *SQL*-u, ali primjenom *JPA* zavisnosti dovoljno je zadati nakon ključne riječi `findBy` naziv stupca (koji je u našem slučaju objekt vanjskog ključa `KorisnikEntitet`) te potom podvlakom definiramo naziv stupca u objektu prema kojem pretražujemo. Kako bi lakše razumjeli način preslikavanja metode i njeno izvršavanje, ispod je prikazana metoda za dohvaćanje transakcija prema korisničkom imenu korisnika čija je transakcija te njen ekvivalent u *SQL* upitnom jeziku.

```
findByKorisnikEntitet_KorisnickoIme(String korisnickoIme)
```

⇩

```
"SELECT * FROM transakcije
 JOIN korisnik ON korisnik.id = transakcije. korisnik_id
 WHERE korisnik.korisnicko_ime = " + korisnickoIme
```

Opisana metoda za dohvaćanje transakcija prema korisničkom imenu specifična je iz razloga što sadrži implementaciju straničenja koju unaprijed nudi *JpaRepository* odnosno *PagingAndSortingRepository* sučelje koje implementira. Parametri metoda za dohvaćanje podataka u većini slučajeva vrijednosti su parametara prema kojem se vrši pretraga (npr. `String korisnickoIme`), ali s obzirom na spomenuta proširenja u pogledu različitih *repository* sučelja, uz pomoć `Pageable` objekta omogućeno je sortiranje i straničenje dohvaćenih podataka. Navedeni objekt sadrži sljedeće parametre: *page* (indeks trenutne stranice), *size* (veličinu jedne stranice, broj zapisa po stranici) te *sort* (objekt za sortiranje koji zaprima stupac i vrstu sortiranja, npr. `sort=datumTerecenja,desc`). *Spring Boot* podatke iz objekta za straničenje pretvara u upit te isti provodi nad bazom podataka što rezultira podacima koji su već sortirani i straničeni na razini baze podataka.

6.3.4. Implementacija mikroservisa

Spring Boot arhitektura temelji se na kreiranju mikroservisa (*engl. Services*) koristeći anotacije: `@Service`, `@Component` ili `@Controler`. Mikroservisi su u srži *Java* zrna (*engl. Bean*) bez stanja koja sadrže poslovnu logiku aplikacije ili predstavljaju vanjski servis koji trenutni sustav koristi, a dostupni su pomoću *dependency injection* mogućnosti *Spring Boot*-a.

Sam razvojni okvir zadužen je za inicijalizaciju i upravljanje mikroservisima u aplikacijskom kontejneru te na kraju životnog ciklusa, uništavanje objekata istih. Primjenom mikroservisa obrada i funkcionalnosti aplikacije mogu se nalaziti na jednom mjestu ili kao što je slučaj u poslužiteljskoj aplikaciji mikroservisi predstavljaju točku u preko koje sustav razmjenjuje podatke. Repozitorij sučelja koriste se za pristup bazi podataka, ali u slučaju promjene zahtjeva te da podaci dolaze s vanjskog servisa, isti mikroservisi imali bi pristup podacima bez da ostatak sustava išta zna o promjeni implementacije.

Poslužiteljska aplikacija koristi tri servisa: `KorisnikServis`, `TransakcijaServis` te `VrstaTransakcijeServis` koji sadrže pružaju točku komunikacije s bazom podataka. Svi mikroservisi realizirani su primjenom *Bridge* uzorka dizajna. Spomenuti uzorak dizajna koristi se u slučajevima kada je potrebno izmjenjivati ponašanje tijekom izvršavanja programa te kako bi izbjegli trajnu ovisnost programa o određenoj implementaciji (Kermek, 2019). *Bridge* uzorak realizira se pomoću sučelja koje definira osnovno ponašanje mikroservisa te klasu koja implementira metode sučelja. Ispod ovog paragrafa prikazano je sučelje `KorisnikServis` i dio implementacije, dok je ostatak implementacije dostupan s ostalim mikroservisima u aplikaciji priloženoj uz diplomski rad.

```
public interface KorisnikServis
{
    Optional<KorisnikEntitet> dohvatiKorisnika(int id);

    Optional<KorisnikEntitet> dohvatiKorisnika(String korisnickoIme);

    boolean kreirajKorisnika(KorisnikEntitet noviKorisnik);

    boolean azurirajKorisnika(KorisnikEntitet azuriraniKorisnik);

    List<KorisnikEntitet> dohvatiSveKorisnike();
}
```

Sučelje `KorisnikServis` sadrži metode za *CRUD* operacije nad podacima korisnika te sve rade s klasom KorisnikEntitet koja definira strukturu podataka za komunikaciju s bazom podataka. Prikazani način implementacije na zahtjeva detalje implementacije komunikacije s bazom ili servisom za dohvaćanje podataka već samo služi kao točka komunikacije sustava s mjestom na kojem su podaci.

```
@Service
public class KorisnikServisImpl implements KorisnikServis
{
    private final KorisnikRepozitorij korisnikRepozitorij;

    public KorisnikServisImpl(final KorisnikRepozitorij korisnikRepo)
    {
        this.korisnikRepozitorij = korisnikRepo;
    }

    @Override
```

```
    public Optional<KorisnikEntitet> dohvatiKorisnika(final int id)
    {
        return this.korisnikRepozitorij.findById(id);
    }

    @Override
    public Optional<KorisnikEntitet> dohvatiKorisnika(final String korisnickoIme)
    {
        return this.korisnikRepozitorij.findByKorisnickoIme(korisnickoIme);
    }

    @Override
    public boolean kreirajKorisnika(final KorisnikEntitet noviKorisnik)
    {
        return this.korisnikRepozitorij.save(noviKorisnik) != null;
    }

    @Override
    public boolean azurirajKorisnika(final KorisnikEntitet korisnikZaAzuriranje)
    {
        return this.korisnikRepozitorij.save(korisnikZaAzuriranje) != null;
    }

    @Override
    public List<KorisnikEntitet> dohvatiSveKorisnike()
    {
        return this.korisnikRepozitorij.findAll();
    }
}
```

Klasa prikazana iznad implementacija je sučelja `KorisnikServis` te ista sadrži svu logiku vezanu uz komunikaciju s mjestom na kojem su pohranjeni podaci, odnosno u ovom slučaju repozitorij sučeljem. Prikazani servis koristi se kao posrednik između poslovne logike sustava i sučelja za pristup i rad s bazom podataka. Prikazani način implementacije primjenom mikroservisa povećava modularnost samog sustava. Mikroservisi skrivaju od samog sustava pristup bazi podataka te pružaju jednu pristupnu točku za rad s bazom odnosno s konkretnim entitetom baze u slučaju poslužiteljske aplikacije

6.3.5. Implementacija fasada

Sva poslovna logika poslužiteljske aplikacije velikim dijelom vezana je uz dohvaćanje, pohranu i izmjenu podataka u bazi. Obrada podataka u pogledu njihovog preslikavanje u klase za *SOAP* i *REST* web servise prevladava u navedenoj obradi stoga je upravo iz tog razloga primijenjen *facade* uzorak dizajna sustava.

Facade uzorak dizajna pruža uniformno sučelje za skup sučelja u podsustavu (Kermek, 2019) točnije rečeno, u slučaju poslužiteljske aplikacije fasada objedinjuje sve mikroservise potrebne za rad s određenim entitetom. Navedeni uzorak, također kao i implementacija mikroservisa koristi uzorak *bridge* to jest sučelje s prefiksom „`Fasada`“ definira ponašanje, a sustavu postoji barem jedna implementacija fasade s prefiksom „`FasadaImpl`“.

```
public interface KorisnikFasada
```

```
    {
        Optional<KorisnikSoap> dohvatiSoapKorisnika(String korisnickoIme);

        Optional<KorisnikEntitet> dohvatiKorisnika(int id);

        boolean kreirajKorisnika(KorisnikEntitet noviKorisnik);

        boolean azurirajKorisnika(KorisnikEntitet azuriraniKorisnik);

        Optional<KorisnikEntitet> dohvatiKorisnika(String korisnickoIme);

        List<KorisnikSoap> dohvatiSveKorisnike();

        boolean promjeniUloguKorisniku(int idKorisnika, String uloga);
    }
```

Programski kod iznad primjer je jednog od više sučelja *facade* uzorka, konkretnije u ovom slučaju riječ je o sučelju fasade za rad s podacima o korisniku. U prikazanom programskom primjeru možemo na temelju metode za dohvaćanje korisnika prema korisničkom imenu uočiti zašto je primijenjen uzorak *facade.* Naime da su u sustavu korišteni samo mikroservisi, jedan mikroservis imao bi dvije metode koje bi radile isti posao, dohvaćale podatke o korisniku korisničkom imenu. Primjenom fasade sustav dobiva podatke u traženom formatu, a sama fasada skriva implementaciju od sustava te u pozadini poziva istu metodu servisa za dohvaćanje korisnika prema korisničkom imenu.

```
@Component
public class KorisnikFasadaImpl implements KorisnikFasada
{
    private final KorisnikServis korisnikServis;
    private final BCryptPasswordEncoder enkoder;

    public KorisnikFasadaImpl(final KorisnikServis korisnikServis,
                              final BCryptPasswordEncoder enkoder)
    {
        this.enkoder = enkoder;
        this.korisnikServis = korisnikServis;
    }

    @Override
    public Optional<KorisnikSoap> dohvatiSoapKorisnika(final String korisnickoIme)
    {
        Optional<KorisnikSoap> korisnikSoapOpt = Optional.empty();

        final Optional<KorisnikEntitet> korisnikOpt =
                 this.korisnikServis.dohvatiKorisnika(korisnickoIme);
        if (korisnikOpt.isPresent())
        {
            final KorisnikSoap korisnikSoap = new KorisnikSoap();
            AlatZaPodatke.kopirajPodatke(korisnikOpt.get(), korisnikSoap);

            korisnikSoapOpt = Optional.of(korisnikSoap);
        }
        return korisnikSoapOpt;
    }

    @Override
    public Optional<KorisnikEntitet> dohvatiKorisnika(final String korisnickoIme)
    {
        return this.korisnikServis.dohvatiKorisnika(korisnickoIme);
    }
```

```
    ...

    @Override
    public boolean kreirajKorisnika(final KorisnikEntitet noviKorisnik)
    {
        boolean korisnikKreiran = false;

        final Optional<KorisnikEntitet> korisnikEntitetOptional =
           this.korisnikServis.dohvatiKorisnika(noviKorisnik.getKorisnickoIme());
        if (!korisnikEntitetOptional.isPresent())
        {
            korisnikKreiran = this.korisnikServis.kreirajKorisnika(noviKorisnik);
        }

        return korisnikKreiran;
    }
}
```

Prikazani programski kod iznad odlomka dio je implementacije sučelja fasade za rad s podacima o korisnicima. Prethodni spomenuti razlog implementacije *facade* uzorka i metode za dohvaćanje korisnika prikazane su u ovom primjeru, a on su: `dohvatiKorisnika` i `dohvatiSoapKorisnika`. Obije metode pozivaju isti mikroservis, točnije istu metodu, dok jedna mora dohvaćene podatke preslikati u *SOAP*-u razumljivi format, druga metoda može samo proslijediti rezultate pretrage. Također prilikom kreiranja ili ažuriranja podataka u bazi *SOAP* servisi koriste vlastitu strukturu podataka stoga fasada preslikava zaprimljene podatke u klase entiteta te pomoću servisa zapisuje u bazu podataka. U programskom primjeru prikazano je kreiranje korisnika prilikom kojega se podaci korisnika zapisuju u entitet, provjeravaju te ukoliko korisnik ne postoji zapisuju.

6.3.6. Primjena mikroservisa i fasada

Primjena mikroservisa nalazi se samo na jednom mjestu u poslužiteljskoj aplikaciji, u fasadama. Mikroservisi implementirani su na način da svaki mikroservis sadrži metode za rad s jednom vrstom entiteta odnosno njihovo propagiranje do baze i natrag u sustav. Fasade implementiraju poslovnu logiku sustava, a s obzirom da je većina obrade podataka vezana uz bazu podataka iste koriste mikroservise kao pristupnu točku za pristup podacima. Na primjeru fasade za transakcije, svaka transakcija veže se za korisnika i sadrži vezu na vrstu transakcija. Navedeni slučaj detaljnije je moguće pojasniti u slučaju kreiranja transakcije, metoda za kreiranje zaprimiti će osnovne podatke transakcije iz kojih se potom preko mikroservisa dohvaćaju objekti vanjskih ključeva (entitet korisnika čija je transakcija i entitet vrste transakcija) te mikroservis transakcija može kreirati entitet transakcije u bazi.

Vezano uz implementaciju samih mikroservisa i fasada poslužiteljske aplikacije važno je reći sljedeće:

- Zavisnosti u pogledu drugih mikroservisa injektirani su pomoću konstruktora iz razloga što je navedeni način ujedno i najsigurniji. Injektiranje preko konstruktora

onemogućava inicijalizaciju servisa bez potrebnih zavisnosti, a *Spring Boot* sam prepoznaje injektiranje te ga izvršava . (Webb i ostali, 2018)

- Metode koje vraćaju objekte koriste `Optional` kontejner kako bi se osigurao sustav od mogući iznimaka zbog `null` vrijednosti. `Optional` kontejner za rukovanje objektima sadrži metodu `isPresent` za provjeru je li objekt `null` te pruža metodu `get` za dohvaćanje objekta.
- Klasa `AlatZaPodatke` klasa je s statičnim (*engl. Static*) metodama za preslikavanje objekata. Navedena klasa kreirana je iz razloga što *SOAP* i *REST* servisi koristi svoje klase za prijenos podataka, a svi podaci u bazi nalaze se u klasama entitetima.
- Klasa `AlatZaDatume` kao i prethodna klasa za podatke sadrži statične metode za promjenu formata i tipa podataka datuma. *SOAP* klase pohranjuju datum u `XMLGregorianCalendar` tipu podataka, dok svi entiteti sadrže datume u `java.util.Date` formatu.
- `BCryptPasswordEncoder` – unaprijed zadana *Spring Boot* klasa, a koristi za kreiranje nečitljivog oblika teksta.

6.3.7. Konfiguracija zaštite poslužiteljske aplikacije

Svi podaci sustava dostupni su pomoću metoda *SOAP* i *REST* web servisa, a bez zaštite istima može bilo tko pristupiti slanjem jednostavnog zahtjeva. Kako bi se podaci zaštitili u sustavu mora postojati prijava, ali problem je što poslužiteljska aplikacija nije standardna web aplikacija s grafičkim sučeljem stoga nije moguće implementirati jednostavnu formu za prijavu. Zaštita podataka u poslužiteljskoj aplikaciji provedena je pomoću *Spring security* zavisnosti te autorizacije na razini zahtjeva.

```
...
@Override
protected void configure(final HttpSecurity http) throws Exception
{
    http.httpBasic()
      .and().authorizeRequests()
      .antMatchers("/rest/korisnik/prijava/**",
                   "/rest/korisnik/kreiraj").permitAll()
      .antMatchers("/rest/**").authenticated()
      .antMatchers(HttpMethod.GET,"/restrepositories/transakcijaEntitets
                           /*/vrstaTransakcijeEntitet").permitAll()
      .antMatchers("/restrepositories/**").authenticated()
      .antMatchers("/soap/**.wsdl").permitAll()
      .antMatchers("/soap/**").authenticated()
      .and()
      .csrf().disable()
      .formLogin().disable();
}
...
```

Programski kod prikazan iznad sigurnosna je konfiguracija poslužiteljske aplikacije kojom su sve eksplicitno definirane metode web servisa zaštićene osnovnom autorizacijom. *Spring security* modul prilikom uključivanja u projekt radi na principu zaštite svih dijelova sustava te onemogućuje pristup neautoriziranom korisniku. Primjenom konfiguracije `authorizeRequests()` konfigurirano je da svaki zahtjev mora biti autoriziran odnosno mora biti provjereno pravo pristupa sustavu. Sama autorizacija nije moguća bez autentikacije korisnika iz razloga što samo korisnici iz baze sustava imaju pravo pristupa web stranicama, osim stranicama prijave i registracije koje su dostupne svim korisnicima. Autorizacija korisnika provodi se uz pomoć dvije metode: `antMatchers()` i `authenticated()`. Primjenom metode `antMatchers()` zaštite s unesenom putanje ili uzorka (*engl. Pattern*) koji ista mora zadovoljiti definira se područje ne kojem je primijenjena zaštita ili onemogućena. Nakon što je definirano područje na primjene zaštite, metodom `.authenticated()` u nastavku definiramo da na prethodno zadanoj putanji svaki pristup mora biti autenticiran. Metoda `antMatchers()` u kombinaciji s `permitAll()` metodom omogućuje pristup svim korisnicima (autenticiranim i neautenticiranim). Opisani pristup *Spring security*-a opis je principa *whitelistinga* odnosno eksplicitnog davanja ovlasti pristupa određenim putanjama i dijelovima sustava. Navedeni načina osiguravanja sustava prvo uzima u obzir autorizaciju zahtjeva, a ista je omogućena autentikacijom korisnika. *Whitelisting* također je vidljiv na postavkama koje dozvoljavaju neprijavljenom korisniku pristup određenim dijelovima sustava Kao što je već rečeno, poslužiteljska aplikacija provodi mjere zaštite autorizacijom i autentifikacijom na razini zahtjeva stoga na kraju primjenom metoda `.formLogin().disable()`. onemogućena je klasična prijava pomoću forme

Nakon što je sustav zaštićen od pristupa neautoriziranog korisnika potrebno je uspostaviti mikroservis kako bi *Spring security* imao pristup korisnicima iz naše baze podataka te u konačnici moga autorizirati zahtjev i autentificirati korisnike. Sučelje `UserDetailsService` koristi se u implementaciji prijave korisnika bila ona u pogled forme ili autorizacije na razini zahtjeva. Navedeno sučelje implementira se kroz vlastiti mikroservis te isto pruža implementaciju metode `loadUserByUsername`. Metodu `loadUserByUsername` poziva sam *Spring* Boot-a ista ima pristup bazi podataka preko mikroservisa za korisnikove podatke. Nakon što je korisnik dohvaćen iz baze u samom objektu `User` provjeravaju se korisnički podaci te u slučaju ispravnih podataka metoda vraća prijavljenog korisnika koji je pohranjen u sustavu.

