Univerzitet u Novom Sadu
Ekonomski fakultet u Subotici

RAZVOJ RESTFUL WEB SERVISA U SPRING BOOT-U

DIPLOMSKI RAD

Autor:
Aleksandar Raletić
PIS039/14

Novi Sad, 2021.

Univerzitet u Novom Sadu
Ekonomski fakultet u Subotici

RAZVOJ RESTFUL WEB SERVISA U SPRING BOOT-U

DIPLOMSKI RAD

Mentor:
doc. dr Lazar Raković

Autor:
Aleksandar Raletić
PIS039/14

Novi Sad, 2021.

Univerzitet u Novom Sadu, Ekonomski fakultet Subotica

KLJUČNA DOKUMENTACIJSKA INFORMACIJA

Autor, AU:		Aleksandar Raletić
Mentor, MN:		doc. dr Lazar Raković
Naslov rada, NR:		Razvoj RESTful Web Servisa u Spring Boot-u
Jezik publikacije, JP:		Srpski
Zemlja publikovanja, ZP:		Republika Srbija
Uže geografsko područje, UGP:		AP Vojvodina
Godina, GO:		2021.
Mesto i adresa, MA:		Zrenjanin, Baranjska 93
Fizički opis rada, FO: (poglavlja/strana/citata/tabela/slika/grafika /priloga)		Broj poglavlja: 11 Broj strana: 52 Broj citata: 31 Broj tabela: 1 Broj slika: 33 Broj grafika: 0 Broj priloga: 0
Naučna oblast, NO:		Poslovna informatika
Naučna disciplina, ND:		Razvoj poslovnih aplikacija
Predmetna odrednica/Ključne reči, PO:		Aplikacija, Poslovna informatika, Poslovni informacioni sistemi, Java, Spring, Spring Boot, Mikroservisi, REST arhitektura, RESTful arhitektura
Čuva se, ČU:		Biblioteka Ekonomskog fakulteta u Subotici
Važna napomena, VN:		-
Izvod, IZ:		1. Uvod 2. Java 3. Web servisi i API 4. REST arhitektura 5. CRUD operacije 6. Anotacije 7. Spring framework 8. Mikroservisi 9. Prikaz aplikacije 10. Zaključak 11. Literatura
Datum prihvatanja teme, DP:		21.05.2021.
Datum odbrane, DO:		
Članovi komisije, KO:	Predsednik, član:	prof. dr Marton Sakal
	Mentor, član:	doc. dr Lazar Raković

Sadržaj

1. Uvod

Mikroservisna arhitektura u izradi aplikacija danas sve više dobija na značaju i popularnosti sa rastom broja kompanija koje dele pozitivna iskustva u vezi sa njenom upotrebom. Za razliku od standardne, monolitne arhitekture, koja podrazumeva izgradnju aplikacije kao jedne celine, mikroservisna arhitektura je način razvoja aplikacija u vidu manjih, izdvojenih i nezavisnih servisa koji komuniciraju međusobno u cilju obezbeđivanja funkcionalnosti celokupne aplikacije.

Predmet ovog diplomskog rada je prikaz jednostavne aplikacije koja je zasnovana na mikroservisnoj arhitekturi u Spring framework-u koji je baziran na programskom jeziku Java. U uvodnom delu biće reči o samom programskom jeziku a potom će teorijski biti analizirano svo štivo koje će biti potrebno za razumevanje ovakvog načina funkcionisanja krajnje aplikacije koja je izrađena u Spring-u. Nakon teorijske obrade, biće prikazano kako aplikacija funkcioniše sa konkretnim primerima i delovima koda koji će biti detaljno objašnjen u cilju razumevanja celokupnog projekta.

2. Java

Java je programski jezik opšte namene koji je kreirala kompanija Sun Microsystems 1995. godine od strane James Gosling-a. Ona poseduje velike biblioteke programskih modula koje omogućavaju komunikaciju sa korisnikom putem prozora (GUI), rad preko interneta, rad sa bazama podataka i slično. (Kraus, 2013) Dizajniran po uzoru na C++, Java jezik je napravljen da bude jednostavan i prenosiv između različitih platformi i operativnih sistema.

Java je danas jedan od najpopularnijih programskih jezika koji je svoju namenu našao i u web programiranju gde se koristi kao posredni jezik koji omogućava implementaciju aplikacija putem određenih framework-ova koji koriste javu kao programski jezik.

Java ima sledeće odlike (Oracle, 1999):

- Objektno orijentisan programski jezik: to znači da je u Javi sve objekat. Svaki objekat poseduje određena stanja i ponašanja i u zavisnosti od tih osobina, sve objekte sa srodnim osobinama svrstavamo u jednu zajedničku grupu koja se naziva klasa.
- Nezavisnost od platforme: java kompajler konvertuje izvorni kod u bytecode koji zapravo predstavlja međujezik. Bytecode se zatim može pokrenuti na bilo kom operativnom sistemu korišćenjem JVM-a (Java Virtual Machine).
- Jednostavnost: Java je dizajnirana da bude laka za učenje. Razumevanjem osnovnih koncepata objektno-orijentisanih programskih jezika nije teško savladati Javu.
- Bezbednost: sa svojim bezbednosnim odlikama omogućava kreaciju i razvoj sistema i sigurnih aplikacija i pruža uvid i bilo kakve neželjene promene. Bezbedno učitavanje klasa i verifikacioni mehanizmi omogućavaju izvršavanje samo legitimnog java koda.

- Arhitekturalna nezavisnost: Java kompajler generiše objekte u arhitekturalno nezavisnom formatu što omogućava da se takav kod izvrši na velikom broju procesora.
- Prenosivost: time što je nezavisna od arhitekture i nema druge implementacione zavisnosti omogućava laku prenosivost.
- Robusnost: Java ima veliku sposobnost da upravlja greškama prilikom kreiranja i izvršavanja koda jer poseduje dobar *exception handler* (eng.) i mehanizme za proveru grešaka prilikom pisanja samog koda.
- Višenitnost: zahvaljujući svojim višenitnim sposobnostima Java omogućava pisanje programa koji može da radi više zadataka odjednom, istovremeno. Ovakav dizajn omogućava programerima da razvijaju aplikacije čiji će delovi funkcionisati glatko i ujednačeno.
- Interpretiranost: Java Virtual Machine koristi tehniku kompilacije Just-In-Time (JIT) da bi kompajlirala bytecode u direktna uputstva koja su razumljiva procesoru u samom trenutku kada se pokreće programski kod.
- Visoke performanse: korišćenjem kompajlerske tehnike Just-In-Time, Java omogućava postizanje visokih performansi.
- Distributivnost: java je dizajnirana za distribuirano okruženje preko interneta korišćenjem TCP/IP protokola.
- Dinamičnost: smatra se da je Java jezik dinamičniji od C i C++ pošto je napravljena tako da može lako da se prilagodi (adaptira) stalnim promenama okruženja.

3. Web servisi

Web servis je softverski sistem koji služi za uspostavljanje interoperabilne interakcije između različitih mašina koje međusobno komuniciraju putem mreže. (W3, 2004)
Oni se mogu kombinovati na specifičan način u cilju izvršavanja kompleksnih operacija. Programi koji pružaju jednostavne servise mogu da komuniciraju jedni sa drugim u cilju izvršavanja kompleksnih operacija i zajedničkim učinkom pružaju usluge višeg kvaliteta.

Prednosti web servisa prema sajtu Test Automation Resources (Test Automation Resources, 2018):

- Interoperabilnost: jedna od prednosti web servisa je njihova interoperabilnost. Web servisi omogućavaju aplikacijama da komuniciraju, razmenjuju podatke i servise između sebe.
- Upotrebljivost: web servisi su dizajnirani tako da mogu da pruže širok spektar usluga, kako od onih najjednostavnijih poput pregleda jednostavnih informacija tako do kompleksnih algoritamskih izračunavanja.
- Ponovna upotrebljivost: web servisi su pravljeni tako da se njihovi kodovi mogu reciklirati u izgradnji drugih servisa a starije aplikacije i programi koji su pisani u zastarelim tehnologijama mogu se implementirati kao web servisi čime se omogućava njihova dalja upotrebljivost.
- Troškovi: zbog mogućnosti ponovnog korišćenja već razvijenih web servisa, novi sistemi se mogu kreirati na bazi već korišćenih web usluga što obezbeđuje uštede u troškovima kako za klijenta tako i za pružaoca usluga.

Postoje dva značajna tipa web servisa (Stackify, 2017)

1. SOAP Web servisi – SOAP (Simple Object Access Protocol) je na standardima baziran protokol zasnovan na XML-u koji se koristi za razmenu informacija između kompjutera. XML je alat nezavisan od softvera i hardvera čiji je zadatak da skladišti i transportuje podatke. (W3Schools, n.d.)
2. REST Web servisi – REST (Representational State Transfer) je standard koji je kreiran da bi prevazišao nedostatke SOAP protokola pružajući mogućnost jednostavnijeg pristupa web servisima korišćenjem JSON sheme za formatiranje zahteva i HTTP transportnog protokola.

4. REST arhitektura

API programski interfejs aplikacije predstavlja listu pravila koja definišu na koji način aplikacije ili uređaji mogu da se povežu i komuniciraju jedni sa drugim. To je pristupna tačka preko koje aplikacija pristupa i preuzima podatke iz baze tj. druge aplikacije. Možemo zaključiti da API služi kao interfejs između dve različite aplikacije pomoću kojeg one uspostavljaju komunikaciju.
REST je arhitekturalni stil koji se može definisati kao dizajn obrazac za izgradnju API-ja. Web servisi koji su bazirani na ovoj arhitekturi nazivaju se RESTful web servisi a programabilni interfejsi ovih servisa nazivaju se RESTful API-ji. (IBM, n.d.)
REST definiše spisak pravila kojih bi se trebalo pridržavati prilikom kreiranja API-ja. Ova pravila su usmerena na povećanje skalabilnosti i robusnosti umreženih, na resursima-orijentisanih sistema koji su bazirani na HTTP-u. (Rodriguez, i drugi, 2016)
Ta pravila se odnose na šest principa koje je potrebno ispuniti prilikom izgradnje aplikacija da bi bilo reči o RESTful aplikaciji.

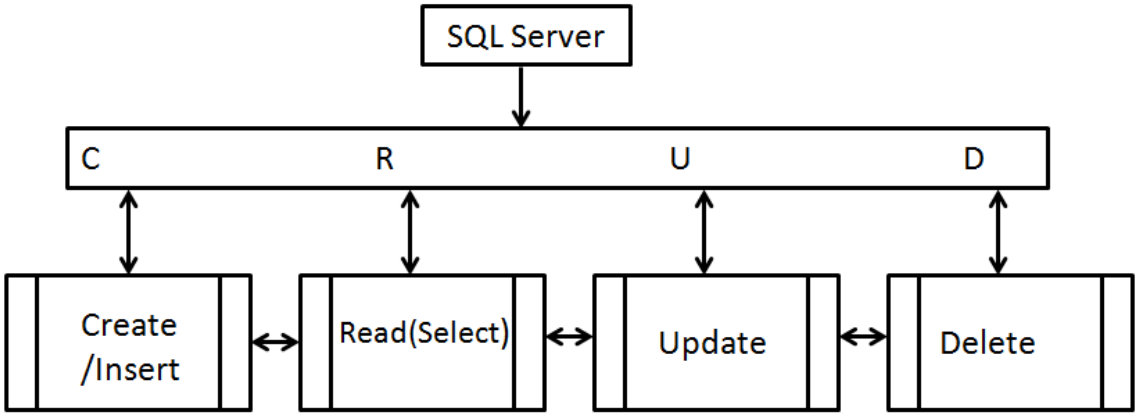
Vodeći principi REST-a (Fielding, 2000):

1. Klijent-server arhitektura: server skladišti i upravlja informacijama da bi ih na najpogodniji način prikazao klijentu na zahtev korisnika. Klijent prima informacije koje šalje server i prikazuje ih korisniku ili koristi datu informaciju na osnovu koje izvršava ostale operacije i radnje u cilju izvršavanja zahteva korisnika. S obzirom da su klijent i server razdvojeni jedan od drugog, to im omogućava da se razvijaju nezavisno zahtevajući samo zajednički interfejs kao posrednu tačku između klijenta i servera.
2. Bez stanja (stateless): svaki zahtev od klijenta ka serveru mora uvek sadržati sve potrebne informacije za izvršenje zahteva, što znači da klijent ne može da iskoristi sadržaj koji se već nalazi na serveru od ranije. Navedeno implicira da se stanje sesije ne čuva na serveru, već u potpunosti na klijentu.
3. Keširanje: klijent, server i sve posredničke komponente mogu da koriste mogućnosti i prednosti keširanja podataka u cilju ostvarivanja većih performansi i bržih obrada zahteva.
4. Ujednačen interfejs: primenom principa softverskog inžinjeringa celokupna arhitektura sistema je pojednostavljena a sve komponente moraju da prate ista pravila i ograničenja da bi mogle da komuniciraju međusobno.
5. Slojevit sistem: kod ovakve vrste sistema koja je izgrađena od slojeva, komponente ne mogu da pristupe nikakvim drugim resursima izvan sloja osim u kojem se ona sama nalazi.

6. Kod na zahtev: REST pruža mogućnost povećavanja funkcionalnosti klijenta preuzimanjem i izvršavanjem koda u formi apleta ili skripti. Ovo pojednostavljuje klijente smanjujući broj funkcija i karaktersitika koje moraju biti ugrađene u klijent.

5. CRUD operacije

Kako bi upravljale podacima, REST API-ji bi trebali da koriste sledeće operacije tj. HTTP metode: POST, GET, PUT, DELETE. Navedene metode omogućavaju interakciju sa bazom podataka putem kreiranja, čitanja, promene i brisanja podataka zbog čega se još nazivaju CRUD operacijama (create, read, update, delete). Pored naznačenih operacija postoje i druge koje neće biti predmet ovog rada a pritom nisi u toliko velikoj upotrebi poput ovih opštih, široko korišćenih operacija.



Slika 1. CRUD operacije (SQLShack, n.d.)

U tekstu ispod su detaljnije opisane metode koje su prikazane na slici 1.

1. CREATE

CREATE metoda omogućava korisnicima da kreiraju novu instancu u bazi podataka. U SQL relacionim bazama podataka ova operacija korespondira INSERT naredbi dok je HTTP metoda za ovu operaciju POST. Prilikom korišćenja POST metode treba imati u vidu njenu (ne)idempotenciju i (ne)bezbednost.
Neki zahtev je idempotentan kada će on rezultirati istim ishodom nezavisno od toga da li je zahtev izvršen jedanput ili bezbroj puta.
Svaka HTTP metoda može pripadati ili grupi bezbednih ili nebezbednih metoda. HTTP metoda je bezbedna onda kada njena upotreba ne menja nikakve podatke na serveru.
Ukoliko nije postavljeno UNIQUE ograničenje, korišćenje POST metode više puta nego jednom rezultiraće kreiranjem nove instance svaki put kad je zahtev izvršen. Ovo se odnosi na neidempotentnost.
Imajući u vidu prethodnu podelu HTTP metoda prema bezbednosti, upotreba POST metode rezultira promenom podataka kreiranjem nove instance u bazi podataka, te se može reći u skladu sa tim da pripada grupi nebezbednih metoda. (sumo logic, n.d.; SymfonyCasts, n.d.)

2. READ

READ metoda omogućava prikazivanje, odnosno iščitavanje podataka koji su već smešteni u bazi podataka pa se u tom smislu može poistovetiti sa funkcijom pretrage podataka. Korisnici treba da unesu određen parametar ili ključnu reč u URL na osnovu čega će imati detaljniji prikaz sa svim atributima koji su povezani sa datim parametrom. U SQL relacionim bazama podataka READ operacija odgovara SELECT naredbi a HTTP metoda na kojoj se zasniva je GET metoda. GET metoda je idempotentna metoda jer će se njenom upotrebom uvek prikazati isti, specifičan podatak, nezavisno od toga da li smo je pozvali jednom ili bezbroj puta. Što se tiče druge podele na bezbednost i nebezbednost metoda, ova metoda pripada grupi bezbednih metoda jer se njenom upotrebom ne menjaju nikakvi podaci već se samo vrši prikaz postojećih podataka. (sumo logic, n.d.; SymfonyCasts, n.d.)

3. UPDATE

UPDATE metoda se koristi da bi se izvršila modifikacija već postojećih podataka koji postoje u bazi podataka. U SQL relacionim bazama podataka UPDATE operacija ima isti naziv kao i UPDATE naredba dok je HTTP metoda na kojoj se zasniva PUT metoda.
UPDATE metoda je idempotentna metoda. Ukoliko izvršimo izmenu određenog podatka, nezavisno od toga da li smo metodu pozvali jednom ili bezbroj puta da bismo izvršili dati zahtev, rezultat će uvek biti isti. Primer: Ako jednom pozovemo ovu metodu, izvršićemo izmenu vrednosti “identifikacionog broja” sa “1” na “2”. Svaki naredni put kada pozovemo ovu metodu, vršićemo izmenu identfikacionog broja na vrednost „2“ čime se potvrđuje hipoteza o idempotenciji jer ova metoda uvek rezultira istim ishodom.
UPDATE metoda pripada grupi nebezbednih metoda jer se njenim pozivom vrši izmena postojećih podataka. S druge strane ako promenimo vrednost identifikacionog broja na vrednost koju već poseduje, tj. ukoliko unesemo isti broj neće se ništa dogoditi. Podaci će i dalje ostati isti. Suština ovoga je da nebezbedni metodi ne moraju, ali mogu imati štetne posledice dok bezbedni metodi nikada neće imati štetne posledice jer se njima ne vrši izmena podataka. (sumo logic, n.d.; SymfonyCasts, n.d.)

4. DELETE

DELETE metoda pruža mogućnost uklanjanja podataka iz baze podataka. Kako i u SQL relacionim bazama podataka tako i u HTTP metodi ova metoda ima isti naziv: delete.
DELETE metoda je idempotentna metoda jer će se uvek izvršiti zahtev zbog kog se poziva. Brisanjem torke gde je vrednost identifikacionog broja: “2” uvek će obrisati datu torku, a ukoliko ona ne postoji neće se nijedna druga torka ukloniti.
DELETE metoda nije bezbedna metoda jer njena upotreba dovodi do izmene postojećih podataka. (sumo logic, n.d.; SymfonyCasts, n.d.)

6. Anotacije

U programskom jeziku Java, anotacije predstavljaju vid metapodataka koji mogu biti dodati izvornom kodu. Anotacije se koriste da bi se automatski generisali meta podaci i one mogu biti povezane sa klasama, metodama, varijablama i parametrima. One pružaju dodatne informacije prilikom kompajliranja koda ali nisu deo samog programa. (Sun Microsystems, 2011) Deklarisanje anotacije vrši se ispisivanjem oznake/karaktera „@“. Ova oznaka daje signal kompajleru da je reč o anotaciji. Ono što se ispisuje posle oznake „@“ predstavlja ime ili naziv anotacije.