6.3.8. Implementacija *SOAP* web servisa

Jednostavan protokol za pristupanje objektu, odnosno *Simple Object Access Protocol* jednostavan je „lagan“ (*engl. Lightweight*) protokol za pristupanje podacima u decentraliziranom i distribuiranom sustavu (Box i ostali, 2000). David Winer, kreator *XML-RPC*-a, uz pomoć kolega nastavio je s radom na novom protokolu temeljenom na razmjeni *XML* poruka. Servisi bazirani na *SOAP* protokolu dizajnirani su na način da nemaju ograničenja u pogledu odabira programskog jezika i načina pisanja koda već su jedina ograničenja samo ona vezana uz razmjenu poruka i dogovora komunikacije. (Box i ostali, 2000)

Spring Boot razvojni okvir u *Web service* zavisnosti nudi veliki broj mogućnosti za implementaciju *SOAP* web servisa i klijenta za pristup servisu. *SOAP* servis moguće je implementirati na dva načina (Poutsma, Evans, Rabbo, & Turnquist, 2019):

- Kreiranje prema „ugovoru“ (*engl. Contract-First*) - generiranje servisa iz postojećeg Opisnog jezika web servisa (*engl. Web Service Definition Language WSDL*) ili Definicije *XML* sheme (*engl. XML Schema Definition - XSD)*
- Kreiranje „ugovora“ iz koda (*engl. Contract-Last*) - implementacijom servisa pomoću *Java* koda te generiranje „ugovora“.

U ovom diplomskom radu korišten je *Contract-First* pristup kreiranja servisa iz postojećeg *XSD* dokumenta, a nakon implementacije pristupnih točaka i metoda primjenom *Contract-Last* pristupa generiran je finalni *WSDL* dokument. *Contract-First* način implementacije primjenjuje se iz nekoliko razloga, a kako navode Poutsma, Evans, Rabbo, & Turnquist u službenoj dokumentaciji *Spring Web Services Reference Documentation* oni u sljedeći:

- Razlika u preslikavanju *Java* objekata u *XML* i obratno (*engl. Object/XML Impedance Mismatch*) – jednostavnije je *XML* preslikati u *Java* kod iz razloga što je manje struktura podataka i više ograničenja, dok poneke strukture podatak i veze između klasa nije moguće „prebaciti“ u *XML*,
- Performanse – preslikavanje *Java* koda u *XML* zahtjeva znatno više resursa
- Ponovna iskoristivost – *XSD* datoteku možemo ponovno iskoristiti u drugom sustavu neovisno o programskom jeziku,
- Verzioniranje – lakše je pratiti promjene na jednom dokumentu alatom za verzioniranje, nego pratiti više klasa.

6.3.8.1. Početna XSD shema

Spring Boot projekt u našem slučaju koristi *Maven* alat za izgradnju projekta koji podržava dodatke. Dodaci se pokreću pri kompiliranju `pom.xml`-a i izgradnje same aplikacije, a jedan od dodataka je `jax2b-maven-plugin` čije su postavke prikazane u *XML* datoteci ispod.

```
<build>
    <plugins>
        ...
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>jaxb2-maven-plugin</artifactId>
            <version>2.3.1</version>
            <executions>
                <execution>
                    <id>xjc-schema</id>
                    <goals>
                        <goal>xjc</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
                <sources>
                    <source>src/main/resources/datoteke/soap.xsd</source>
                </sources>
                <packageName>hr.foi.anddanzan.soap.podaci</packageName>
                <clearOutputDir>true</clearOutputDir>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Pokretanjem naredbe `mvn compile` izgrađuje se aplikacije te dodatak generira sve potrebne klase definirane na putanji unutar oznake `source`. Pomoću oznake `packageName` definiran je odredišni direktorij u kojem će se nalaziti generirane klase, a oznaka `clearOutputDir` naznačava da se sadržaj odredišnog direktorija obriše prije ponovnog generiranja.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:tns="http://foi.hr/anddanzan/diplomski/soap/podaci"
           targetNamespace="http://foi.hr/anddanzan/diplomski/soap/podaci"
           elementFormDefault="qualified">
    ...
    <xs:element name="dohvatiVrsteTransakcijaZahtjev">
        <xs:complexType>
            <xs:complexContent>
                <xs:restriction base="xs:anyType"/>
            </xs:complexContent>
        </xs:complexType>
    </xs:element>

    <xs:element name="dohvatiVrsteTransakcijaOdgovor">
        <xs:complexType>
            <xs:sequence>
                <xs:element maxOccurs="unbounded" name="vrsteTransakcija"
                            type="tns:vrstaTransakcijeSoap" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    ...
    <xs:complexType name="vrstaTransakcijeSoap">
        <xs:sequence>
```

```
            <xs:element name="id" type="xs:int" />
            <xs:element name="naziv" type="xs:string" />
            <xs:element name="opis" type="xs:string" />
        </xs:sequence>
    </xs:complexType>
    ...
</xs:schema>
```

Prikazani *XML* dokument sadrži strukturu zahtjeva i odgovora samo za dohvaćanje svih vrsta transakcija iz razloga što navedeni elementi sadrže sve oznake i atribute koje je potrebno poznavati za razumijevanje `soap.xsd` datoteke. Korijenski element svake *XSD* datoteke je `schema` koja definira potrebne imenske prostore (*engl. Namespace*) za tumačenje oznaka unutar korijenske oznake. Osnovni *namespace* koji definira dokument kao *XSD* dokument je zadan atributom `xmlns:xs` te isti propisuje dozvoljene oznake i atribute unutar dokumenta (Thompson, Mendelsohn, Beech, & Maloney, 2012). Atributi `xmlns:tns` i `targetNamespace` definiraju imenski prostor u kojem se nalaze posebni kompleksni tipovi podataka koje smo sami definirali. Putanja `http://foi.hr/anddanzan/diplomski/soap/podaci` definira imenski prostor u kojem se nalazi definicija ulaznih i izlaznih tipova podataka. Oznake koje se koriste za definiranje svih zahtjeva, odgovora i složenih struktura podataka sljedeće su (Thompson i ostali, 2012):

- `element` – osnovna gradivna jedinica *XSD* dokumenta, a ista može koristiti jedan od jednostavnih tipova podataka koje pruža osnovni imenski prostor ili može sadržavati druge oznake. Svaki *element* mora biti definiran imenom (*engl. name*) te vrstom podataka (*engl. type*) ili referencom (atribut *ref*) na neki drugi element.
- `complexType` – definira kompleksnu strukturu podataka, a mora sadržavati atribut *name.* U ovom diplomskom radu navedena oznaka koristi se za definiranje strukture kompleksnih podataka odnosno klasa koje se koriste za prijenos podataka.
- `sequence` – kao što i samo ime kaže, definira sekvencu odnosno slijed. Svi elementi definirani unutar ove oznake moraju se nalaziti u generiranim klasama u istom redoslijedu.
- `complexContent` – ukoliko `complexType` ne definira određeni slijed polja, može definirati i određeni složeni sadržaj u pogledu proširenja ili ograničenja
- `restriction` – u našem slučaju oznaka `complexContent` sadrži ograničenje da navedeni zahtjev može primiti bilo koji parametar
- `maxOccurs`– definira da se određena oznaka može ponavljati jednom ili više puta. Navedeni atribut u *Java* kodu manifestira se kao lista podataka ukoliko sadrži vrijednost `unbound`, a u *SOAP* porukama omogućava da se oznaka s tim atributom pojavljuje više puta unutar svojeg roditelja.

6.3.8.2. Preslika *XSD* datoteke u *Java* klase

Izgradnja aplikacije početni *XSD* dokument tumači te generira klase koje će se koristiti u *SOAP* servisu. *SpringBoot* koristi anotacije iz paketa `javax.xml.bind.annotation` kako bi *SOAP* klase definirao kao klase za mapiranje *XML*-a odnosno kako bi prilikom zaprimanja zahtjeva mogao preslikati *XML* u *java* klasu, a prilikom vraćanja odgovara učiniti obratno. Tri prikazane *Java* klase ispod ovog paragrafa preslika su definicija iz prethodno prikazanog *XML*-a, a sve koriste sljedeće anotacije (Oracle, 2019):

- `@XmlAccessorType` - klasa s anotacijom može biti serijalizirana u X*ML* dokument, a u *XML* se preslikava svaka varijabla koja nije statična ili s konačnom vrijednošću.
- `@XmlType` - koristi se za preslikavanje klase u *XML* shemu, a atribut `propOrder` označava redoslijed kojim se preslikavanje varijabli vrši iz *XML*-a u *Java* klasu i obratno. Navedeni atribut implementacija je oznake `sequence` iz *XSD* sheme.
- `@XmlRootElement` - korijenska oznaka, a vrijednost `name` biti će ime iste.
- `@XmlElement` - pruža dodatne mogućnosti poput imena oznake koje će se preslikati u varijablu s anotacijom, mogućnost da bude bez vrijednosti (*null*) ili kao što je u našem slučaju, definira da je varijabla obavezna.
- `@XmlSchemaType` - koristi se za preslikavanje tip podataka u određenu varijablu. *XSD* shema koristi svoje tipove podataka, kao na primjer `dateTime` u našem slučaju, a `javax.xml.bind` isti preslikava u `XMLGregorianCalendar` stoga je potrebno povezati tip podataka iz sheme s tipom u *Java* klasi.

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "")
@XmlRootElement(name = "dohvatiVrsteTransakcijaZahtjev")
public class DohvatiVrsteTransakcijaZahtjev {
}
```

Klasa zahtjeva koristi se za pozivanje metode pristupne točke te za prijenos parametara u navedenu metodu. Kao što je prethodno rečeno, pozivanje metode za dohvaćanje svih vrsta transakcija poziva se bez dodatnih parametara te se oznake `complexContent` i `restriction` s vrijednošću `xs:anyType` u *Java* kodu preslikavaju u praznu klasu. Pozivanje metode u pristupnoj točki vrši se preko `name` atributa zadanog u `XmlRootElement` anotaciji.

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
    "vrsteTransakcija"
})
@XmlRootElement(name = "dohvatiVrsteTransakcijaOdgovor")
public class DohvatiVrsteTransakcijaOdgovor {
    @XmlElement(required = true)
```

```
    protected List<VrstaTransakcijeSoap> vrsteTransakcija;
    ...
    public List<VrstaTransakcijeSoap> getVrsteTransakcija() {
        if (vrsteTransakcija == null) {
            vrsteTransakcija = new ArrayList<VrstaTransakcijeSoap>();
        }
        return this.vrsteTransakcija;
    }
}
```

Prikazana klasa predstavlja odgovor koji *SOAP* operacija pristupna točke vraća. Za navedenu klasu valja napomenuti da predstavlja primjer implementacije `maxOccurs` atributa iz *XSD* sheme, a to je implementacije korištenjem `java.util.List` tipa podataka.

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "vrstaTransakcijeSoap", propOrder = {
    "id",
    "naziv",
    "opis"
})
public class VrstaTransakcijeSoap {
    protected int id;
    @XmlElement(required = true)
    protected String naziv;
    @XmlElement(required = true)
    protected String opis;
    ...
```

Posljednji element početno *XSD*-a predstavlja podatak koji se prenosi unutar samog odgovora. Oznaka `complexType` koristi se za definiranje kompleksnih tipova koji se potom mogu koristiti na više mjesta. Navedena oznaka preslikava se u klasu `VrstaTransakcijeSoap` koja sadrži sve podatke kao i sama klasa entiteta. Prethodno spomenuti `AlatZaPodatke` koristi se u ovakvim slučajevima u kojima imamo klasu `VrstaTransakcijeEntitet` pomoću koje čitamo podatke iz baze, ali *SOAP* operacija mora vratiti `DohvatiVrstaTransakcijaOdgovor` odgovor koji prenosi listu objekata tipa `VrstaTransakcijeSoap` što u konačnici zahtjeva kopiranje podataka iz jednog objekta u drugi.

Slika 17. Klase zahtjeva, odgovora i podataka za SOAP servis

Slika iznad prikazuje konačni rezultat pokretanja naredbe `mvn compile` i generiranje klasa pomoći `jaxb` *Maven* dodatka.

6.3.8.3. Konfiguracija *SOAP* servisa

Sva komunikacija *SOAP* servisa vrši se u porukama *XML* formata, stoga prije same implementacije pristupnih točaka i operacija koje iste nude potrebno je definirati pokretanje nekoliko mikroservisa *Spring Boot*-a za komunikaciju sa *SOAP* servisom.

Kako bi *SOAP* poruke poslane od klijenta stigle do pristupne točke implementiran je dispečer poruka definiran klasom `MessageDispatcherServlet`. Navedena klasa instancira se u metodi s anotacijom `@Bean` pomoću koje pri završetku metode objekt koji je vraćen kao rezultat postaje *Java bean.* Unutar metode za kreiranje dispečera, spomenuta klasa zaprima aplikacijski kontekst kao parametar pri inicijalizaciji kako bi sve pristupne točke koje postoje u sustavu bile raspoložive dispečeru. Nakon što dispečer dohvati listu svih pristupnih točaka potrebno je isti registrirati kao *servlet* na putanji tako da sve *SOAP* poruke idu preko zadane putanje.

Uz samog dispečera poruke, *Spring Boot* nudi unaprijed pripremljene klase za rad s *WSDL*-om servisa. U ovom diplomskom radu korištene su dvije klase (*Java beana*) za generiranje *WSDL*-a, jedna za učitavanje početnog *XSD* dokumenta te druga za skeniranje *Java* koda i kreiranje konačnog *WSDL*-a s svim operacijama i *SOAP* porukama. `DefaultWsdl11Definition` klasa je čiji *Java bean* pokreće konfiguracija *SOAP* servisa, a ista radi na principu skeniranja svih klasa servisa. Skeniranje klasa rezultira operacijama koje servis nudi, a pomoću već definiranog `XsdSchema` *Java beana* početna *XSD* datoteka učitana je u sustav i *SOAP* poruke povezane su s operacijama. Važna napomena vezana uz *bean* `DefaultWsdl11Definition` je ta da isti sve podatke iz *XSD*-a skenira prema sufiksu koji su inače na engleskom jeziku, ali u našem slučaju *request* je zahtjev, a *response* je odgovor stoga isti moraju biti zadani prilikom konfiguracije klase *beana.*

6.3.8.4. Implementacija pristupne točke

Spring Boot definira *SOAP* pristupnu točku pomoću anotacije `@Endpoint`, a potom unutar klase definiramo „potrošače“ odnosno metode koje obrađuju zahtjeve. Zahtjevi koji su poslani u obliku *SOAP* poruka preslikavaju se u objekte pomoću *Jackson* biblioteke i pozadinskih mikroservisa te postaju ulazni parametri metode pristupne točke. U nastavku poglavlja prikazan je dio implementacije pristupne točke za manipulaciju podataka o vrstama transakcija, a isti je prikazan iz razloga što sadrži osnovne gradivne elemente *SOAP* servisa *Spring Boot* aplikacije.

Sva poslovna logika vezana uz obradu podataka te rad s bazom podataka premještene je u fasade. Fasade su *Java bean*-ovi koji postaje u kontejneru na aplikacijskom serveru, a njihovo injektiranje obavlja sam razvojni okvir. U prikazanom slučaju, s obzirom da je riječ o vrstama transakcija, potrebno je dobaviti fasadu za obradu podataka, a svo injektiranje navedene i sličnih zavisnosti vrši se pomoću konstruktora, kao što je opisano u poglavlju „Implementacija mikroservisa“.

```
@Endpoint
public class VrstaTransakcijePristupnaTocka
{
    private final VrstaTransakcijeFasada vrstaTransakcijeFasada;

    public VrstaTransakcijePristupnaTocka(
                final VrstaTransakcijeFasada vrstaTransakcijeFasada)
    {
        this.vrstaTransakcijeFasada = vrstaTransakcijeFasada;
    }

    @PayloadRoot(namespace = Konstante.PODACI_IMENSKI_PROSTORA,
                localPart = "dohvatiVrsteTransakcijaZahtjev")
    @ResponsePayload
    public DohvatiVrsteTransakcijaOdgovor dohvatiVrsteTransakcija(@RequestPayload
```

```
                        final DohvatiVrsteTransakcijaZahtjev zahtjev)
    {
        final DohvatiVrsteTransakcijaOdgovor odgovor =
                                    new DohvatiVrsteTransakcijaOdgovor();

        final List<VrstaTransakcijeSoap> sveVrsteTransakcija =
                 this.vrstaTransakcijeFasada.dovatiSveSoapVrsteTransakcija();
        odgovor.getVrsteTransakcija().addAll(sveVrsteTransakcija);

        return odgovor;
    }

    ...
```

Svaki *SOAP* servis definiraju četiri anotacije, prethodno spomenuta `@Endpoint` te `@PayloadRoot`, `@RequestPayload` i `@ResponsePayload` anotacije pomoću kojih prisutna točka definira operaciju za obradu zahtjeva, odgovor koji vraća te tumačenje samog zahtjeva. `@Endpoint` anotacija u svojoj implementaciji koristi prethodno spomenutu `@Controller` anotaciju što u konačnici znači da je pristupna točka *Java bean* u kontejneru aplikacijskog servera namijenjen za rukovanje *XML SOAP* porukama.

Anotacija `@PayloadRoot` definira potrošača zahtjeva odnosno metodu koja se poziva kad pristupna točka zaprimi *SOAP XML* poruku s oznakom definiranom u atributu `localPart`. Navedena anotacija uz atribut `localPart` sadrži i putanju imenskog prostora, atribut `namesapce`, kako bi se definirao imenski prostor za tumačenje oznaka. Kako bi mogli tumačiti ulazne i izlane podatke koriste se anotacije `@RequestPayload` i `@ResponsePayload`. Kao što je pojašnjeno u poglavlju „Preslika *XSD* datoteke u Java klase“, generirane klase sadrže *XML* anotacije kojima je definirano preslikavanje *XML* oznaka i vrijednosti unutar istih u *Java* objekte i njegove varijable. Navedene anotacije za zahtjev i odgovor definiraju da se podaci iz klasa s *XML* anotacijama preslikaju u *XML* odgovor ili iz *XML* zahtjeva u klasu.

Nakon što su sve pristupne točke implementirane završen je *Contract-Last* pristup implementiranja te je moguće generirati *WSDL* datoteku našeg servisa. Zavisnost *Web services* u sebi već sadrži potrebne klase za kreiranje *WSDL* dokumenta koji predstavlja naš servis, a potrebno je samo implementirati servis točnije kreirati *Java bean* koji će se pokretati na poziv putanje s ekstenzijom *wsdl*. U ovom diplomskom radu korišten je već opisani mikroservis `DefaultWsdl11Definition`.

6.3.8.5. Generirani *WSDL*-a servisa

SOAP servisi omogućili su standardiziranu razmjenu podataka između klijenta i poslužitelja ili između dva poslužitelja (servisa). Standardizacija komunikacije između servisa sa *SOAP* servisima postignuta je pomoću *WSDL*-a odnosno Opisnog jezika web servisa.

Svaki *XML* dokument koji opisuje *SOAP* servis mora sadržavati podatkovne strukture (ukoliko se prenose kompleksni podaci), ulazne i izlazne tipove podataka, operacije koje obrađuju navedene podatke te tehničke detalje vezane uz povezivanje sa samim servisom. Christensen, Curbera, Meredith i Weerawarana u službenoj dokumentaciji *WSDL*-a navode sljedeće oznake koje svaki ispravni *WSDL* mora imati: tipovi podataka (*engl. types*), poruke (*eng. message*), vrsta mrežnog priključka *(engl. portType*), način povezivanja (*engl. binding*) te na kraju servis (*engl. service*). Svaka od navedenih oznaka pod sobom sadrže dodatne oznake kojima definiraju sve potrebne parametre *SOAP* servisa, a isti su opisani i pojašnjeni na danom primjeru.

Generirani *WSDL* sadrži cijelu definiciju koja se nalazi i u početnom *XSD*-u, ali proširen je dodatnim parametrima koji su opisani u nastavku. Važno je napomenuti da je prikazani *WSDL* dokument samo dio cijelog *WSDL*-a iz razloga što se veći broj oznaka i atributa ponavlja, pa nisu potrebni za opisivanje dokumenta.

```
<wsdl:definitions
   xmlns:wsdl=http://schemas.xmlsoap.org/wsdl/
   xmlns:sch=http://foi.hr/anddanzan/posluzitelj/soap/podaci
   xmlns:soap=http://schemas.xmlsoap.org/wsdl/soap/
   xmlns:tns=http://foi.hr/anddanzan/posluzitelj/soap/podaci
   targetNamespace="http://foi.hr/anddanzan/posluzitelj/soap/podaci">
   <wsdl:types>
      <xs:schema
         xmlns:xs=http://www.w3.org/2001/XMLSchema
         elementFormDefault="qualified"
         targetNamespace="http://foi.hr/anddanzan/posluzitelj/soap/podaci">
         ...
         <xs:element name="dohvatiVrsteTransakcijaZahtjev">
            <xs:complexType>
               <xs:complexContent>
                  <xs:restriction base="xs:anyType"/>
               </xs:complexContent>
            </xs:complexType>
         </xs:element>

         <xs:element name="dohvatiVrsteTransakcijaOdgovor">
            <xs:complexType>
               <xs:sequence>
                  <xs:element maxOccurs="unbounded" name="vrsteTransakcija"
                              type="tns:vrstaTransakcijeSoap"/>
               </xs:sequence>
            </xs:complexType>
         </xs:element>
         ...
         <xs:complexType name="vrstaTransakcijeSoap">
            <xs:sequence>
               <xs:element name="id" type="xs:int"/>
               <xs:element name="naziv" type="xs:string"/>
               <xs:element name="opis" type="xs:string"/>
            </xs:sequence>
         </xs:complexType>
         ...
```

```
      </xs:schema>
   </wsdl:types>
   ...
   <wsdl:message name="dohvatiVrsteTransakcijaZahtjev">
      <wsdl:part element="tns:dohvatiVrsteTransakcijaZahtjev"
                 name="dohvatiVrsteTransakcijaZahtjev">
      </wsdl:part>
   </wsdl:message>

   <wsdl:message name="dohvatiVrsteTransakcijaOdgovor">
      <wsdl:part element="tns:dohvatiVrsteTransakcijaOdgovor"
                 name="dohvatiVrsteTransakcijaOdgovor">
      </wsdl:part>
   </wsdl:message>

   <wsdl:portType name="SoapPort">
      ...
      <wsdl:operation name="dohvatiVrsteTransakcija">
         <wsdl:input message="tns:dohvatiVrsteTransakcijaZahtjev"
                       name="dohvatiVrsteTransakcijaZahtjev">
         </wsdl:input>
         <wsdl:output message="tns:dohvatiVrsteTransakcijaOdgovor"
                        name="dohvatiVrsteTransakcijaOdgovor">
         </wsdl:output>
      </wsdl:operation>
      ...
   </wsdl:portType>
   <wsdl:binding name="SoapPortSoap11" type="tns:SoapPort">
      <soap:binding style="document"
                    transport="http://schemas.xmlsoap.org/soap/http"/>
       ...
      <wsdl:operation name="dohvatiVrsteTransakcija">
         <soap:operation soapAction=""/>
         <wsdl:input name="dohvatiVrsteTransakcijaZahtjev">
            <soap:body use="literal"/>
         </wsdl:input>
         <wsdl:output name="dohvatiVrsteTransakcijaOdgovor">
            <soap:body use="literal"/>
         </wsdl:output>
      </wsdl:operation>
      ...
   </wsdl:binding>

   <wsdl:service name="SoapPortService">
      <wsdl:port binding="tns:SoapPortSoap11" name="SoapPortSoap11">
         <soap:address location="http://localhost:8080/soap"/>
      </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Osnovna oznaka svakog ispravnog *WSDL* dokumenta je oznaka *definitions*, a naziv je dobila po tome što je *WSDL* skup definicija koje opisuju web servis. Kao korijenska oznaka, *definitions* sadrži imenske prostore kojima definira dozvoljene oznake i atribute. Osim osnovnih imenskih prostora poput *targetNamespace* i *xmlns:tns* koje se nalaze i u *XSD* shemi, *WSDL* sadrži vlastitu shemu kojom propisuje prethodno spomenut oznake poput: *portType*-a, *types*-a i ostalih. Kod navedenih imenskih prostora najvažnije je spomenuti imenski prostor `wsdl` koji

definira osnovne gradivne elemente *WSDL* dokumenta te `soap` pomoću kojih se definiraju *SOAP* poruke i detalji komunikacije sa servisom.

Prva oznaka po redu unutar korijenske oznake *definitions* je oznaka *types* čija je namjena definirati tipove podataka odnosno podatkovne strukture koje se koriste u razmjeni podataka (Christensen i ostali, 2001). Ukoliko malo bolje pogledamo strukturu unutar same oznake *types* možemo uočiti da je riječ o početnoj *XSD* datoteci koja je korištena za generiranje klasa vezanih uz razmjenu podataka. Definirane strukture podataka nalaze se u imenskom prostru *xmlns:tns* te se preko iste *tns* kratice referenciraju u oznakama nakon.

Nakon što su definirane strukture podataka te ulazni i izlazni podaci (zahtjevi i odgovori) potrebno je konstruirati poruke iz osnovnih podatkovnih struktura. Prethodno opisana oznaka *types* definira je korištene strukture podatka, a oznaka *messages* iste te strukture povezuje sa u poruke koje će se razmjenjivati sa servisom. Svaka poruka definira se pomoću oznake *part* koja, kako joj i samo ime kaže, određuje dio poruke. Dio poruke sadrži referencu na tip podataka, odnosno u našem slučaju zahtjev ili odgovor te jedinstveno ime. (Christensen i ostali, 2001)

Kreiranjem definicija tipova podataka i potom poruka koje će se razmjenjivati potrebno je definirati oznaku *portType* i njen sadržaj. Oznaka p*ortType* definira apstraktne metode s ulaznim i izlaznim parametrima koje servis stavlja na korištenje. Operacije su definirane oznakom *operation* i jedinstvenim imenom operacije. Prikazani primjer *WSDL*-a sadrži definiciju operacije za dohvaćanje svih vrsta transakcija koja nema ulaznih parametara, a kao odgovor vraća sve vrste transakcija u sustavu. Poruka zahtjeva na temelju koje se poziva operacija servisa navodi se unutar oznake *input*, dok odgovor operacije i servisa definiran je oznakom *output*, a jedinstveno ime *inputa* koristi se za povezivanje metode unutar pristupne točke*.* (Christensen i ostali, 2001)

Format *SOAP* poruke te detalji o protokolima za razmjenu podataka za određeni *portType* definiran je u tijelu oznake *binding* (Christensen i ostali, 2001). Oznaka *binding* sadrži referencu na *portType* koji popisuje i definira sve operacije servisa te za iste određuje format i protokol slanja *SOAP* poruka. Oznaka povezivanja unutar sebe prvo definira *SOAP* povezivanje i to na način da je određuje sljedeće elemente (Christensen i ostali, 2001):

- *style* – može poprimiti vrijednosti *rpc* (podaci se prenose kao poziv udaljene metode) ili *document* kao u našem slučaju (podaci se prenose *SOAP* porukama koje sadrže zaglavlje, tijelo i omotnicu) i
- *transport* – definira protokol kojim se prenose podaci, a u naše slučaju to je *HTTP* protokol.

Nakon što je definiran način povezivanja na razini servisa, potrebno je za svaku operaciju i njene ulazne i izlazne podatke definirati *SOAP* postavke. Za svaku operaciju definiranu unutar *portType*-a zadane su sljedeće oznake i postavke (Christensen i ostali, 2001):

- *soap:operation* - definira vrstu operacije (npr. *rpc*).
    - `SOAPAction` – atribut dio je *SOAP* zaglavlja i zadužen je za definiranje akcije koja će se pokretati prilikom poziva operacije. Vrijednosti koje može poprimiti su (Box i ostali, 2000):
        - `"action-uri"` ili `"myapp.sdl"` – definirana akcija
        - `""` – naznačava da je akcija definirana u samom imenu operacije
        - `bez vrijednosti` - akcija je definirana u samom uniformnom identifikatoru resursa (*engl. Uniform Resource Identifier – URI*) *HTTP* zahtjeva.
- *soap:body* – oznaka zadužena za definiranje kako će se dijelovi (*part*) poruke definirane u *WSDL*-u prikazivati unutar same *SOAP* poruke. Zadani atribut *use* s vrijednošću *literal*, kao što i sam prijevod riječi govori, definira da se referencirani tip podataka koristi doslovno kako je zadan u *WSDL*-u (nema apstraktnih tipova podataka).

Posljednja oznaka svakog *WSDL* dokumenta definira sam SOAP servis odnosno objedinjuje sve pristupne točke servisa na koje se možemo spojiti. Oznaka *port* referencira definirani *wsdl:binding* te za isti određuje *URI* preko kojega možemo pristupiti *SOAP* servisu. Određivanje samog *URI*-a izvedeno je postavljanjem atributa *location* oznake *address* iz *SOAP* imenskog prostora. (Christensen i ostali, 2001)

6.3.8.6. *SOAP* poruke servisa

Sve klase koje su dosad generirane i kreirane koriste se u jednu svrhu razmjenu poruka između klijenta i poslužitelja odnosno pristupne točke. Svaka komunikacija vrši se preko *SOAP* poruka temeljenih na *XML* jeziku oznaka s definiranim imenskim prostorima za tumačenje oznaka.

Svaka poruka (*engl. SOAP message*), bila zahtjev ili dogovor, mora sadržavati „omotnicu“ (*engl. SOAP envelope*) i sam sadržaj, odnosno tijelo poruke (*engl. SOAP body*), dok je zaglavlje (*engl. SOAP header*) neobavezno. *SOAP* omotnica definira imenske prostore (*engl. namespace*) pomoću kojih, kako je već rečeno, propisuje dozvoljene XML oznake i atribute. Osnovni imenski prostor *SOAP* poruka definira se kao `SOAP-ENV` imenski prostor koja se koristi za definiranje pravila ispravnog *SOAP XML*-a. Kako *SOAP* omotnica u sebi sadrži tijelo poruke s podacima koji se razlikuju od sustava do sustava, mora postojati imenski

prostor oznaka za prijenos podataka, u našem slučaju `xmlns:pod` *namespace.* Omotnica može definirati i dodatne atribute poput stila znakovne stranice (*engl. styleEncoding*), atribut za već dozvoljene *XML* oznake *(engl. tag*) i njihovo tumačenje pri serializaciji (čitanju poruke). (Box i ostali, 2000).

Unutar SOAP omotnice nalazi se zaglavlje, ono je neobavezno te omogućava proširenje poruke u pogledu uzajamno dogovorenih dodatnih mogućnosti definiranih od strane poslužitelja. U prikazanom primjer zaglavlja možemo uočiti da ono ne sadrži dodatne postavke te je iz tog razloga samo prazna oznaka u svrhu očuvanja strukture *SOAP* poruke. Za oznaku zaglavlja možemo također zadati dodatne atribute poput (Box i ostali, 2000; Mitra & Lafon, 2007): `role` (definira daljnju obradu poruke), `mustUnderstand` (zastavica koja naglašava da podaci zaglavlju moraju biti obrađen) te `relay` (podaci zaglavlja se ne obrađuju nego prosljeđuju).

```
POST http://localhost:8080/soap HTTP/1.1
Accept-Encoding: gzip,deflate
Content-Type: text/xml;charset=UTF-8
SOAPAction: ""
Content-Length: 263
Host: localhost:8080
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.1.1 (java 1.5)
Cookie: JSESSIONID=9E708E9F20F1D88B658677E71155D82A
Cookie2: $Version=1
Authorization: Basic YW5kZGFuemFuOmxvemlua2E=

<SOAP-ENV:Envelope
       xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
       xmlns:pod="http://foi.hr/anddanzan/posluzitelj/soap/podaci">
       <SOAP-ENV:Header/>
              <SOAP-ENV:Body>
                     <pod:dohvatiVrsteTransakcijaZahtjev/>
       </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Prikazani *XML* dokument primjer je *SOAP* poruke zahtjeva s pripadajućim *HTTP* zaglavljem koji se koriste za dohvaćanje svih vrsta transakcija u sustavu. *HTTP* zaglavlje značajno je za prijenos *SOAP* poruka, au ovom slučaju i za autorizaciju odnosno autentifikaciju korisnika. Sve *SOAP* poruke koje se šalju pristupnoj točki poslane se pomoću *HTTP POST* metode obliku `text/xml` sadržaja. `SOAPAction` je prazan *string* tip podataka što, kao što je već spomenuto, naznačava da se prema nazivu zahtjeva odabire operacija servisa (metoda servisa definira da zaprima `dohvatiVrsteTransakcijaZahtjev` tip zahtjeva). Vrlo važan parametar *HTTP* zaglavlja je `Authorization` parametar, a isti prosljeđuje podatke za osnovnu autentifkaciju korisnika. Parametar za autorizaciju sadrži dvije vrijednosti: `Basic` vrijednost koja naznačava da je riječ o osnovnoj autentifikaciji te enkriptirane podatke

`YW5kZGFuemFuOmxvemlua2E=`. Enkriptirani podaci kreiraju se na sljedeći način, korisničko ime i lozinka povezani su dvotočkom (*korisnickoIme:lozinka*) te potom primjenom unaprijed dostupnih metoda trenutni znakovi enkodiraju se u base-64 *ASCII* (*engl. American Standard Code for Information Interchange*) niz znakova što u konačnici rezultira nerazumljivim nizom.

```
HTTP/1.1 200
Set-Cookie: JSESSIONID=EF2F840A5B89CFB04A09DC343616FEDD; Path=/; HttpOnly
Access-Control-Allow-Credentials: true
Access-Control-Allow-Methods: POST, PUT, GET, OPTIONS, DELETE
Access-Control-Max-Age: 3600
Access-Control-Allow-Headers: Content-Type, Accept, X-Requested-With, remember-me
Accept: text/xml, text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
SOAPAction: ""
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Frame-Options: DENY
Content-Type: text/xml;charset=utf-8
Content-Length: 763
Date: Tue, 16 Jul 2019 18:55:31 GMT

<SOAP-ENV:Envelope
     xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
     <SOAP-ENV:Header/>
     <SOAP-ENV:Body>
          <ns2:dohvatiVrsteTransakcijaOdgovor
             xmlns:ns2="http://foi.hr/anddanzan/posluzitelj/soap/podaci">
                    <ns2:vrsteTransakcija>
                           <ns2:id>1</ns2:id>
                           <ns2:naziv>Uplata</ns2:naziv>
                           <ns2:opis>Uplata iznosa na korisnikov račun.</ns2:opis>
                    </ns2:vrsteTransakcija>
                    <ns2:vrsteTransakcija>
                           <ns2:id>2</ns2:id>
                           <ns2:naziv>Isplata</ns2:naziv>
                           <ns2:opis>Isplata iznosa s korisnikovog računa.</ns2:opis>
                    </ns2:vrsteTransakcija>
                    ...
     </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Posljednji dokument prikazan iznad ovog odlomka primjer je *SOAP* odgovora operacije za dohvaćanje svih vrsta transakcija. Metoda pristupne točke pozvana je na temelju prethodno prikazanog `dohvatiVrsteTransakcijaZahtjev` zahtjeva te kao odgovor vraća tip odgovora `dohvatiVrsteTransakcijaOdgovor`. Svaki odgovor sadrži *HTTP* zaglavlje s obzirom da se odgovor *SOAP* operacija također šalje kroz *HTTP* protokol, ali u ovom slučaju nema posebnih parametara zaglavlja. Prikazani dokument sadrži korijensku oznaku definira imenski prostor `ns2` koji propisuju dozvoljene oznake za prijenos podataka, a unutar iste nalaze se objekti s podacima vrsta transakcija. Veći broj zapisa u odgovoru primjer je kako dispečer *SOAP* poruka *Java* listu objekata mapira u niz oznaka i u njima pohranjuje podatke iz objekta.