Postoje četiri osnovne kategorije anotacija (Oracle, 2020):

1. normalne anotacije
2. obeleživačke anotacije (marker annotations)
3. anotacije sa jednim elementom (single element annotations)
4. anotacije sa više elemenata (multiple element annotations)

Pored navedenih kategorija anotacija postoje i ugrađene anotacije koje se mogu podeliti na predefinisane anotacije i meta anotacije (Oracle; java2novice, n.d.)

1. predefinisane anotacije

   @Deprecated – ova anotacije predstavlja marker anotaciju koja ukazuje da je ono što je njome deklarisano (klasa, metoda, polje itd.) zastarelo i zamenjeno novijom formom datog elementa ili će biti zamenjeno u skorijoj budućnosti.

   @Override – Override anotacija može biti samo korišćena nad metodama. Metoda podklase koja je anotirana Override anotacijom nadjačava metodu istog imena iz superklase.

   @SuppressWarnings – koristi se da bi se naznačilo kompajleru da suzbije određena upozorenja i obaveštenja.

   @SafeVarargs – anotacija čijom primenom se obezbeđuje da anotirana metoda ili konstruktor ne izvršava operacije koje nisu bezbedne na svoje varijabilne argumente (Varargs).

   @FunctionalInterface – ovom anotacijom se označava da je deklaracija nad kojom je upotrebljena funkcionalni interfejs. Funkcionalni interfejs može imati samo jednu apstraktnu metodu. (Programiz, n.d.)

2. meta anotacije su anotacije koje se primenjuju na druge anotacije

   @Retention – ova anotacija služi za označavanje tačke/nivoa do kog će se anotacija primenjivati nakon čega će se njena upotreba odbaciti. Java definiše 3 smernice za zadržavanje (retention policies): SOURCE, CLASS i RUNTIME.

   @Documented – ova anotacija se koristi da bi se naznačilo da klase koje koriste ovu anotaciju to i prikažu u svojim generisanim JavaDoc dokumentima.

   @Target – ovom anotacijom ograničava se upotreba anotacija na specifične elemente (mete) koji mogu biti: anotacije, konstruktori, polja, lokalne varijable, metode, paketi, parametri ili bilo koji element klase

   @Inherited – koristi se za nasleđivanje anotacije iz superklase u podklasu.

   @Repeatable – ponavljajuće anotacije omogućavaju korišćenje iste anotacije više puta u okviru iste klase, metode ili polja što pre Java 8 verzije nije bilo moguće i rezultiralo bi greškom prilikom kompajliranja koda.

7. Spring framework

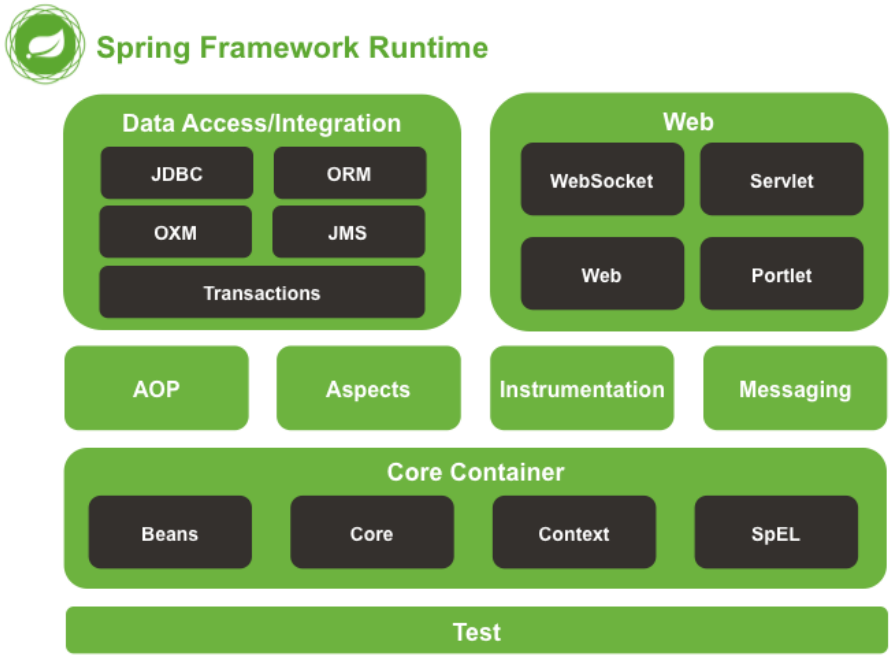
Spring je radni okvir (framework) za razvoj java web aplikacija. Ovaj radni okvir se koristi za razvoj enterprise aplikacija uz upotrebu POJO (plain old java object) klasa. Spring omogućava brže, lakše i sigurnije java programiranje. Fokus na brzinu, jednostavnost i produktivnost, uticali su da on tokom godina postane najpopularniji radni okvir za programiranje u Javi. Jedni od najvažnijih aspekata Spring radnog okvira su ubrizgavanje zavisnosti (Dependency Injection) i inverzija kontrole (Inversion of Control).
Ubrizgavanje zavisnosti predstavlja tehniku gde jedan objekat prima druge objekte od kojih zavisi. To je šablon putem kojeg se implementira inverzija kontrole (IoC). On omogućava kreiranje zavisnih objekata izvan klasa i omogućava pristup kreiranim objektima u okviru klase. (TutorialsTeacher, n.d.)

Primajući objekat se naziva klijent (client) a onaj koji se prosleđuje ili ubrizgava se zove servis (service). Ono što ubrizgava servis klijentu se naziva injektor (injector). Umesto da klijent odabira koji će servis koristiti, injektor govori klijentu koji servis da koristi. Injekcija (injection) se odnosi na prosleđivanje zavisnosti (servisa) u objekat (klijentu) koji će je koristiti. Dakle, prosleđivanje servisa klijentu, umesto da mu dozvoli da sam kreira ili pronađe servis, predstavlja osnovu ubrizgavanja zavisnosti. (Wikipedia, n.d.)

Spomenuli smo da ubrizgavanje zavisnosti predstavlja formu za implementaciju inverzije kontrole. Spring poseduje kontejner za inverziju kontrole (IoC Container) pomoću kog se vrši automatsko ubrizgavanje zavisnosti. Pomoću njega se upravlja kreacijom objekata i reguliše njihov životni vek kao i ubrizgavanje zavisnosti u klase. IoC Container kreira objekat određene klase i takođe ubrizgava sve zavisne objekte putem konstruktora, promenljive ili metode prilikom izvršenja koda (runtime) i odlaže ga (odbacuje) u odgovarajuće vreme. Ovo se radi automatski tako da korisnik ne mora manuelno da kreira i upravlja objektima. (TutorialsTeacher, n.d.)

7.1. Moduli

Spring radni okvir poseduje oko 20 funkcionalnosti. Ovi moduli se mogu podeliti u određene grupe prikazane na slici 2 a to su "Core Container", "Data Access/Integration", "Web", "AOP" (Aspect oriented programming), "Instrumentation" i "Test".



Slika 2. Moduli Spring framework-a (Spring, n.d.)

1. Core Container

Core Container je sačinjen od sledećih modula: Core, Beans, Context i Expression Language
Core (jezgro) i Beans (zrna) predstavljaju središnje, fundamentalne delove radnog okvira uključujući i inverziju kontrole i ubrizgavanje zavisnosti. BeanFactory predstavlja kontejner koji instancira, konfiguriše i upravlja zrnima. Ova zrna međusobno sarađuju i imaju definisane zavisnosti među sobom.
Context modul predstavlja sredstvo za pristup objektima. Ono nasleđuje svoje karakteristike iz Core i Beans modula i koristi se za instanciranje i konfiguraciju zrna a to čini iščitavanjem metapodataka iz XML-a, java anotacija ili java koda iz konfiguracionih fajlova. Interfejs ApplicationContext je glavna tačka Context modula.
Expression Language modul predstavlja moćno sredstvo, odnosno jezik za postavljanje upita i upravljanje objektima prilikom izvršavanja koda (runtime). Ovaj jezik služi za postavljanje (setting) i dobijanje (getting) vrednosti varijabli, invokaciju tj. pozivanje metoda, pristup nizovima, kolekcijama, logičkim i aritmetičkim operacijama, imenovanim varijablama kao i za pronalaženje objekata po imenu iz Spring-ovog kontejnera za inverziju kontrole (IoC Container).

2. Data Access/Integration

Data Access/Integration sloj se koristi za upravljanje podacima a sastoji se od JDBC (Java Database Connectivity), ORM (Object Relationship Mapping), OXM (Object XML Mapping), JMS (Java Messaging Service) i transakcionih modula. Ovi moduli pružaju podršku za interakciju sa bazom podataka. Svrha ovog sloja je da smanji količinu koda i umanji dugačko i zamorno kodiranje za pristup bazi podataka. Korišćenjem odgovarajućih radnih okvira poput Hibernate-a, JPA (Java Persistence API), JDO (Java Data Objects) i sličnih koji se mogu integrisati u Spring, eliminiše se potreba za pisanjem koda za otvaranje konekcije, pokretanje transakcije, izvršavanje transakcije i zatvaranje konekcije. Pored ovoga, navedeni radni okviri kada se integrišu sa Springom olakšavaju testiranje kreiranih aplikacija i pružaju bolju podršku za "exception handling" i upravljanje transakcijama.

3. Web

Web sloj se sastoji od Web, Web-Servlet, Web-Socket i Web-Portlet modula.

Springov web modul pruža osnovne web orijentisane funkcionalnosti poput otpremanja jedne datoteke iz više delova (multipart upload) kao i inicijalizaciju IoC kontejnera korišćenjem servlet osluškivača (listeners) i web orijentisanog application context-a.
Web-Servlet modul sadrži Spring MVC (model-view-controller) radni okvir koji je neophodan deo web aplikacija.
MVC radni okvir se koristi za logičku separaciju i podelu koda, odnosno razdvajanje različitih aspekata aplikacije gde imamo "input logic", "business logic" i "UI logic".
Model vrši enkapsulaciju podataka i sastoji se od POJO-a.
View je odgovoran za prikazivanja podataka iz modela i generisanje HTML autputa koji će biti prikazan na pretraživaču klijenta.
Controller se koristi za obradu korisničkih zahteva i kreiranje funkcionalnog modela koji se otprema za prikaz (rendering). (Tutorialspoint, n.d.)
Web-Socket modul omogućava bidirekcionu komunikaciju između klijenta i servera u web aplikacijama.
Web-Portlet modul omogućava korišćenje MVC implementacije u portlet okruženju.

4. AOP i instrumentalizacija

Spring ima odvojen AOP (Aspect-oriented programming) radni okvir koji ima za cilj da poveća modularnost cross-cutting problema. Cross-cutting brige su delovi programa koji zavise ili utiču na veliki broj delova sistema. Ovaj radni okvir se koristi za presretanje nekih procesa, na primer, pri izvršavanju neke metode Spring AOP presreće metodu i dodaje joj dodatnu funkcionalnost pre ili posle izvršavanja navedene metode.
Instrumentation modul se koristi za nadgledanje nivoa performansi, dijagnostikovanje grešaka i za pisanje informacionih poruka u okviru osluškivača (trace listener). Svrha osluškivača jeste da prima i skladišti poruke.

5. Test

Test modul se koristi za testiranje komponenti Springa ili aplikacija. Moguće je lako instancirati objekte korišćenjem "new" operatora i upotrebiti lažne objekte (mock objects) umesto stvarnog ubrizgavanja zavisnosti za testiranje koda u izolaciji, odvojeno od aplikacije a pored toga pruža i mogućnosti primene ugrađene baze podataka čime se ubrzava vreme testiranja aplikacije.

7.2 Spring Boot

Spring Boot predstavlja ekstenziju Spring radnog okvira koji poseduje unikatna svojstva koja olakšavaju razvoj aplikacija. Dok je Spring tu da pruži fleksibilnost prilikom programiranja, Spring Boot je više fokusiran na smanjivanje dužine koda čime se pojednostavljuje razvoj aplikacije. Korišćenjem Spring Boot-a, programeri ne moraju da definišu individualno svaku stvar koja im je potrebna za izradu funkcionalne aplikacije već mogu upotrebiti Spring Boot Configuration anotacije čime se smanjuje vreme za izradu a pored toga uopšte nije obavezno koristiti XML za definisanje zavisnosti između objekata jer se to može uraditi kroz anotacije.

7.3 Maven

Apache Maven je softver za upravljanje projektima koji je baziran na konceptu project object model (POM). Maven upravlja izradom projekta, izveštavanjem i dokumentacijom. Za izradu aplikacije moguće je koristiti određene zavisnosti (dependencies) koje olakšavaju rad sa zrnima (beans), bazama podataka i smanjuju količinu ukupnog koda za pisanje aplikacije. Upotreba ovih zavisnosti i njihov uvoz u sam Spring se pojednostavljuje kroz Maven. Pretpostavimo da naš projekat koristi određenu verziju MySQL-a za pristup bazi podataka. U međuvremenu će izaći novija verzija MySQL-a sa određenim poboljšanjima koju bismo želeli da dodamo u naš projekat ili aplikaciju. Bez project manager-a poput Maven-a morali bismo da odemo na web sajt MySQL-a i pronađemo noviju verziju, pritom proverimo da li je kompatibilna sa našom verzijom Spring radnog okvira i zatim ručno zamenimo stariju verziju MySQL-a novijom verzijom. Ovo i ne predstavlja toliki problem ukoliko radimo sa malim brojem zavisnosti u projektu ali ozbiljnije aplikacije sadrže veliki broj zavisnosti koje bismo morali ručno da ažuriramo ukoliko ne bismo koristili project manager poput Maven-a.
Svaki projekat sadrži POM datoteku koja je XML fajl u kom se nalaze detalji projekta. Ovi detalji se odnose na ime projekta, verziju, tip paketa, zavisnosti, maven plugine i slično. Upravo modifikovanjem ovog pom.xml fajla možemo uvoziti zavisnosti i upravljati projektom na lak način uz pomoć Maven-a. (Robinson)

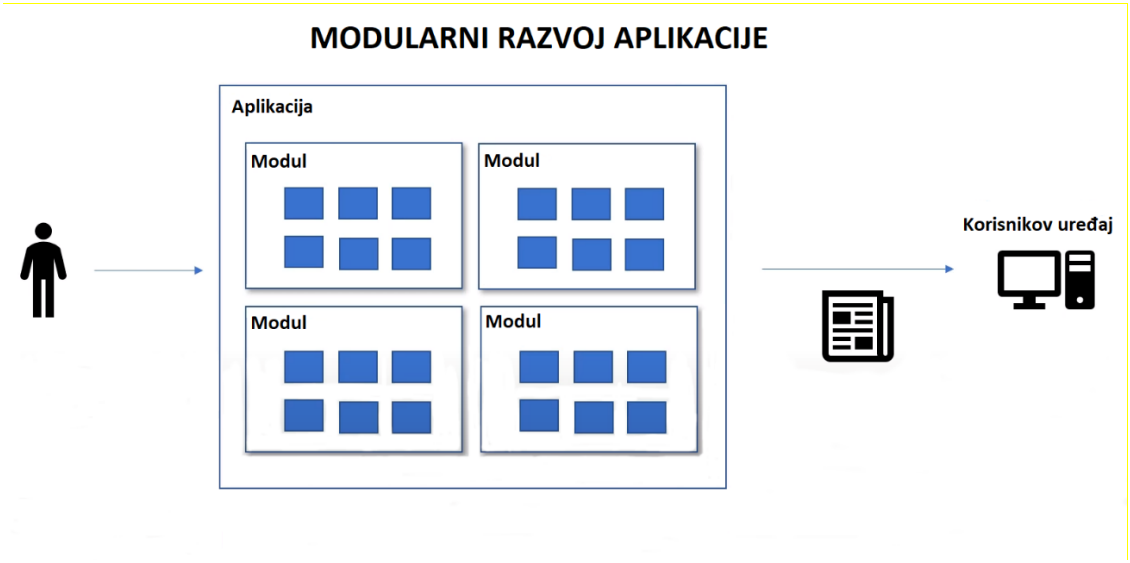
8. Mikroservisi

Da bismo ispričali priču o mikroservisima i njihovom uticaju na današnji razvoj aplikacija potrebno je vratiti se neko vreme unazad i skrenuti pažnju na njihove preteče.

Tradicionalni razvoj aplikacija se još nazivao i monolitnim s obzirom da je celina bila sačinjena od jednog velikog dela. Po ovom principu, ako je bilo potrebno dodati nove funkcionalnosti u aplikaciju to je podrazumevalo pisanje i dodavanje novog koda u tu celinu tako da se vremenom veličina svake aplikacije sve više povećavala sa dodavanjem novih funkcionalnosti. (Brains, 2019)
Tradicionalni razvoj aplikacije prikazan je na slici 3.

Slika 3. Tradicionalni razvoj aplikacije (Brains, 2019)

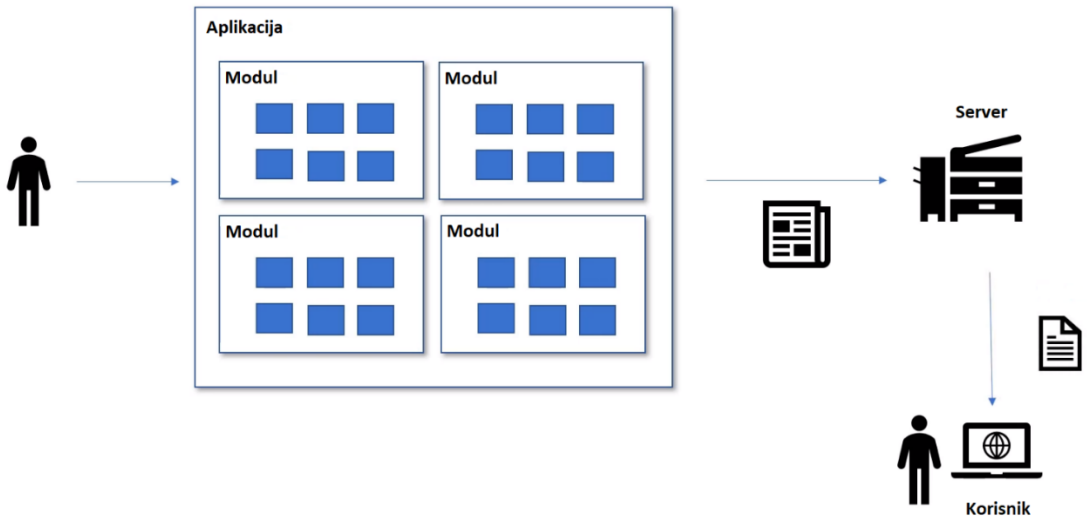
Vremenom, programeri su počeli da razbijaju celokupan kod na manje delove koji se nazivaju moduli kako bi lakše upravljali većim, kompleksnijim projektima. Moduli predstavljaju manje, nezavisne delove aplikacije koji obezbeđuju određeni deo funkcionalnosti. Oni su bili namenjeni za višekratnu upotrebu, tako da jednom napisan modul u ranijem projektu može da se iskoristi u nekom drugom projektu kako bi upotpunio neku drugu aplikaciju. Međutim, čak i kada se logika aplikacije razbije na module, na samom kraju ona se ipak razvija kao jedna celina, koja u krajnjoj instanci predstavlja jednu datoteku koja se pokreće na korisnikovom uređaju. Bez obzira što je kod bio struktuiran na manje delove, krajnji rezultat je bila jedna aplikacija za koju nije bilo važno kako izgleda i koja je trebala da funkcioniše samo na jednom uređaju. (Brains, 2019)
Modularni razvoj aplikacije prikazan je na slici 4.



Slika 4. Modularni razvoj aplikacije (Brains, 2019)

Ovakav način razvoja aplikacija je trajao neko vreme dok se nije popularizovala ideja o web aplikacijama. Umesto instaliranja aplikacija na korisnikovom uređaju kako je do tada bilo rađeno, web aplikacije bi bile otpremljene/instalirane na serverima. Da bi funkcionisala, web aplikaciji je potreban web server, aplikacioni server (application server) i u nekim slučajevima baza podataka. Web server upravlja zahtevima koji dolaze od korisnika (klijenta) i prosleđuje ih odgovarajućem aplikacionom serveru. Aplikacioni server izvršava zahtev korisnika i ako je potrebno pristupa bazi podataka da bi taj zahtev izvršio nakon čega šalje rezultate zahteva web serveru i na samom kraju web server respondira korisniku sa zahtevanim informacijama koji se prikazuju na displeju korisnika. (Brains, 2019)

Uključivanje servera u celokupnu priču nije na početku dovelo do nekih većih promena. Modifikacija koja je usledila jeste da aplikacija više nije direktno instalirana na uređajima korisnika već samo na jednom serveru kojem bi ti korisnici pristupili. Osim toga, struktura koda se nije izmenila. Iako je logika aplikacije bila lepo organizovana i podeljena na sitnije module ipak je krajnji ishod bila jedna aplikacija i jedna datoteka u kojoj je čitav kod bio isprepleten. Jedina razlika u odnosu na prethodnu priču jeste to da je aplikacija kao takva instalirana na serveru dok je ranije u istoj takvoj formi bila instalirana na uređaju korisnika. (Brains, 2019)
Serverski razvoj aplikacije prikazan je na slici 5.

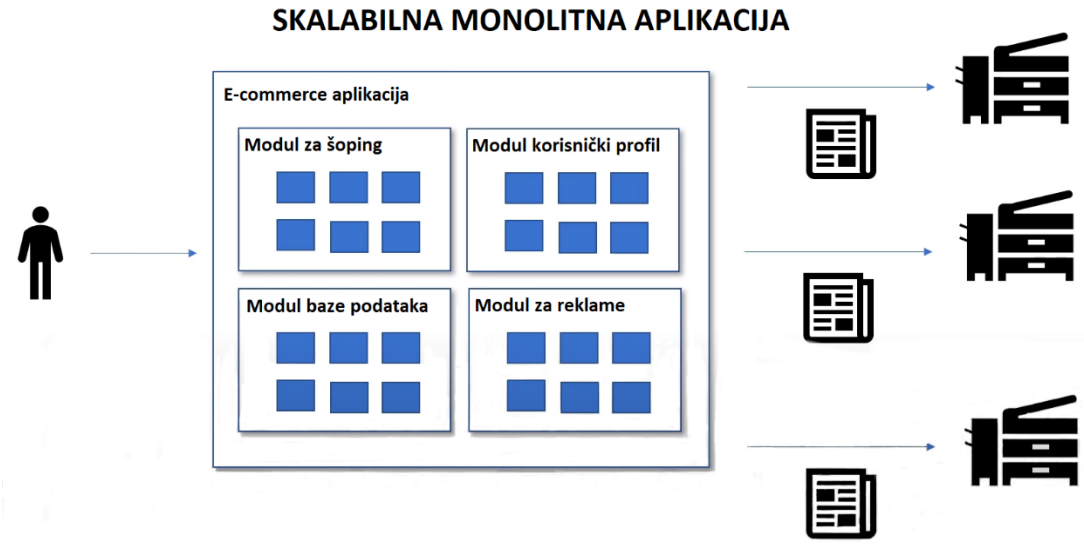


Slika 5. Serverski razvoj aplikacija (Brains, 2019)

Sa pojavom web aplikacija one su postepeno postajale veće, brže i kompleksnije te monolitski pristup razvoju aplikacija gde imamo jednu aplikaciju nije više bio zadovoljavajući.
Jedan od najvećih nedostataka ovakve arhitekture jeste što dodavanje novog koda ili greška u kodu može uticati na nemogućnost pokretanja aplikacije tako da je uvek potrebno testirati celu aplikaciju jer se ne može sa sigurnošću znati na koji deo aplikacije je novi kod uticao. S obzirom da se čitava aplikacija pokreće iz jednog dela, potrebno je testirati uvek celu aplikaciju.
(Brains, 2019)

Drugi nedostatak sa monolitskim arhitekturama je skalabilnost. Kao primer možemo uzeti neku kompaniju koja se bavi prodajom artikala široke potrošnje. U vreme većih rasprodaja i sniženja na njihovim proizvodima može se očekivati veći pristup njihovom web sajtu za razgledanje i poručivanje datih proizvoda. U tim periodima promet je značajno uvećan nego što je uobičajeno. Ovaj problem sa opterećenjem sajta u periodima kada mu veći broj ljudi pristupa u istom vremenskom periodu se rešava takozvanim "elastičnim serverima". Kada se promet poveća, povećava se i broj servera da bi se zadovoljila novonastala rastuća tražnja, tj. da bi se omogućio nesmetani pristup sajtu svim korisnicima, a kada se promet smanji, smanjuje se i broj servera. (Brains, 2019)

Pretpostavimo da imamo web sajt koji je izgrađen po monolitnoj arhitekturi. Ovaj monolit se sastoji iz više modula: modul za šoping, modul korisničkog profila, modul za interakciju sa bazom podataka i modul za reklamni materijal. Sada pretpostavimo da je došlo do rasta prometa na web stranicama za šoping. Umesto da se poveća broj servera samo za šoping modul jer se one u tom periodu i najviše koriste, pošto je reč o monolitnoj arhitekturi moraće da se pojačaju (scale-up) i sve ostale funkcionalnosti odnosno drugi moduli s obzirom da je reč samo o jednoj aplikaciji. (Brains, 2019)
Skalabilna monolitna aplikacija prikazana je na slici 6.



Slika 6. Skalabilna monolitna aplikacija (Brains, 2019)

Nakon ovakvog tipa arhitekture gde je svaka funkcionalnost bila isprepletena sa svim ostalim funkcionalnostima u okviru jedne aplikacije, počelo se sa razgrađivanjem tako velike aplikacije na "mini aplikacije" koje bi bile otpremljene na različite servere koji su međusobno mogli da komuniciraju putem mreže da bi zajedno formirali jednu celinu, odnosno kompletnu aplikaciju. (Brains, 2019)

Pretpostavimo primer web sajta za kupovinu proizvoda online. Moguće je izgraditi jednu aplikaciju koja će sadržati katalog proizvoda i koja ima tu jednu jedinu funkcionalnost koja bi bila otpremljena na jednom serveru. Zatim aplikaciju za procesiranje porudžbine, koja će takođe biti zasebna i nalaziti se na drugom serveru i profil korisnika – koji bi predstavljao treću aplikacija na trećem serveru. (Brains, 2019)

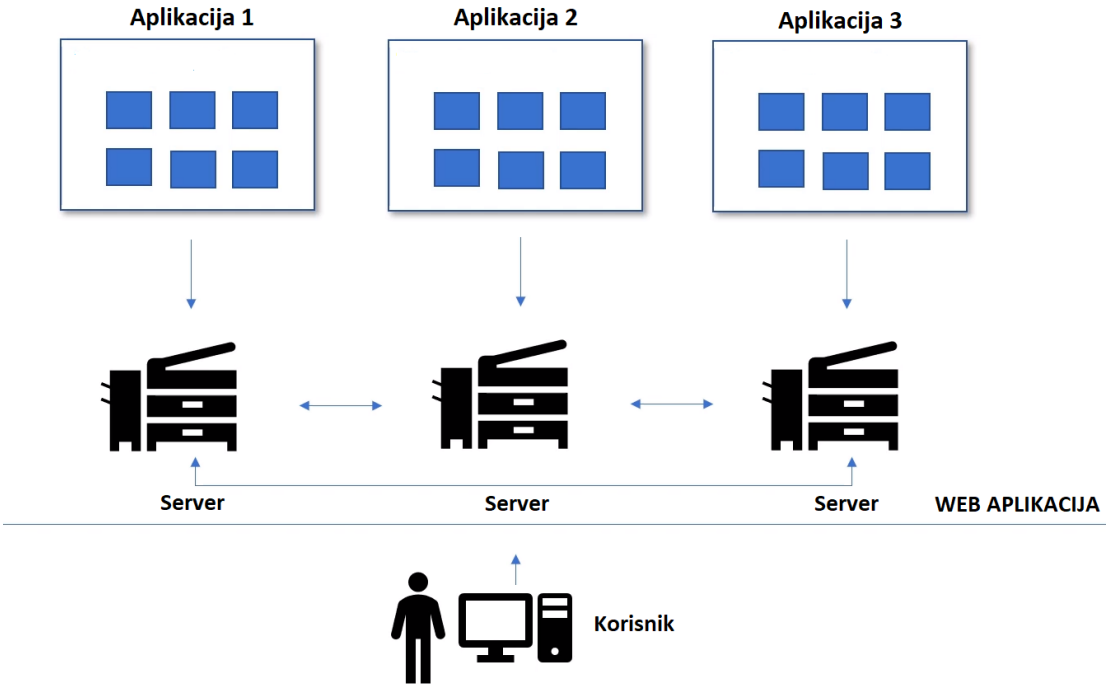
Kada korisnik želi da pogleda katalog proizvoda on to zapravo čini putem "view" aplikacije koja putem REST API poziva šalje zahtev aplikaciji za katalog proizvoda koja zatim vraća katalog svih proizvoda aplikaciji koja je učinila zahtev, odnosno view aplikaciji koja na kraju prikazuje korisniku HTML stranicu sa listom datih proizvoda. Na taj način se ostvaruje komunikacija između ovih mini aplikacija na mreži. (Brains, 2019)

Prednost ovakvog tipa arhitekture se odnosi na lakoću testiranja i vršenja izmena u okviru mini aplikacija. Ako su izvršene promene u okviru aplikacije za katalog proizvoda, potrebno je samo testirati samo taj deo aplikacije s obzirom da ona predstavlja nezavisnu aplikaciju.
Druga prednost se odnosi na skalabilnost. Prilikom povećanog prometa potrebno je samo kreirati više instanci servera za onu mini aplikaciju gde je promet i uvećan, u ovom primeru za katalog proizvoda a ne za čitavu aplikaciju. (Brains, 2019)

Ove mini aplikacije su zapravo mikroservisi. Mikroservisi nastaju razbijanjem aplikacije ili servisa na samostalne, nezavisne aplikacije. Oni se mogu pokrenuti na različitom hardveru i različitim serverima ali pritom komuniciraju međusobno putem REST API-a kako bi povezali sve samostalne funkcionalnosti u jednu celinu koju će predstavljati celokupna aplikacija. Dok smo kod monolitne arhitekture imali jednu veliku aplikaciju, kod mikroservisa imamo više manjih aplikacija koje su povezane i rade zajedno prilikom rada programa da bi formirale kompletnu aplikaciju. (Brains, 2019)
Struktura mikroservisne aplikacije prikazana je na slici 7.

Ovakav tip arhitekture pogodan je za više različitih timova, gde će svaki biti zadužen za izgradnju svoje manje aplikacije. Pored toga, mogu biti napisani i različitim programskim jezicima, a REST će se pobrinuti za njihovo funkcionisanje. (Brains, 2019)

Osim ovih prednosti može se očekivati da i ovakav tip arhitekture prate neki nedostaci. Dok smo ranije bili zaduženi za jednu veliku aplikaciju, sada se možemo suočiti sa desetinama ili stotinama malih aplikacija u formi mikroservisa tako da je neophodno postarati se ne otići u drugu krajnost i bespotrebno zakomplikovati stvari. Potrebno je izvršiti dobru procenu prilikom razdvajanja aplikacije na mikroservise. Ukoliko je za svaku novu funkcionalnost neophodno izvršiti izmene koda u deset ili više mikroservisa da bi sve funkcionisalo kako treba takva organizacija projekta poražava svoju svrhu i ne može se reći da je u pitanju efikasan razvoj aplikacije. (Brains, 2019)

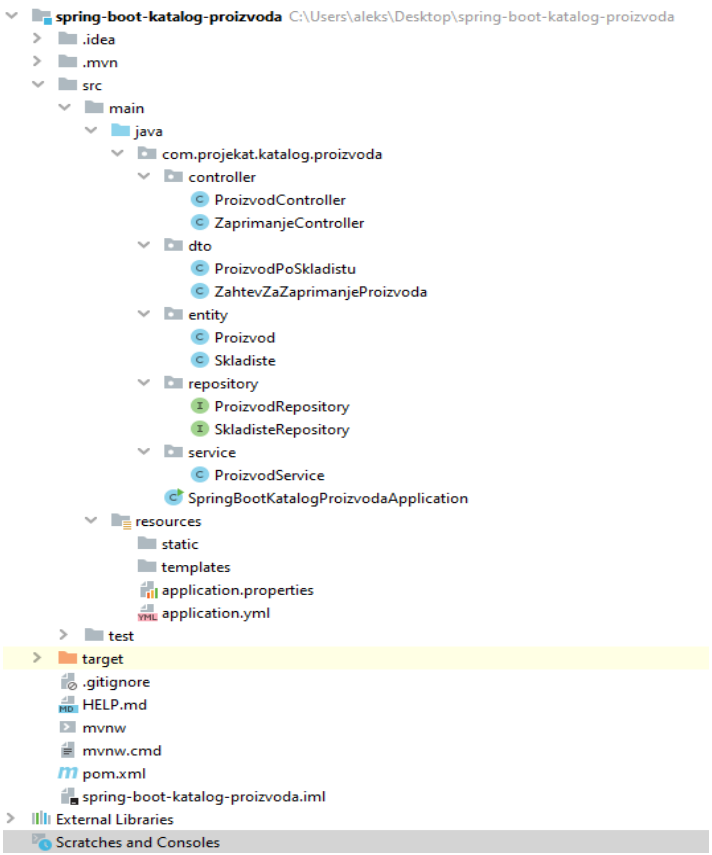


Slika 7. Mikroservisi (Brains, 2019)

9. Prikaz aplikacije

Za izradu aplikacije korišćeno je okruženje IntelliJ i Spring framework koji se zasniva na Java programskom jeziku. Sam projekat je podeljen u više delova kako bi se prikazala suština mikroservisa koji bi trebalo da funkcionišu kao nezavisne i odvojene aplikacije koje formiraju jednu celinu. Celokupan projekat treba da pruži uvid u katalog proizvoda i poručivanje proizvoda sa osnovnim funkcijama i metodama za upravljanje podacima u bazi podataka. Pored toga, u projekat je integrisan lokator servisa (service discovery) i mrežni prolaz (gateway) pomoću kojih ovi servisi prepoznaju jedni druge na mreži i zajedno obezbeđuju funkcionalnosti koje će biti dalje objašnjene tokom komentarisanja samog projekta. Za testiranje API-a korišćen je Postman. Postman predstavlja platformu za razvijanje i testiranja API-a koja poseduje prijateljski korisnički interfejs za kreiranje HTTP zahteva, bez pisanja opširnog koda da bi se testirala funkcionalnost aplikacije. U okviru adress bar-a potrebno je upisati URL koji smo napisali u projektu u okviru anotacija za mapiranje zahteva. Posle toga je potrebno izabrati tip HTTP zahteva (POST, GET, UPDATE, DELETE itd.) i izabrati željeni tip formata u kom će biti prikazani podaci.
MySQL Workbench aplikacija je korišćena za kreiranje i interakciju sa bazom podataka.
Za celokupan projekat kreirana je jedna baza podataka pod nazivom „proizvod“ sa 3 tabele, „proizvod_tbl“, „skladiste_tbl“ i „porudzbina_tbl“.

Projekat: katalog proizvoda



Slika 8. Struktura aplikacije katalog proizvoda

Na slici 8 je prikazana struktura aplikacije katalog proizvoda.

Pre nego što počnemo sa komentarisanjem programske logike treba napomenuti da je aplikacija rađena u Maven project manager-u tako da svaki projekat sadrži POM datoteku koja je XML fajl u kom se nalaze detalji projekta. U pom.xml fajlu se navode zavisnosti koje su neophodne za izgradnju projekta.

Sadržaj pom.xml fajla:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Spring Web zavisnost čiji je kod prikazan iznad je osnova izgradnje svake web, uključujući i RESTful aplikacije koja koristi Spring MVC (model-view-controller) koji pruža osnovne funkcionalnosti Spring okvira poput inverzije kontrole i ubrizgavanja zavisnosti i omogućava da aplikacija funkcioniše na servletu (serveru).

```
<dependency>
   <groupId>mysql</groupId>
   <artifactId>mysql-connector-java</artifactId>
   <scope>runtime</scope>
</dependency>
```

Za izradu ove aplikacije korišćena je MySQL baza podataka tako da je okviru zavisnosti potrebno uvesti MySQL konektor koji je zapravo drajver putem kojeg se omogućava povezivanje sa MySQL bazom podataka. Ova zavisnost prikazana je u kodu koji se nalazi iznad.