6.3.9. Implementacija REST web servisa

U disertaciji pisanoj na Sveučilištu Kalifornija u gradu Irvine, Roy Thomas Fielding 2000.-e godine stvorio je novu arhitekturu web servisa. Vođen samim razvojem World Wide Web-a (WWW), Roy Fielding uočio je prednosti i nedostatke u njegovom svakodnevnom radu i sve većem broju korisnika i povećanoj kompleksnosti podataka. Fielding navodi kako distribuirana hipermedijska arhitektura onog vremena počiva na tri načina dohvaćanja resursa: renderiranje multimedijskih podataka na mjestu pohrane te slanje pripremljene multimedije klijentu, učahurivanje hipermedijskih podataka sa samim procesorom za renderiranje (*engl. rendering engine*) te slanje podataka s opisnim podacima (*engl. metadata*) kako se hipermedija na klijentskoj strani mora obraditi za prikaz

REST arhitektura temeljena je na prethodno navedenim opcijama, a opisana kroz sljedećih šest ograničenja (Fielding, 2000):

- **Klijent-Poslužitelj** (*engl. Client-Server*) – ideja web-a počiva na komunikaciji klijent-poslužitelj. Uvođenjem poslužitelja, obrada podataka više nije na razini same baze podataka, a slanje *rendering engine*-a s podacima klijent preuzima veliki dio obrade čime je poboljšana skalabilnost.
- **Bez stanja** (*engl. Stateless*) – dohvaćanje multimedijskih podataka i razmjena metapodataka o istima uklanja stanje na poslužitelju jer su sad svi potrebni podaci za obradu u samom zahtjevu odnosno odgovoru. Navedeno ograničenje poboljšalo je:
    - vidljivost (*engl. Visibility*) - poslužitelj „gleda“ sve u kontekstu jednog zahtjeva,
    - pouzdanost (*engl. Reliability*) - nema pohrane podatka na poslužitelju čime je lakše vratiti sustav u stanje u kojem je bio te
    - skalabilnost (*engl. Scalability*) - poslužitelj ne mora alocirati resursi za pohranu podataka o klijentima i statusu komunikacije s istima.
- **Privremena memorija** (*engl. Cache*) – koliko god su poboljšane performanse klijenta i poslužitelja i dalje je najslabija točka sam internetska veza točnije latentnost mreže. Uvođenjem privremene memorije na klijentskoj strani omogućena je privremena pohrana podataka koji se često dohvaćaju, a poslužitelj definirana koji podaci mogu biti pohranjeni u *cache.* Spremanjem podataka smanjuje se broj zahtjeva na server čime je poboljšana efikasnost, skalabilnost te performanse u pogledu manje ovisnosti o samoj mreži.
- **Jedinstveno sučelje** (*engl. Uniform Interface*) – poslužitelj nudi jedinstveno i standardizirano sučelje za pristup podacima. Navedeni način omogućava jednostavniju arhitekturu poslužitelja, a klijent i poslužitelj nisu više „usko“ povezani. Jedina mana

jedinstvenog sučelja je ta da se korisnik mora prilagoditi sučelju odnosno znati koje podatke zaprima.

- **Slojeviti sustav** (*engl. Layered System*) – po uzoru na *pipe-and-filter* arhitekturu, *REST* primjenjuje implementaciju sustava pomoću slojeva (manjih servisa), ali s dvosmjernom komunikacijom za razliku od *pipe-and-filter*-a. Svaki sloj zna samo za sloj ispod sebe, a pruža uslugu sloju iznad u hijerarhiji. Slojevita arhitektura umanjuje kompleksnost sustava jer sloj mora znat samo za sloj koji direktno koristi, a također omogućena je ponovna iskoristivost i zamjenjivost jer servis nije „čvrsto“ povezan s cijelim sustavom.
- **Kod na zahtjev** (*engl. Code-On-Demand*) – način je dohvaćanja hipermedijskih podataka s dodatnim *rendering engine*-om za obradu istih. Kod na zahtjev koristi se u slučajevima kada klijent ima multimedijske podatke, ali nema potrebne alate i znanje za obradu stoga mora sa servera dohvaćati aplikaciju (*engl. Applet*) ili skriptu za obradu istih. Jednostavnost je umanjena korištenjem koda na zahtjev, ali skalabilnost poslužitelja je povećana jer se obrada prebacuje na klijentsku stranu. Ograničenje koda na zahtjev, Roy Fielding definirao je kao opcionalno iz razloga što servis ne može znati detaljne podatke o sustavu koji će zaprimiti kod za obradu, ali ukoliko znamo detalje implementacija klijenta vrlo jednostavno možemo na zahtjev slati dodatne klase koje mogu proširiti klijenta dodatnim funkcionalnostima.

6.3.9.1. Implementacija *REST* upravljača

Implementacija *REST* web servisa znatno je jednostavnija od implementacije *SOAP*-a iz razloga što *REST* predstavlja arhitekturu web servisa koja nema propisana ograničenja u pogledu razmjene poruka. *SOAP* predstavlja zasebni protokol za razmjenu podataka s web servisom što u konačnici zahtjeva dodatne mikroservise, konfiguracije te oblikovanje poruka prema pravilima protokola. *REST*, za razliku od *SOAP*-a, ne propisuje dodatna pravila ili način komunikacije već najbolju praksu za kreiranje web servisa za razmjenu hipermedijskih podataka. Implementacija *REST* web servisa temeljena je na *HTTP* zahtjevima i odgovorima najčešće u *JSON* ili rjeđe *XML* formatu, za razliku od *SOAP*-a koji striktno definira *XML.* (Cornelißen i ostali, 2018)

```
@RestController
public class TransakcijaRestKontroler
{
    private final TransakcijaFasada transakcijaFasada;

    public TransakcijaRestKontroler(final TransakcijaFasada transakcijaFasada)
    {
        this.transakcijaFasada = transakcijaFasada;
    }

    @RequestMapping(value = "/rest/transakcije/{korisnickoIme}", method = GET)
    public ResponseEntity<Object> dohvatiTransakcije(
```

```
        @PathVariable("korisnickoIme") final String korisnickoIme,
        @RequestParam(name = "page", required = false) final Integer stranica,
        @RequestParam(name = "size", required = false) final Integer velicina,
        @RequestParam(name = "sort", required = false) final String sort)
    {
        final OdgovorRest odgovorRest =
                new OdgovorRest("/rest/transakcije/{korisnickoIme}", new Date());

        Pageable stranicenje = null;
        if (stranica != null && velicina != null)
        {
            if (StringUtils.isNotEmpty(sort))
            {
                final String[] sortParams = sort.split(",").length == 2
                        ? sort.split(",") : new String[] { sort };
                final Sort sortiranje =
                 new Sort(Sort.Direction.fromString(sortParams[1]), sortParams[0]);
                stranicenje = PageRequest.of(stranica, velicina, sortiranje);
            }
            else
            {
                stranicenje = PageRequest.of(stranica, velicina);
            }
        }

        final Optional<Page<TransakcijaEntitet>> korisnikoveTransakcijeOpt =
                this.transakcijaFasada
                .dohvatiKorisnikoveTransakcije(korisnickoIme, stranicenje);
        if (!korisnikoveTransakcijeOpt.isPresent())
        {
            odgovorRest.setPoruka("Transakcije za korisničko ime '" + korisnickoIme
                                  + "' ne postoje.");
            odgovorRest.setIzvrseno(false);
        }
        else
        {
            odgovorRest.setIzvrseno(true);
            odgovorRest.setSadrzaj(korisnikoveTransakcijeOpt.get());
        }

        return ResponseEntity.ok(odgovorRest);
    }
    ...
```

REST upravljači, kontroleri *(engl. controller)* ekvivalent su pristupnim točkama u implementaciji *SOAP* servisa s jedinom razlikom što u konačnici *REST* kontroleri čine aplikacijsko programsko sučelje (*engl. Application Programming Interface - API*) (Cornelißen i ostali, 2018). Primjer iznad prikazuje *API* za rad s transakcijama, a operacija unutar samog kontrolera koristi se za dohvaćanje transakcija korisnika.

Osnovna anotacija koja definira svaki *REST* web servis je anotacija `@RestController` koja ujedno objedinjuje anotacije `@ResponseBody` te `@Controller`. `@Controller` anotacija, kao što je već pojašnjeno, kreira *Java bean* u aplikacijskom kontejneru na serveru, dok pomoću anotacije `@ResponseBody` na razini klase definirano je da svaka metoda unutar iste generira web (*HTTP*) odgovor.

Razlika operacija *SOAP* i *REST* operacija je u samom pozivu iste, operacije *REST* web servisa pozivaju se na temelju vrste *HTTP* zahtjeva te same putanje na kojoj je dostupna.

Anotacijom `@RequestMapping` povezujemo operaciju servisa na putanju pomoću atributa *value,* a atributom *method* zadajemo vrstu zahtjeva odnosno je li riječ o dohvaćanju podataka (*GET*), zapisivanju (*POST*), brisanju (*DELETE*) ili ažuriranju (*PUT*). (Cornelißen i ostali, 2018)

Parametri operacija u *SOAP* servisima moraju biti zadani unutar samog zahtjeva, dok *REST* servisi koriste prednosti primjene *HTTP* protokola te isti koriste za prosljeđivanje parametara operaciji. Parametri *REST* metode mogu biti proslijeđeni na sljedeće načine:

- `@PathVariable` anotacija koristi se u načinu dohvaćanja pomoću putanje i to na način da se u samoj putanji operacije unutar vitičastih zagrada zadaje varijabla (`rest/transakcije/{korisnickoIme}`). *Spring Boot* parametar iz putanje preslikava u varijablu u nastavku anotacije (`@PathVariable("korisnickoIme") final String korisnickoIme`) te ista postaje dostupna unutar metode.
- `@RequestParam` anotacija omogućava pristupanje podacima proslijeđenim pomoću samog zaglavlja. Preuzimanje vrijednosti željenog parametra iz zaglavlja izvršava se na isti način kao kod prosljeđivanja vrijednosti preko putanje (`@RequestParam("sort") final String sort`).
- `@RequestBody` anotacija omogućuje dohvaćanje podataka kao parametra metode preslikavanjem cijelog tijela *HTTP* poruke. Razvojni okvir *SpringBoot* sam po sebi sadrži veliki broj servisa koji olakšavaju rad i programiranje, pa tako i postoji servis `HttpMessageConverter` koji je zadužen za preslikavanje *HTTP* poruka u *Java* klase odnosno objekte. Navedeni servis zaprimljeni sadržaj tijela poruke preslikava u objekt definiran nakon anotacije (`@RequestBody final VrstaTransakcijeEntitet novaVrsta`).

Metode *REST* servisa implementirane u poslužiteljskoj aplikaciji kreirane su na način da konstruiraju i vraćaju objekt *HTTP* odgovora. Klasa `ResponseEntity` koristi se za kreiranje prethodno spomenutog odgovora, a ista omogućava postavljanje svih informacija koje standardni *HTTP* odgovor sadrži. Kako bi se odgovor servisa proslijedio preko *HTTP* odgovora, korištena je klasa za prijenos podataka (*engl. Data Transfer Object - DTO*).

6.3.9.2. REST odgovor servisa

DTO klasa `OdgovorRest` koristi se kako bi operacija web servisa uz podatke mogla vratiti informacije o izvođenju same operacije.

```
public class OdgovorRest
{
    @JsonProperty("sadrzaj")
    private Object sadrzaj;

    @JsonProperty("operacija")
```

```
    private String operacija;

    @JsonProperty("vrijemeIzvrsavanja")
    @JsonFormat(pattern="dd-MM-yyyy HH:mm:ss")
    private Date vrijemeIzvrsavanja;

    @JsonProperty("poruka")
    private String poruka;

    @JsonProperty("izvrseno")
    private boolean izvrseno;
    ...
```

Java klasa prikazana iznad odlomka primjer je *DTO* klase koja se koristi u *REST* kontrolerima poslužiteljske aplikacije. Na prethodnom primjeru metode za dohvaćanje transakcija korisnika, lista transakcija pohranjuje se u varijablu `Object sadrzaj` objekta klase `OdgovorRest` te uz podatke nadodane su sljedeće informacije: vrijeme izvršavanja, putanja do operacije, status izvršenosti operacije te poruka o izvršavanju. Anotacije `@JsonProperty` i `@JsonFormat` u klasi odgovora koriste se iz razloga što navedeni objekt mora biti preslikan u *JSON* u samom tijelu *HTTP* odgovora. Spomenute anotacije dio su *Jackson* biblioteke za rad s *JSON* formatom, a koriste se za definiranje *JSON* objekta odnosno definiranje formata zapisa za određeni *JSON* objekt.

```
{
   "sadrzaj":{...},
   "operacija":"/rest/transakcije/{korisnickoIme}",
   "vrijemeIzvrsavanja":"18-07-2019 18:42:02",
   "poruka":null,
   "izvrseno":true
}
```

JSON dokument prikazan iznad odlomka primjer je strukture odgovora *REST* operacije za dohvaćanje transakcija korisnika. Sam sadržaj odgovora nije prikazan iz razloga što je naglasak na samoj strukturi odgovora, ali isti se nalazi pod poljem `sadrzaj` odnosno `content`, dok `operacija`, `vrijemeIzvrsavanja`, `poruka` te `izvrseno` sadrže dodatne informacije o samom pozivu operacije i odgovoru. Primjenom opisane strukture odgovora, *REST* operacija u slučaju greške može vratiti odgovor s porukom greške te polje `izvršeno` postavljeno na `false` kako bi pozivatelj znao da operacija nije izvršena ispravno.

6.3.10. Primjena *Rest repositories* zavisnosti

REST repositories zavisnost modul je koji možemo uključiti u *Spring Boot* projekt kako bi olakšali kreiranje *REST* servisa. Primjenom navedene zavisnosti sve klase odnosno svi *Java bean*-ovi s anotacijom `@Repository` postaju javno dostupni (Webb i ostali, 2018). Uključivanjem navedene zavisnosti u projekt ovog diplomskog rada te postavljanjem `spring.data.rest.basePath` parametra u datoteci `application.properties` na

putanju `/restrepositories` svi repozitoriji javno su dostupni na zadanoj putanji. (Brisbin, Gierke, Turnquist, & Bryant, 2019)

```
{
  "_links": {
      "korisnikEntitets": {
          "href":"http://localhost:8080/restrepositories
                    /korisnikEntitets{?page,size,sort}",
          "templated": true
      },
      "transakcijaEntitets": {
          "href": "http://localhost:8080/restrepositories
                    /transakcijaEntitets{?page,size,sort}",
          "templated": true
      },
      "vrstaTransakcijeEntitets": {
          "href": "http://localhost:8080/restrepositories
                    /vrstaTransakcijeEntitets{?page,size,sort}",
          "templated": true
      },
      "profile": {
          "href": "http://localhost:8080/restrepositories/profile"
      }
  }
}
```

JSON dokument prikazan iznad predstavlja odgovor poslužiteljske aplikacije na *GET* zahtjev na korijensku putanju *rest repositories* servisa (u testnom slučaju riječ je o putanji `http://localhost:8080/restrepositories`). Iz priloženog možemo uočiti da *rest repositories* kreira *API* na putanjama imena entiteta sa sufiksom „s“ za množinu jer je riječ o operaciji koja radi sa svim entitetima. Prikazani *JSON* objekt `_links` vrlo je važna značajka primjene *rest repositories* zavisnosti iz razloga što je tu riječ o Hipertekstualnom aplikacijskom jeziku (*engl. Hypertext Application Language - HAL*). *HAL* je hipertekstualni jezik s nizom pravila i konvencija koji se koristi za definiranje poveznica u *JSON* dokumentima. Primjenom hiperpoveznica u dokumentu odgovora znatno je olakšan pristup i rad na klijentskoj strani jer svaki *JSON* koji sadrži kompleksne podatke u sebi sadrži i poveznicu za dohvaćanje dodatnih podataka poput vanjskih ključeva. U prikazanom dokumentu hiperoveznice `_links` *JSON* objekta sadrže dodatne parametre poput `{?page,size,sort}` iz razloga što sva repozitorij sučelja u ovom diplomskom radu proširuju `JpaRepository`. Navedeno sučelje objedinjuje više sučelja, među ostalim i sučelje `PagingAndSortingRepository<T,ID>` koje unaprijed priprema spomenute parametre i operacije za paginaciju i sortiranje. (Brisbin i ostali, 2019)

```
{
  "_embedded": {
      "transakcijaEntitets": [
          {
              "bankovniRacun": "HR8984890681102331690",
              "opis": "Uplata po dogovoru korisnika Pero Kos.",
              "datumKreiranja": "2019-06-02T08:06:48.000+0000",
              "datumTerecenja": "2019-06-04T08:06:48.000+0000",
              "iznos": 1500.45,
              "uplata": true,
```

```
            "_links": {
                "self": {
                    "href": "http://localhost:8080/restrepositories
                              /transakcijaEntitets/1"
                },
                "transakcijaEntitet": {
                    "href": "http://localhost:8080
                              /restrepositories /transakcijaEntitets/1"
                },
                "korisnikEntitet": {
                    "href": "http://localhost:8080
                              /restrepositories /transakcijaEntitets/1/
                              korisnikEntitet"
                  },
                  "vrstaTransakcijeEntitet": {
                      "href": "http://localhost:8080/restrepositories/
                              transakcijaEntitets/1/vrstaTransakcijeEntitet"
                }
            }
        },
        ...
    ]
  },
  "_links": {
      "self": {
          "href": "http://localhost:8080/restrepositories/
                    transakcijaEntitets{?page,size,sort}",
          "templated": true
      },
      "profile": {
          "href": "http://localhost:8080/restrepositories/profile/transakcijaEntitets"
      },
      "search": {
          "href": "http://localhost:8080/restrepositories/transakcijaEntitets/search"
      }
  },
  "page": {
      "size": 20,
      "totalElements": 4,
      "totalPages": 1,
      "number": 0
  }
}
```