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

JPA (Java Persistence API) definiše koncepte za skladištenje, pristup i upravljanje java objektima u relacionoj bazi podataka. Termin *persistance* (eng.) odnosi se na kreiranja java objekata koji će nadživeti aplikacioni proces koji ih je kreirao, dakle za njihovo skladištenje. JPA se koristi za skretanje fokusa sa tehnološke logike te omogućava kreatoru aplikacije da se više fokusira na pisanje biznis logike smanjujući količinu koda za upravljanje bazom podataka. Ova zavisnost prikazana je u kodu koji se nalazi iznad.
Spring Data JPA ne prikazuje detalje i mehanizme za skladištenje podataka kako bi se programer više fokusirao na biznis kod, a da bi se koristio potrebno je kreirati interfejs za entitet koji će naslediti JPA. Pored toga on dolazi sa predefinisanim metodama tako da nije potrebno pisati sam kod za upravljanje podacima (insert, read i slično) sve dok se ne radi o kompleksnijim metodama i operacijama.
(Janssen, n.d.)

```
<dependency>
   <groupId>org.projectlombok</groupId>
   <artifactId>lombok</artifactId>
   <optional>true</optional>
</dependency>
```

Lombok čija zavisnost je prikazan u kodu koji se nalazi iznad predstavlja java biblioteku koja olakšava rad sa POJO-ima (plain old java objects). U ovom projektu su korišćene Lombok anotacije koje olakšavaju enkapsulaciju objekata putem *getter-a* (eng.) i *setter-a* (eng.) kao i za kreiranje konstruktora sa određenim argumentima.

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

Eureka server i klijent predstavljaju REST servis koji se koristi za lociranje (mikro)servisa i njihovu komunikaciju kao i za *load balancing* (eng.) koji se koristi za efikasnu distribuciju mrežnog saobraćaja kako ne bi dolazilo do većih zastoja prilikom slanja zahteva na mreži. Load balancer je zadužen za ispunjenje ovih zahteva uz maksimalne brzine vodeći računa da ne dođe do opterećenja nijednog servera. Ako server nije dostupan, load balancer preusmerava mrežni saobraćaj i zahteve na raspolagajuće servere koji su dostupni (online).
Ova zavisnost je predstavljena u kodu koji se nalazi iznad.

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-test</artifactId>
   <scope>test</scope>
</dependency>
```

Spring Boot Starter Test je integrisan u okviru bilo kog Springovog projekta i sadrži funkcionalnosti koje olakšavaju testiranje aplikacija.
Ova zavisnost čiji kod je prikazan iznad obezbeđuje i H2 integrisanu bazu podataka koja eliminiše potrebu za kreiranjem i konfigurisanjem stvarne baze podataka čime se ubrzava testiranje aplikacija.

Ovde se završava objašnjenje zavisnosti koje su ključni deo pom.xml fajla

application.properties fajl je korišćen za podešavanje konekcije sa bazom podataka dok u application.yml fajlu navodimo port 8500 na kom će aplikacija biti aktivna kada bude pokrenuta i aplikaciji dajemo naziv “KATALOG-PROIZVODA”. To je i naziv pod kojim će ona biti registrovana na eureka serveru o kojoj će kasnije biti reči.

U paketu entity nalaze se dve klase – “Proizvod” i “Skladiste”. U ove klase se unose polja koja će predstavljati sve kolone tabela – “proizvod_tbl” i “skladiste_tbl”. Sva polja su privatna čime je obezbeđena enkapsulacija podataka. Kako bi se obezbedio pristup ovim poljima spolja, izvan same klase, koriste se getter-i i setter-i ali za razliku od standardnog načina za njihovo kreiranje koriste se Lombok anotacije za njihovo aktiviranje.

U nastavku je prikazan kod klase „Proizvod“ koja se nalazi u paketu „entity“:

```
package com.projekat.katalog.proizvoda.entity;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import javax.persistence.*;
@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity
@Table(name = "proizvod_tbl")

public class Proizvod {
    @Id
    private int idProizvoda;
    private String nazivProizvoda;
    private int kolicina;
    private double cena;
    private int id_skladista_FK;
}
```

Lombok java library koji smo dodali u zavisnosti u okviru pom.xml fajla nam omogućava korišćenje anotacija @Data, @AllArgsConstructor i @NoArgsConstructor. Ove anotacije često idu jedna uz drugu i smanjuju količinu ponavljajućeg koda koji se često javlja u klasama a odnose se na podešavanje getter-a i setter-a.
Anotacija @Entity je neophodna jer naznačava Java Persistance API-u (JPA) da klasa sa datom anotacijom treba biti mapirana kao tabela. Svaka instanca entiteta će predstavljati jedan red u tabeli baze podataka.
@Table anotacija se koristi za definisanje naziva tabele u bazi podataka. U ovom slučaju naziv tabele će biti “proizvod_tbl”. Ukoliko ne bismo ovom anotacijom naznačili naziv tabele JPA bi sam dodelio naziv tabele koji bi se podudarao sa nazivom entitet klase.
@Id anotacija se koristi da se naznači da će polje sa datom anotacijom predstavljati primarni ključ.

Ispod je prikazan kod klase „Skladiste“ koja se nalazi u paketu „entity“:

```
package com.projekat.katalog.proizvoda.entity;
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import javax.persistence.*;
import java.util.List;
@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity
@Table(name = "skladiste_tbl")
```

```
public class Skladiste {
    @Id
    @GeneratedValue
    private int idSkladista;
    private String nazivSkladista;
    private String adresaSkladista;

    @OneToMany(targetEntity = Proizvod.class, cascade = CascadeType.ALL)
    @JoinColumn(name = "id_skladista_FK", referencedColumnName =
"idSkladista")
    @JsonIgnoreProperties("id_skladista_FK")
    private List<Proizvod> proizvodi;
}
```

Klasa "Skladiste" koristi iste anotacije koje su navedene i u klasi "Proizvod" a pored toga u okviru primarnog ključa je dodata anotacija @GeneratedValue što znači da će primarni ključ biti automatski generisan od strane JPA prilikom kreiranja instance.

Pošto želimo da zapremimo više proizvoda u okviru jednog skladišta koristi se anotacija za specifikaciju odnosa između entiteta @OneToMany gde se u okviru "targetEntity" unosi klasa "Proizvod". "CascadeType" predstavlja određeni mehanizam JPA/Hibernate-a kojim se upravlja odnosima između entiteta i konfiguriše se njihov životni vek, brisanje, i istrajanje zavisnog entiteta (dete entitet) kada je obrisan nezavisan entitet (roditelj entitet) i slično.
@JoinColumn predstavlja kolonu koja će predstavljati strani ključ u tabeli "proizvod_tbl" a to će biti polje "idSkladista".
Kada god kreiramo objekat klase "Skladiste" i dodamo listu proizvoda, "idSkladista" će se pojaviti kao strani ključ u tabeli "proizvod_tbl" pod nazivom "id_skladista_FK".

Paket "repository" sadrži dva interfejsa: "ProizvodRepository" i "SkladisteRepository". Sve klase koje će predstavljati repozitorijum treba označiti anotacijom @Repository i one treba da naslede klasu "JpaRepository" navođenjem ključne reči *extends* (eng.). Repozitorijum sadrži mehanizme za enkapsulaciju podataka, i obezbeđuje metode za upravljanje podacima poput metoda za pretragu "findById", brisanje "deleteById", čuvanje "save" i slično.

Dalje je prikazaan kod interfejsa ProizvodRepository u okviru paketa „repository":

```
package com.projekat.katalog.proizvoda.repository;
import com.projekat.katalog.proizvoda.entity.Proizvod;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
@Repository
public interface ProizvodRepository extends JpaRepository <Proizvod,
Integer>{

    Proizvod findBynazivProizvoda(String nazivProizvoda);
}
```

U nastavku je prikazan kod interfejsa „SkladisteRepository“ iz paketa „repository“:

```
package com.projekat.katalog.proizvoda.repository;
import com.projekat.katalog.proizvoda.dto.ProizvodPoSkladistu;
import com.projekat.katalog.proizvoda.entity.Skladiste;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;
import java.util.List;

@Repository
public interface SkladisteRepository extends JpaRepository <Skladiste,
Integer>{

    @Query("SELECT new
com.projekat.katalog.proizvoda.dto.ProizvodPoSkladistu(s.nazivSkladista ,
p.nazivProizvoda) FROM Skladiste s JOIN s.proizvodi p")
    public List<ProizvodPoSkladistu> prikaziProizvodeUSkladistu();
}
```

U interfejsu SkladisteRepository, unutar izlomljenih zagrada u ovom slučaju navodimo entitet klasu „Skladiste“ iz paketa „entity“ a tip vrednosti primarnog ključa je Integer (ceo broj) jer je u datoj klasi primarni ključ polje tipa Integer sa nazivom „idSkladista“.
Upit za bazu podataka biće kasnije objašnjen u tekstu.

U paketu „dto“ nalaze se dve klase: „ProizvodPoSkladistu“ i „ZahtevZaZaprimanjeProizvoda“. Naziv paketa „dto“ predstavlja skraćenicu za *data transfer object* (eng.). To su jednostavni objekti koji služe za enkapsulaciju podataka i njihovo slanje iz jednog podsistema aplikacije u drugi podsistem. Ne sadrže nikakvu biznis logiku i služe za prenos podataka.

Dalje je prikazan kod klase „ZahtevZaZaprimanjeProizvoda“ iz paketa „dto“:

```
package com.projekat.katalog.proizvoda.dto;
import com.projekat.katalog.proizvoda.entity.Skladiste;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.ToString;
@Data
@AllArgsConstructor
@NoArgsConstructor
@ToString
public class ZahtevZaZaprimanjeProizvoda {

    private Skladiste skladiste;
}
```

Ispod je prikazan kod klase „ZaprimanjeController“ u okviru paketa „controller“:

```
package com.projekat.katalog.proizvoda.controller;
import com.projekat.katalog.proizvoda.dto.ProizvodPoSkladistu;
import com.projekat.katalog.proizvoda.dto.ZahtevZaZaprimanjeProizvoda;
import com.projekat.katalog.proizvoda.entity.Skladiste;
import com.projekat.katalog.proizvoda.repository.ProizvodRepository;
import com.projekat.katalog.proizvoda.repository.SkladisteRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import java.util.List;

@RestController
@RequestMapping("/katalogproizvoda")
public class ZaprimanjeController {
    @Autowired
    private SkladisteRepository skladisteRepository;

    @Autowired
    private ProizvodRepository proizvodRepository;

    @PostMapping("/zaprimiproizvoduskladiste")
    public Skladiste zaprimanje(@RequestBody ZahtevZaZaprimanjeProizvoda
zahtev){
        return skladisteRepository.save(zahtev.getSkladiste());
    }
    @GetMapping("/prikazisve")
    public List<Skladiste> nadjiSve(){
        return skladisteRepository.findAll();
    }
    @GetMapping("/prikaziproizvodeposkladistu")
    public List<ProizvodPoSkladistu> nadjiProizvodeUSkladistu(){
        return skladisteRepository.prikaziProizvodeUSkladistu();
    }
}
```

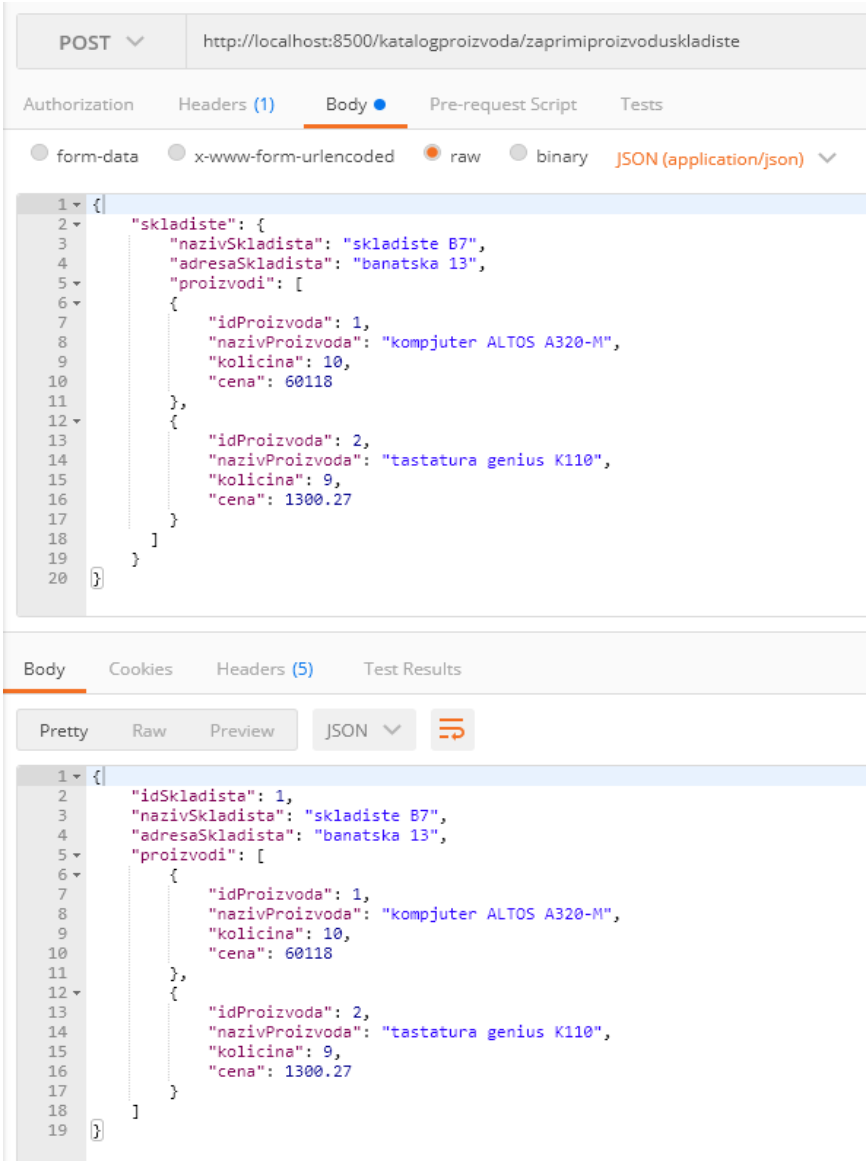
@RestController anotacija se koristi za kreiranje RESTful web servisa korišćenjem Spring MVC-a. Ova anotacija je neophodna za mapiranje zahteva i dodeljivanje tih zahteva definisanim handler-ima. Kada je generisan odgovor od strane handler-a, on se konvertuje u JSON ili XML odgovor.
@RequestMapping se koristi za mapiranje HTTP zahteva odgovarajućim handler metodama. Pošto je u zagradi pod navodnicima napisano „/katalogproizvoda“ to znači da ćemo ovom controller-u pristupiti putem URL adrese, navođenjem porta na kojoj je aplikacija aktivna (što je u ovom slučaju localhost 8500), zatim upisivanjem „/katalogproizvoda“ posle navođenja porta i na kraju se unosi naziv metode kojoj želimo pristupiti a koji je naznačen u okviru @GetMapping anotacija.

Korišćenjem @Autowired anotacije prepuštamo kreiranje objekata Springu čime se eliminiše potreba da ih samostalno kreiramo navođenjem ključne reči „new“. Prilikom inicijalizacije Spring kreira instance označenih klasa.

Kasnije kada želimo da iskoristimo objekte koji su označeni sa @Autowired anotacijom, Spring će ubrizgati ove prethodno kreirane instance u program. Autowire predstavlja metodu ubrizgavanje zavisnosti o kojoj je ranije bilo reči.
U okviru ove klase potrebno je ubrizgati objekte oba repozitorijuma koje smo ranije kreirali u okviru paketa „repository“ – „SkladisteRepository“ i „ProizvodRepository“ jer oni predstavljaju objekte koji nasleđuju klasu „JpaRepository“ tako da će oni biti zaduženi za interakciju sa bazom podataka.

Metoda sa imenom „zaprimanje“ treba da snimi objekat klase „Skladiste“ i listu proizvoda koji će se nalaziti u tom konkretnom skladištu. Iz tog razloga je napravljena klasa „ZahtevZaZaprimanjeProizvoda“ u „dto“ (data transfer object) paketu. Iz te klase možemo „izvući“ objekat klase „Skladiste“ a iz objekta klase „Skladiste“ možemo „izvući“ listu proizvoda jer smo u okviru klase „Skladiste“ definisali listu sa nazivom „proizvodi“. To ćemo uraditi preko objekta „zahtev“ koji navodimo u okviru @RequestBody anotacije. Kao povratnu vrednost treba proslediti „skladisteRepository“ jer taj objekat nasleđuje „JpaRepository“ klasu koja ima metodu za snimanje objekata. U okviru naziva metoda „save“ kao parametar prosleđujemo „zahtev“ koji smo naveli u okviru anotacije @RequestBody. @RequestBody iščitava i pamti ono što je napisano u HTTP zahtevu i koristi određeni konvertor (HttpMessageConvertor) da bi zapisao RequestBody u objekat, u ovom slučaju „zahtev“.
@PostMapping anotacija je neophodna jer naznačava da je u pitanju HTTP POST zahtev i u skladu sa tim ga prosleđuje određenom handler-u koji je zadužen za tu metodu. U metodi „zaprimanje“ mi prosleđujemo određeni tekst serveru u JSON formatu koji metoda „skladisteRepository.save“ upisuje u bazu podataka.

Izlaz date metode prikazan je na slikama 9, 10 i 11.



Slika 9. Metoda zaprimanje, postman aplikacija

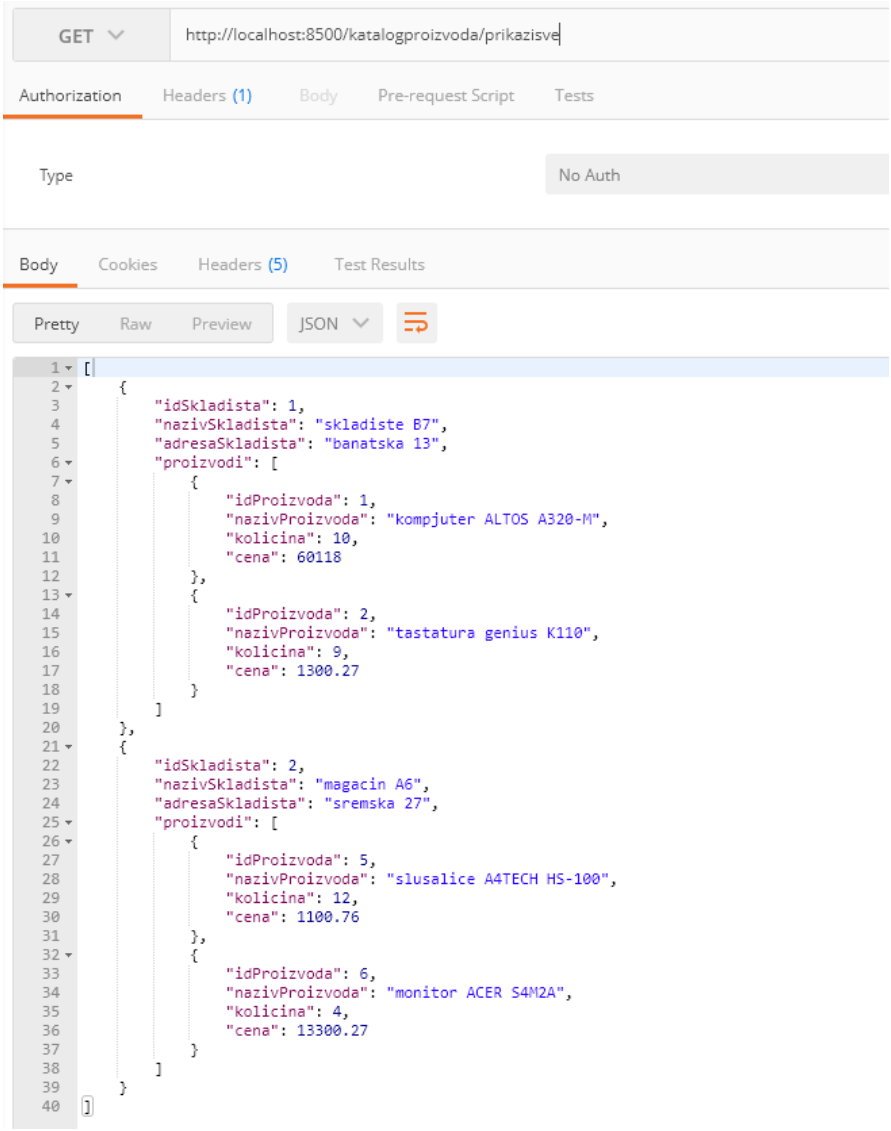
id_proizvoda	naziv_proizvoda	kolicina	cena	id_skladista_fk
1	kompjuter ALTOS A320-M	10	60118	1
2	tastatura genius K110	9	1300.27	1
5	slusalice A4TECH HS-100	12	1100.76	2
6	monitor ACER S4M2A	4	13300.27	2
NULL	NULL	NULL	NULL	NULL

Slika 10. Metoda zaprimanje, tabela proizvod_tbl u bazi podataka

id_skladista	adresa_skladista	naziv_skladista
1	banatska 13	skladiste B7
2	sremska 27	magacin A6
NULL	NULL	NULL

Slika 11. Metoda zaprimanje, tabela skladiste_tbl u bazi podataka

Metoda sa imenom „nadjisve“ i njen izlaz prikazani su na slici 12. Ova metoda treba da prikaže sve informacije i kolone iz baze podataka, kako za sva skladišta, tako i za sve proizvode. Kao i u prethodnoj metodi, kao povratna vrednost se navodi „skladisteRepository“ koja nasleđuje „JpaRepository“ klasu koja ima pristup metodi za prikaz podataka „findAll“. Ova metoda će prikazati sve podatke iz obe tabele na osnovu primarnog ključa iz jedne tabele.
@GetMapping anotacija je neophodna jer naznačava da je u pitanju HTTP GET zahtev i u skladu sa tim ga prosleđuje određenom handler-u koji je zadužen za tu metodu. Pošto je u pitanju zahtev za prikaz svih podataka i nemamo nikakve argumente, ne navodi se anotacija @RequestBody.



Slika 12. Metoda nadjiSve, postman aplikacija

Metoda „nadjiProizvodeUSkladistu“ treba da prikaže naziv svih skladišta i naziv svih proizvoda koji se nalaze u datim skladištima. Da bi se ovo izvršilo, potrebno je preciznije specificirati zahtev u okviru repozitorijuma „SkladisteRepository“.

@Query anotacija je neophodna pre pisanja HQL zahteva da označi da je u pitanju metoda koja će komunicirati sa bazom podataka. HQL je skraćenica za *Hibernate Query Language* (eng.) koji je deo Spring Data JPA sličan SQL-u ali se od njega razlikuje jer ne vrši direktno operacije nad tabelama i kolonama već nad objektima i njihovim svojstvima. Umesto naziva tabele u bazi podataka, koristi se naziv klase tako da predstavlja query jezik koji je nazavisan od baze podataka.

Sledeći upit treba da izvrši JOIN operaciju i da kao rezultat prikaže naziv skladišta iz tabele „skladiste_tbl“ i naziv proizvoda iz tabele „proizvod_tbl“. Da bi se uspešno izvršio neophodno je da napravimo klasu koja će imati dva polja sa nazivom „nazivSkladista“ i „nazivProizvoda“.

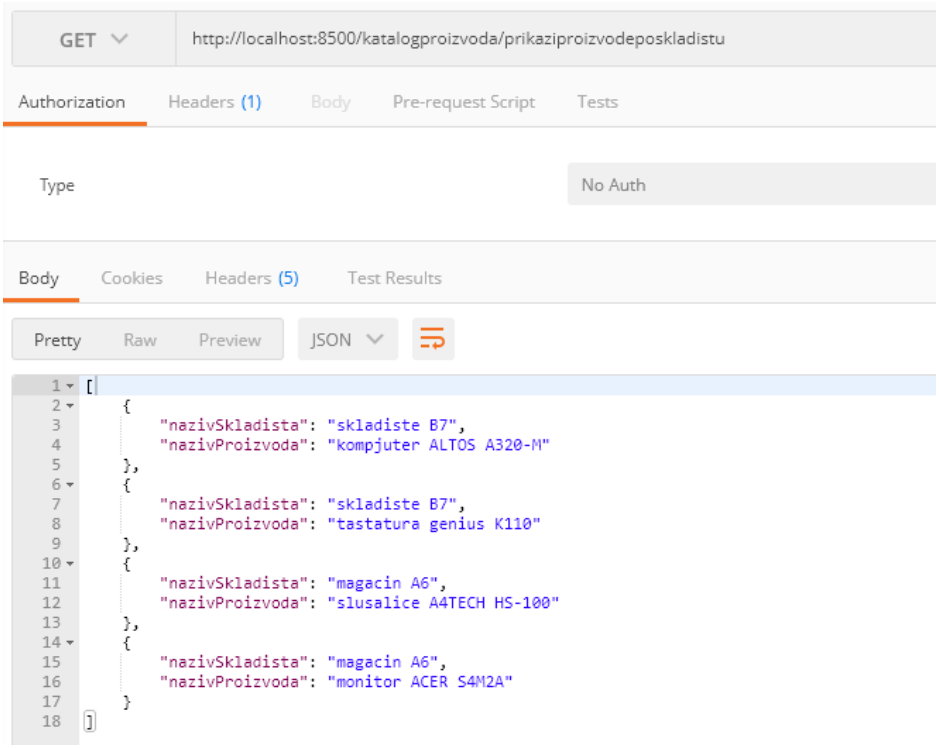
Kreiramo klasu „ProizvodPoSkladistu“ u okviru paketa „dto“ čiji kod je prikazan ispod:

```
public class ProizvodPoSkladistu {

    private String nazivSkladista;
    private String nazivProizvoda;
}
```

Da bismo dva prethodno navedena polja „nazivSkladista“ i „nazivProizvoda“ mapirali sa klasom „ProizvodPoSkladistu“ koju smo kreirali potrebno je vratiti se u klasu „SkladisteRepository“ i sada objasniti kod za bazu podataka koji smo spomenuli u tekstu ranije.
U okviru @Query anotacije potrebno je nakon SELECT naredbe dodati reč „new“, naziv paketa i naziv klase a zatim u okviru zagrade, kao deo konstruktora upisati ova dva polja.

```
@Repository
public interface SkladisteRepository extends JpaRepository <Skladiste,
Integer>{
    @Query("SELECT new
com.projekat.katalog.proizvoda.dto.ProizvodPoSkladistu(s.nazivSkladista ,
p.nazivProizvoda) FROM Skladiste s JOIN s.proizvodi p")
    public List<ProizvodPoSkladistu> prikaziProizvodeUSkladistu();
}
```

Ovaj zahtev će kao rezultat vratiti listu objekata klase „ProizvodPoSkladistu“ sa podacima dobijenim iz kolona koje se nalaze u tabelama baze podataka.
Izlaz metode pod nazivom „nadjiProizvodeUSkladistu“ prikazan je na slici 13.



Slika 13. Metoda prikaziProizvodeUSkladistu, postman aplikacija

U paketu „service“ se nalazi klasa „ProizvodService“ u kojoj će biti ispisana biznis logika i sve standardne CRUD operacije nad jednom tabelom dok će klasa „ProizvodController“ biti zadužena za prezentacioni deo i rukovati zahtevima koje će usmeriti na određenu URL adresu.

Dalje u nastavku je prikazan kod klase „ProizvodService“ iz paketa „service“:

```
package com.projekat.katalog.proizvoda.service;
import com.projekat.katalog.proizvoda.entity.Proizvod;
import com.projekat.katalog.proizvoda.repository.ProizvodRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;

@Service
public class ProizvodService {

    @Autowired
    private ProizvodRepository proizvodRepository;

    public Proizvod sacuvajProizvod(Proizvod proizvod){
        return proizvodRepository.save(proizvod);
    }
    public List<Proizvod> nadjiSveProizvode(){
        return proizvodRepository.findAll();
    }
```

```
    public Proizvod nadjiProizvodPoIdu(int idProizvoda){
        return proizvodRepository.findById(idProizvoda).orElse(null);
    }
    public Proizvod nadjiProizvodPoNazivu(String nazivProizvoda){
        return proizvodRepository.findBynazivProizvoda(nazivProizvoda);
    }
    public String obrisiProizvod(int idProizvoda){
        proizvodRepository.deleteById(idProizvoda);
        return "Obrisan je proizvod koji ima id: " + idProizvoda;
    }
    public Proizvod izmeniProizvod (Proizvod proizvod){
        Proizvod postojeciProizvod =
proizvodRepository.findById(proizvod.getIdProizvoda()).orElse(null);
        postojeciProizvod.setNazivProizvoda(proizvod.getNazivProizvoda());
        postojeciProizvod.setCena(proizvod.getCena());
        postojeciProizvod.setKolicina(proizvod.getKolicina());
        postojeciProizvod.setId_skladista_FK(proizvod.getId_skladista_FK());

        return proizvodRepository.save(postojeciProizvod);
    }
}
```

Anotacija @Service pripada @Component anotaciji koja se koristi da bi označila klasu kao bean (zrno). Pojednostavljeno, to znači da će Spring automatski skenirati našu aplikaciju tražeći klase koje su označene sa @Component anotacijom i njenim pripadnicama kako bi automatski izvršio njihovo instanciranje i ubrizgavanje zavisnosti.
Već smo spomenuli kako funkcionišu POST i GET metode koje predstavljaju create i read operacije u prethodnim primerima.

Ukratko, u POST metodi „sacuvajProizvod“ prosledimo objekat klase proizvod kao parametar u okviru argumenta metoda i kao povratnu vrednost unesemo „proizvodRepository“ objekat koji je nasledio JPARepository metode za upravljanje podacima. Pristupimo metodi „save“ i prosledimo objekat „proizvod“ koji smo već naveli kao parametar u okviru metoda. Izlaz ove metode prikazan je na slici 14 i 15.
Slično prethodnom, u GET metodi „nadjiSveProizvode“ koristimo JPARepository metodu „findAll“ koja će prikazati sve proizvode koji se nalaze u tabeli „proizvod_tbl“. Izlaz ove metode prikazan je na slici 16.
Metodi „nadjiProizvodPoIdu“ kao argument prosleđujemo „idProizvoda“ i koristimo JPARepository metodu „findById“ koja će prikazati proizvod iz baze podataka na osnovu unetog id-a. Izlaz ove metode prikazan je na slici 17.
Metoda „nadjiProizvodPoNazivu“ radi isto što i prethodna metoda ali se umesto id-a prosledi naziv proizvoda. Izlaz ove metode prikazan je na slici 18.
Metoda „izmeniProizvod“ kao argument prima objekat „proizvod“ koji želimo da izmenimo. Kreiramo novi objekat klase „Proizvod“ koji se zove „postojeciProizvod“ koji će na osnovu JPARepository metode „findById“ naći torku u bazi podataka sa id-em proizvoda koji smo mi uneli u okviru zahteva.

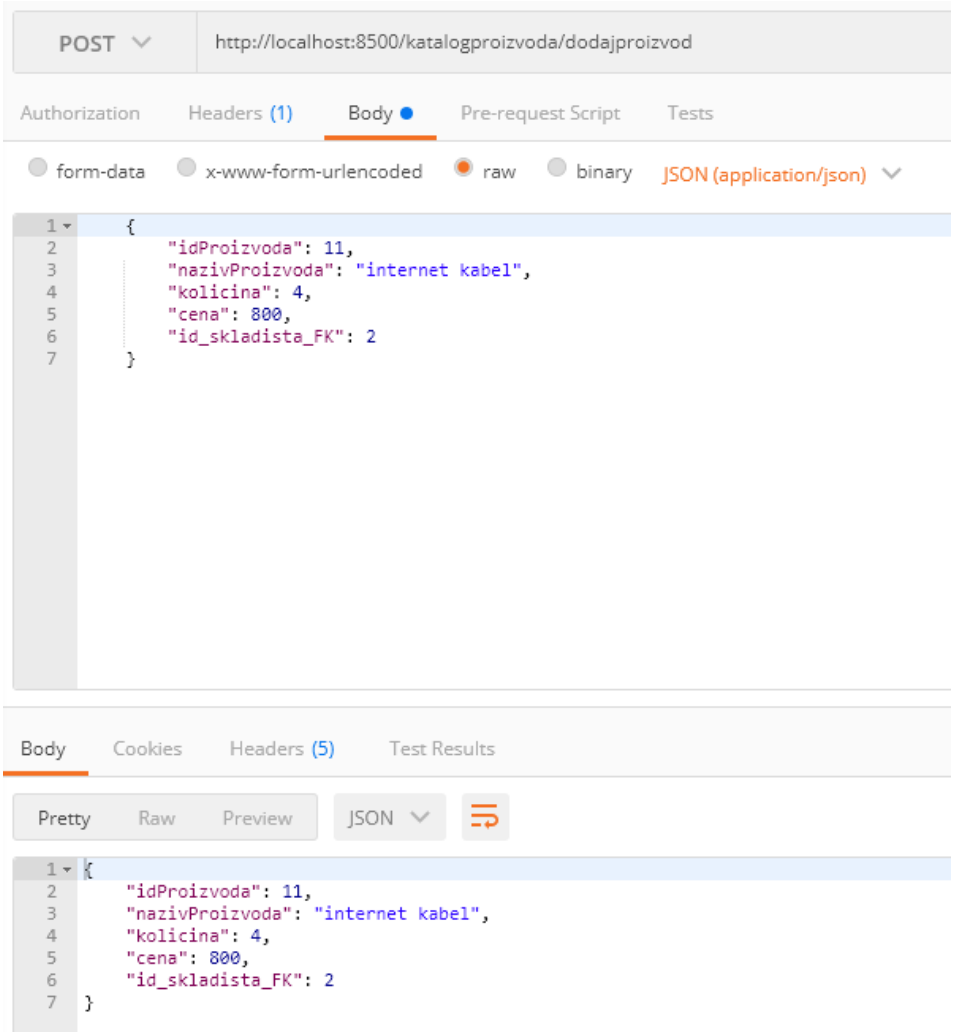
S obzirom da je „idProizvoda“ primarni ključ i da se ta vrednost ne može menjati, potrebno je iskoristiti getter-e i setter-e za sve ostale vrednosti osim za „idProizvoda“ i na kraju kao povratnu vrednost iskoristiti metodu JPARepository „save“ kojoj ćemo proslediti „postojeciProizvod“ čije smo atribute izmenili, osim „idProizvoda“ jer je on primarni ključ. Izlaz ove metode prikazan je na slici 19 i 20.
Metodi „obrisiProizvod“ prosleđujemo parametar „idProizvoda“ i pozivamo JPARepository metodu „deleteById“ gde se na osnovu unetog id-a proizvoda briše dati proizvod iz baze podataka. Kao povratnu vrednost dobijamo poruku koja će ispisati da je obrisan proizvod sa tim id-em ukoliko je metoda izvršena. Izlaz ove metode prikazan je na slici 21 i 22.

Ispod je prikazan kod klase „ProizvodController“ iz paketa „controller“:

```
package com.projekat.katalog.proizvoda.controller;
import com.projekat.katalog.proizvoda.entity.Proizvod;
import com.projekat.katalog.proizvoda.service.ProizvodService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import java.util.List;
@RestController
@RequestMapping("/katalogproizvoda")