Dokument iznad je *JSON* odgovor poslužiteljske aplikacije na *GET* zahtjev na link `http://localhost:8080/restrepositories/transakcijaEntitets`. Navedena putanja korijenska je putanja za rad s transakcijama te slanje zahtjeva na istu vraća sve entitete transakcije koji postoje u bazi podataka. Uz osnovne podatke koji se nalaze u *JSON* objektu pod ključem `_embedded`, *repositories* zavisnost dodaje i *JSON* objekte `_links` i `page`. *JSON* objekt `_links` sadrži sve dostupne putanje vezane uz entitet s kojim radimo, u ovom slučaju možemo pronaći putanju za pretraživanje entiteta ili putanju za detaljnu strukturu entiteta. (Brisbin i ostali, 2019)

```
{
  "alps": {
      "version": "1.0",
      "descriptor": [
          {
              "id": "transakcijaEntitet-representation",
              "href": "http://localhost:8080/restrepositories/profile/transakcijaEntitets",
              "descriptor": [
```

```
            {
                "name": "bankovniRacun",
                "type": "SEMANTIC"
            },
            {
                "name": "opis",
                "type": "SEMANTIC"
            },
            {
                "name": "datumKreiranja",
                "type": "SEMANTIC"
            },
            {
                "name": "datumTerecenja",
                "type": "SEMANTIC"
            },
            {
                "name": "iznos",
                "type": "SEMANTIC"
            },
            {
                "name": "uplata",
                "type": "SEMANTIC"
            },
            {
                "name": "korisnikEntitet",
                "type": "SAFE",
                "rt": "http://localhost:8080/restrepositories/profile
                          /korisnikEntitets#korisnikEntitet-representation"
            },
            {
                "name": "vrstaTransakcijeEntitet",
                "type": "SAFE",
                "rt": "http://localhost:8080/restrepositories/profile
                          /vrstaTransakcijeEntitets
                          #vrstaTransakcijeEntitet-representation"
            }
        ]
    },
    {
        "id": "create-transakcijaEntitets",
        "name": "transakcijaEntitets",
        "type": "UNSAFE",
        "rt": "#transakcijaEntitet-representation"
    },
    {
        "id": "get-transakcijaEntitets",
        "name": "transakcijaEntitets",
        "type": "SAFE",
        "descriptor": [
            {
                "name": "page",
                "type": "SEMANTIC",
                "doc": {
                    "format": "TEXT",
                    "value": "The page to return."
                }
            },
            {
                "name": "size",
                "type": "SEMANTIC",
                "doc": {
                    "format": "TEXT",
                    "value": "The size of the page to return."
                }
            },
            {
                "name": "sort",
```

```
                    "type": "SEMANTIC",
                    "doc": {
                        "format": "TEXT",
                        "value": "The sorting criteria to use to calculate the
                                  content of the page."
                    }
                }
            ],
            "rt": "#transakcijaEntitet-representation"
        },
        {
            "id": "get-transakcijaEntitet",
            "name": "transakcijaEntitet",
            "type": "SAFE",
            "rt": "#transakcijaEntitet-representation"
        },
        {
            "id": "delete-transakcijaEntitet",
            "name": "transakcijaEntitet",
            "type": "IDEMPOTENT",
            "rt": "#transakcijaEntitet-representation"
        },
        {
            "id": "update-transakcijaEntitet",
            "name": "transakcijaEntitet",
            "type": "IDEMPOTENT",
            "rt": "#transakcijaEntitet-representation"
        },
        {
            "id": "patch-transakcijaEntitet",
            "name": "transakcijaEntitet",
            "type": "UNSAFE",
            "rt": "#transakcijaEntitet-representation"
        },
        {
            "name": "findByKorisnikEntitet_KorisnickoIme",
            "type": "SAFE",
            "descriptor": [
                {
                    "name": "korisnickoIme",
                    "type": "SEMANTIC"
                }
            ]
        }
    ]
  }
}
```


Objekt `alps` *JSON* dokumenta prikazanog iznad skraćenica je naziva formata podataka za definiranje opisa semantike na aplikacijskoj razini odnosno *Application-Level Profile Semantics*. Slanjem *GET* zahtjeva na *rest resources* putanju s dodatkom *profil*, kao odgovor dobivamo prikazani dokument s metapodacima metoda servisa vezanih uz dani entitet.

Na samom početku dokumenta nalazi se *descriptor* (opisnik), a isti sadrži strukturu entiteta, semantiku polja entiteta te sve operacije koje mogu biti provedene nad entitetima. Prvi opisnik u nizu je reprezentacija klase entiteta čiji je repozitorij javno dostupan, a zatim se nižu opisnici metoda i parametara koje zaprimaju te odgovora koje vraćaju. Atribut `id` definira vrstu operacije, `name` određuje izvršava li se operacija na jednom entitetu ili na svima

(prepoznajemo prema množini ili jednini imena entiteta), potom `type` određuje je li metoda sigurna (samo čita) ili nesigurna (izmjenjuje podatke), također svaka operacija može imati svoj `descriptor` atribut za dodatne opisnike koji vrijede za tu operaciju te na kraju `rt` atribut određuje odgovor operacije. Ručno napisane operacije unutar repozitorij sučelja nalaze se na kraju niza, a one nemaju *id* atribut. Unutar *descriptor-a* mogu se nalaziti atribut *doc* koji pohranjuje ljudima razumljive opise operacija, a u ovom slučaju to sadrže samo metode sortiranja i paginacije. Dokumentacija metoda koje smo samostalno kreirali u repozitorij sučeljima morali bi opisati u zasebnom *properties* dokumentu. (Brisbin i ostali, 2019)

```
{
  "_links": {
      "findByKorisnikEntitet_KorisnickoIme": {
          "href": "http://localhost:8080/restrepositories
                    /transakcijaEntitets/search/
                    findByKorisnikEntitet_KorisnickoIme{?korisnickoIme}",
          "templated": true
      },
      "self": {
          "href": "http://localhost:8080/restrepositories/transakcijaEntitets/search"
      }
  }
}
```

Pristup linku unutar *JSON* objekta `search`, dohvaća *JSON* odgovor prikazan na slici iznad, a riječ je popisu svih ručno kreiranih metoda koje se koriste za pretraživanje entiteta. U ovom diplomskom radu, za transakcije kreirana je jedna metoda pretraživanja, a to je prema korisničkom imenu korisnika čija je transakcija (prema vrijednosti varijable objekta korisnik koji je vanjski ključ). Vrijednosti *JSON* objekta `page` sadrže informacije o paginaciji entiteta iz baze podataka. Navedene vrijednosti definiraju broj stranica koji je dohvaćen (`totalPages`), koliko je elemenata dohvaćeno (`totalElements`), broj trenutne stranice (`number`) te veličina jedne stranice (`size`). Navedeni atributi kreiraju se zbog prethodno spomenutog korištenja `JpaRepository` sučelja koje omogućava navedene operacije nad repozitorijem podataka.

6.4. Implementacija klijentske aplikacije

Klijentska aplikacija implementirana je kako bi korisnik pomoću intuitivnog grafičkog sučelja mogao pristupati web servisima na poslužiteljskoj aplikaciji. Svi podaci i poslovna logika implementirani su na poslužiteljskoj strani stoga ova aplikacija pojednostavljuje korisniku pregled i rad te prenosi podatke i zahtjeve na poslužiteljsku aplikaciju. Svaka stranica, osim prijave i registracije, sadrži tipku za prikaz *SOAP* ili *REST* poruka odnosno *XML* ili *JSON* sadržaja koji se izmjenjuje. Ovisno o vrsti poruka, klikom na plavu tipku pokraj naslova stranice otvara se skočni prozor s odgovorom poslužiteljske aplikacije te zahtjevom ukoliko isti postoji.

6.4.1. Tehnički detalji implementacije

Klijentska aplikacija, u pogledu strukture, implementirana je na jednak način kao i poslužiteljska aplikacija s jedinom razlikom, onom u načinu pristupa podacima. Obije aplikacije koriste fasade za poslovnu logiku, fasada komunicira sa mikroservisom kao posrednikom, a potom komunikacija ide na sloj za pristup podacima. Poslužiteljska aplikacija ima vezu na bazu podataka stoga u tom slučaju sloj za pristup podacima su repozitorij sučelja, dok klijentska aplikacija na istom sloju implementira klijente za web servise.

6.4.1.1. Implementacija SOAP klijenta

Implementacija *SOAP* klijenta provedena je kroz tri mikroservisa, svaki entitet sustava posjeduje svoj klijenta za slanje i zaprimanje *SOAP* poruka. Prvi korak u implementaciji klijenta je inicijalizacija *marshaller-a* odnosno mikroservisa za serijalizaciju i deserijalizaciju *XML* poruka. Programski kod prikazan ispod primjer je inicijalizacije navedenog mikroservisa, a isti je korišten u svim klasama klijenata.

```
public class SoapKlijentKonfiguracija
{
    @Bean
    public Jaxb2Marshaller marshaller()
    {
        final Jaxb2Marshaller marshaller = new Jaxb2Marshaller();
        marshaller.setContextPath("hr.foi.anddanzan.soap.podaci");

        return marshaller;
    }
}
```

Jedinu konfiguraciju koju mikroservis za preslikavanje *XML-a* zahtjeva je putanja paketa na kojoj se nalaze generirane klase iz početne *XSD* datoteke. Kao što je već opisano, na navedenoj putanji nalaze se klase zahtjeva, odgovora i klase za prijenos podataka sa svim potrebnim anotacijama za serijalizaciju i deserijalizaciju *XML* poruka.

```
@Service
public class VrstaTransakcijeSoapKlijent extends WebServiceGatewaySupport
{
    public VrstaTransakcijeSoapKlijent()
    {
        final AnnotationConfigApplicationContext context = new
             AnnotationConfigApplicationContext(SoapKlijentKonfiguracija.class);
        final Jaxb2Marshaller marshaller = context.getBean(Jaxb2Marshaller.class);

        this.setDefaultUri("http://localhost:8080/soap");
        this.setMarshaller(marshaller);
        this.setUnmarshaller(marshaller);
    }

    public Map<String, Object> dohvatiVrsteTransakcija()
    {
        final DohvatiVrsteTransakcijaZahtjev zahtjev =
             new DohvatiVrsteTransakcijaZahtjev();

        final DohvatiVrsteTransakcijaOdgovor odgovor =
```

```
            (DohvatiVrsteTransakcijaOdgovor) posaljiZahtjev(zahtjev);

        return AlatZaXml.oblikujOdgovor(zahtjev, odgovor, getMarshaller());

    }
    ...

    private Object posaljiZahtjev(final Object zahtjev)
    {
        final WebServiceTemplate template = getWebServiceTemplate();
        final ClientInterceptor[] interceptors =
             new ClientInterceptor[] { new SoapAutorizacijskiPresretac() };
        template.setInterceptors(interceptors);

        return template.marshalSendAndReceive(zahtjev);
    }
}
```

Nakon inicijalizacije potrebnog mikroservisa, implementacija *SOAP* klijenta vrlo je jednostavna. Klijenta mora sadržavati anotaciju `@Service` kako bi se u sustavu kreirao *Java bean*, a svaki klijent mora proširiti klasu `WebServiceGatewaySupport` koja je unaprijed nudi metode za slanje *SOAP* poruka. Navedena klasa omogućava `WebServiceTemplate` čija je namjena slanje poruka, a ista čini srži *Spring* klijenata web servisa. Nakon što je objekt zahtjeva kreiran, proslijeđen je metodi `posaljiZahtjev` koja poziva prethodno inicijalizirani *marshaller* te šalje zahtjev.

Implementacija sigurnosti na poslužiteljskoj aplikaciji provedena je u pogledu autentifikacije svakog zahtjeva. Kako bi se prilikom slanja *SOAP* poruka na poslužiteljsku aplikaciju mogli postaviti podaci za autentifikaciju kreiran je presretač (*engl. interceptor*) koji presreće *SOAP* poruke te u *HTTP* zaglavlje postavlja parametar.

```
public class SoapAutorizacijskiPresretac implements ClientInterceptor
{
    @Override
    public boolean handleRequest(final MessageContext messageContext)
             throws WebServiceClientException
    {

        final TransportContext context =
             TransportContextHolder.getTransportContext();
        final HttpUrlConnection connection =
             (HttpUrlConnection) context.getConnection();

        try
        {
            final Authentication autentifikacija =
                   SecurityContextHolder.getContext().getAuthentication();
            final String korisnickoIme = autentifikacija.getName();
            final String lozinka = autentifikacija.getCredentials().toString();

             connection.addRequestHeader("Authorization",
                   parametriZahtjeva(korisnickoIme, lozinka));
        }
        catch (final IOException e)
        {
            e.printStackTrace();
            return false;
```

```
        }
        return true;
    }

    private String parametriZahtjeva(final String korisnickoIme,
       final String lozinka)
    {
        final String auth = korisnickoIme + ":" + lozinka;
        final byte[] encodedAuth =
             Base64.encodeBase64(auth.getBytes(Charset.forName("US-ASCII")));

        return "Basic " + new String(encodedAuth);
    }
```

Programski kod prikazan iznad primjer je presretača *HTTP* zahtjeva s *SOAP* zahtjevom koji se šalju s klijentske aplikacije. Za svaki zahtjev dohvaćaju se podaci iz sigurnosnog konteksta *Spring Boot-a* te se isti spajaju u `Authorization` tekst za *HTTP* zaglavlje, kao što je opisano u poglavlju „*SOAP* poruke servisa“. Nakon što su provedeni svi navedeni postupci, zahtjev je poslan te odgovor dolazi u obliku klase odgovora generirane iz početnog *XSD-a.* Pomoćna klasa `AlatZaXML` i njezine metode `oblikujOdgovor` već zaprimljena *SOAP* poruka serijalizira se u obliku teksta kako bi se u konačnici prikazala na stranici.

6.4.1.2. Implementacija REST klijenta

Implementacija *REST* klijenta slična je implementaciji *SOAP* klijenta s jedinom razlikom u načinu postavljanja parametra za autentifikaciju. *SOAP* klijent koristi presretač zahtjeva dok *REST* klijenti pozivaju postojeći mikroservis za kreiranje web klijenta. Svaki zahtjev koji se šalje na *REST* servis poslan je pomoću `WebClient` klase, a kako bi se podaci za autentifikaciju nalazi u *HTTP* zaglavlju kreiran je mikroservis `RestKlijentInicijalizatorImpl` čija je zadaća dodati autorizacijsko zaglavlje prilikom slanja. Navedeni mikroservis koristi se za dohvaćanje klijenta za slanje *HTTP* zahtjeva s podacima ili bez podataka za autentifikaciju. Parametar autorizacije u zaglavlju kreira se na isti način kao što je opisno u poglavlju „*SOAP* poruke servisa“.

```
@Service
public class RestKlijentInicijalizatorImpl implements RestKlijentInicijalizator
{
    @Override
    public WebClient dohvatiAutoriziraniKlijent()
    {
        final Authentication autentifikacija =
             SecurityContextHolder.getContext().getAuthentication();
        final String korisnickoIme = autentifikacija.getName();
        final String lozinka = autentifikacija.getCredentials().toString();

        return WebClient.builder()
                        .defaultHeader(HttpHeaders.AUTHORIZATION,
                                       parametriZahtjeva(korisnickoIme, lozinka))
                        .build();

    }

    @Override
```

```
    public WebClient dohvatiNeautoriziraniKlijent()
    {
        return WebClient.builder().build();
    }

    private String parametriZahtjeva(final String korisnickoIme,
             final String lozinka)
    {
        final String auth = korisnickoIme + ":" + lozinka;
        final byte[] encodedAuth =
             Base64.encodeBase64(auth.getBytes(Charset.forName("US-ASCII")));

        return "Basic " + new String(encodedAuth);
    }
}
```

Programski kod iznad implementacija je mikroservisa za inicijalizaciju web klijenta. Podaci za prijavu dohvaćaju se iz sigurnosnog konteksta, kao što je slučaj kod *SOAP* presretača zahtjeva, a sam parametar `Authorization` kreirana je na isti način. Autorizirani klijent koristi se za metode koje su zaštićene prijavom, dok neautorizirani klijent mora biti korišten za metode prijave i registracije iz razloga što su iste dostupne neprijavljenom korisniku.

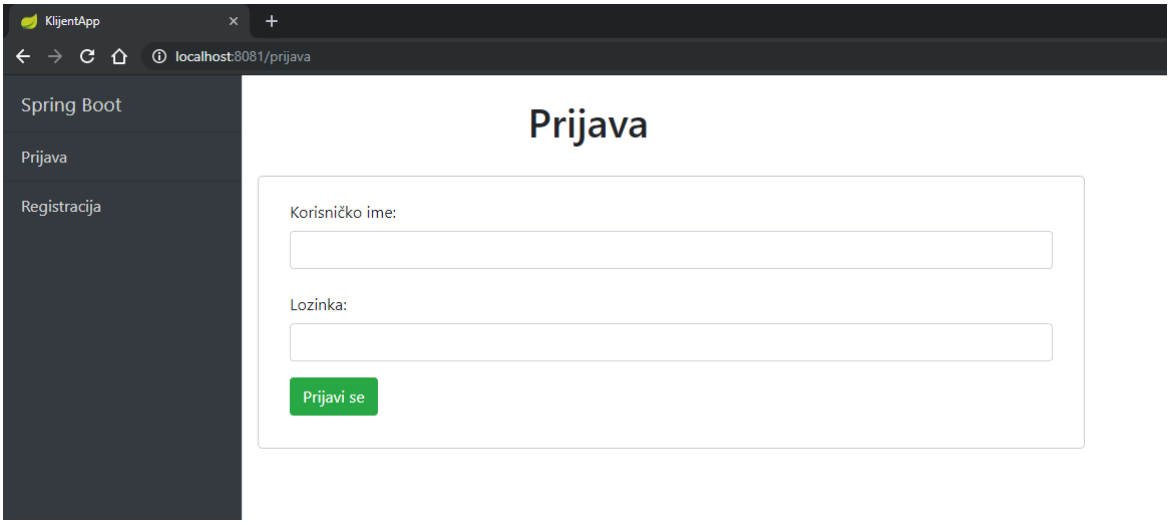
```
@Service
public class VrstaTransakcijeRestKlijent
{
    private final RestKlijentInicijalizator restKlijentInicijalizator;

    public VrstaTransakcijeRestKlijent(
       final RestKlijentInicijalizator restKlijentInicijalizator)
    {
        this.restKlijentInicijalizator = restKlijentInicijalizator;
    }

    public Map<String, Object> kreirajVrstuTransakcije(
             final NovaVrstaTransakcijeForma novaVrsta)
    {
        final WebClient client =
             this.restKlijentInicijalizator.dohvatiAutoriziraniKlijent();
        OdgovorRestVrstaTransakcije odgovorRest = null;

        if (client != null)
        {
            final String uri = "http://localhost:8080/rest/vrstaTransakcije/kreiraj";
            odgovorRest = client.method(HttpMethod.POST).uri(uri)
                                .body(BodyInserters.fromObject(novaVrsta))
                                .exchange().block()
                                .bodyToMono(OdgovorRestVrstaTransakcije.class)
                                .block();
        }

        return AlatZaJson.oblikujOdgovor(novaVrsta, odgovorRest);
    }
    ...
```

Primjer iznad odlomka dio je implementacije *REST* klijenta za pristup podacima vrsta transakcije. Svaki klijent pomoću inicijalizatora dohvaća autoriziranog ili neautoriziranog web klijenta za slanje *HTTP* zahtjeva. Metode za slanje kreiraju se korak po korak, prvo se postavlja *URI* parametar i *HTTP* metoda, potom pokrećemo slanje metodom `exchange()`. Nakon što

odgovor stiže s poslužiteljske aplikacije, tijelo poruke serijalizira se u klasu koju smo definirali u metodi `bodyoMono` te primjenom `AlatZaJson` i metode `oblikujOdgovor` kreiramo *JSON* odgovor i zahtjev za prikaz na stranici.

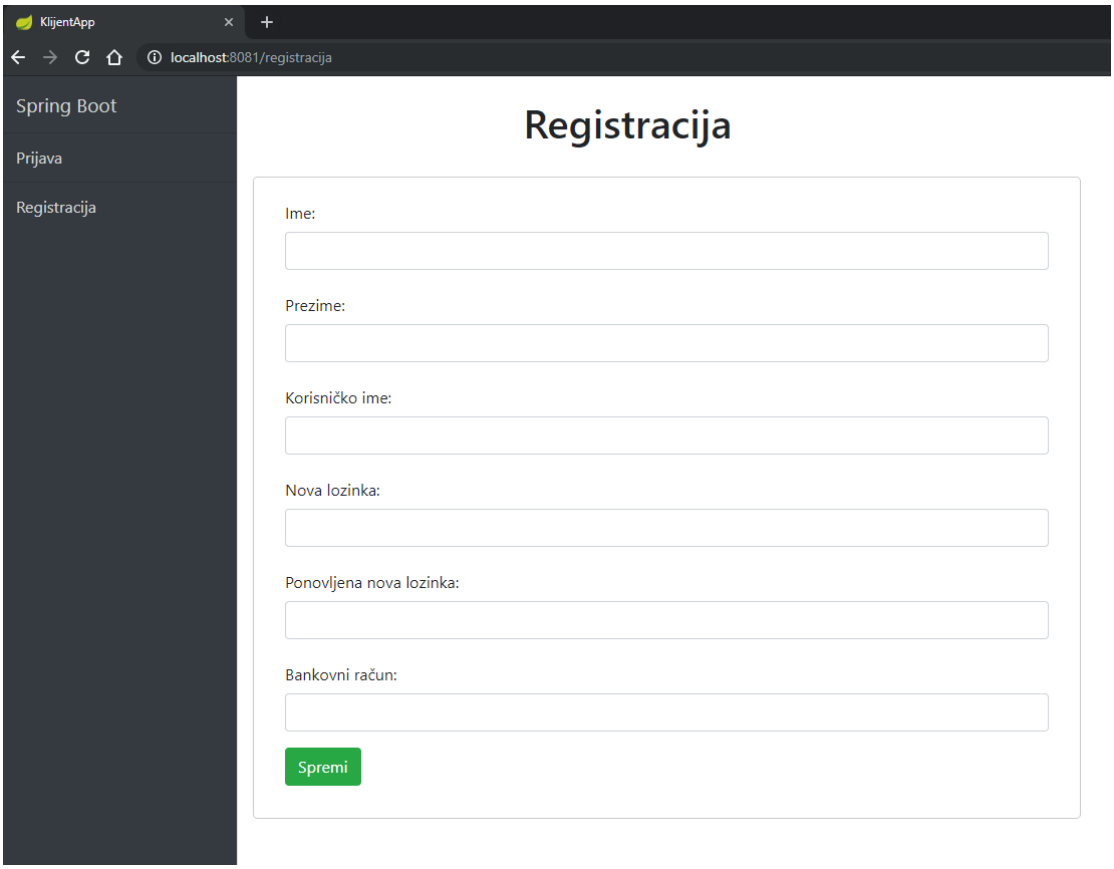
6.4.2. Stranice aplikacije i mogućnosti korisnika

6.4.2.1. Prijava, registracija i sigurnosna konfiguracija



Slika 18. Stranica za prijavu

Prilikom pristupanja aplikaciji korisnik nema pravo koristiti istu dokle god ne ispuni podatke za prijavu pomoću forme prikazane iznad. Primjenom *Spring security* zavisnosti neprijavljenom korisniku onemogućene su sve stranice aplikacije osim prijave i registracije. Ukoliko korisnik nema korisnički račun za aplikaciju, isti može kreirati pomoću stranice za registraciju prikazan ispod.



Slika 19. Stranica registracije

Zabrana pristupa sustavu implementirana je pomoću već spomenute *Spring security* zavisnosti. *Spring security* koristi se također na poslužiteljskoj strani za primjenu osnovne autorizacije i autentifikacije korisnika prilikom pristupanja metodama web servisa. Konfiguracija navedene zavisnosti temeljena je na zadavanju putanje te ovlasti odnosno akcija koje se moraju provesti prije samog pristupanja.

```
...
@Override
protected void configure(final HttpSecurity http) throws Exception
{
   http.csrf().disable().authorizeRequests()
      .antMatchers("/css/**", "/js/**", "/slike/**",
                   "/vendor/**";).permitAll()
      .antMatchers("/prijava", "/registracija").permitAll()
      .antMatchers("/pregledKorisnika", "/oduzmiAdminPrava",
                   "/dajAdminPrava", "/kreirajVrstuTransakcije",
                   "/azurirajVrstuTransakcije", "/obrisiVrsteTransakcija")
                    .hasAuthority("ADMIN")
      .anyRequest().authenticated()
      .and()
      .formLogin().loginPage("/prijava")
      .defaultSuccessUrl("/pocetna")
      .failureUrl("/prijava" + "?greska=true").and()
      .logout().logoutRequestMatcher(new AntPathRequestMatcher("/odjava"))
      .logoutSuccessUrl("/prijava" + "?odjava=true");
}
...
```

Programski kod prikazan iznad dio je sigurnosne konfiguracije klijentske aplikacije kojim su definirane dozvole korisnicima. Kao što su prijava i registracija jedine dostupne neprijavljenom korisniku u poslužiteljskoj aplikaciji, isto mora biti u klijentskoj aplikaciji kako bi neprijavljeni korisnik imao pristup stranicama. Osim navedenih stranica prijave i registracije, sve putanje vezane uz resurse u pogledu slika i vanjskih biblioteka (*CSS, JS ...*) također moraju biti odobrene za sve zahtjeve. Zaštita klijentske aplikacije temeljena je na istim principima kao i poslužiteljska, svi zahtjevi koji nisu dozvoljeni neautoriziranom korisniku primjenom `.permitAll()` zaštićeni su autentifikacijom korisnika primjenom metode `.anyRequest().authenticated()`. Prijavljeni korisnik može imati jednu od dvije moguće uloge u sustavu (administrator ili običan korisnik) stoga osim navedene zabrane pristupa dijelovima sustava, konfiguracija dozvoljava pristup određenim putanjama samo korisnicima s zadanim ulogama. Primjenom `hasAuthority("ADMIN")` konfiguracije, sve putanje navedene u `antMatchers` dijelu dostupne su korisniku s ulogom `ADMIN`.

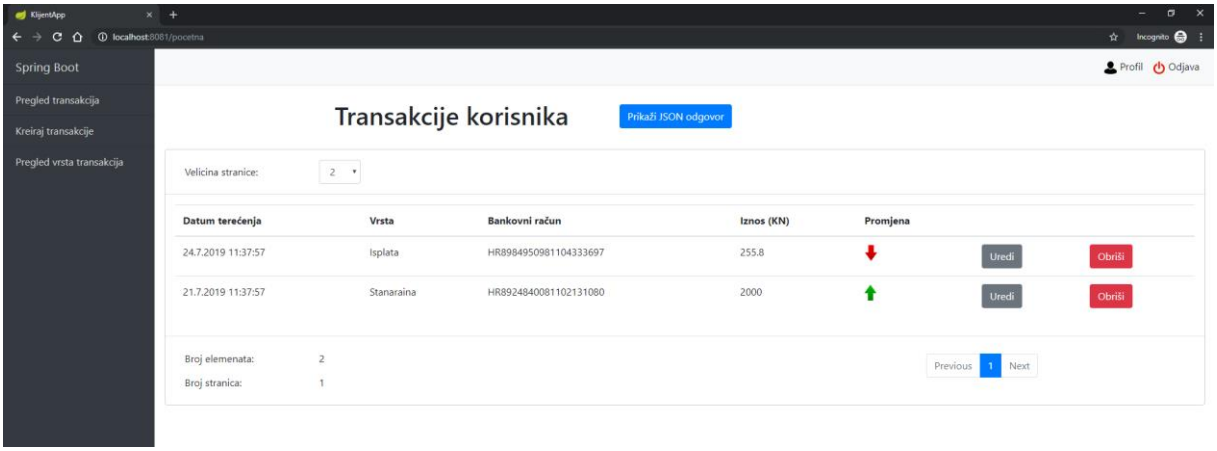
Prijava u klijentskoj aplikaciji različita je od one na poslužiteljskoj iz razloga što je sad primijenjena prijava pomoću forme. Primjenom `.formLogin().loginPage("/prijava")` metode postavljena je stranica na putanji `"/prijava"` kao stranica koja sadrži formu za prijavu. Nakon definiranja forme za prijavu postavlja ju se putanje u slučaju uspješne (`defaultSuccessUrl`) ili neuspješne (`failureUrl`) prijave. S obzirom da postoji prijava mora i odjava postojati u sustavu, a ista je definirana na putanji zadanoj u metodi `logoutRequestMatcher()` nakon koje slijedi putanja u slučaju uspješne odjave (`logoutSuccessUrl`).

```
...
<form method="POST" action="#" th:action="@{/prijava}">
   <div class="form-group">
      <label for="username" class="col-form-label">Korisničko ime:</label>
      <input type="text" class="form-control" id="username" name="username" >
   </div>
   <div class="form-group">
      <label for="password" class="col-form-label">Lozinka:</label>
      <input type="password" class="form-control" id="password" name="password" >
   </div>
   <div class="form-group">
      <button id="spremi" type="submit" class="btn btn-success"
             data-dismiss="modal">Prijavi se</button>
   </div>
</form>
...
```

Spring Boot razvojni okvir pruža veliki broj mikroservisa kojima možemo olakšati implementiranje funkcionalnosti, ali i primijeniti najbolju praksu *Spring* temeljenih aplikacija. Jedan on mikroservisa *Spring Boot-a* je `UserDetailsService` čija je osnovna namjena implementacija autentifikacije korisnika, kao što je opisano i u implementaciji na poslužiteljskoj aplikaciji. Isječak programskog koda iznad prikazuje formu za prijavu na putanji `/prijava`

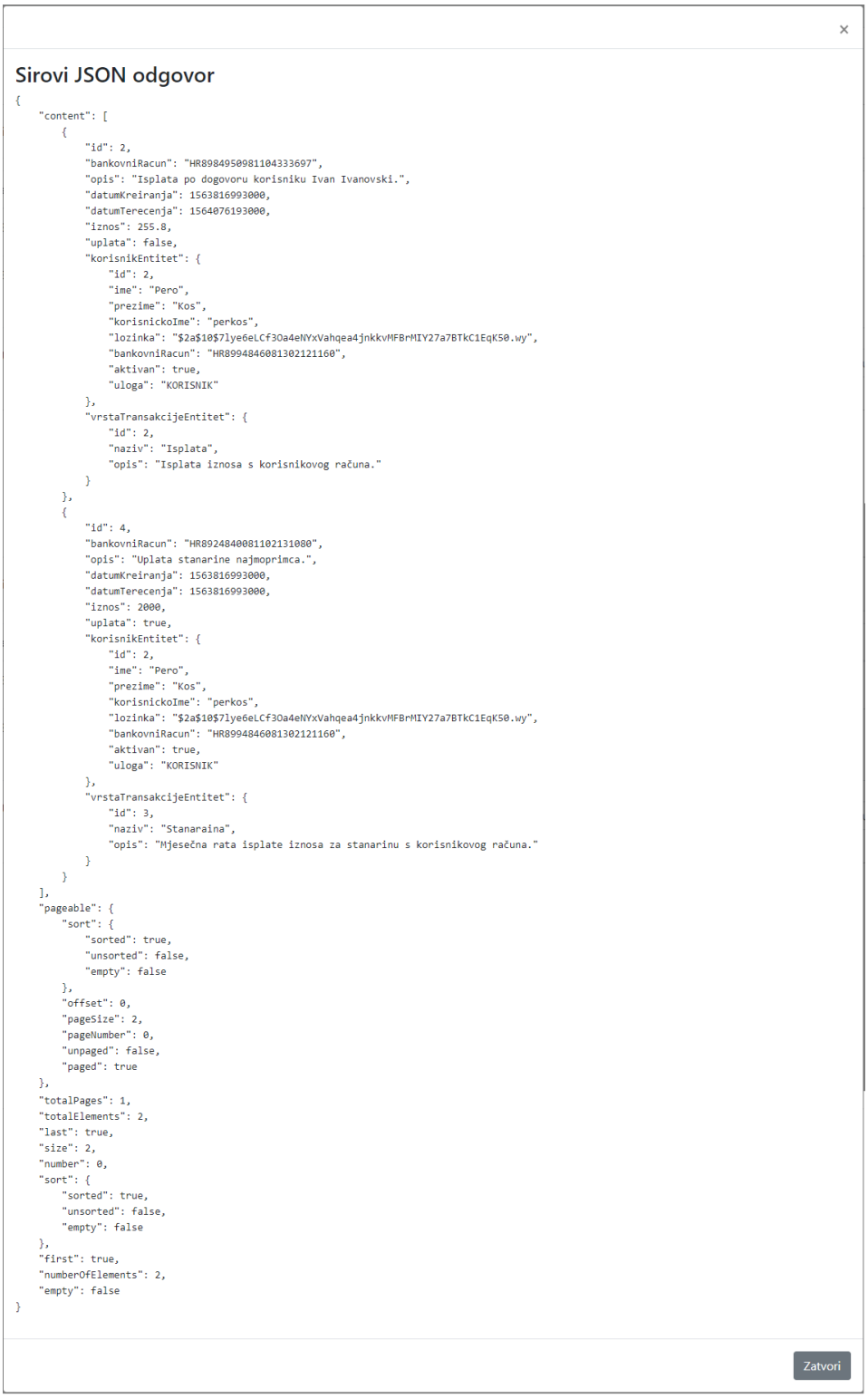
opisanoj prethodno. Kako bi mikroservis za prijavu u *Spring Boot-u* mogao dohvatiti podatke koje korisnik unese, polja korisničkog imena i lozinke moraju sadržavati *HTML* oznaku `input` s atributom `name` i vrijednostima `username` za korisničko ime i `password` za lozinku. Sam proces prijave u pozadini preuzima *Spring Boot* na jednak način primjenom `UserDetailsService` mikroservisa opisanog u poglavlju „Konfiguracija zaštite poslužiteljske aplikacije“.

6.4.2.2. Pregled transakcija



Slika 20. Stranica pregleda transakcija

Stranica na slici iznad sadrži sve transakcije korisnika, a kako bi veći broj zapisa mogao biti prikazan implementirano je straničenje nad podacima. Straničenje je implementirano u poslužiteljska aplikacija, *REST API* metoda zaprima parametre za straničenje, kreira se `Pageable` objekt koji je podržan od strane `JpaRepository` sučelja i samo sortiranje i paginacija provedeni su na razini baze.