public class ProizvodController {
    @Autowired
    private ProizvodService proizvodService;
    @PostMapping("/dodajproizvod")
    public Proizvod dodajProizvod(@RequestBody Proizvod proizvod){
        return proizvodService.sacuvajProizvod(proizvod);
    }
    @GetMapping("/proizvodi")
    public List<Proizvod> traziSveProizvode(){
        return proizvodService.nadjiSveProizvode();
    }
    @GetMapping("/proizvodpoidu/{id}")
    public Proizvod traziProizvodPoIdu(@PathVariable ("id") int idProizvoda)
{
        return proizvodService.nadjiProizvodPoIdu(idProizvoda);
    }
    @GetMapping("/proizvodponazivu/{ime}")
    public Proizvod traziProzivodPoNazivu(@PathVariable String ime){
        return proizvodService.nadjiProizvodPoNazivu(ime);
    }
    @DeleteMapping("/obrisiproizvod/{id}")
    public String ukloniProizvod(@PathVariable ("id") int idProizvoda){
        return proizvodService.obrisiProizvod(idProizvoda);
    }
    @PutMapping("/promeniproizvod")
    public Proizvod promeniProizvod(@RequestBody Proizvod proizvod){
        return proizvodService.izmeniProizvod(proizvod);
    }
}
```

U „ProizvodController“ klasu smo ubrizgali servis kako bismo mogli da pristupimo metodama klase „ProizvodService“.
Ono što je ovde još potrebno objasniti jesu vitičaste zagrade u okviru poslednje tri metode. Kod metode „ukloniProizvod“ navedena je anotacija @PathVariable unutar argumenata metode. Ta anotacija se odnosi na „idProizvoda“. Kada budemo pristupili URL-u http://localhost:8500/katalogproizvoda/obrisiproizvod/, ono što upišemo nakon poslednje kose crte (/) koristiće se kao argument te metode.
http://localhost:8500/katalogproizvoda/obrisiproizvod/2 - u ovom primeru kao deo zahteva, obrisaće se proizvod iz baze podataka čiji je “idProizvoda: 2”.

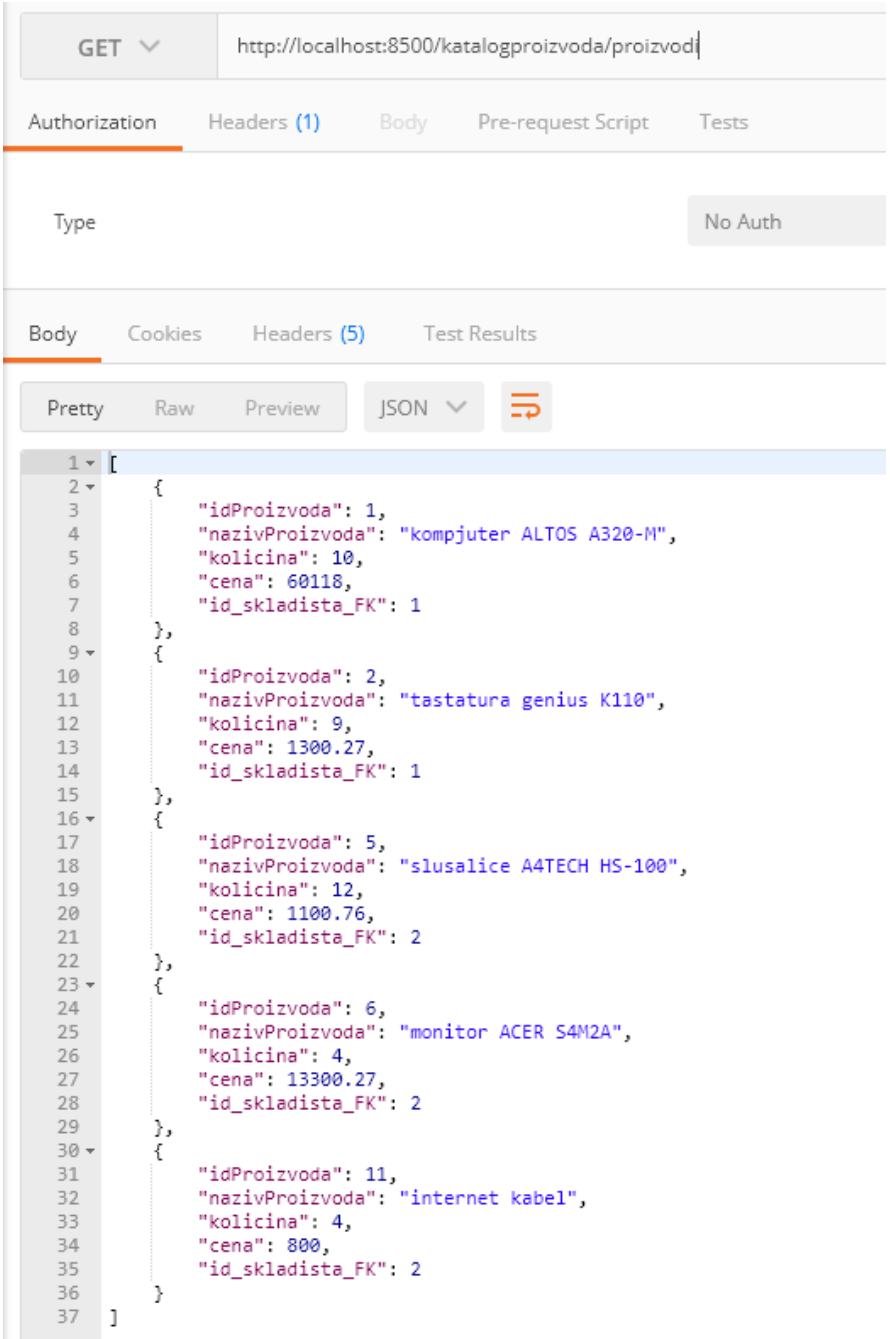
U nastavku su prikazane sve CRUD operacije koje su napomenute a odnose se na metode za izmenu podataka u tabeli baze podataka “proizvod_tbl” čiji se kodovi nalaze u klasama “ProizvodService” i “ProizvodController”.



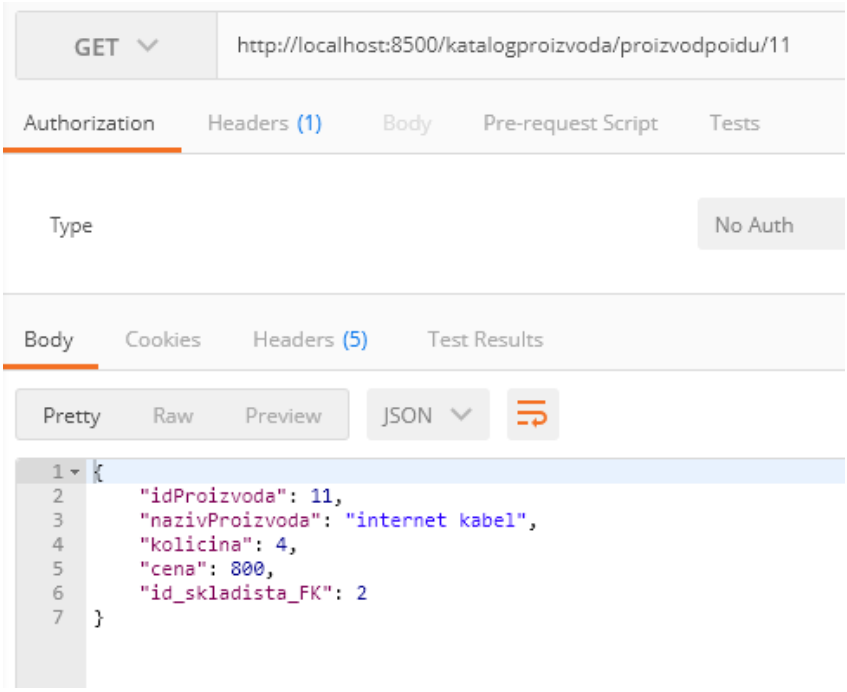
Slika 14. Metoda dodajProizvod, postman aplikacija

id_proizvoda	naziv_proizvoda	kolicina	cena	id_skladista_fk
1	kompjuter ALTOS A320-M	10	60118	1
2	tastatura genius K110	9	1300.27	1
5	slusalice A4TECH HS-100	12	1100.76	2
6	monitor ACER S4M2A	4	13300.27	2
11	internet kabel	4	800	2
NULL	NULL	NULL	NULL	NULL

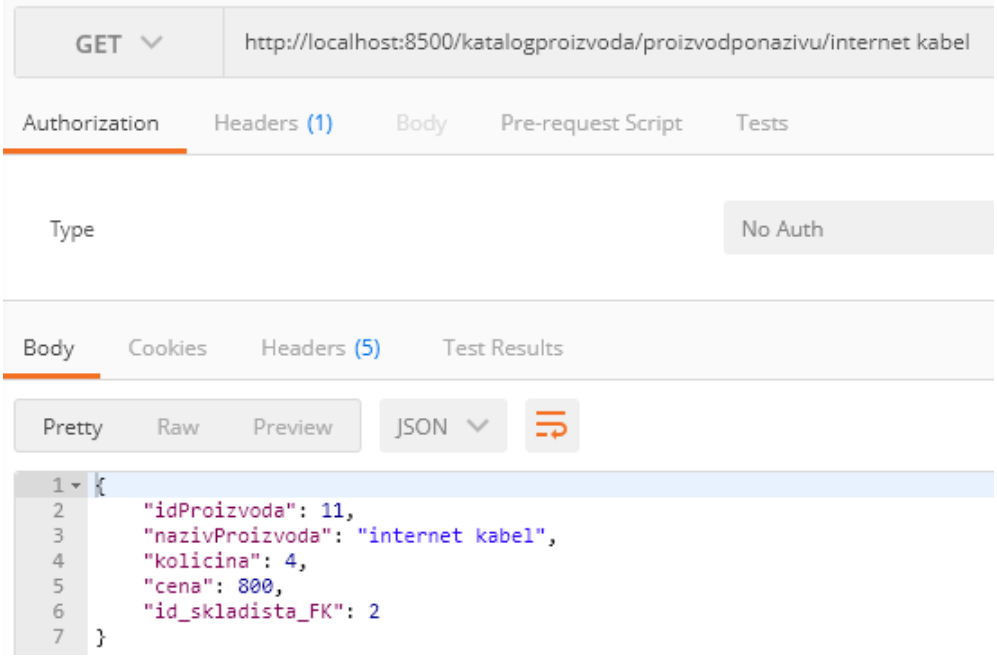
Slika 15. Metoda dodajProizvod, tabela proizvod_tbl u bazi podataka



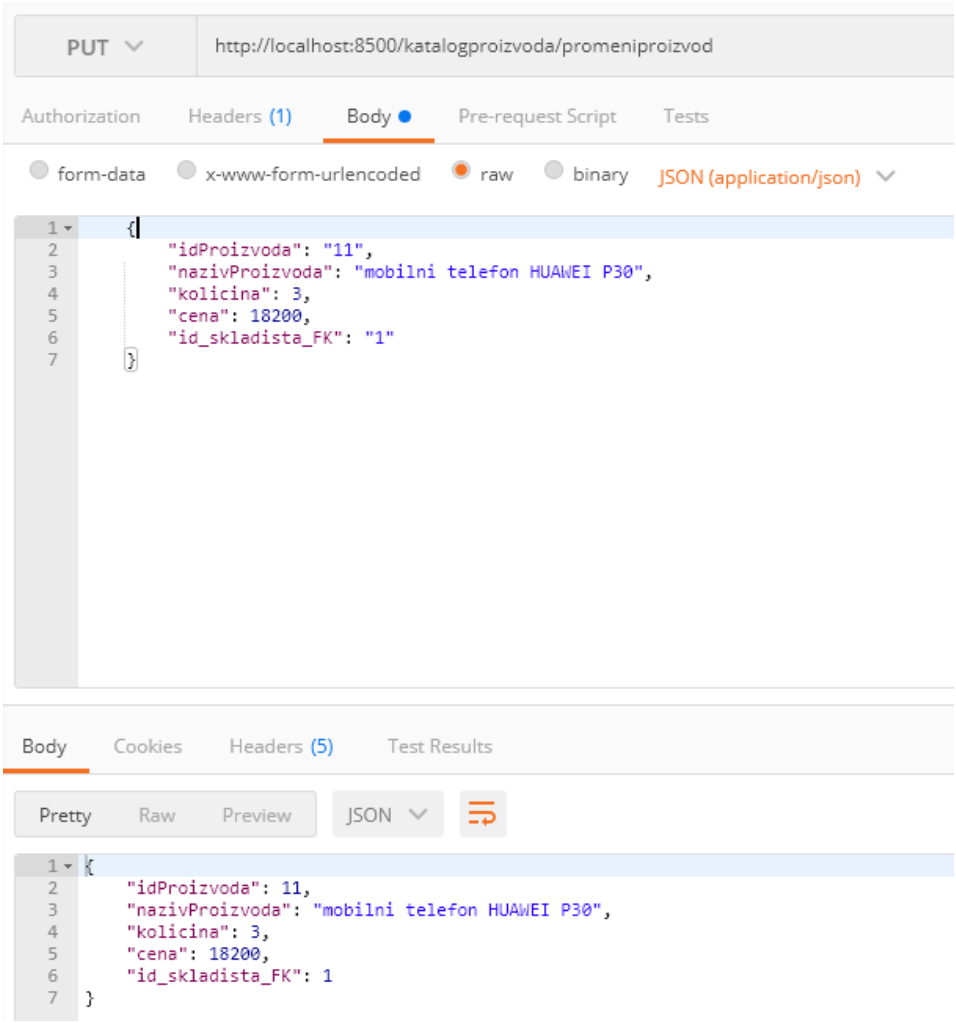
Slika 16. Metoda traziSveProizvode, postman aplikacija



Slika 17. Metoda traziProizvodPoIdu, postman aplikacija



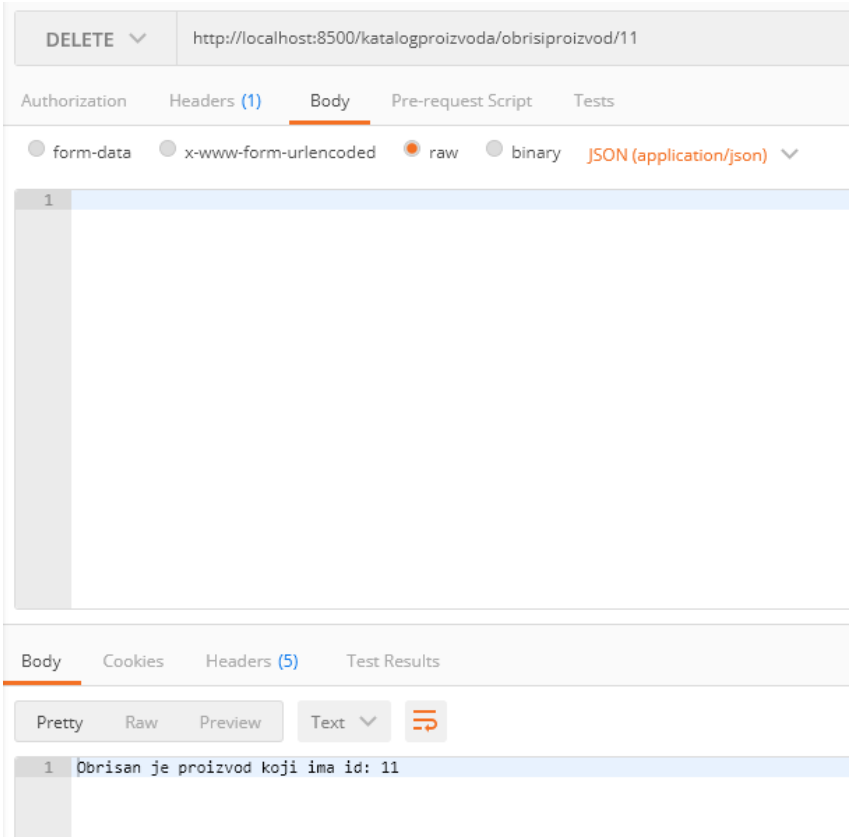
Slika 18. *Metoda* traziProizvodPoNazivu, postman aplikacija



Slika 19. Metoda promeniProizvod, postman aplikacija

id_proizvoda	naziv_proizvoda	kolicina	cena	id_skladista_fk
1	kompjuter ALTOS A320-M	10	60118	1
2	tastatura genius K110	9	1300.27	1
5	slusalice A4TECH HS-100	12	1100.76	2
6	monitor ACER S4M2A	4	13300.27	2
11	mobilni telefon HUAWEI P30	3	18200	1
NULL	NULL	NULL	NULL	NULL

Slika 20. Metoda promeniProizvod, tabela proizvod_tbl u bazi podataka



Slika 21. Metoda ukloniProizvod, postman aplikacija

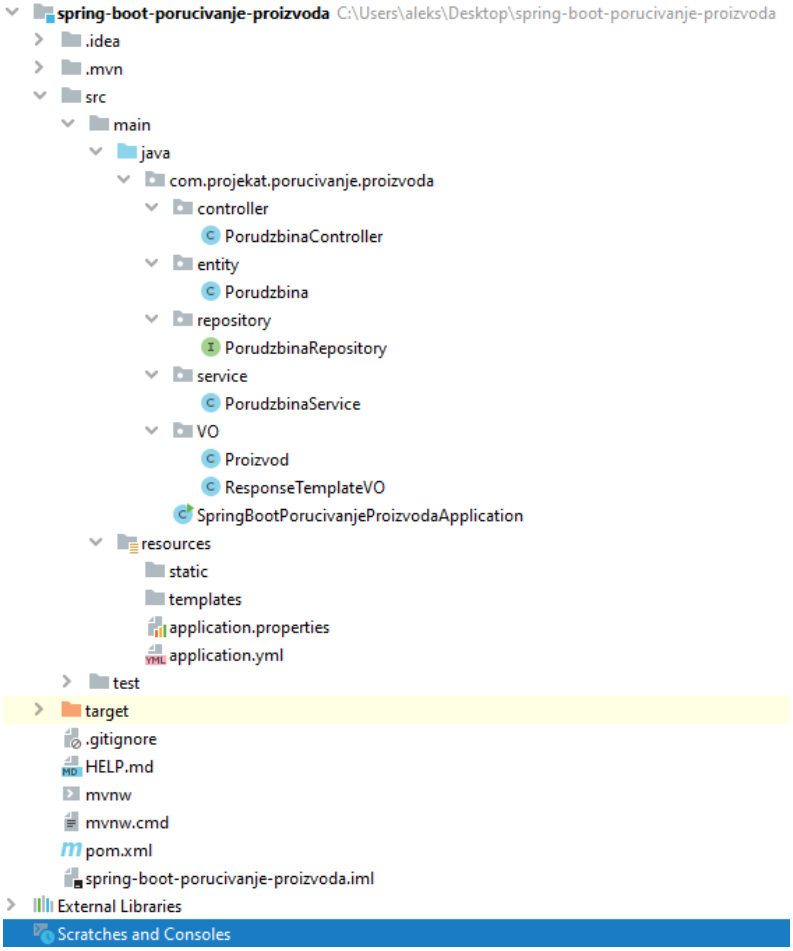
id_proizvoda	naziv_proizvoda	kolicina	cena	id_skladista_fk
1	kompjuter ALTOS A320-M	10	60118	1
2	tastatura genius K110	9	1300.27	1
5	slusalice A4TECH HS-100	12	1100.76	2
6	monitor ACER S4M2A	4	13300.27	2
NULL	NULL	NULL	NULL	NULL

Slika 22. Metoda ukloniProizvod, tabela proizvod_tbl u bazi podataka

Ispod je prikazan kod „main“ klase ovog projekta:
Paket com.projekat.katalog.proizvoda, klasa: SpringBootKatalogProizvodaApplication

```
package com.projekat.katalog.proizvoda;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringBootKatalogProizvodaApplication {

   public static void main(String[] args) {
      SpringApplication.run(SpringBootKatalogProizvodaApplication.class,
args);
   }

}
```

Projekat: porucivanje proizvoda



Slika 23. Struktura aplikacije poručivanje proizvoda

Projekat poručivanje proizvoda čija je struktura prikazana na slici 23 ima iste zavisnosti kao i katalog proizvoda koji je prethodno komentarisan a pored toga koristi i istu programsku logiku i metode za pisanje i čitanje podataka. Fajl u okviru kog se podešava konekcioni string sa bazom podataka je takođe identičan jer koristi istu bazu podataka kao i prethodni projekat a jedino što se razlikuje je port na kom će biti registrovana aplikacija kada bude aktivna i naziv aplikacije u application.yml fajlu koji je „PORUCIVANJE-PROIZVODA“. Katalog proizvoda je registrovan na portu 8500, dok će ovaj projekat (poručivanje proizvoda) biti registrovan na portu 8600.

U skladu sa tim, mislim da nije potrebno detaljno objašnjavati iste stvari koje su već obrađene, već spomenuti one stavke koje predstavljaju novinu a koje bi trebalo da upotpune priču o mikroservisima koji će međusobno komunicirati preko mreže da bi ispunili korisničke zahteve.
U nastavku sledi kod svih klasa sa tim da će klasa „PorudzbinaService“ biti najznačajnija jer su u njoj napisane metode o kojima bih rekao par reči.

U nastavku je prikazan kod klase „Porudzbina“ iz paketa „entity“:

```
package com.projekat.porucivanje.proizvoda.entity;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import javax.persistence.*;
@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
@Table(name = "porudzbina_tbl")
public class Porudzbina {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int idKorisnika;
    private String imeKorisnika;
    private String prezimeKorisnika;
    private String adresaKorisnika;
    private int idProizvoda;
}
```

Dalje je prikazan kod klase „PorudzbinaRepository“ u okviru paketa „repository“:

```
package com.projekat.porucivanje.proizvoda.repository;
import com.projekat.porucivanje.proizvoda.entity.Porudzbina;import
org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
@Repository
public interface PorudzbinaRepository extends JpaRepository<Porudzbina,
Integer> {
    Porudzbina findByidKorisnika(int idKorisnika);
}
```

Ispod je prikazan kod klase „Proizvod“ iz paketa „VO“ (skraćeno od „value objects“):

```
package com.projekat.porucivanje.proizvoda.VO;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
@Data
@AllArgsConstructor
@NoArgsConstructor

public class Proizvod {
    private Long idProizvoda;
    private String nazivProizvoda;
    private int kolicina;
    private double cena;
}
```

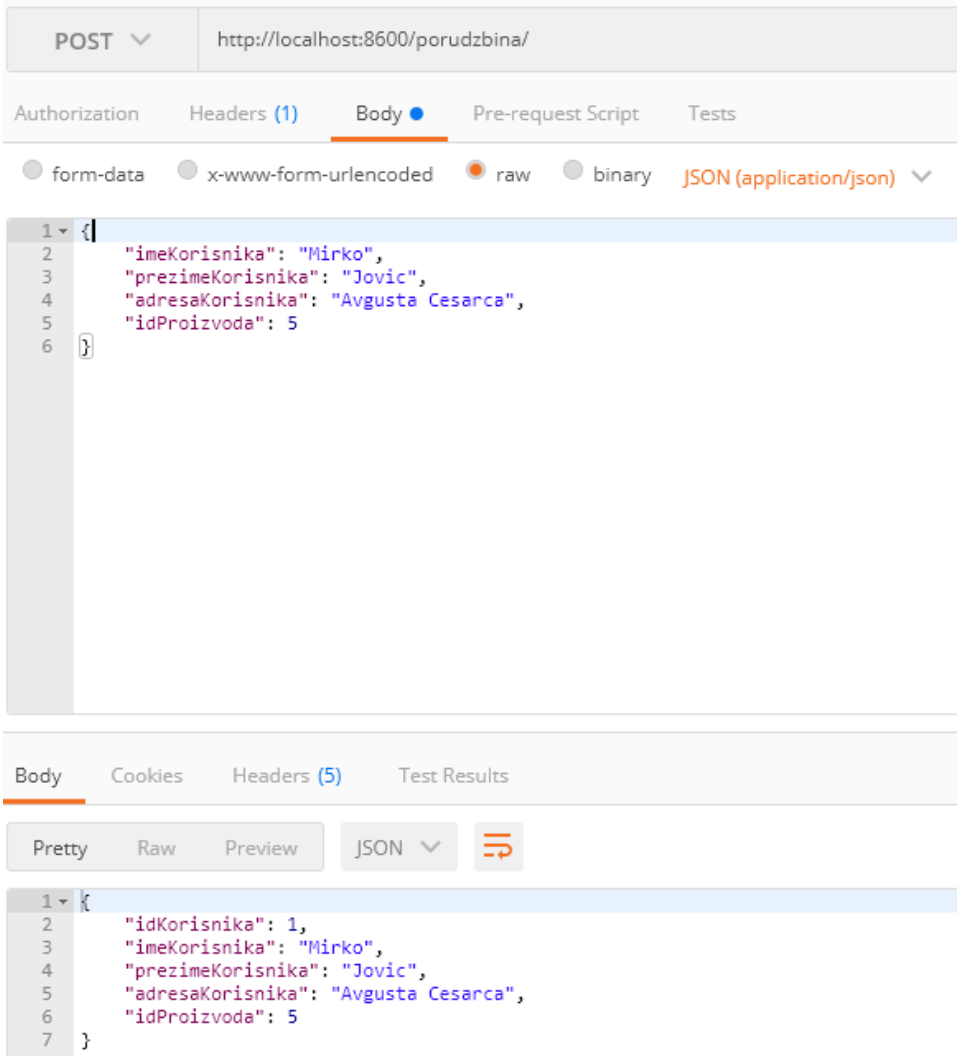
Dalje je prikazan kod klase „ResponseTemplateVO“ iz paketa „VO“:

```
package com.projekat.porucivanje.proizvoda.VO;
import com.projekat.porucivanje.proizvoda.entity.Porudzbina;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor

public class ResponseTemplateVO {
    Porudzbina porudzbina;
    Proizvod proizvod;
}
```

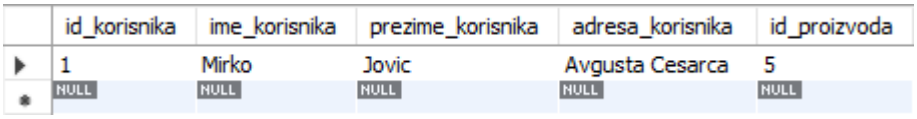
U nastavku je prikazan kod klase „PorudzbinaService“ u okviru paketa „service“:

```
package com.projekat.porucivanje.proizvoda.service;
import com.projekat.porucivanje.proizvoda.VO.Proizvod;
import com.projekat.porucivanje.proizvoda.VO.ResponseTemplateVO;
import com.projekat.porucivanje.proizvoda.entity.Porudzbina;
import com.projekat.porucivanje.proizvoda.repository.PorudzbinaRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.client.RestTemplate;
@Service
public class PorudzbinaService {
    @Autowired
    PorudzbinaRepository porudzbinaRepository;

    public Porudzbina napraviPorudzbinu(@RequestBody Porudzbina porudzbina){
        return porudzbinaRepository.save(porudzbina);
    }
    @Autowired
    private RestTemplate restTemplate;
    public ResponseTemplateVO prikazDetaljnePorudzbine(int idKorisnika){
        ResponseTemplateVO vo = new ResponseTemplateVO();

        Porudzbina porudzbina =
porudzbinaRepository.findByidKorisnika(idKorisnika);
        Proizvod proizvod =
restTemplate.getForObject("http://localhost:8500/katalogproizvoda/proizvodpo
idu/"
 + porudzbina.getIdProizvoda(), Proizvod.class);
        vo.setPorudzbina(porudzbina);
        vo.setProizvod(proizvod);
        return  vo;
    }
}
```

Metodi „napraviPorudzbinu“ u klasi „PorudzbinaService“ prosledimo objekat klase „Porudzbina“ kao parametar u okviru argumenta metode i kao povratnu vrednost unesemo „porudzbinaRepository“ objekat koji je nasledio JPARepository metode za upravljanje podacima. Pristupimo metodi „save“ i prosledimo objekat „porudzbina“ koji smo već naveli kao parametar u okviru metode.
Izlaz ove metode prikazan je na slikama 24 i 25.



Slika 24. Metoda napraviPorudzbinu, postman aplikacija



id_korisnika	ime_korisnika	prezime_korisnika	adresa_korisnika	id_proizvoda
1	Mirko	Jovic	Avgusta Cesarca	5
NULL	NULL	NULL	NULL	NULL

Slika 25. Metoda napraviPorudzbinu, tabela porudzbina_tbl u bazi podataka

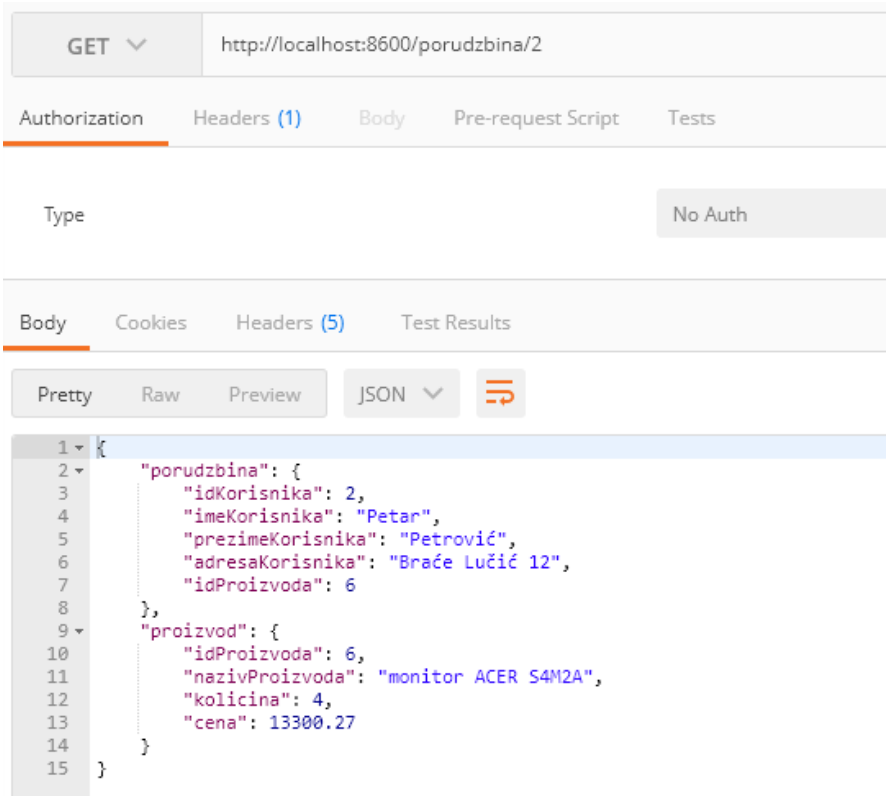
Metoda prikazDetaljnePorudzbine u klasi „PorudzbinaService“

Ova metoda, čiji izlaz je prikazan na slici 26 pokazuje kako dva različita projekta mogu zajedno da komuniciraju kada su uključeni paralelno na različitim portovima a to se čini putem RestTemplate objekta. RestTemplate klasa se koristi za HTTP pozive, odnosno da poziva REST servise. Ovaj RestTemplate objekat je kreiran u okviru „main“ klase i kasnije pozvana njegov metoda „getForObject(url, classType)“ u klasi „ProizvodService“. Metoda „getForObject“ snima (zapisuje) podatke iz navedenog URL-a korišćenjem HTTP GET metode.

U klasi ResponseTemplateVO imamo dva objekta; jedan je objekat klase „Proizvod, a drugi je objekat klase „Porudzbina“ pošto želimo da prikažemo sve podatke o jednom korisniku, i sve podatke o proizvodu koji je naručio, te se ovo navodi u okviru povratne vrednosti metoda „prikazDetaljnePorudbine“.

Objekat „porudzbina“ klase „Porudzbina“ kao argument uzima „idKorisnika“ i na osnovu tog id-a iščitava celu torku u bazi podataka.
Objekat „proizvod“ klase „Proizvod“ poziva restTemplate metodu getForObject kojoj prosleđujemo url: http://localhost:8500/katalogproizvoda/proizvodpoidu/ i dodaje „idProizvoda“ koji se nalazi u torki koja je iščitana na osnovu „idKorisnika“.

Ako kao primer unesemo sledeći url: http://localhost:8600/porudzbina/2 , ovaj metod će potražiti u bazi podataka u tabeli „porudzbina_tbl“ korisnika čiji je id: 2; videti koji je proizvod naručio dati korisnik na osnovu „idProizvoda“ i taj „idProizvoda“ će biti unesen u link http://localhost:8500/katalogproizvoda/proizvodpoidu/ nakon kose crte (/) jer nakon linka sledi kod: “+ porudzbina.getIdProizvoda()”. Kao rezultat, metod će prikazati sve podatke o jednoj porudžbini iz baze podataka “porudzbina_tbl” i sve podatke o konkretnom proizvodu koji je poručen ali koji nije dobijen iz baze podataka, već sa URL-a:
http://localhost:8500/katalogproizvoda/proizvodpoidu/2



Slika 26. Metoda prikazDetaljnePorudzbine, postman aplikacija

Dalje će ovaj servis biti implementiran u klasi "PorudzbinaController" kako bi se definisao URL putem kojeg će mu se pristupiti i kako bismo odvojili biznis logiku u service klasu, a način pristupanja u controller klasu.

Ispod je prikazan kod klase „PorudzbinaController“ iz paketa „controller“:

```
package com.projekat.porucivanje.proizvoda.controller;
import com.projekat.porucivanje.proizvoda.VO.ResponseTemplateVO;
import com.projekat.porucivanje.proizvoda.entity.Porudzbina;
import com.projekat.porucivanje.proizvoda.service.PorudzbinaService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/porudzbina")

public class PorudzbinaController {

    @Autowired
    PorudzbinaService porudzbinaService;
    @PostMapping("/")
    public Porudzbina napraviPorudzbinu(@RequestBody Porudzbina porudzbina)
{
        return porudzbinaService.napraviPorudzbinu(porudzbina);
    }
    @GetMapping("/{id}")
    public ResponseTemplateVO prikazPorudzbine(@PathVariable ("id") int
idKorisnika){
        return porudzbinaService.prikazDetaljnePorudzbine(idKorisnika);
    }
}
```

Dalje je prikazan kod „main klase“:
paket: com.projekat.porucivanje.proizvoda,
klasa: SpringBootPorucivanjeProizvodaApplication

```
package com.projekat.porucivanje.proizvoda;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;
@SpringBootApplication
public class SpringBootPorucivanjeProizvodaApplication {

   public static void main(String[] args) {
      SpringApplication.run(SpringBootPorucivanjeProizvodaApplication.class,
args);
   }

   @Bean
   @LoadBalanced
   public RestTemplate restTemplate(){
      return new RestTemplate();
   }
}
```

Eureka server predstavlja aplikaciju koja sadrži informacije o svim klijent-servis aplikacijma. Svaki mikroservis se registruje na eureka server tako da server zna na kom portu je aktivna koja aplikacija i njenu IP adresu. Nakon registrovanja, svakih 30 sekundi servis pinguje eureka server da bi ga obavestio da je servis dostupan. Ako eureka ne dobije nikakav ping (signal) od strane servisa u određenom vremenskom periodu, taj servis se odjavljuje sa eureka servera.

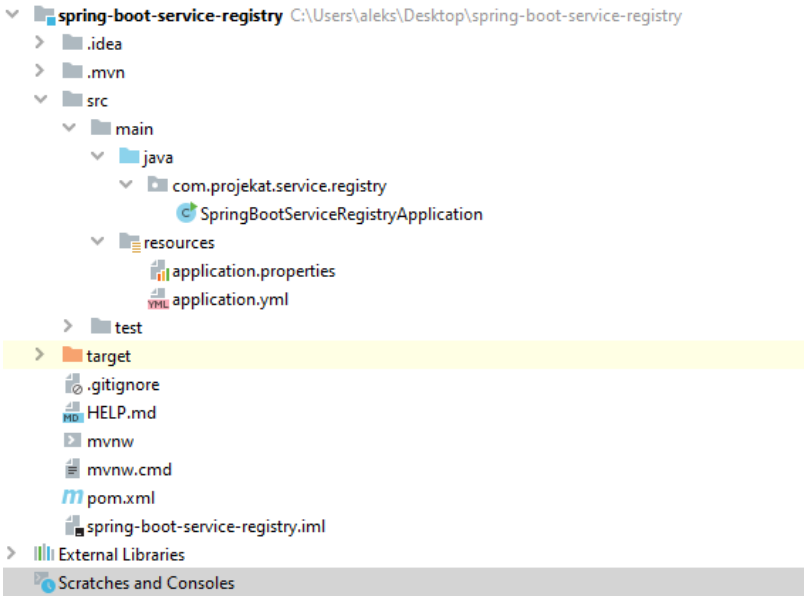
Da bismo implementirali eureka server, potrebno je da uvezemo zavisnost (dependency) koja se na nju odnosi a koja je prikazana u kodu koji se nalazi ispod. U pom.xml fajlu neophodno je dodati sledeće linije teksta:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

Nakon toga potrebno je dodati anotaciju @EnableEurekaServer u okviru "main" klase i podesiti određena svojstva vezana za portove i konekciju servera sa klijent aplikacijama.

U nastavku je prikazan projekat pod nazivom "spring-boot-service-registry" u kom su navedeni koraci ispunjeni u cilju podizanja eureka servera.

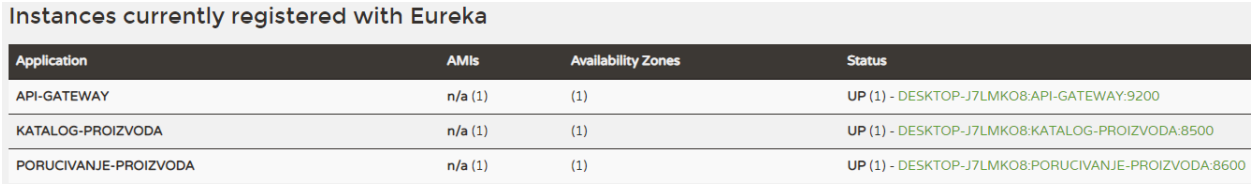
Projekat: service registry



Slika 27. Struktura aplikacije service registry

Na slici 27 prikazana je struktura projekta “service registry” koji zapravo predstavlja konfiguraciju eureka servera.

U fajlu application.yml podešen je port 8761 na kom će eureka server biti aktivan kada je pokrenuta aplikacija.
Nakon ovih podešavanja potrebno je pokrenuti sva tri projekta: katalog proizvoda, poručivanje proizvoda i service registry pa zatim u brauzeru u okviru URL adrese ukucati port na kom je eureka aktivna što je port 8761.
Ovo će nas odvesti na stranicu gde možemo videti da su se obe aplikacije uspešno registrovale kao servisi pod nazivom kako smo i podesili u application.yml fajlu: “KATALOG-PROIZVODA” i “PORUCIVANJE-PROIZVODA” što je prikazano na slici 28.


Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
API-GATEWAY	n/a (1)	(1)	UP (1) - DESKTOP-J7LMKO8:API-GATEWAY:9200
KATALOG-PROIZVODA	n/a (1)	(1)	UP (1) - DESKTOP-J7LMKO8:KATALOG-PROIZVODA:8500
PORUCIVANJE-PROIZVODA	n/a (1)	(1)	UP (1) - DESKTOP-J7LMKO8:PORUCIVANJE-PROIZVODA:8600

Slika 28. Eureka Server – registrovane aplikacije

Sada kada smo uspešno podesili eureku vratićemo se u “PorudbzinaService” klasu aplikacije koja je registrovana kao “PORUCIVANJE-PROIZVODA”. U okviru metode “napraviPorudzbinu” u objektu proizvod zamenićemo broj porta nazivom aplikacije, tako da će ova metoda raditi nezavisno od toga na kom portu će biti aktivna aplikacija “KATALOG-PROIZVODA”

Pre izmene koda:

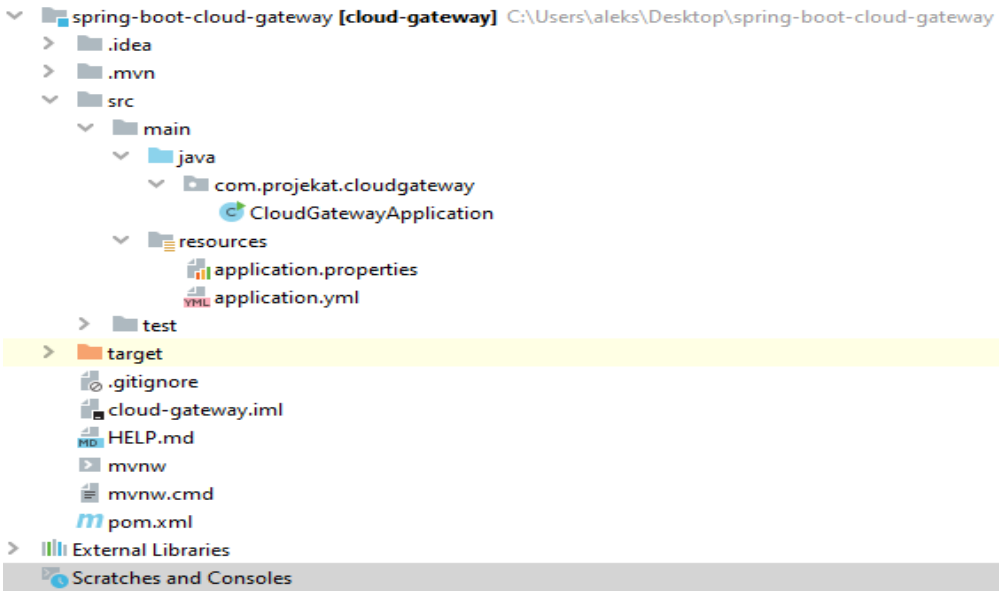
```
Proizvod proizvod =
restTemplate.getForObject("http://localhost:8500/katalogproizvoda/proizvodpo
idu/"
 + porudzbina.getIdProizvoda(), Proizvod.class);
```

Posle izmene koda:

```
Proizvod proizvod = restTemplate.getForObject("http://KATALOG-
PROIZVODA/katalogproizvoda/proizvodpoidu/" + porudzbina.getIdProizvoda(),
Proizvod.class);
```

Ovo je urađeno zato što ću za kraj predstaviti još jednu aplikaciju koja će funkcionisati kao mrežni prolaz (gateway – eng.) i preusmeravaće sve zahteve sa različitih portova na jedan port, a ono što je potrebno navesti u okviru URL-a biće broj porta na kom se nalazi mrežni prolaz i zatim naziv “katalogproizvoda” ako želimo pristupiti metodama kataloga proizvoda, odnosno naziv “porudzbina” ako želimo pristupiti metodama poručivanja proizvoda. Dakle, neće više biti potrebno unositi port 8500 ako je u pitanju katalog proizvoda, odnosno 8600 ako je u pitanju poručivanje proizvoda, već će obe aplikacije raditi na novom portu, koji je u konfiguraciji podešen kao port 9200.

Projekat: cloud gateway

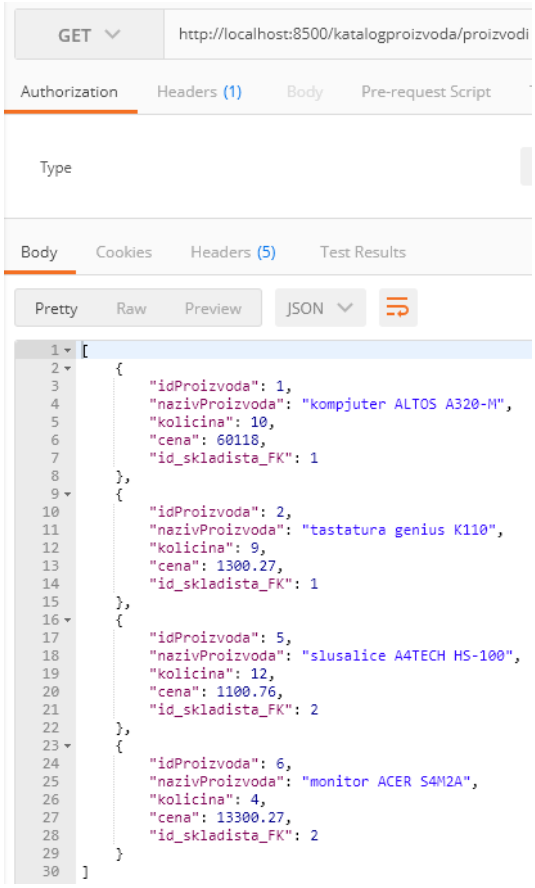


Slika 29. Struktura aplikacije cloud gateway

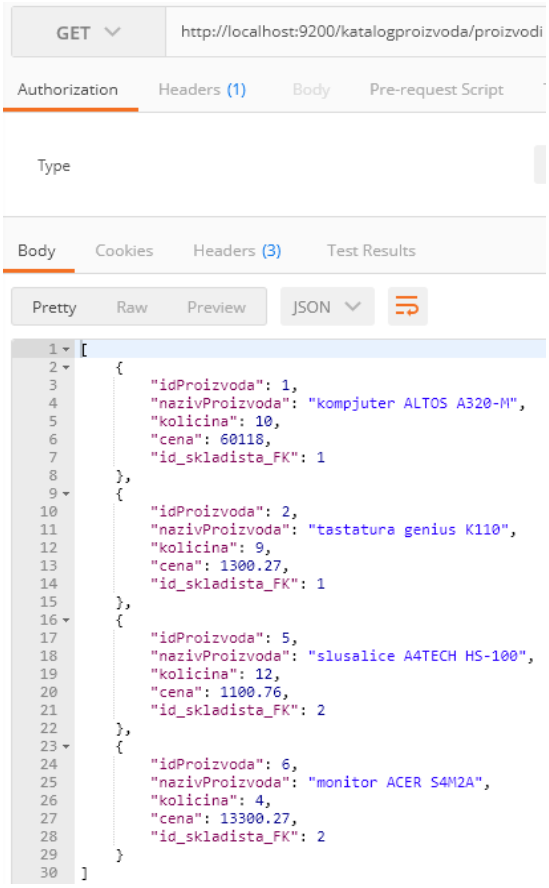
Struktura aplikacije cloud gateway prikazana je na slici 29.

Za uspešnu konfiguraciju mrežnog prolaza neophodno je dodati u pom.xml fajl zavisnost za mrežni prolaz koja se koristi za usmeravanje API-a i eureka discovery client da bi eureka locirala i ovu aplikaciju.
U main klasi potrebno je dodati anotaciju @EnableEurekaClient i uraditi još neka podešavanja vezana za eureku kao i za usmeravanje zahteva i umrežavanje servisa. Nakon ovih podešavanja svaki zahtev na portu 9200 koji u URL-u ima naziv "katalogproizvoda" treba da bude preusmeren na aplikaciju "KATALOG-PROIZVODA", dok svaki zahtev na istom portu koji u URL-u ima naziv "porudzbina" treba da bude preusmeren na aplikaciju "PORUCIVANJE-PROIZVODA"

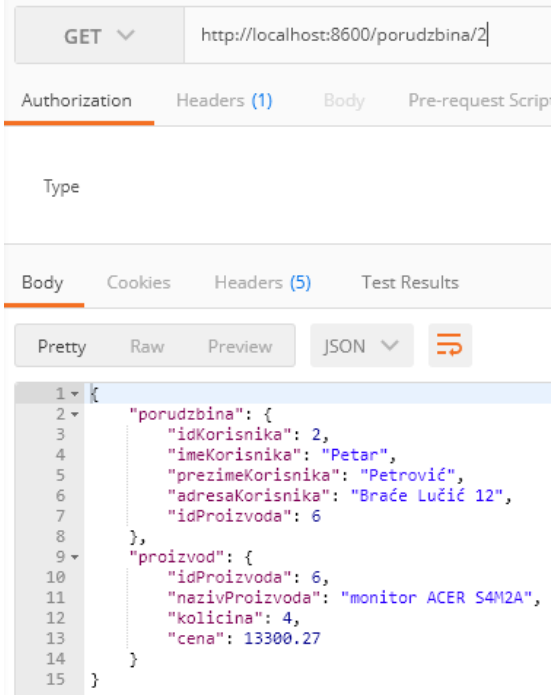
Da bismo potvrdili da mrežni prolaz funkcioniše, u nastavku su prikazane metode na portovima 8500 i 8600, koje odgovaraju aplikacijama katalog proizvoda i poručivanje proizvoda, respektivno.
Bilo koja metoda iz obe aplikacije radiće i na portovima samih aplikacija ali i na portu 9200 na kom je konfigurisan mrežni prolaz.
Zahtev koji treba da prikaže sve proizvode u aplikaciji katalog proizvoda koja je registrovana na portu 8500, radiće i na portu 9200. Izlaz datog zahteva, odnosno metode prikazan je na slikama 30 i 31.
Shodno tome, zahtev koji treba da prikaže porudžbinu korisnika čiji je id: 2 u aplikaciji porudžbina koja je registrovana na portu 8600 radiće i na portu 9200. Izlaz ovog zahteva, odnosno metode prikazan je na slikama 32 i 33.



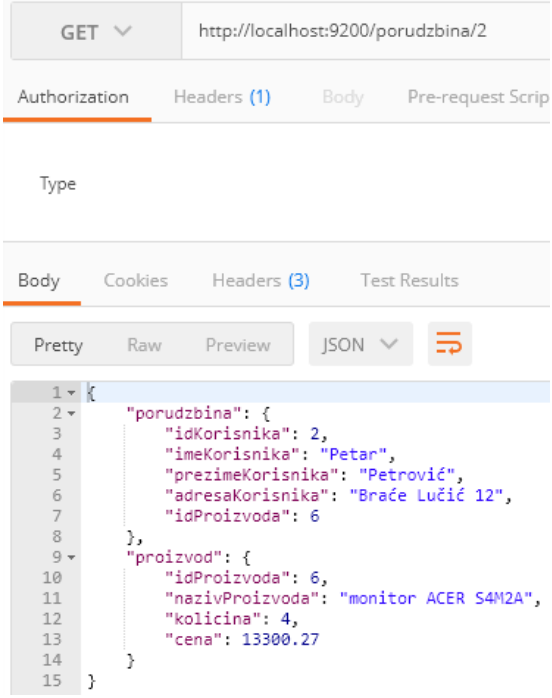
Slika 30. Metoda aplikacije katalog proizvoda na portu 8500



Slika 31. Metoda aplikacije katalog proizvoda na portu 9200



Slika 32. Metoda aplikacije porudzbina na portu 8600



Slika 33. Metoda aplikacije porudzbina na portu 9200

10. Zaključak

Predmet ovog diplomskog rada bila je izrada web aplikacije zasnovane na RESTful arhitekturi. Kako se RESTful arhitektura i mikroservisi dobro kombinuju, a imajući u vidu i njihovu rastuću popularnost odlučio sam se za izradu projekta koji je sačinjen od mikroservisa gde imamo više manjih aplikacija koje su povezane i rade zajedno čime se obezbeđuje funkcionalnost projekta.
Kroz celokupan rad bilo je neophodno obraditi i predstaviti razne pojmove i steći znanja iz različitih oblasti na kojima se projekat zasniva pre nego što je predstavljena aplikacija.
Iako je u ovom projektu predstavljen katalog proizvoda i poručivanje proizvoda, bilo koja aplikacija može da bude struktuirana kao mikroservis.
Podela projekta na više aplikacija olakšalo mi je da se fokusiram na konkretnu funkcionalnost. Prvo je izrađena aplikacija katalog proizvoda a tek kasnije, nakon istraživanja o mikroservisima došao sam na ideju da izradim i aplikaciju za poručivanje proizvoda.
Nadam se da je čitalac, kroz rad i komentarisanje koda mogao da vidi pogodnosti prikazanih arhitektura ali i Spring framework-a koji se ne sme zanemariti a koji mi je u velikoj meri olakšao posao smanjujući količinu koda a naročito i time što je pojednostavio i ubrzao rad sa samim podacima i bazom podataka.

11. Literatura

[1] Brains, J. (2019, 1 1). *What are microservices really all about? - Microservices Basics Tutorial*. Preuzeto sa Youtube: https://www.youtube.com/watch?v=j1gU2oGFayY

[2] cpp-netlib. (n.d.). *Generic URI Syntax Overview¶*. Preuzeto sa cpp-netlib: https://cpp-netlib.org/0.8-beta/uri.html

[3] Fielding, R. (2000). *Architectural styles and the design of network-based software architectures.* Irvine: University of California.

[4] Guru99. (n.d.). *API vs Web Service: What's the Difference?* Preuzeto sa Guru99: https://www.guru99.com/api-vs-web-service-difference.html

[5] IBM. (n.d.). *REST APIs*. Preuzeto sa IBM: https://www.ibm.com/cloud/learn/rest-apis

[6] Janssen, T. (n.d.). *What is Spring Data JPA? And why should you use it?* Preuzeto sa Thorben Janssen: https://thorben-janssen.com/what-is-spring-data-jpa-and-why-should-you-use-it/

[7] java2novice. (n.d.). *What is Retention policy in java annotations?* Preuzeto sa java2novice: https://www.java2novice.com/java-annotations/retention-policy/#:~:text=Annotation%20with%20retention%20policy%20SOURCE,to%20the%20JVM%20through%20runtime.

[8] Javatpoint. (n.d.). *Types of Web Services*. Preuzeto sa Javatpoint: https://www.javatpoint.com/restful-web-services-types-of-web-services

[9] Kraus, L. (2013). *Programski jezik Java sa rešenim zadacima.* Beograd: Akademska misao.

[10] Masse, M. (2011). *REST API Design Rulebook.* California: O'Reilly Media.

[11] Oracle. (1999, January 1). *Design Goals of the Java Programming Language*. Preuzeto sa Oracle: https://www.oracle.com/java/technologies/introduction-to-java.html
[12] Oracle. (2020, Septembar). *Java Language Specification*. Preuzeto sa Oracle: https://docs.oracle.com/javase/specs/jls/se15/html/jls-9.html#jls-NormalAnnotation
[13] Oracle. (n.d.). *Predefined Annotation Types*. Preuzeto sa Oracle: https://docs.oracle.com/javase/tutorial/java/annotations/predefined.html
[14] Pautasso, C. (2009, October 22). Some REST Design Patterns (and Anti-Patterns). Lugano, Switzerland.
[15] Programiz. (n.d.). *Java Annotation Types*. Preuzeto sa Programiz: https://www.programiz.com/java-programming/annotation-types
[16] Programiz. (n.d.). *Programiz*. Preuzeto sa Java Annotations: https://www.programiz.com/java-programming/annotations
[17] Robinson, S. (n.d.). *What is Maven?* Preuzeto sa Stack Abuse: https://stackabuse.com/what-is-maven/
[18] Rodriguez, C., Bez, M., Daniel, F., Casati, F., Trabucco, C. J., Canali, L., & Percannella, G. (2016). *REST APIs: A Large-Scale Analysis of*. Milan: University of Trento.
[19] Spring. (n.d.). *Overview of Spring Framework*. Preuzeto sa Spring: https://docs.spring.io/spring-framework/docs/5.0.0.RC2/spring-framework-reference/images/spring-overview.png
[20] SQLShack. (n.d.). *CRUD operations in SQL Server*. Preuzeto sa SQLShack: https://www.sqlshack.com/crud-operations-in-sql-server/
[21] Stackify. (2017, March 14). *SOAP vs. REST: The Differences and Benefits Between the Two Widely-Used Web Service Communication Protocols*. Preuzeto sa Stackify: https://stackify.com/soap-vs-rest/
[22] sumo logic. (n.d.). *CRUD*. Preuzeto sa sumo logic: https://www.sumologic.com/glossary/crud/
[23] Sun Microsystems. (2011, September 25). *Annotations*. Preuzeto sa Sun Microsystems: https://web.archive.org/web/20110925021948/http://download.oracle.com/javase/1,5.0/docs/guide/language/annotations.html
[24] SymfonyCasts. (n.d.). *PUT Versus POST*. Preuzeto sa SymfonyCasts: https://symfonycasts.com/screencast/rest/put-versus-post#:~:text=Idempotent%C2%B6&text=PUT%20and%20DELETE%20are%20idempotent,results%20in%20the%20same%20state
[25] Test Automation Resources. (2018, May 21). *6 Differences between Web Services vs API (SOAP & REST examples)*. Preuzeto sa Test Automation Resources: https://testautomationresources.com/api-testing /differences-web-services-api/
[26] Tutorialspoint. (n.d.). *Spring - MVC Framework*. Preuzeto sa tutorialspoint: https://www.tutorialspoint.com/spring/spring_web_mvc_framework.htm
[27] TutorialsTeacher. (n.d.). *Dependency Injection*. Preuzeto sa TutorialsTeacher: https://www.tutorialsteacher.com/ioc/dependency-injection#:~:text=Dependency%20Injection%20(DI)%20is%20a,class%20that%20depends%20on%20them.

[28] W3. (2004, February 11). *Web Services Architecture*. Preuzeto sa W3: https://www.w3.org/TR/ws-arch/
[29] W3Schools. (n.d.). *XML Introduction*. Preuzeto sa W3Schools: https://www.w3schools.com/xml/xml_whatis.asp
[30] Wikipedia. (n.d.). *Aspektno-orijentisano programiranje*. Preuzeto sa Wikipedia: https://sr.wikipedia.org/sr-el/%D0%90%D1%81%D0%BF%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D1%98%D0%B5%D0%BD%D1%82%D0%B8%D1%81%D0%B0%D0%BD%D0%BE_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%B8%D1%80%D0%B0%D1%9A%D0%B5
[31] Wikipedia. (n.d.). *Dependency Injection*. Preuzeto sa Wikipedia: https://en.wikipedia.org/wiki/Dependency_injection

Preuzete slike

Slika 1: https://www.sqlshack.com/crud-operations-in-sql-server/
Slika 2: https://docs.spring.io/spring-framework/docs/5.0.0.RC2/spring-framework-reference/images/spring-overview.png
Slika 3: https://www.youtube.com/watch?v=j1gU2oGFayY
Slika 4: https://www.youtube.com/watch?v=j1gU2oGFayY
Slika 5: https://www.youtube.com/watch?v=j1gU2oGFayY
Slika 6: https://www.youtube.com/watch?v=j1gU2oGFayY
Slika 7: https://www.youtube.com/watch?v=j1gU2oGFayY