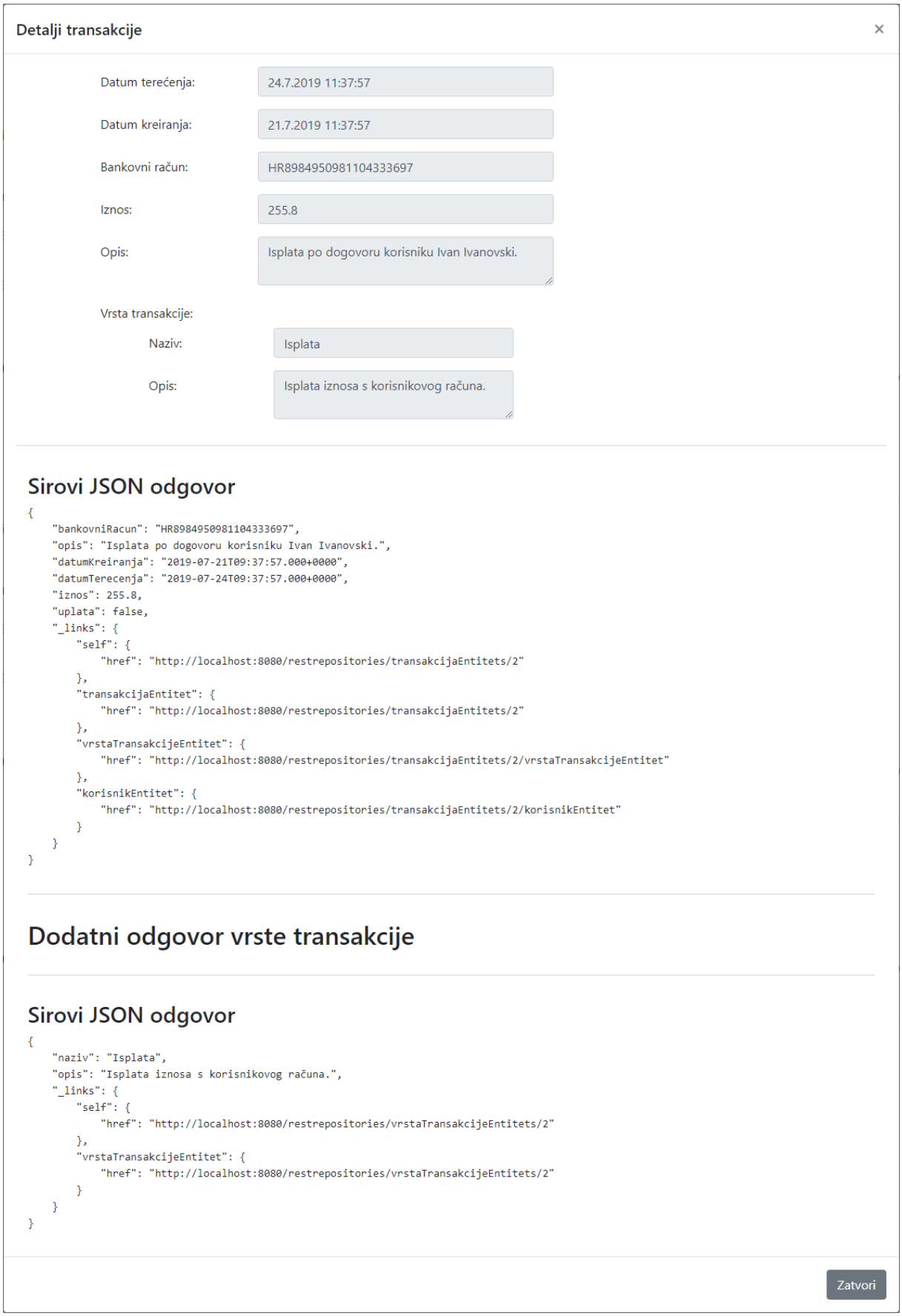
Slika 21. JSON podaci svih transakcija korisnika

Slika prikazana iznad skočni je prozor dobiven klikom na tipku „Prikaži JSON odgovor“ koja se nalazi pokraj naslova stranice. Prilikom učitavanja podataka o transakcijama, prikazanih na slici ispod, dostupni su sljedeći podaci o straničenju: `totalPages`, *boolean* varijablama `last` i `first` kako bi znali je li stranica posljednja ili prva, varijablom `empty` kako bi bez provjere sadržaja provjerili imamo li uopće rezultate za prikazati te dodatnim podacima

o samom sortiranju. Na temelju dostupnih podataka o straničenju popunjena su polja „Broj elemenata“ te Broj stranica“, a sama paginacija na stranici generira se svakim dohvaćanjem podataka ili promjenom stranice.

Prilikom pregledavanja svih transakcija korisnik klikom na zapis u tablici može pregledati detalje transakcije. Slika prikazana na sljedećoj stranici primjer je detalja transakcije, a riječ je o skočnom prozoru koji uz detalje sadrži *JSON* odgovor web servisa. Pregled detalja transakcije implementiran je pomoću *rest reposiories* mogućnosti *Spring Boot* razvojnog okvira. Primjena *rest repositories* web servisa od velike je koristi u ovom slučaju iz razloga što se podaci skočnog prozora učitavaju na klijentskoj strani pomoću *Javascript* programskog koda. Kao što je već rečeno, *rest repositories* web servisi uz podatke u samom odgovoru dostavljaju dodatne linkove odnosno svaka podaci o transakciji sadrže putanje na referenciranu vrstu transakcije i korisnika koji je kreirao transakciju. Uz asinkroni poziv za dohvaćanje podataka o transakciji, *Javascript* šalje još jedan asinkroni poziv na link u polju `vrstaTransakcijeEntitet` pod `_links` poljem pomoću kojeg dohvaća odgovor web servisa prikazan u dijelu „Dodatni odgovor vrste transakcija“.

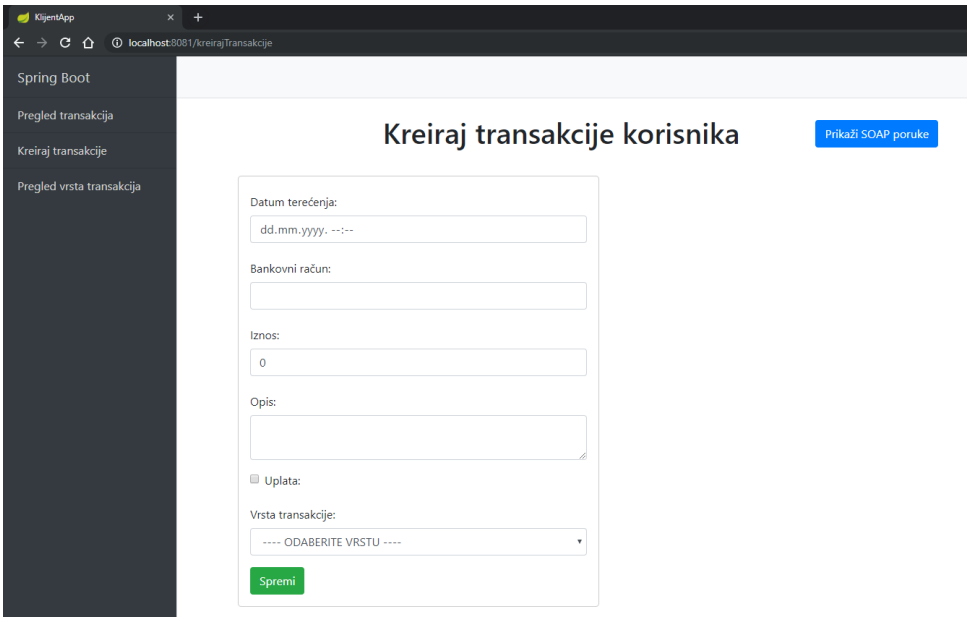
Osim navedenih mogućnosti, početna stranica nudi još dvije operacije nad zapisom transakcije: uređivanje i brisanje. Brisanje transakcije moguće je klikom na crvenu tipku Obriši, a klik na istu šalje *Javascript* asinkroni poziv na *REST* web servis na poslužiteljskoj aplikaciji. Uređivanje odnosno ažuriranje podataka implementirano je pomoću forme za kreiranje transakcije stoga će navedena operacija biti opisana u potpoglavlju Kreiranje i ažuriranje transakcije.



Slika 22. Prikaz REST poruke s poslužiteljske aplikacije

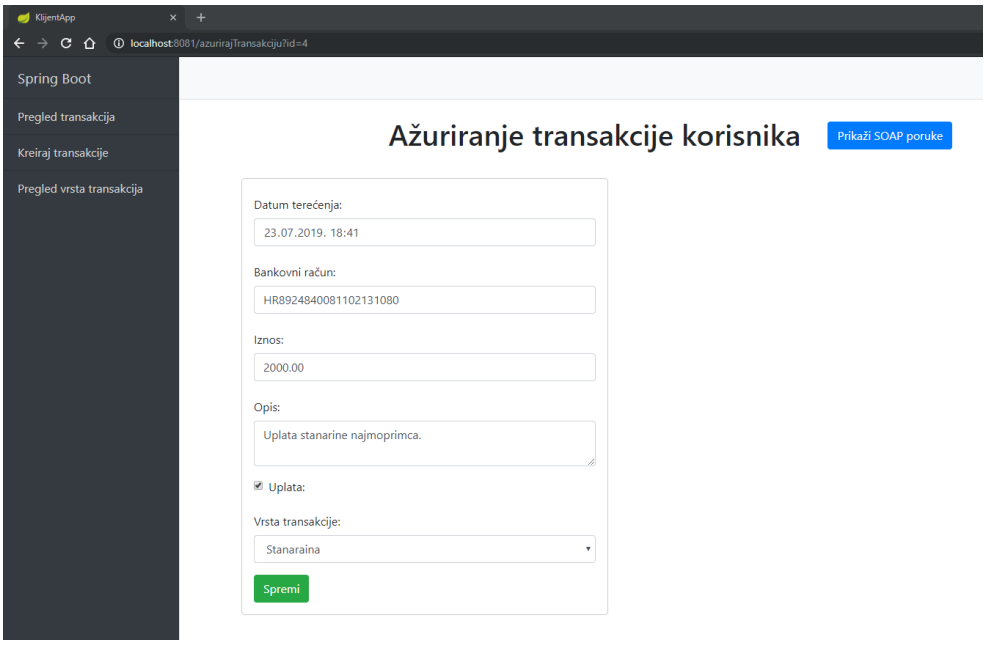
6.4.2.3. Kreiranje i ažuriranje transakcije

Stranica prikazana na slici ispod forma je za unos nove transakcija. Svaki korisnik za svoju transakciju može unijeti datum terećenja, bankovni račun s kojeg ili na koji se primjenjuje transakcija, iznos, opis, je li riječ o uplati te vrstu transakcije.



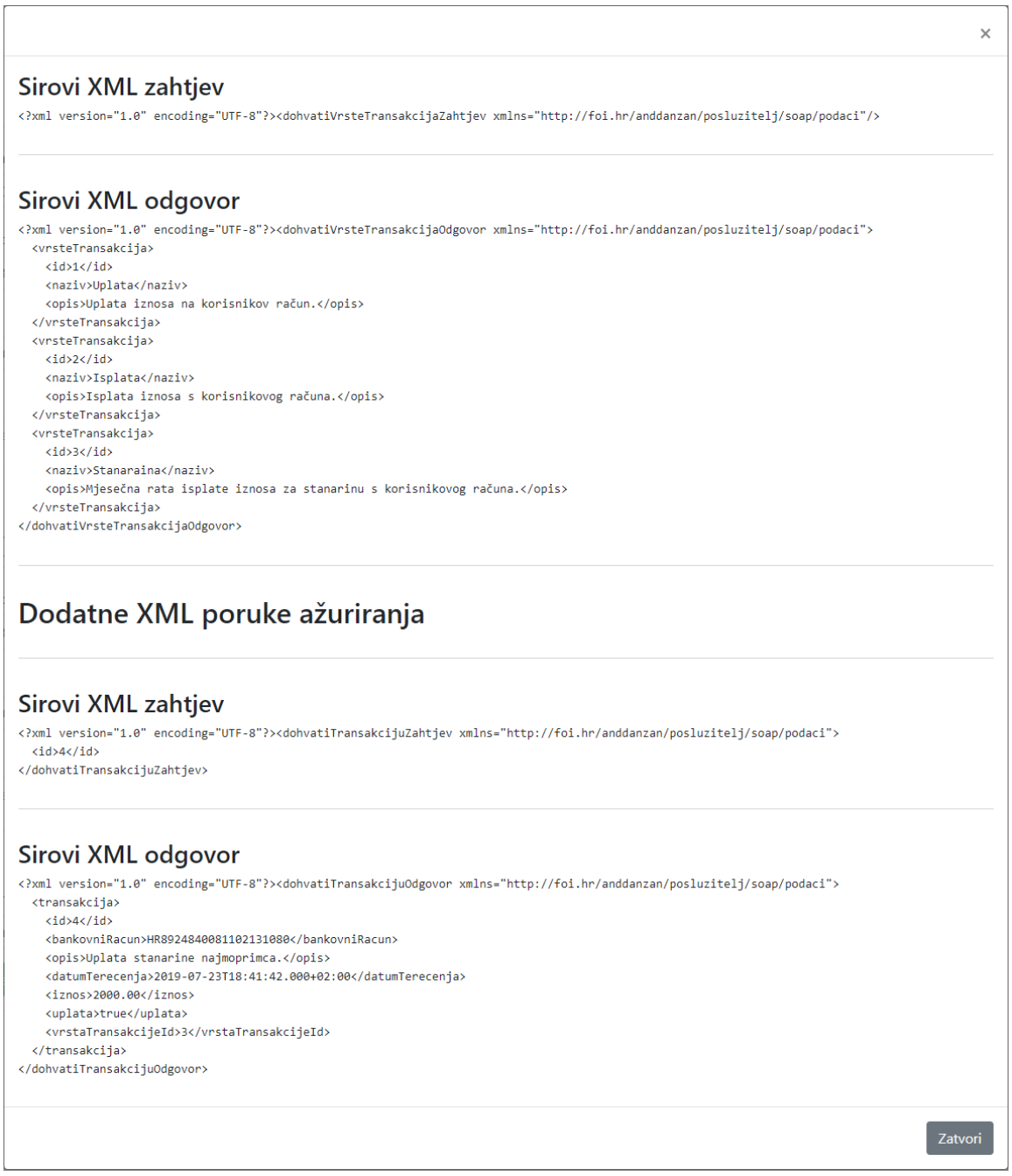
Slika 23. Stranica kreiranja transakcija

Forma za unos također je korištena za ažuriranje transakcije jedina razlika je što u slučaju ažuriranja forma je unaprijed popunjena postojećim podacima, a i sam naslov stranice je izmijenjen.



Slika 24. Stranica ažuriranja transakcije

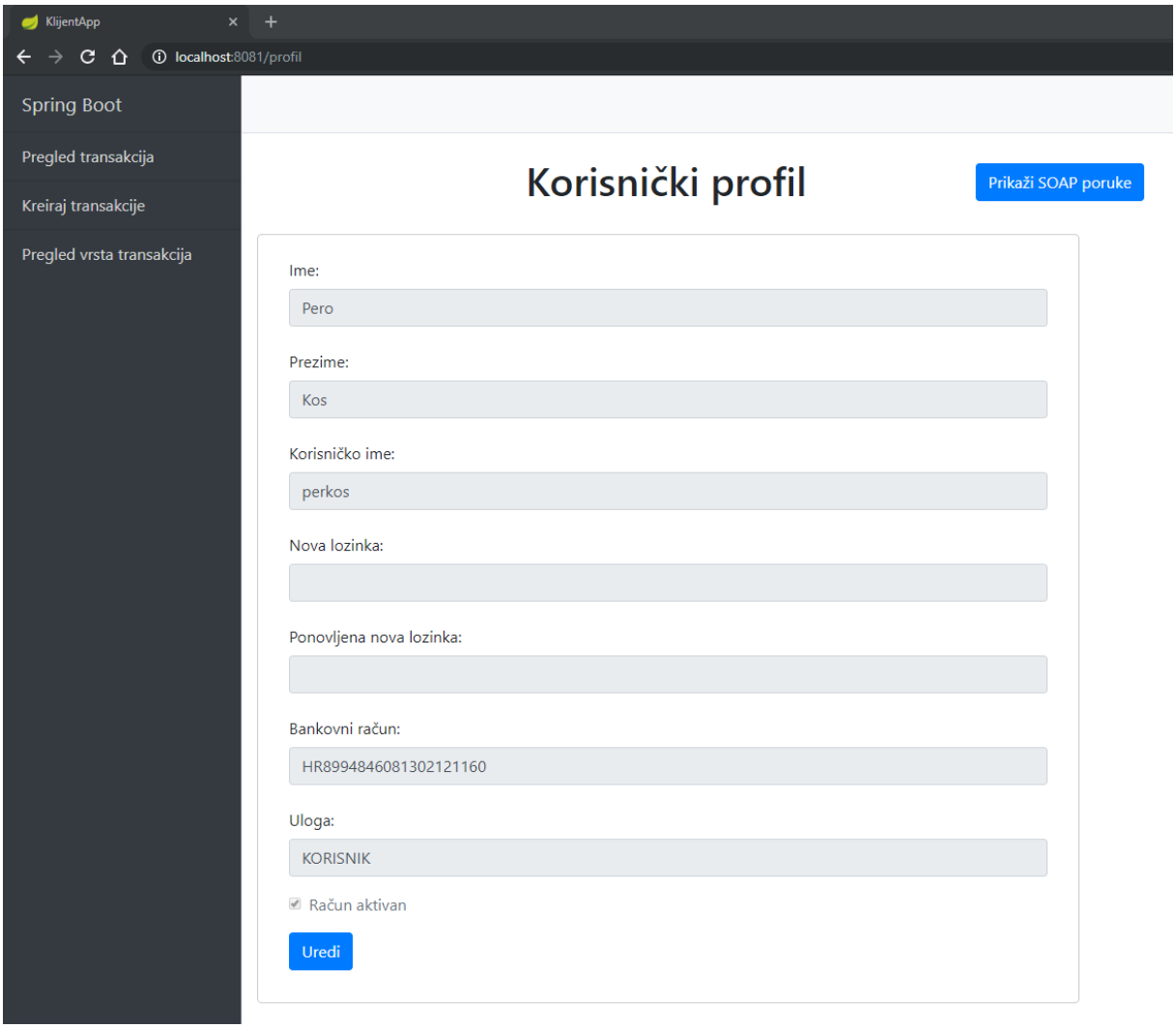
Obije operacije nad transakcijom dohvaćaju podatke preko *SOAP* web servisa, a prilikom kreiranja i ažuriranja transakcija potrebni su podaci o vrstama transakcije za padajući izbornik. *SOAP* zahtjev za dohvaćanje svih vrsta transakcija padajućeg izbornika i odgovor na isti prikazani su na slici ispod pod naslovima „Sirovi *XML* zahtjev“ i „Sirovi *XML* odgovor“. Prilikom ažuriranja podataka o transakciji na samu stranicu prenosi se identifikator transakcije stoga je potreban još jedan zahtjev za dohvaćanje podataka o transakciji. Dohvaćanje transakcije također je implementirano pomoću *SOAP* web servisa, a zahtjev i odgovor prikazani su ispod zahtjeva i odgovora za vrste transakcija u istom skočnom prozoru.



Slika 25. SOAP poruke ažuriranja transakcije

6.4.2.4. Profil korisnika

Svaki korisnik sustava može svoje podatke izmijeniti na stranici profila. Slika ispod prikazuje profil korisnika, svaki korisnik može klikom na tipku „Uredi“ osposobiti sva polja te izmjenjivati podatke. Klik na tipku „Odustani“ prikazanu na slici 29 ponovno učitava podatke s poslužiteljske aplikacije odnosno vraća stare podatke u polja. Ukoliko je korisnik završio s izmjenama, klik na zelenu tipku „Spremi“ također prikazanu na slici 29 pohranjuje podatke u bazu podataka. Jedino polje koje korisnik ne može mijenjati je polje „Uloga“.



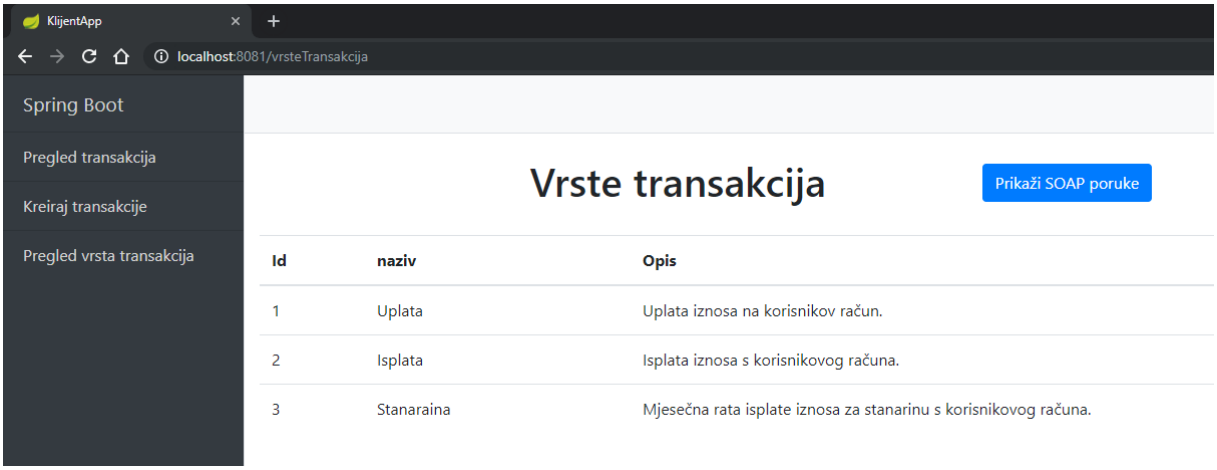
Slika 26. Stranica profila korisnika

Slika 27. Prikaz mogućnosti nakon klika na tipku Uredi

6.4.2.5. Pregled vrsta transakcija

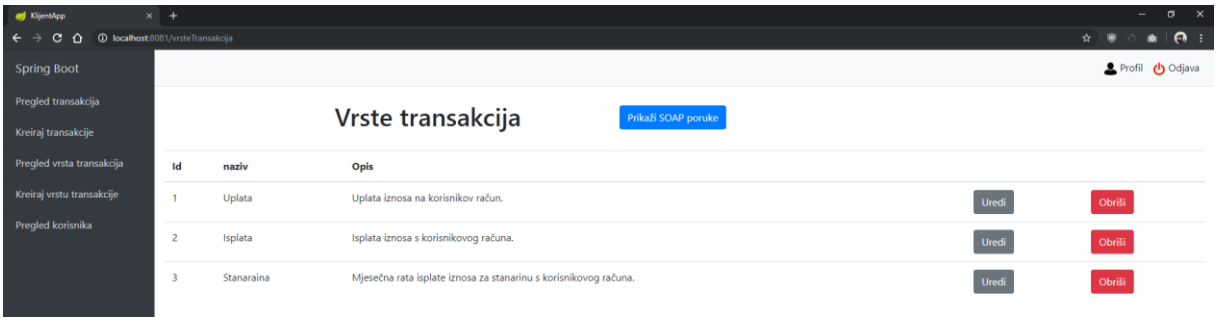
Pregled vrsta transakcije stranica je specifična po tome što je različita za svaku vrstu korisnika. Slika ispod prikazuje izgled stranice za običnog korisnika koji ima prava samo

pregledavati postojeće vrste transakcija u sustavu. Navedeni način onemogućuje da bilo tko u sustavu kreira, briše ili izmjenjuje podatke o vrstama transakcija koje se koriste već samo nekolicina osoba koje su administratori imaju prava provoditi navedene operacije.



Slika 28. Pregled vrsta transakcija običnog korisnika

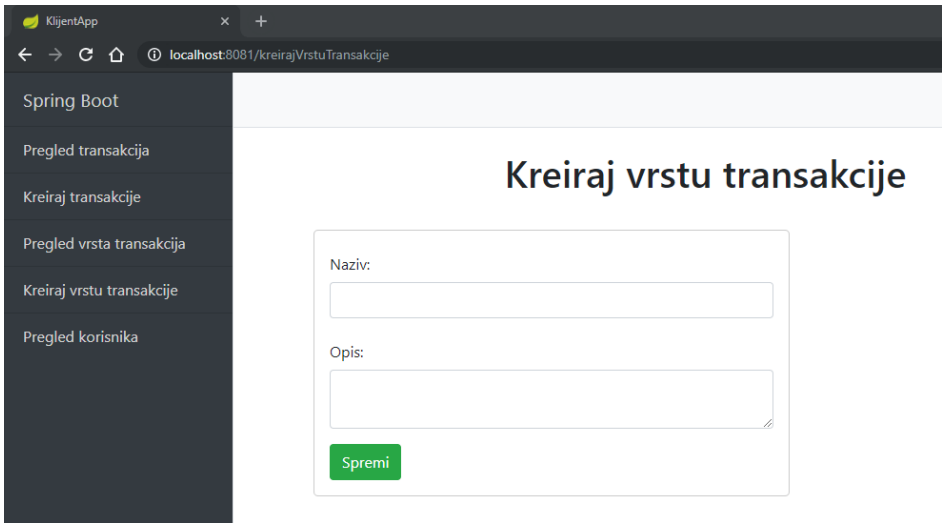
Administrator sustava ima prava kreirati, brisati te ažurirati transakcije. Navedene operacije provode se na sličan način kako u slučaju pregleda transakcija. Klik na crvenu tipku „Obriši“ šalje zahtjev na klijentsku aplikaciju koja potom dobiveni identifikator vrste transakcije unutar *SOAP* zahtjeva šalje na web servis. U slučaju da se vrsta transakcije ne koristi već odnosno nije referencirana u nijednoj transakciji, ista se briše. Kreiranje i ažuriranje same vrste transakcije opisano je u nastavku.



Slika 29. Pregled vrsta transakcija administratora

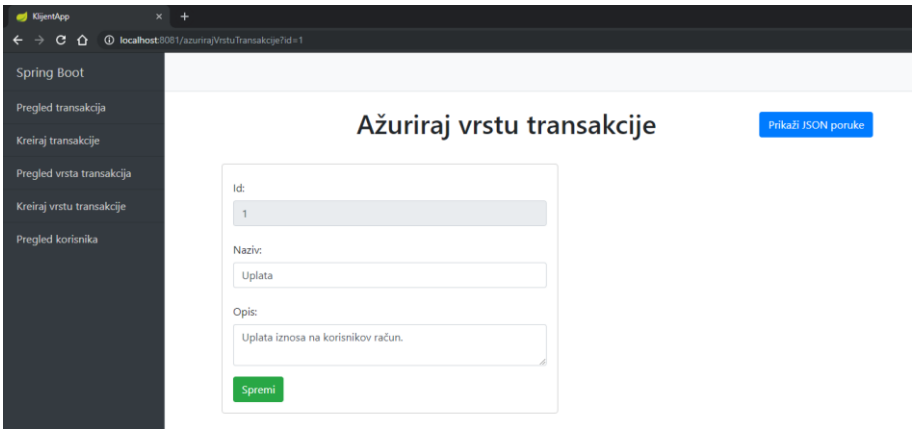
6.4.2.6. Kreiranje i uređivanje vrste transakcije

Kreiranje vrste transakcije vrlo je jednostavan proces iz razloga što ista sadrži samo polja za naziv i opis vrste transakcije. Slika ispod odlomka prikazuje stranicu za kreiranje vrste transakcije, ista nema tipku za pregled *SOAP* ili *JSON* poruka jer je riječ o jednostavnoj formi koja nema prikaza podataka.



Slika 30. Kreiranje vrste transakcije

U slučaju ažuriranja vrste transakcije postoji tipka *SOAP* odgovora iz razloga što se prilikom klika na tipku „Uredi“, na stranici pregleda vrsta transakcija, prosljeđuje samo identifikator vrste. Prema dobivenom identifikatoru dohvaćaju se podaci o vrsti transakcije te su isti prikazani u formi ispod, ali s dodatnim polje „id“ za prikaz identifikatora koje nije moguće izmijeniti.

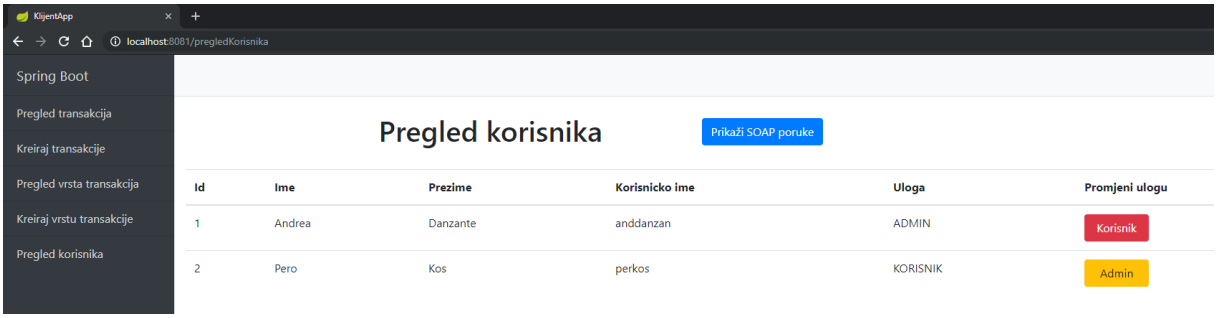


Slika 31. Ažuriranje vrste transakcija

6.4.2.7. Pregled i promjena uloge korisnika

Posljednja mogućnost sustava i stranica kojoj ima pristup samo administrator je pregled i promjena uloge korisnika. Pregled svih korisnika i njihove uloge u sustavu ima pravo samo administrator, stupac „Uloga“ u tablici prikazuje trenutnu ulogu u sustavu, a sama promjena uloge implementirana je pomoću dvije tipke prikazane na slici ispod. Ukoliko je korisnik administrator sustava, u redu zapisa nalaziti će se crvena tipka s nazivom „Korisnik“ pomoću koje oduzimamo administratorske ovlasti. U slučaju kad je korisnik običan korisnik sustava, kraj njegovog imena nalazi se žuta tipka s tekstom „*Admin*“ te pomoću iste korisnik može dobiti

administratorske ovlasti. Navedena tablica također sadrži i trenutnog korisnika kako bi korisnik koji je trenutno administrator mogao sam sebi oduzeti prava te postati običan korisnik.



Slika 32. Stranica pregleda korisnika

7. Zaključak

Predmet ovog diplomskog rada bio je prikazati implementaciju *SOAP* i *REST* web servisa u *Spring Boot* razvojnom okruženju. Implementacija je provedena u obliku dviju nezavisnih aplikacija, poslužiteljske aplikacije koja je srž sustava i klijentske aplikacije kao primjer korištenja web servisa u stvarnom sustavu.

U samom praktičnom dijelu korišten je veći broj programske podrške i alata, obije aplikacije koriste isti set alata i zavisnosti osim u nekoliko slučajeva. *Maven* alat za izgradnju aplikacije korišten je u obije aplikacije iz razloga što isti omogućava vrlo jednostavnu izgradnju aplikacije, dodavanje zavisnosti te omogućava dodatke. Dodatak za *Maven* koji je korišten u aplikacijama je *Jaxb* koji *XSD* datoteku preslikava u *Java* klase na poslužiteljskoj strani, a na klijentskoj preslikava *WSDL* datoteku *SOAP* servisa. Zavisnosti i alati potrebni za pokretanje, spajanje i rad s bazom korišteni su na poslužiteljskoj strani, a zavisnost poput *Thymleaf*-a korištena je samo na klijentskoj strani iz razloga što je riječ o zavisnosti za implementaciju grafičkog sučelja.

Nakon implementacije obiju vrsta web servisa u *Spring Boot-u* otkrio sam prednosti i mane istih. *SOAP* web servisi temeljeni su na unaprijed definiranom skupu pravila komunikacije stoga je implementacija klijenta znatno lakša. Mana navedenog je ta što svaki zahtjev mora sadržavati odgovarajuću *XML* strukturu *SOAP* poruke s imenskim prostorima stoga ukoliko bi *SOAP* servis željeli koristiti u *Javascript* srodnom programskom jeziku morali bi imali veliki broj zavisnosti i podosta implementacije. Navedeni razlog ide u prilog *REST* servisa koji se pozivaju preko jedinstvene putanje u obliku *HTTP* zahtjeva, a parametri mogu biti proslijeđeni u putanji, u zaglavlju ili preko tijela poruke. Mana jednostavnosti pozivanja *REST* web servisa je ta što nema propisanih pravila komunikacije kao što je slučaj sa *SOAP* web servisom, stoga ukoliko je odgovor *REST* metode promijenjen sustav nema informacija o tome i neće raditi s novim podacima dok ne prilagodimo programski kod. Razliku *SOAP-a* i *REST-a* u pogledu *Spring Boot* razvojnog okruženja uočio sam prilikom konfiguriranja sigurnosnih postavka. *Spring security* omogućava implementaciju sigurnosnih ograničenja na putanjama, *SOAP* pristupne točke sve rade na istoj putanji, a pozivaju se na temelju vrste zahtjeva koji dolazi dok *REST* radi na principu da svaka metoda ima jedinstvenu putanju i *HTTP* metodu. Navedeno ograničenje *SOAP-a* diktira da se sve metode moraju zaštititi u sustavu jer se nalaze na istoj putanji što je nezgodno za metode prijave i registracije koje moraju biti dostupne svim korisnicima.

Popis literature

1. Apache. (2019a). Maven. Preuzeto 25. travanj 2019., od https://maven.apache.org/what-is-maven.html
2. Apache. (2019b). Tomcat. Preuzeto 25. travanj 2019., od https://tomcat.apache.org/tomcat-9.0-doc/introduction.html
3. ApacheFriends. (2019). XAMPP. Preuzeto 02. svibanj 2019., od https://en.wikipedia.org/wiki/XAMPP
4. Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., … Winer, D. (2000). Simple Object Access Protocol (SOAP) 1.1. Preuzeto 07. travanj 2019., od https://www.w3.org/TR/2000/NOTE-SOAP-20000508/
5. Brisbin, J., Gierke, O., Turnquist, G., & Bryant, J. (2019). Spring Data REST Reference. Preuzeto 01. lipanj 2019., od PivotalSoftware website: https://docs.spring.io/spring-data/rest/docs/3.1.8.RELEASE/reference/html
6. Christensen, E., Curbera, F., Meredith, G., & Weerawarana, S. (2001). Web Services Description Language (WSDL) 1.1. Preuzeto 14. travanj 2019., od https://www.w3.org/TR/2001/NOTE-wsdl-20010315
7. Cornelißen, P., Piefel, M., & Sparkowsky, A. (2018). *Spring Boot 2 Fundamentals*. Packt Publishing.
8. Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. Irvine: University of California, Irvine.
9. JetBrains. (2019). *InteliJ IDEA*. JetBrains.
10. Kalim, M. (2013). *Java Web Services: Up and Running, 2nd Edition*. Elsevier.
11. Kermek, D. (2019). UZORCI DIZAJNA - Uzorci strukture. Preuzeto 04. lipanj 2019., od FOI website: https://elf.foi.hr/pluginfile.php/13870/mod_resource/content/10/predavanja/Kermek_UzDiz_03.pdf
12. Mitra, N., & Lafon, Y. (2007). SOAP Version 1.2 Part 0: Primer (Second Edition). Preuzeto 13. travanj 2019., od https://www.w3.org/TR/soap12-part0/
13. Oracle. (2015). *javax.persistance*.
14. Oracle. (2019). javax.xml.bind.annotation. Preuzeto 20. svibanj 2019., od https://docs.oracle.com/javase/8/docs/api/javax/xml/bind/annotation/package-frame.html
15. Ort, E. (2005). Service-Oriented Architecture and Web Services: concepts, Technologies, and Tools. *Sun Technical Articles*, (April 2005).
16. PivotalSoftware. (2019a). Spring Boot Data JPA Starter. Preuzeto 07. svibanj 2019.,

od https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-data-jpa/2.1.4.RELEASE
17. PivotalSoftware. (2019b). Spring Boot Data REST Starter. Preuzeto 03. svibanj 2019., od https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-data-rest/2.1.4.RELEASE
18. PivotalSoftware. (2019c). Spring Boot Security Starter. Preuzeto 28. travanj 2019., od https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-security/2.1.4.RELEASE
19. PivotalSoftware. (2019d). Spring Boot Thymeleaf Starter. Preuzeto 13. srpanj 2019., od https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-thymeleaf/2.1.6.RELEASE
20. PivotalSoftware. (2019e). Spring Boot Web Services Starter. Preuzeto 28. travanj 2019., od https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-web-services/2.1.4.RELEASE
21. PivotalSoftware. (2019f). Spring Boot Web Starter. Preuzeto 28. travanj 2019., od https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-web/2.1.4.RELEASE
22. PivotalSoftware. (2019g). Spring Initializr. Preuzeto 24. svibanj 2019., od https://start.spring.io/
23. Poutsma, A., Evans, R., Rabbo, T. A., & Turnquist, G. (2019). Spring Web Services Reference Documentation. Preuzeto 22. svibanj 2019., od https://docs.spring.io/spring-ws/docs/current/reference/
24. Thompson, H. S., Mendelsohn, N., Beech, D., & Maloney, M. (2012). W3C XML Schema Definition Language (XSD) 1.1. Preuzeto 18. svibanj 2019., od https://www.w3.org/TR/xmlschema11-1/
25. Webb, P., Syer, D., Long, J., Nicoll, S., Winch, R., Wilkinson, A., … Madhura Bhave. (2018). *Spring Boot Reference Guide* (str. 443). str. 443.

Popis slika